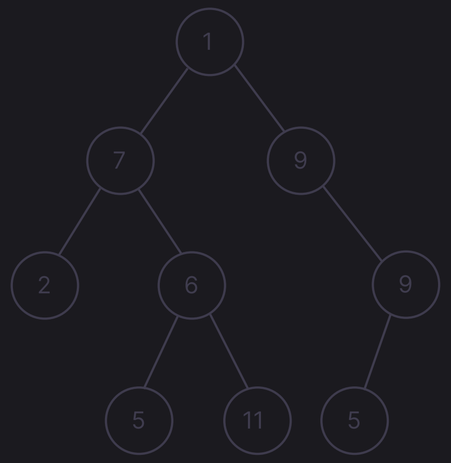
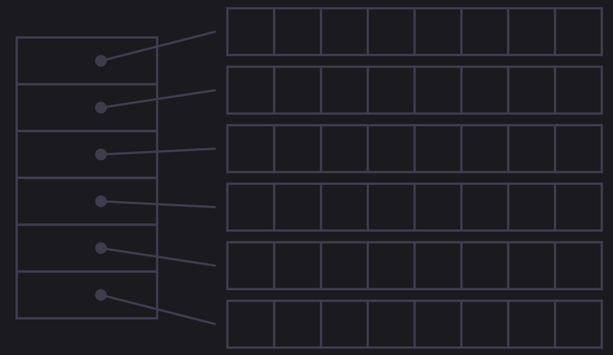
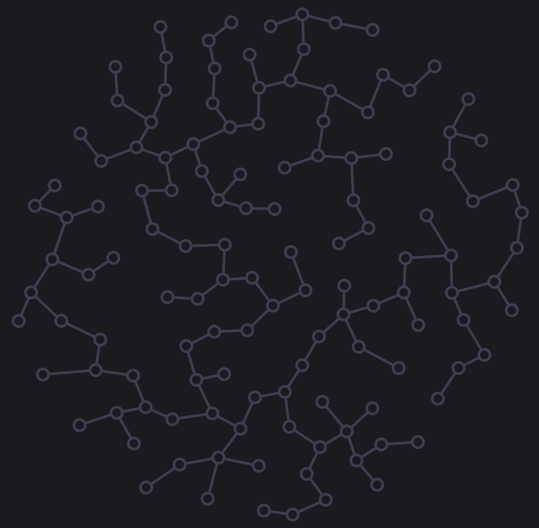
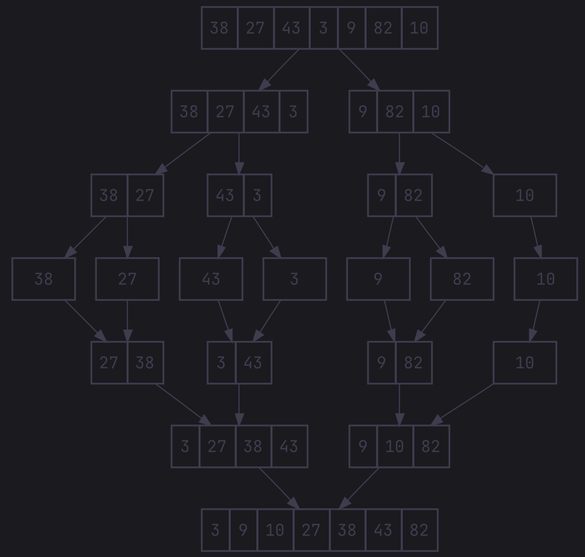




Welcome to

data structures & algorithms





Karl Rombauts

- PhD Student at UTS (Quantum Computing)
- Software Developer
- Co-parent of Molly

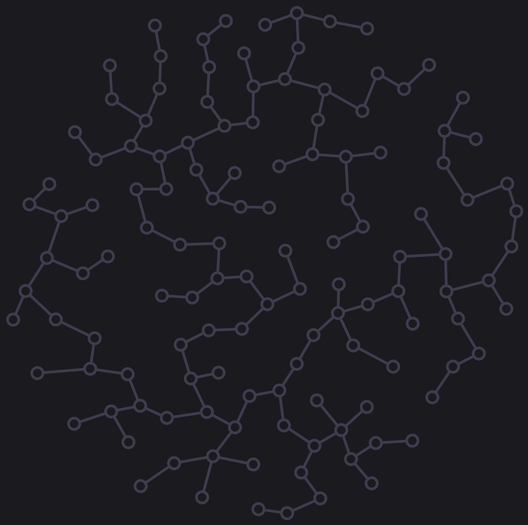
Contact

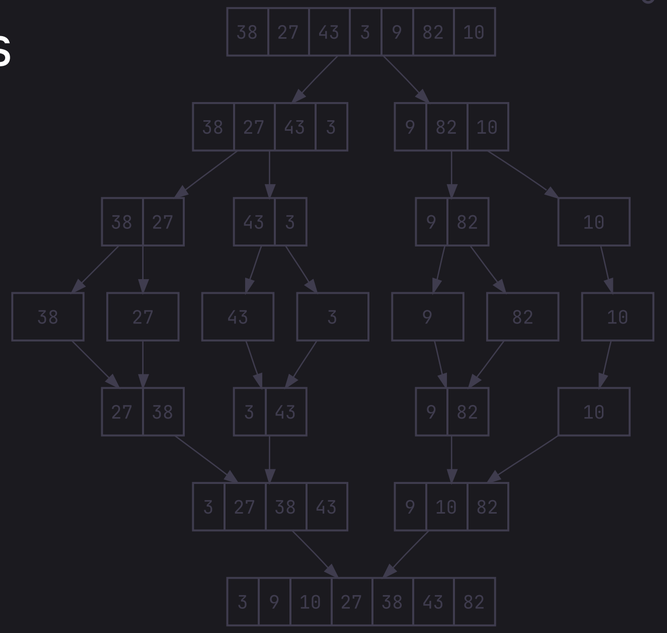
karel.rombauts@uts.edu.au

On the **Ed discussion** board

What to Expect From DSA

- We are going to go through A LOT
- The C++ language
- Pointers and references
- Big O notation
- Complexity theory
- Basic list/array like data structures
- Sorting algorithms
- Algorithm design paradigms
- Binary tree data structure
- Hashing, sets and maps
- Graph data structures
- Graph traversal algorithms
- **And more!**





The Important Stuff!

Programming Assignment 1	20%
Programming Assignment 2	30%
Weekly Exercises (10 in total)	**20%**
Exam	30%

This week’s lab

The **building blocks** for algorithms and data structures

- Input and output
- Functions
- Pointers and references
- Classes

Output (Printing to the console)

```
std::cout
```

- **std** stands for "standard library"
- **cout** stands for "character output"
- Allows us to print text to the console

Output (Printing to the console)

```
std::cout << "Hello\n";
```

- The `<<` symbol is the insertion operator
- The data is flowing from right to left

Output (Printing to the console)

```
std::string msg = "Hello World!\n";
std::cout << msg;
```

- The `<<` symbol is the insertion operator
- The data is flowing from right to left
- We are flowing data from **`msg`** to **`std::cout`**

Input (Reading input from a user)

```
std::cin
```

- **std** stands for "standard library"
- **cin** stands for "character input"
- Allows us to get text that the user types in the console

Input (Reading input from a user)

```
std::string userInput {};
std::cin >> userInput;
```

- The `>>` symbol is the extraction operator
- The data is flowing from left to right
- We are flowing data from **`std::cout`** to **`userInput`**

Input types are important

```
int userInput {};
std::cin >> userInput;
```

- The type is important!
- Now we are expecting an int

Input types are important

```
float userInput {};
std::cin >> userInput;
```

- The type is important!
- Now we are expecting a float

Input types are important

```
std::string userInput {};
std::cin >> userInput;
```

- The type is important!
- Now we are expecting a string

Input & Output

Give the exercises a go

and ask questions if you get stuck :)

Functions

```
int add(int a, int b) {
    return a + b
}
```

Exercise: Factorial Function

```
int factorial(int n) {
    // implementation
}
```

Factorial

$$n!$$

A factorial is the multiplication of all positive integers from 1 up to a given number.

Factorial

$$3! = 3 \times 2 \times 1 = 6$$

A factorial is the multiplication of all positive integers from 1 up to a given number.

Factorial

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

A factorial is the multiplication of all positive integers from 1 up to a given number.

Factorial

$$0! = 1$$

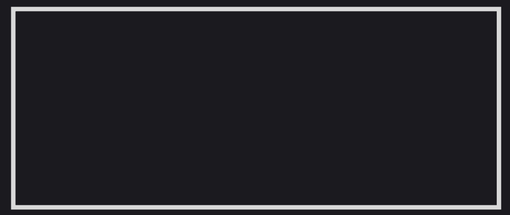
And because it makes the maths nicer,
mathematicians define 0! to be equal to 1

Factorial

Give the exercise a go

and ask questions if you get stuck :)

Variables



- Variables are like a **box**

Variables

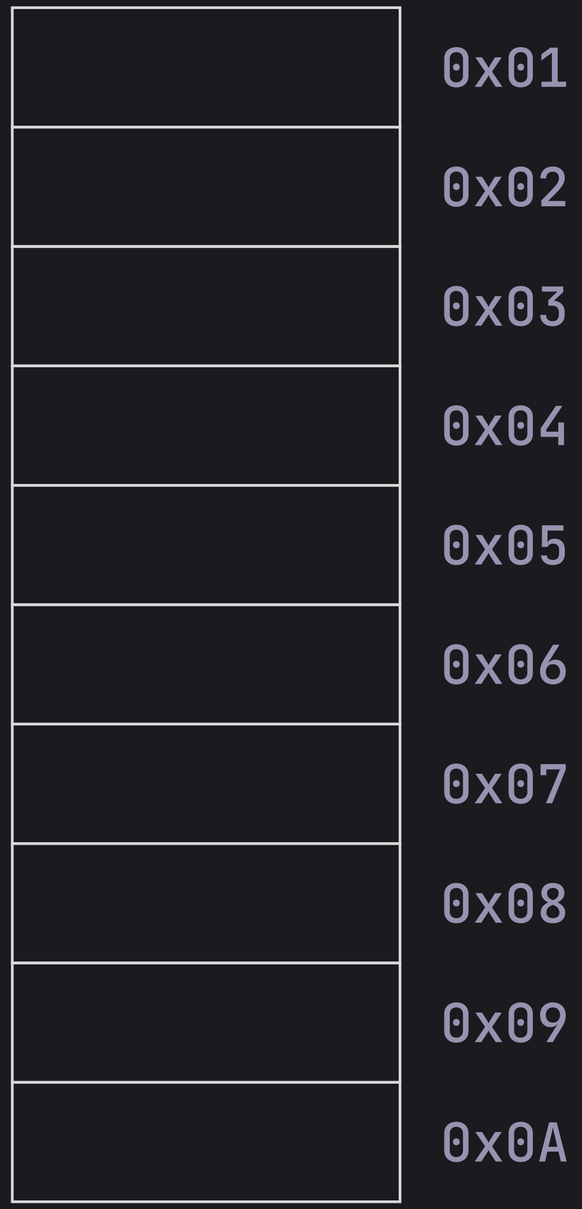
- Variables are like a **box**
- You can put a **value** in the box

Variables

- Variables are like a **box**
- You can put a **value** in the box
- and the box as a unique **address** (hexadecimal)

Variables

- Variables are stored in memory (RAM)
- All in a big line

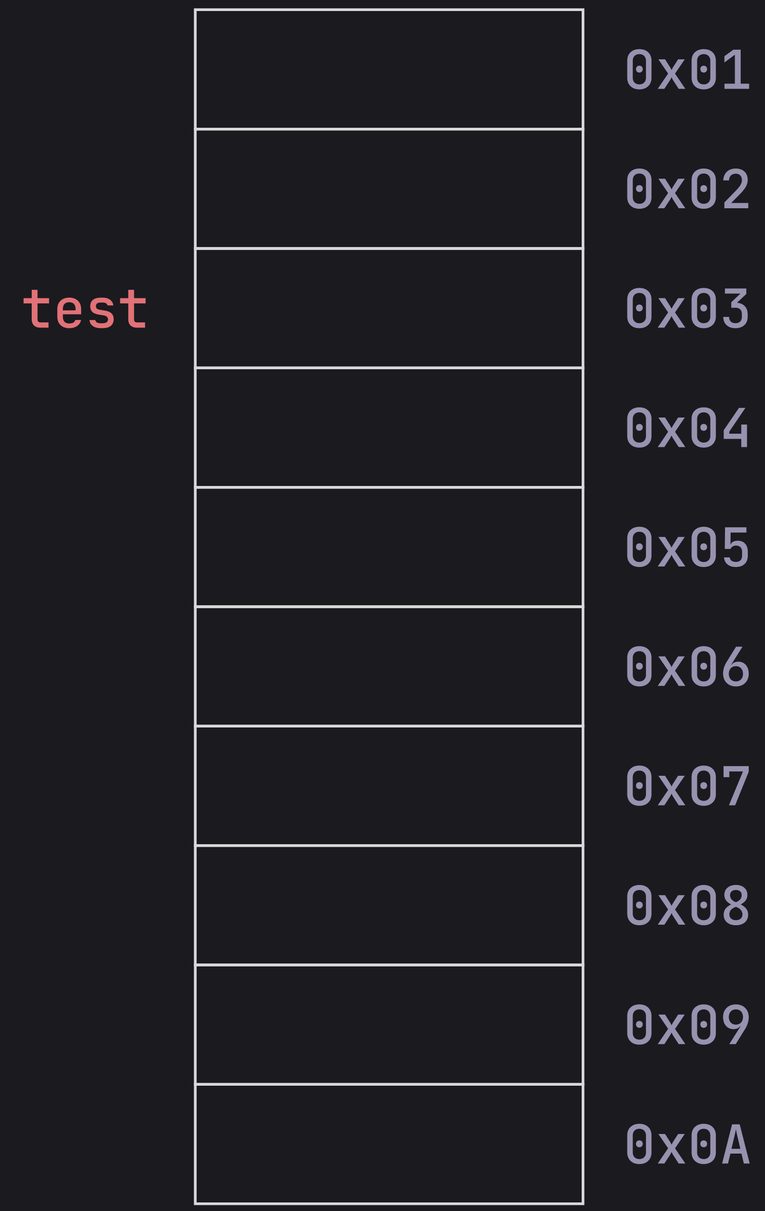


Variables

variable declaration

```
char test;
```

- A free box is selected (0x03)
- We assign a name to the box

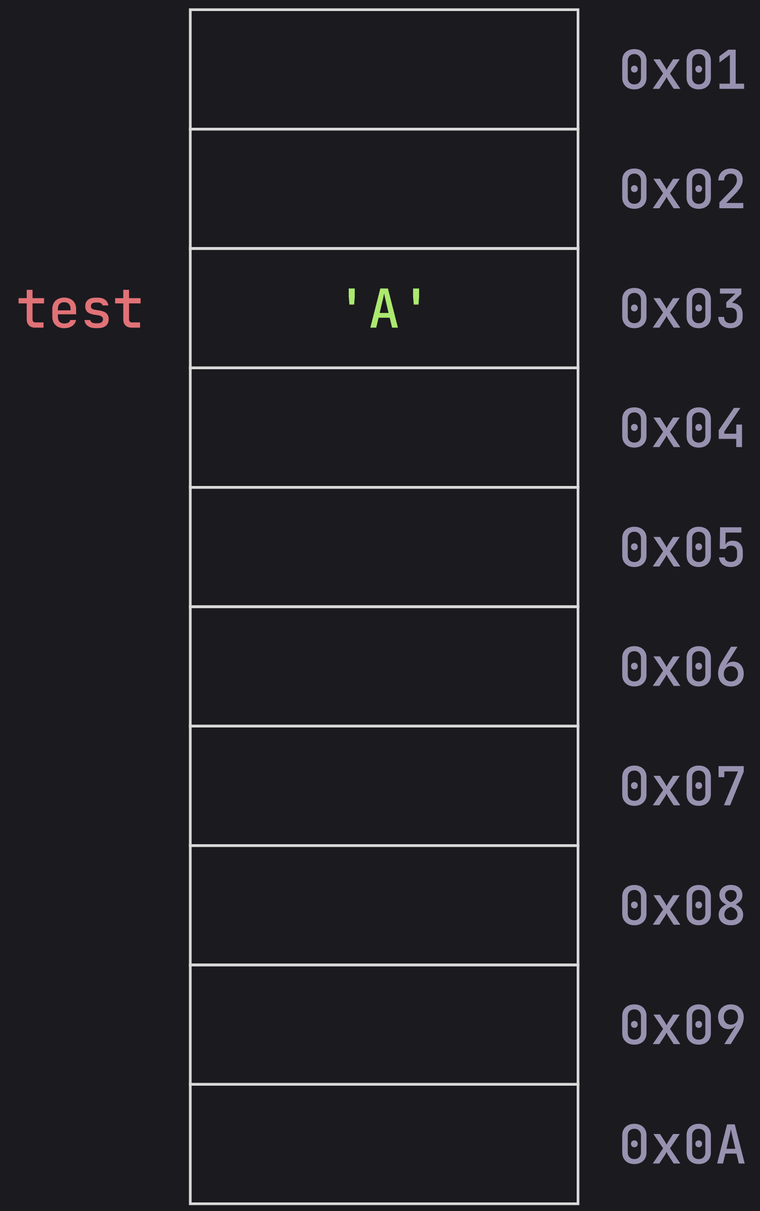


Variables

variable initialisation

```
char test = 'A';
```

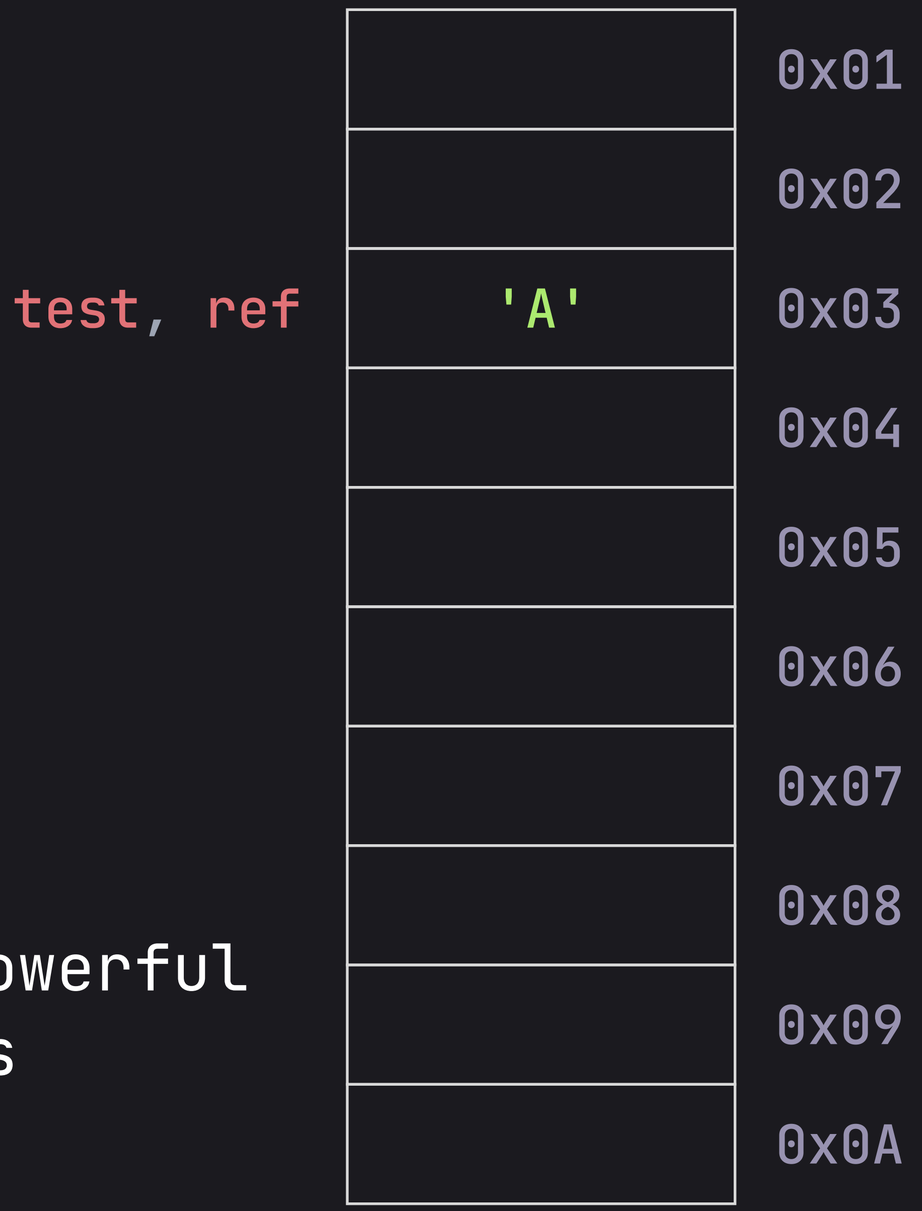
- A free box is selected (**0x03**)
- We assign a **name** to the box
- And put a **value** in it



References

Variable aliasing

```
char test = 'A';
char & ref = test;
```

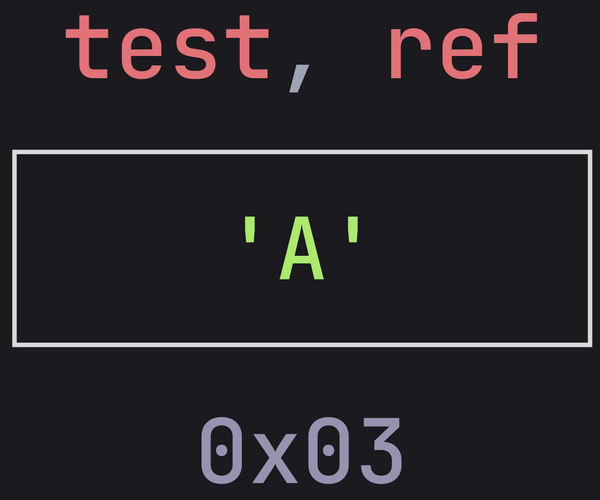


- References are very useful and powerful
- You can think of them as an alias (or a nickname)

References

Variable aliasing

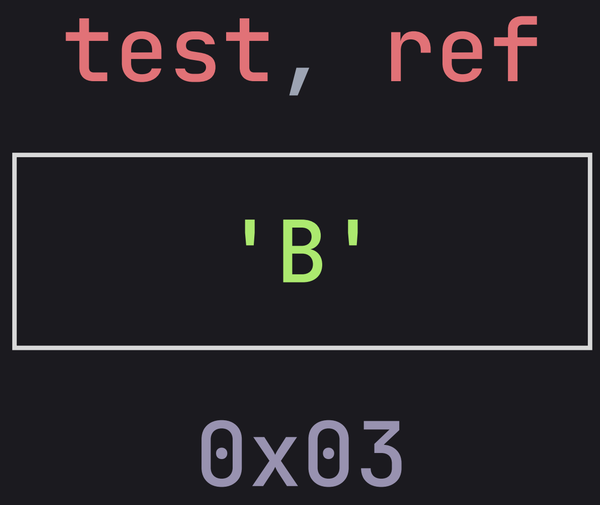
```
char test = 'A';
char & ref = test;
```



References

Variable aliasing

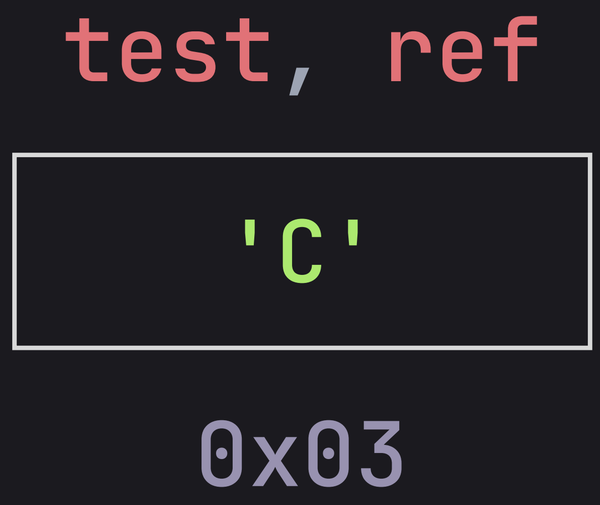
```
char test = 'A';
char & ref = test;

test = 'B';
```



References

Variable aliasing

```
char test = 'A';
char & ref = test;

test = 'B';
ref = 'C';
```

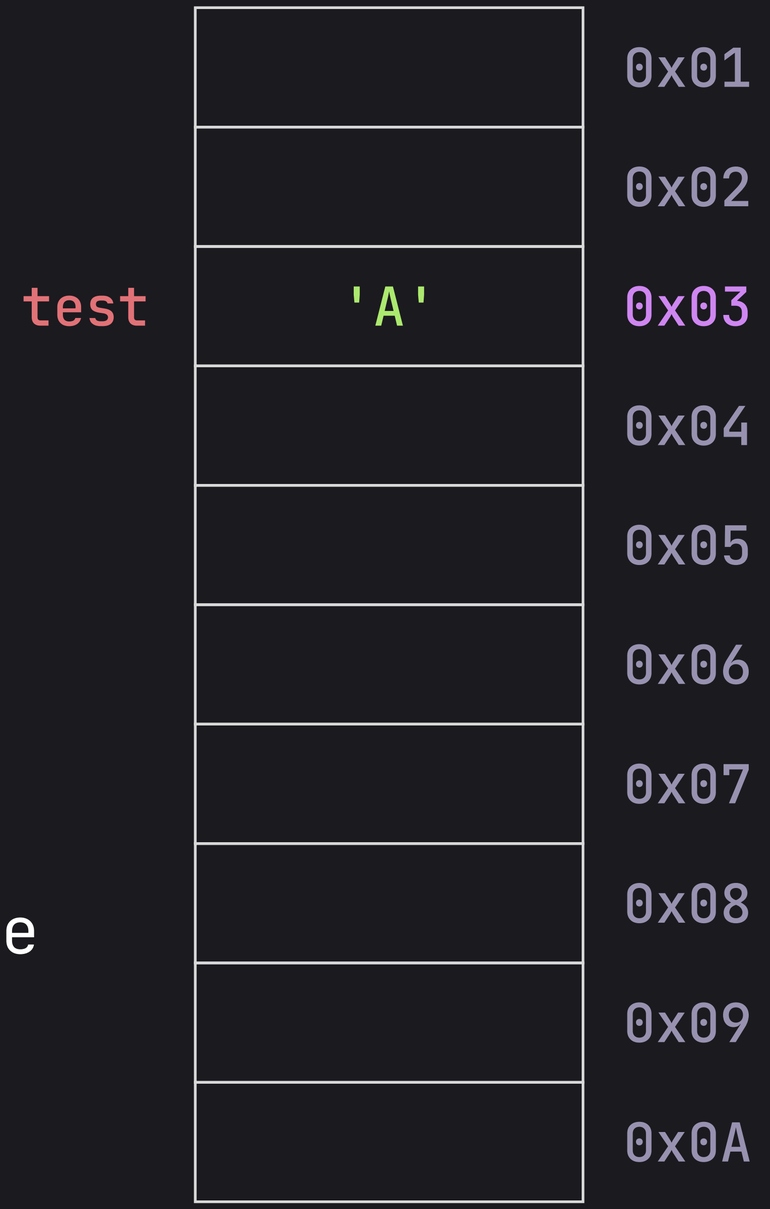


Pointers

Getting the memory address

`&test;` ⟶ `0x03`

- The `&` symbol is the address operator
- Gets the **memory address** of a variable



Pointers

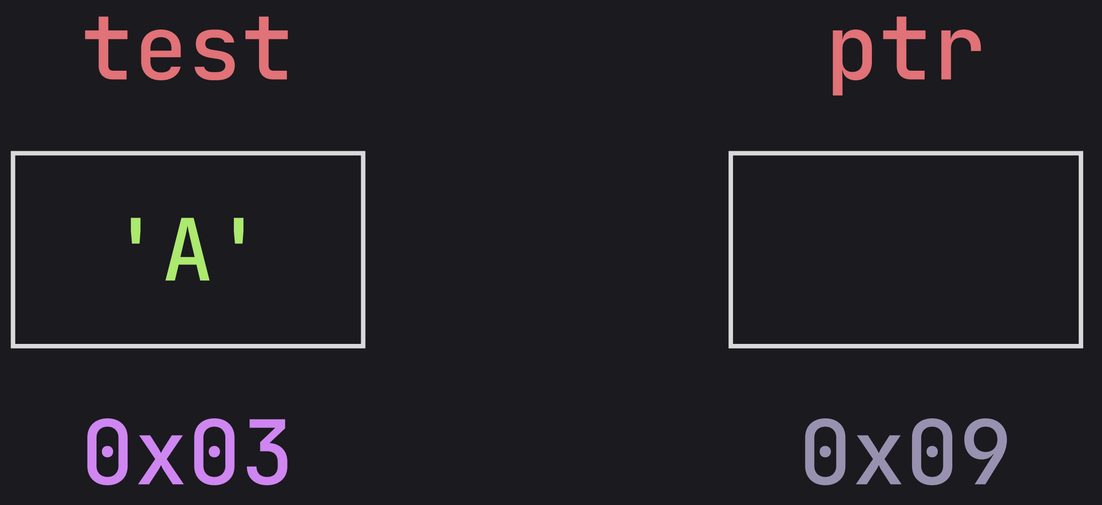
Storing the memory address

```
char test = 'A';
```

Pointers

Storing the memory address

```
char test = 'A';
char * ptr;
```

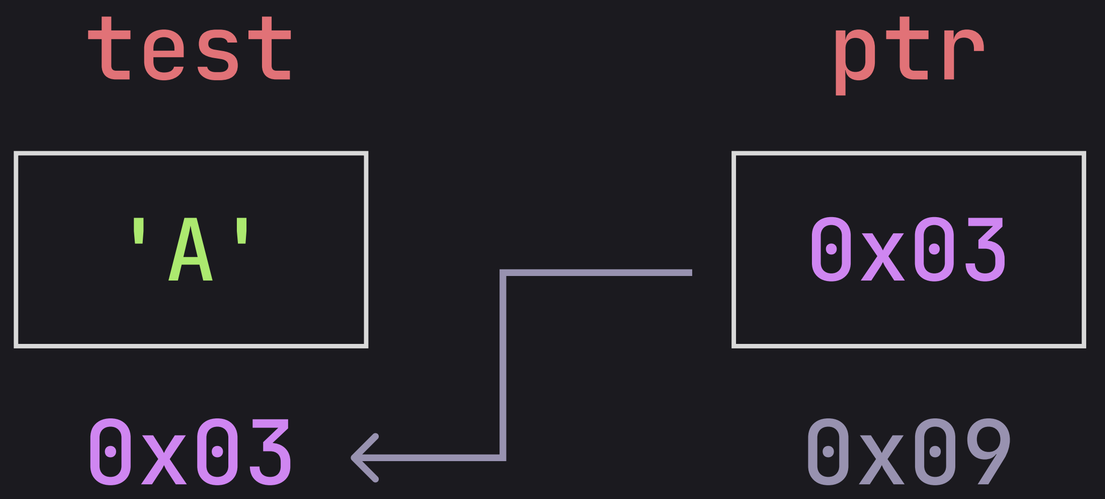


- The * symbol tells the compiler the **variable** is a **pointer**
- We again assign a **name** to an empty box

Pointers

Storing the memory address

```
char test = 'A';
char * ptr = &test;
```

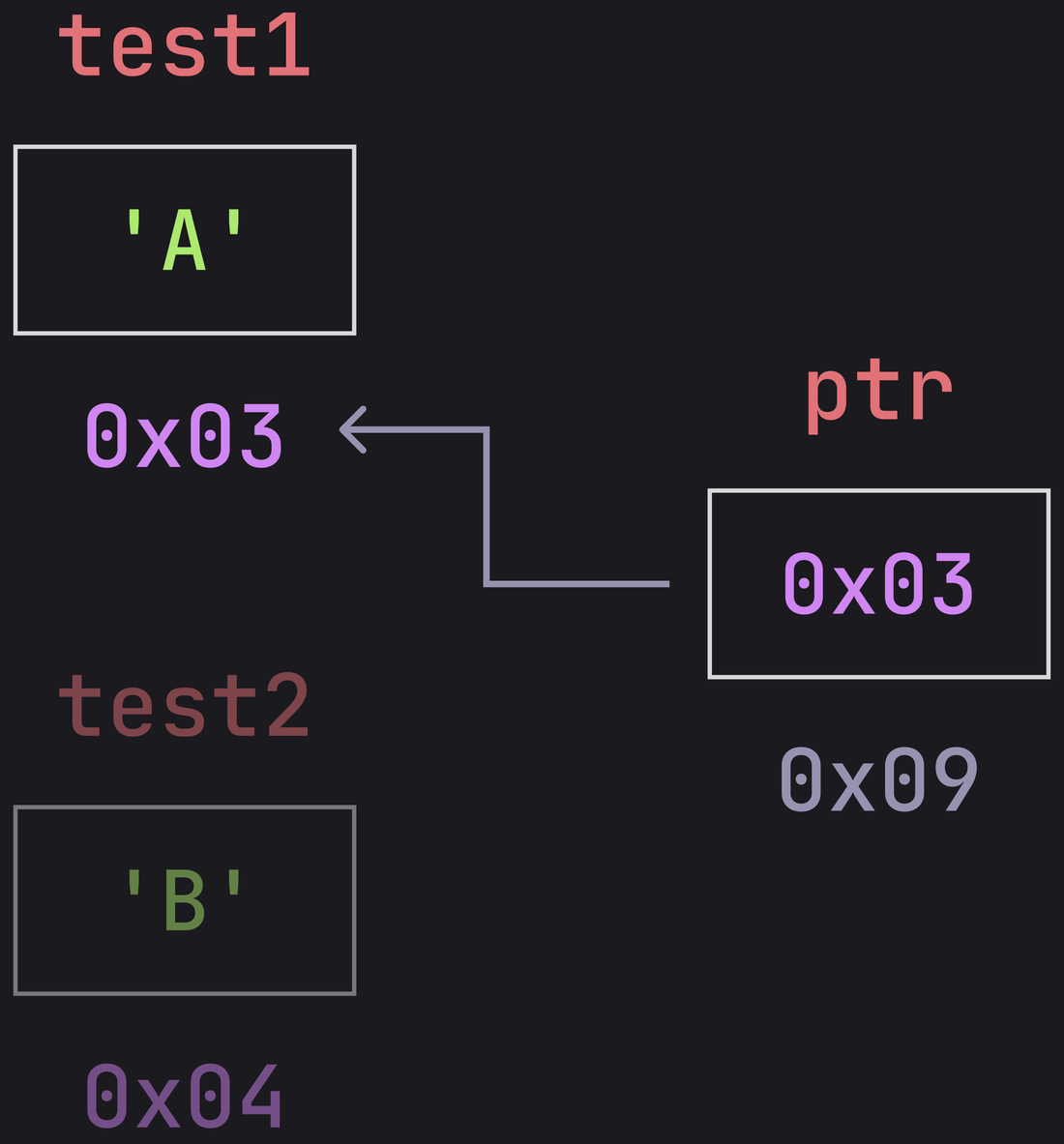


- The * symbol tells the compiler the **variable** is a **pointer**
- We again assign a **name** to an empty box
- Then we set the value to the **memory address** of **test**

Pointers

Storing the memory address

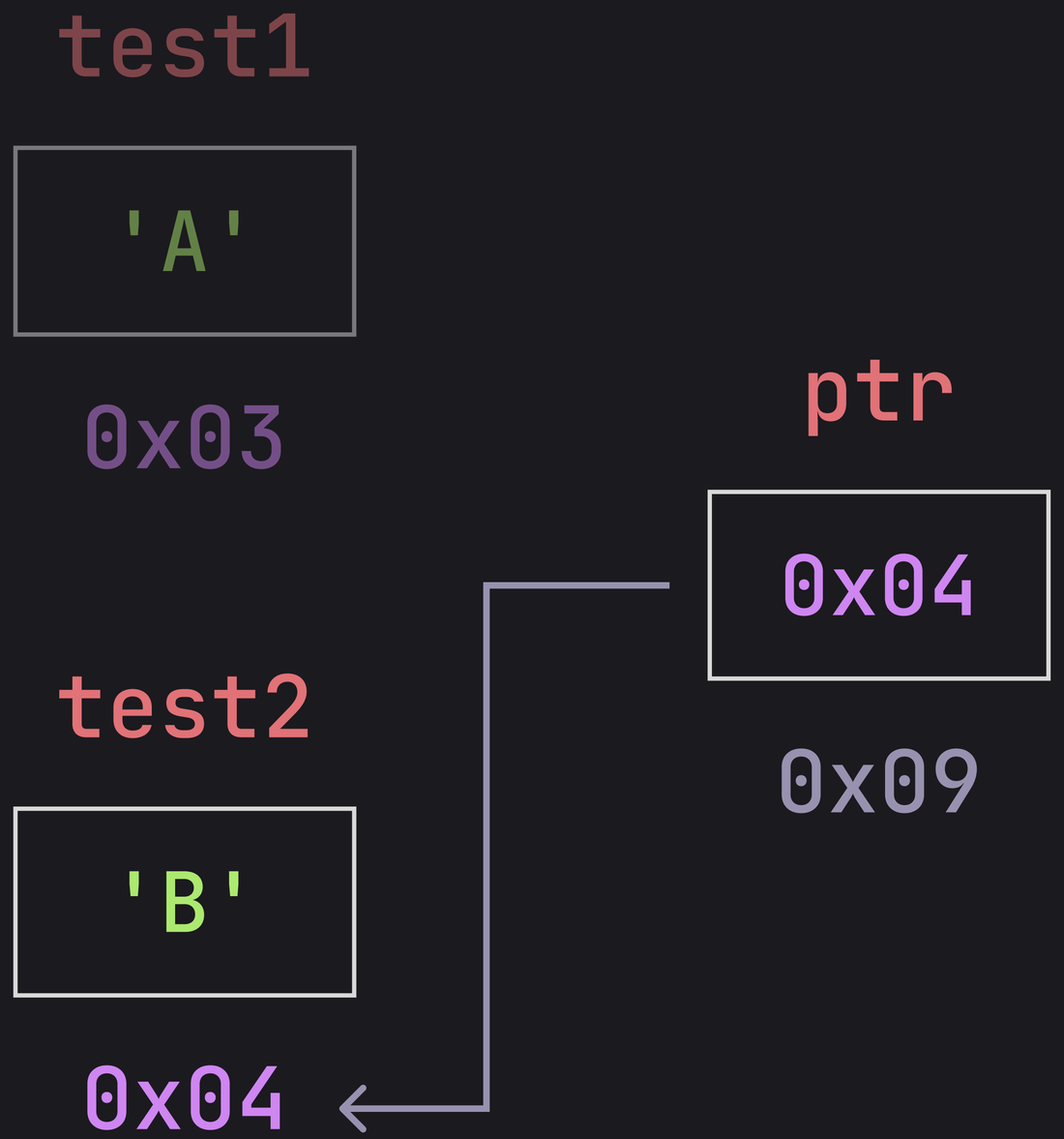
```
char test1 = 'A';
char test2 = 'B';

char * ptr = &test1;
```



Pointers

Storing the memory address

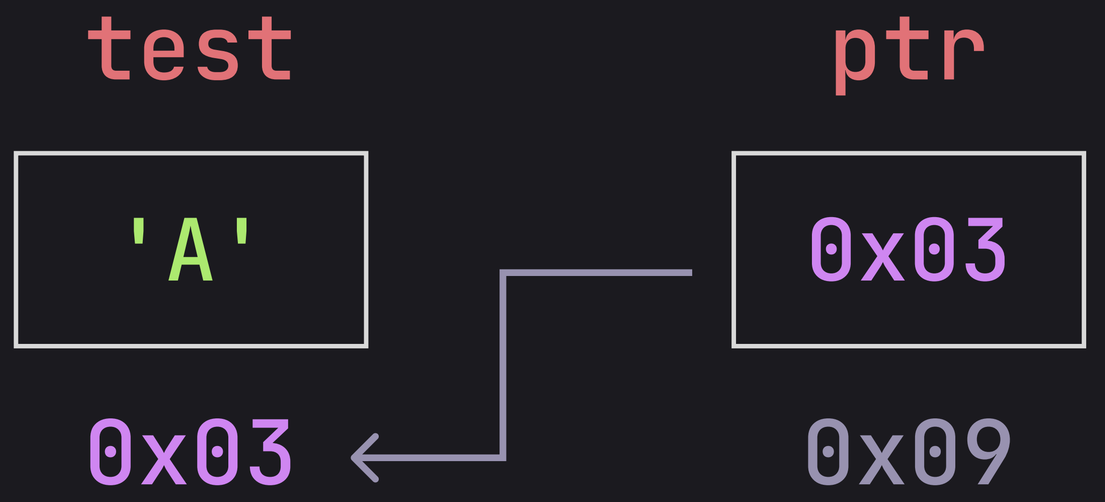
```
char test1 = 'A';
char test2 = 'B';

char * ptr = &test1;
ptr = &test2;
```



Pointers

Storing the memory address

```
char test = 'A';
char * ptr = &test;
```



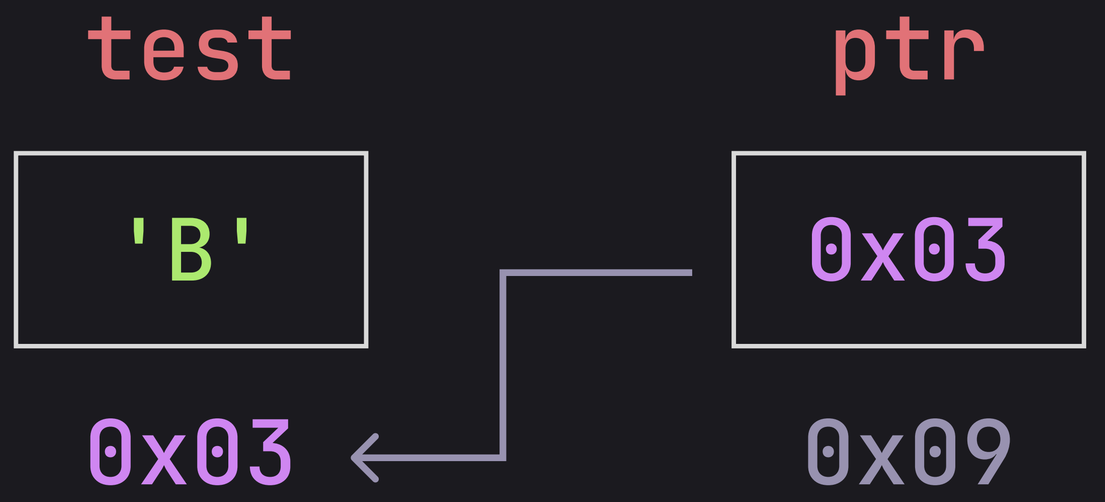

```
*ptr; ⟶ 'A'
```

- We can now use the * symbol to get the value stored at the memory address that the pointer is pointing to

Pointers

Storing the memory address

```
char test = 'A';
char * ptr = &test;
```



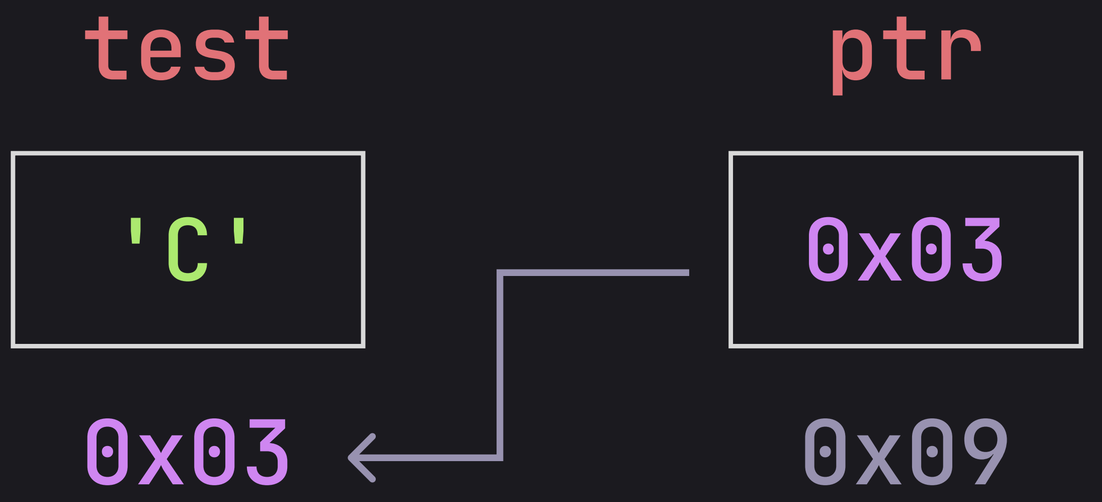

```
*ptr = 'B';
```

- We can even change the value at that memory address

Pointers

Storing the memory address

```
char test = 'A';
char * ptr = &test;
```

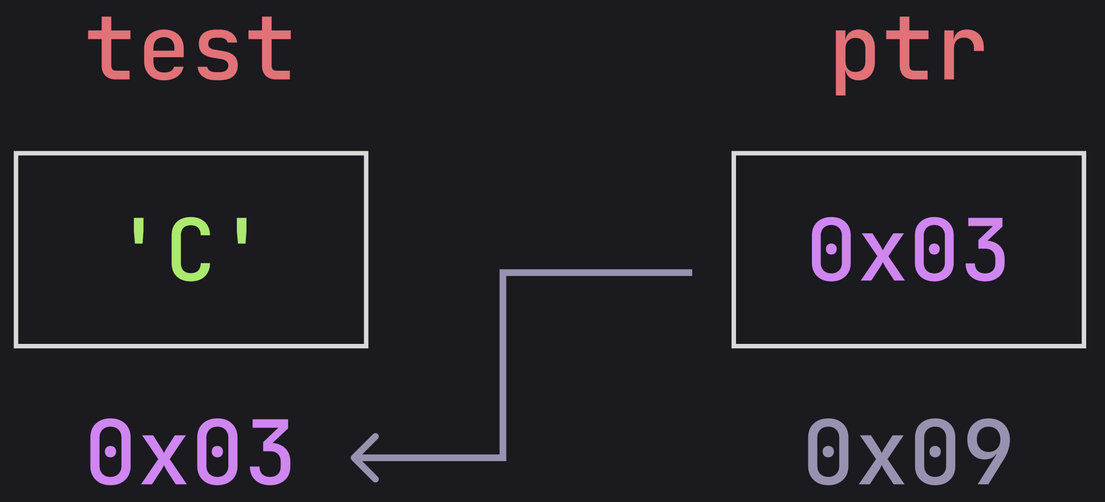



```
*ptr = 'B';
*ptr = 'C';
```

Pointers

Storing the memory address

```
char test = 'A';
char * ptr = &test;
```




```
*ptr = 'B';
*ptr = 'C';
test; ⟶ 'C'
```

Passing variables to functions

- Pass by value
- Pass by reference
- Pass by pointer

Pass by value

```
void doubleValue(int x)
```

- A **copy** of the value is created and passed to the function.
- Any changes within the function **will not affect** the original value of the argument.
- Like making a **photocopy** of a piece of paper, and then working on the photocopy instead of the original.

Pass by value

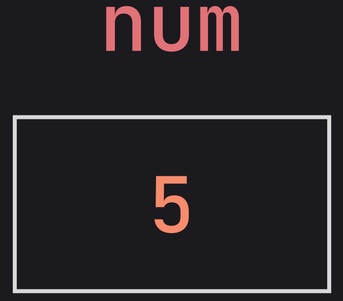
```
void doubleValue(int x) {
  x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```

Pass by value

```
void doubleValue(int x) {
  x = x * 2;
}

int main() {
    int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```

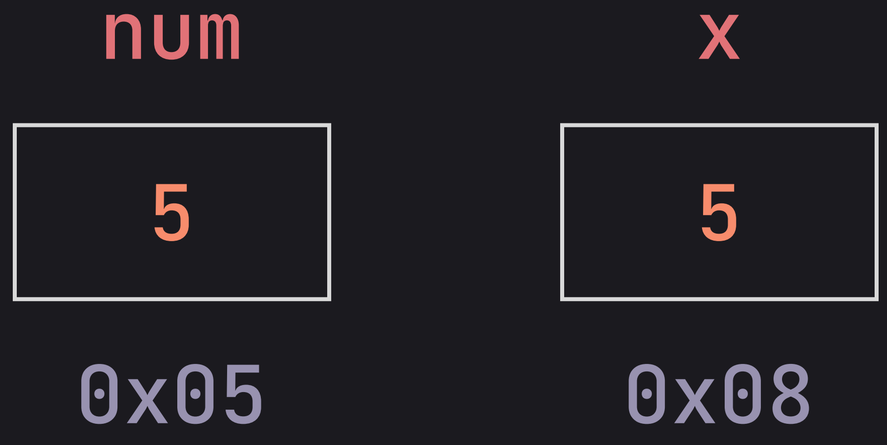
Pass by value

```
void doubleValue(int x) {
  x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```



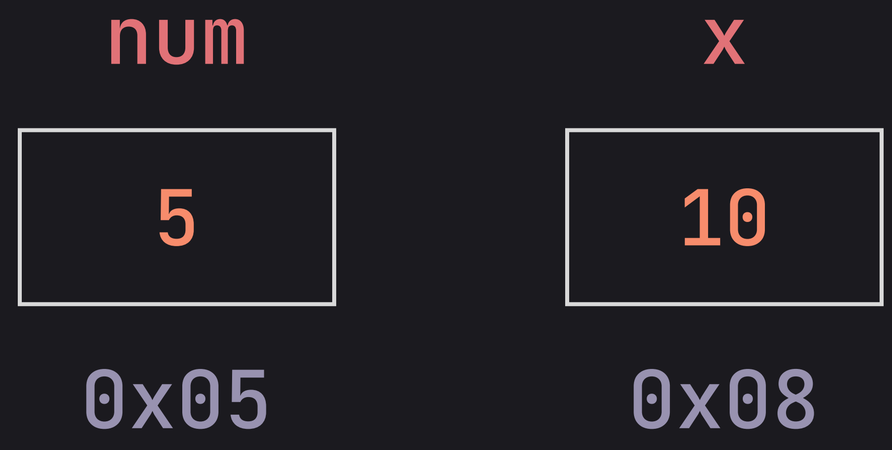
Pass by value

```
⟶  void doubleValue(int x) {
      x = x * 2;
    }

    int main() {
      int num = 5;
      doubleValue(num);
      std::cout << num << '\n';
      return 0;
    }
```



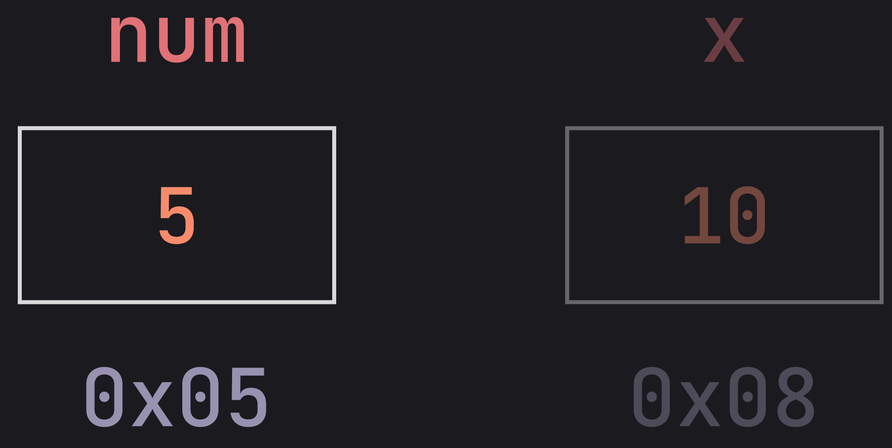
Pass by value

```
void doubleValue(int x) {
    x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```



Pass by value

```
void doubleValue(int x) {
  x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
→   std::cout << num << '\n'; // Output: 5
  return 0;
}
```



Pass by reference

```
void doubleValue(int & x)
```

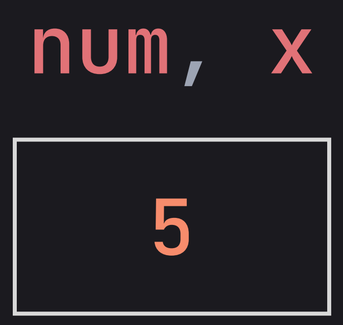
- The function receives a **reference** to the original variable, not a copy
- Any changes made to the parameter within the function **will affect** the original value of the variable outside the function
- Like working on the **original** piece of paper directly

Pass by reference

```
void doubleValue(int & x) {
  x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```

Pass by reference

```
⟶  void doubleValue(int & x) {
      x = x * 2;
    }

    int main() {
      int num = 5;
      doubleValue(num);
      std::cout << num << '\n';
      return 0;
    }
```



Pass by reference

```
void doubleValue(int & x) {
    x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
  std::cout << num << '\n';
  return 0;
}
```

Pass by reference

```
void doubleValue(int & x) {
  x = x * 2;
}

int main() {
  int num = 5;
  doubleValue(num);
→   std::cout << num << '\n'; // Output: 10
  return 0;
}
```

Pass by pointer

```
void doubleValue(int * x)
```

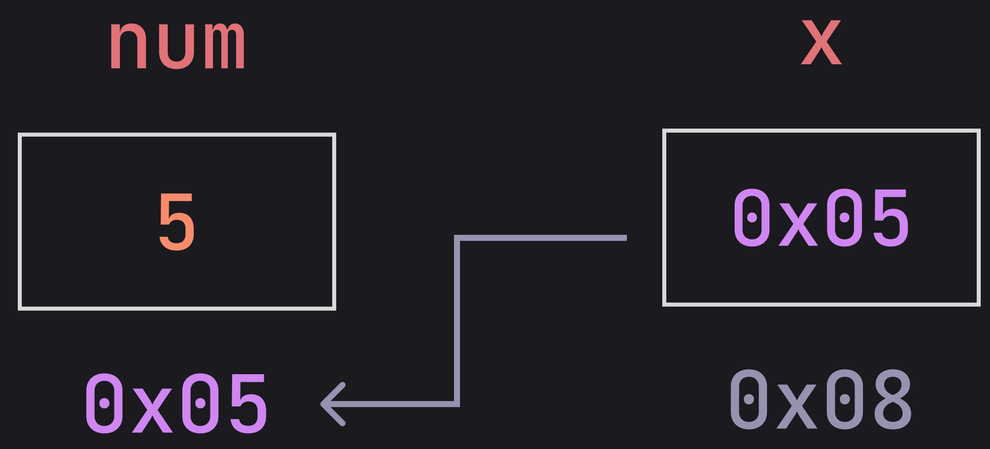
- Similar to pass by reference, but instead of passing a reference to the original variable, we pass a **pointer**
- Any changes made to the parameter within the function **will affect** the original value of the argument outside the function

Pass by pointer

```
void doubleValue(int * x) {
  *x = (*x) * 2;
}

int main() {
  int num = 5;
  doubleValue(&num);
  std::cout << num << '\n';
  return 0;
}
```

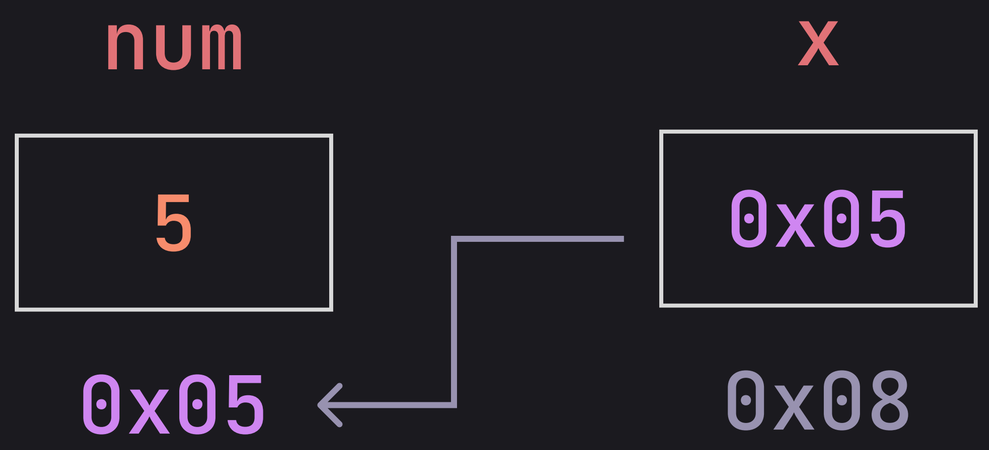
Pass by pointer

```
⟶  void doubleValue(int * x) {
     *x = (*x) * 2;
   }

   int main() {
     int num = 5;
     doubleValue(&num);
     std::cout << num << '\n';
     return 0;
   }
```



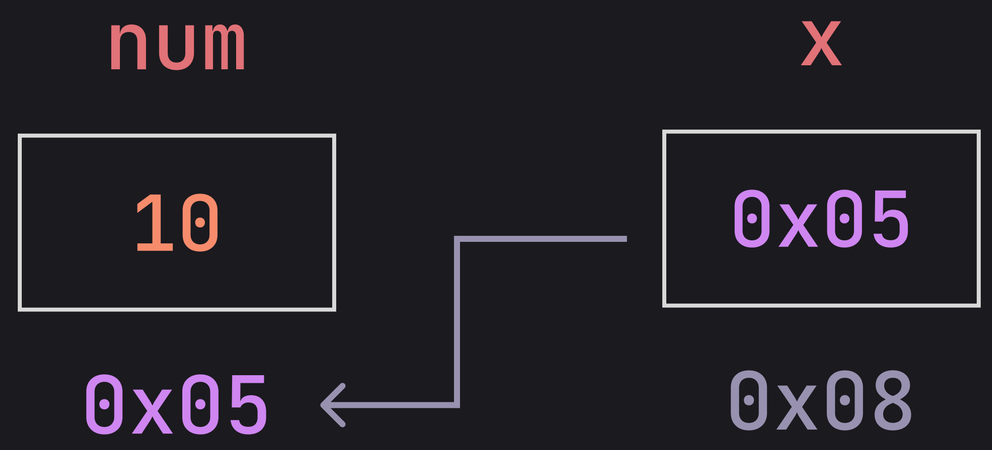
Pass by pointer

```
void doubleValue(int * x) {
    *x = (*x) * 2;
}

int main() {
  int num = 5;
  doubleValue(&num);
  std::cout << num << '\n';
  return 0;
}
```



Pass by pointer

```
void doubleValue(int * x) {
    *x = (*x) * 2;
}

int main() {
  int num = 5;
  doubleValue(&num);
  std::cout << num << '\n';
  return 0;
}
```



Pass by pointer

```
void doubleValue(int * x) {
  *x = (*x) * 2;
}

int main() {
  int num = 5;
  doubleValue(&num);
  std::cout << num << '\n'; // Output: 10
  return 0;
}
```