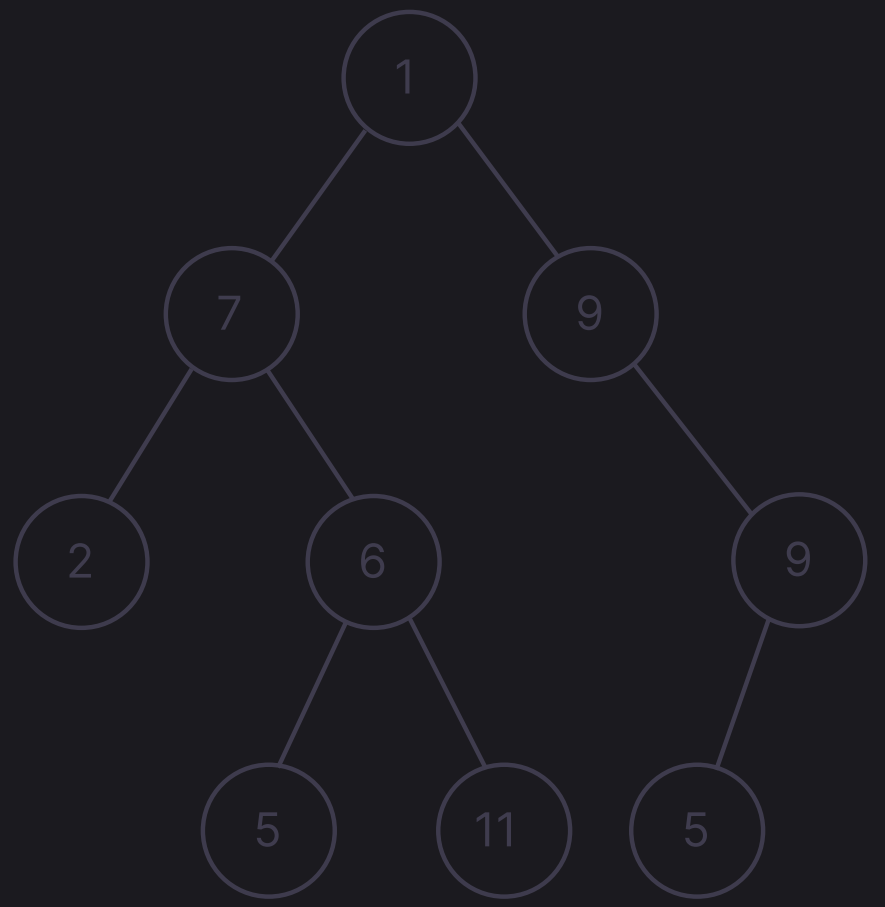
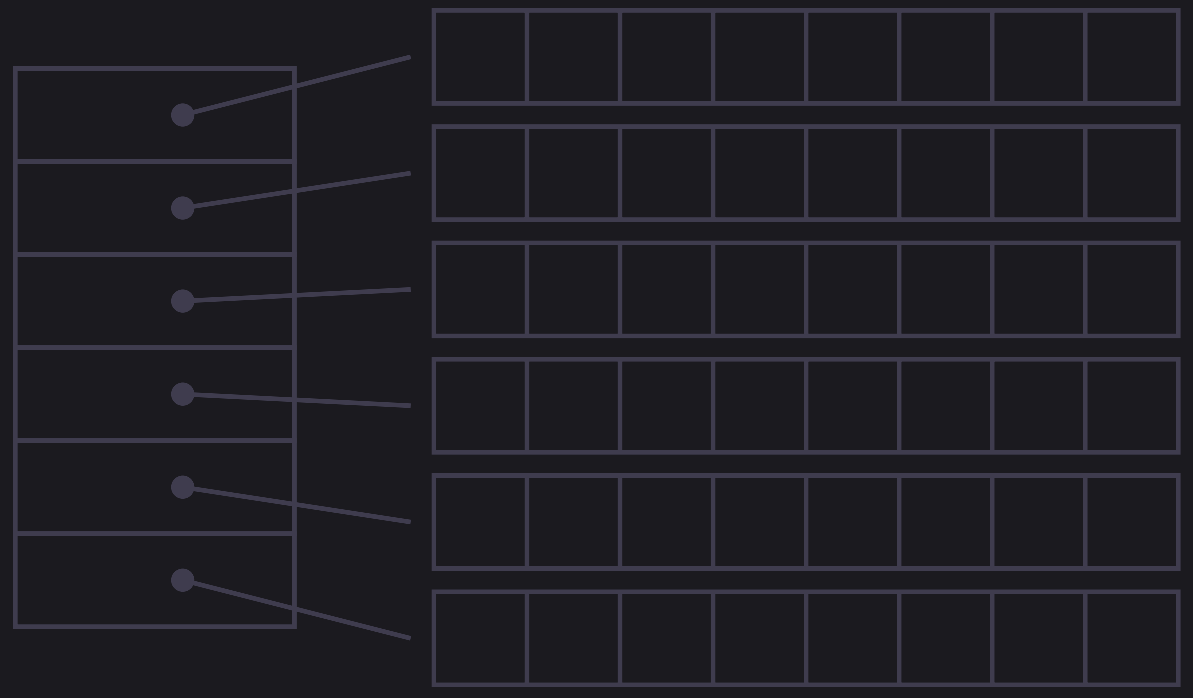
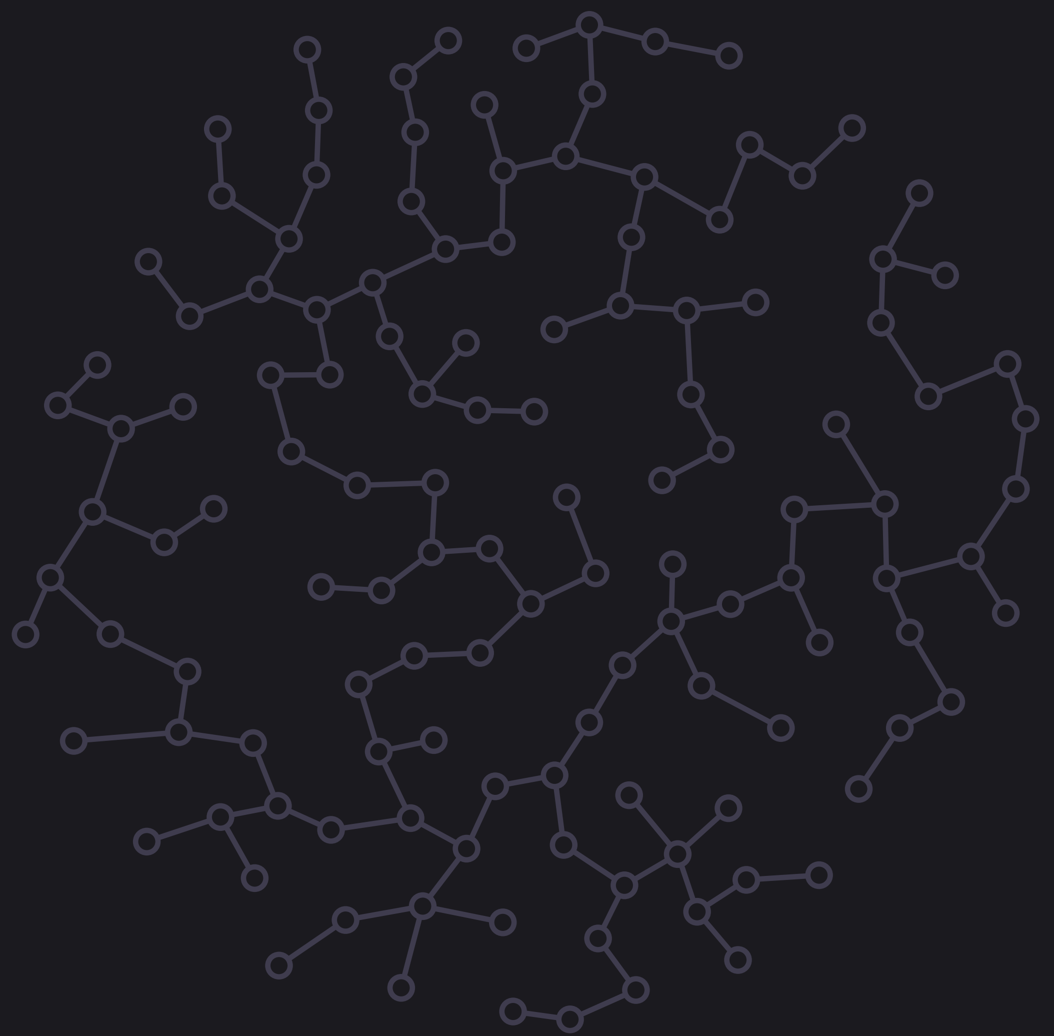
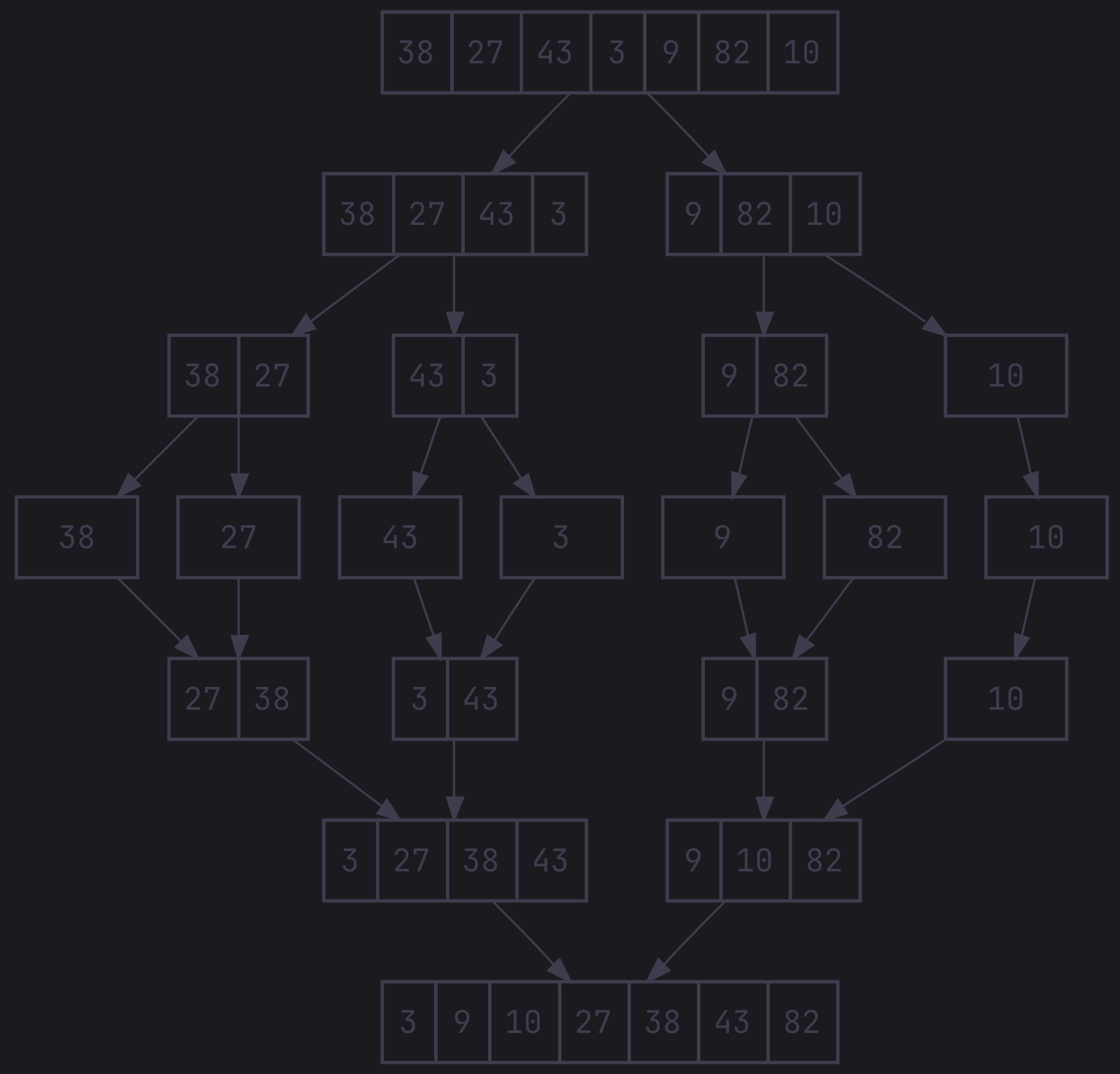




data structures & algorithms

Tutorial 3





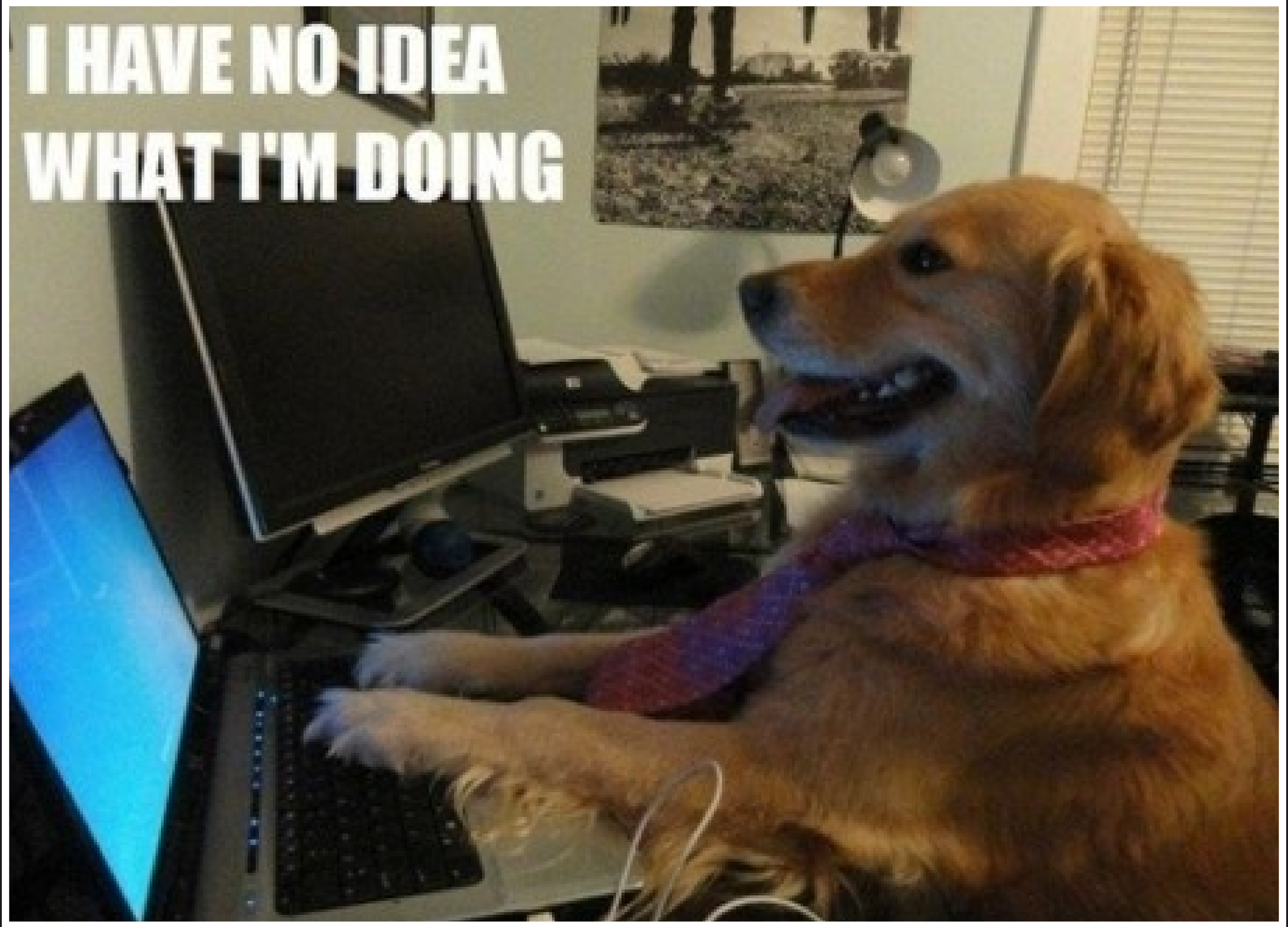
π

Happy Pi Day!

🔥🙋‍♀️

Burning questions from last week?

This week’s lab



This was me at this point in the course

Today we are levelling up our C++ with templates, class constructors and learning about more data structures

- Templates
- Memory: Stack vs Heap
- Linked Lists
- Constructors

Templates

Templates are like a **blueprint** where you **fill in the blank**

(Similar to Generics in Java, but more powerful) *motivation*

Write less code

Templates let you write a ***single*** function that supports ***multiple*** types

Templates

Imagine we are writing an **add** function

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

int sum = add(1, 2);
```

Templates

But now we want to also add **floats** together...

```cpp
// int implementation
int add(int a, int b) {
    return a + b;
}

// float implementation
float add(float a, float b) {
    return a + b;
}

int sum1 = add(1, 2);
float sum2 = add(3.14f, 2.71f);
```

Templates

And now we want to also add **doubles** together...

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

// float implementation
float add(float a, float b) {
    return a + b;
}

// double implementation
double add(double a, double b) {
    return a + b;
}

int sum1 = add(1, 2);
float sum2 = add(3.14f, 2.71f);
double sum3 = add(1.234, 5.678);
```

Templates

And what about...
long, or long long or
unsigned int, etc. 😭

Templates

Ahh... much better

😌

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}

int sum1 = add(1, 2);
float sum2 = add(3.14f, 2.71f);
double sum3 = add(1.234, 5.678);
unsigned int sum4 = add(10u, 20u);
```

Templates

This would work too!

Usually we just use **T**

(T stands for type)

```cpp
template <typename Blank>
Blank add(Blank a, Blank b) {
    return a + b;
}

int sum1 = add(1, 2);
float sum2 = add(3.14f, 2.71f);
double sum3 = add(1.234, 5.678);
unsigned int sum4 = add(10u, 20u);
```

Templates

And if you want to explicitly specify the type you can do that like so

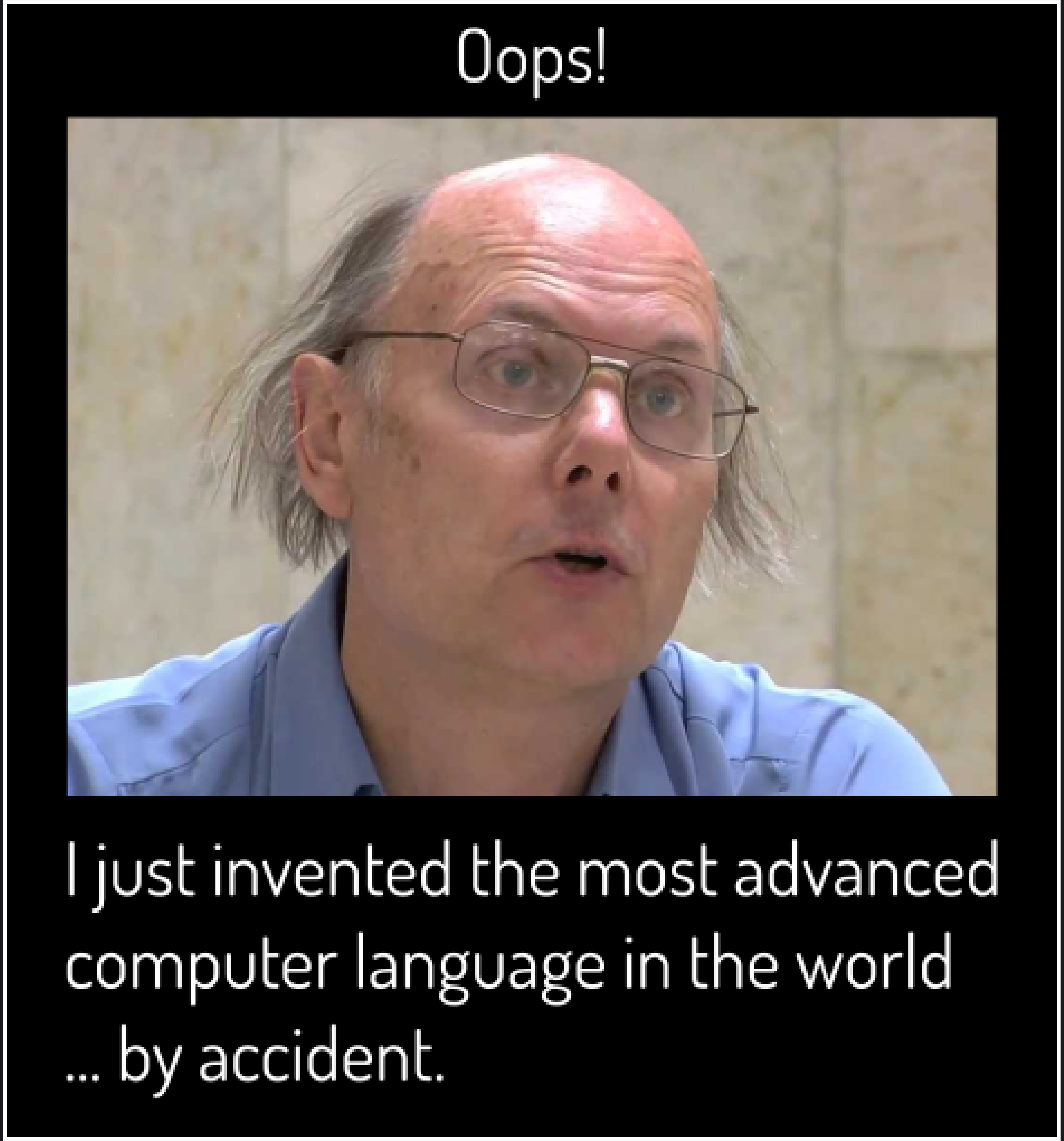
```
template <typename T>
T add(T a, T b) {
    return a + b;
}

int sum1 = add<int>(1, 2);
float sum2 = add<float>(3.14f, 2.71f);
double sum3 = add<double>(1.234, 5.678);
unsigned int sum4 = add<unsigned int>(10u, 20u);
```

Templates

```cpp
std::vector<int> {};
```

You’ve actually been using
templates all along 👀

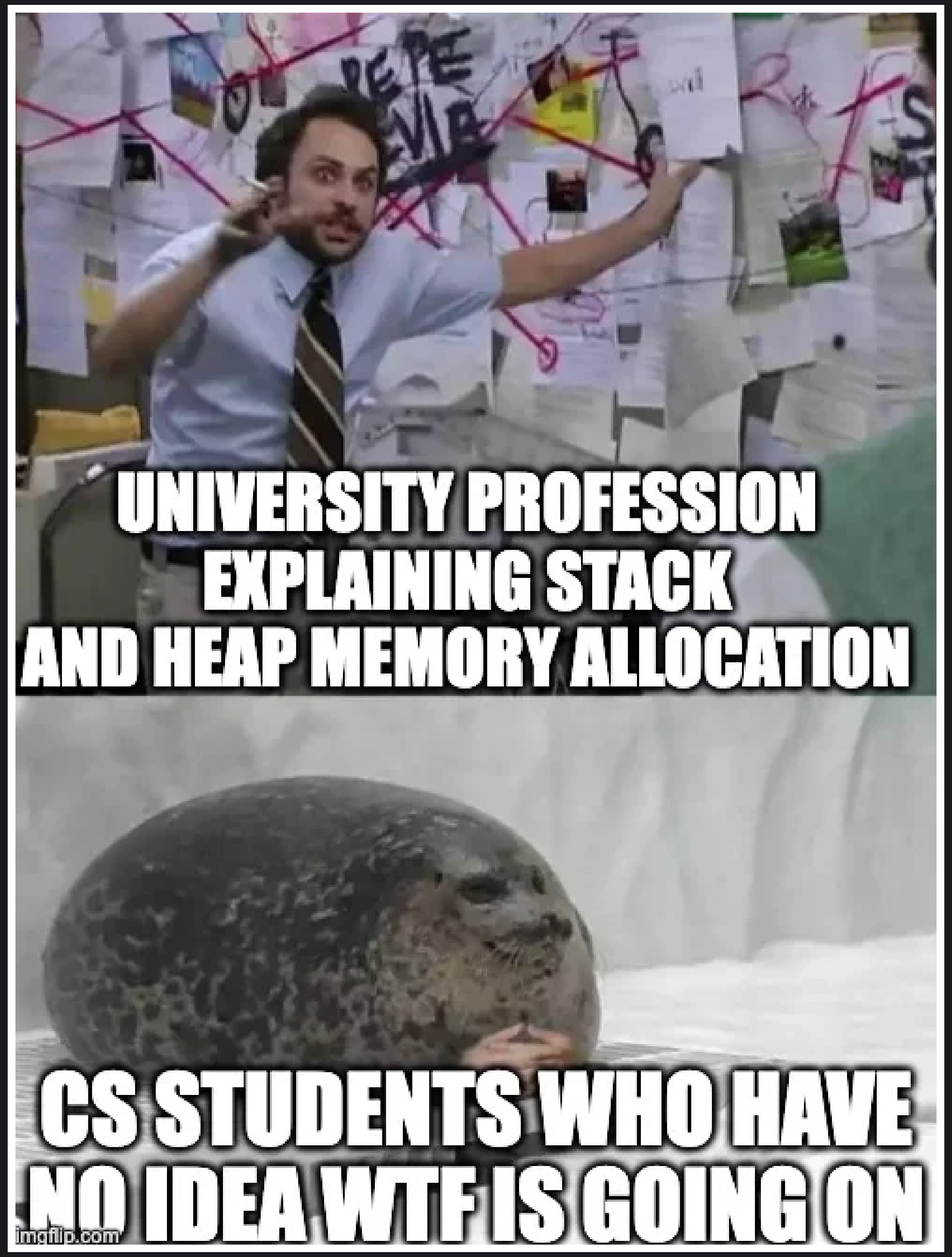
Templates



Templates are stupidly OP. We are only just scratching the surface here...

Let’s give the first activity a go

Stack vs Heap



tl;dr

Stack vs Heap

The stack and heap are both locations in **memory** were we can store data

tl;dr

Variables

Whenever we create a variable we put ***something*** on the **stack**

tl;dr

new keyword

Whenever we use the **new** keyword we put ***something*** on the **Heap**

tl;dr

Dynamic Memory

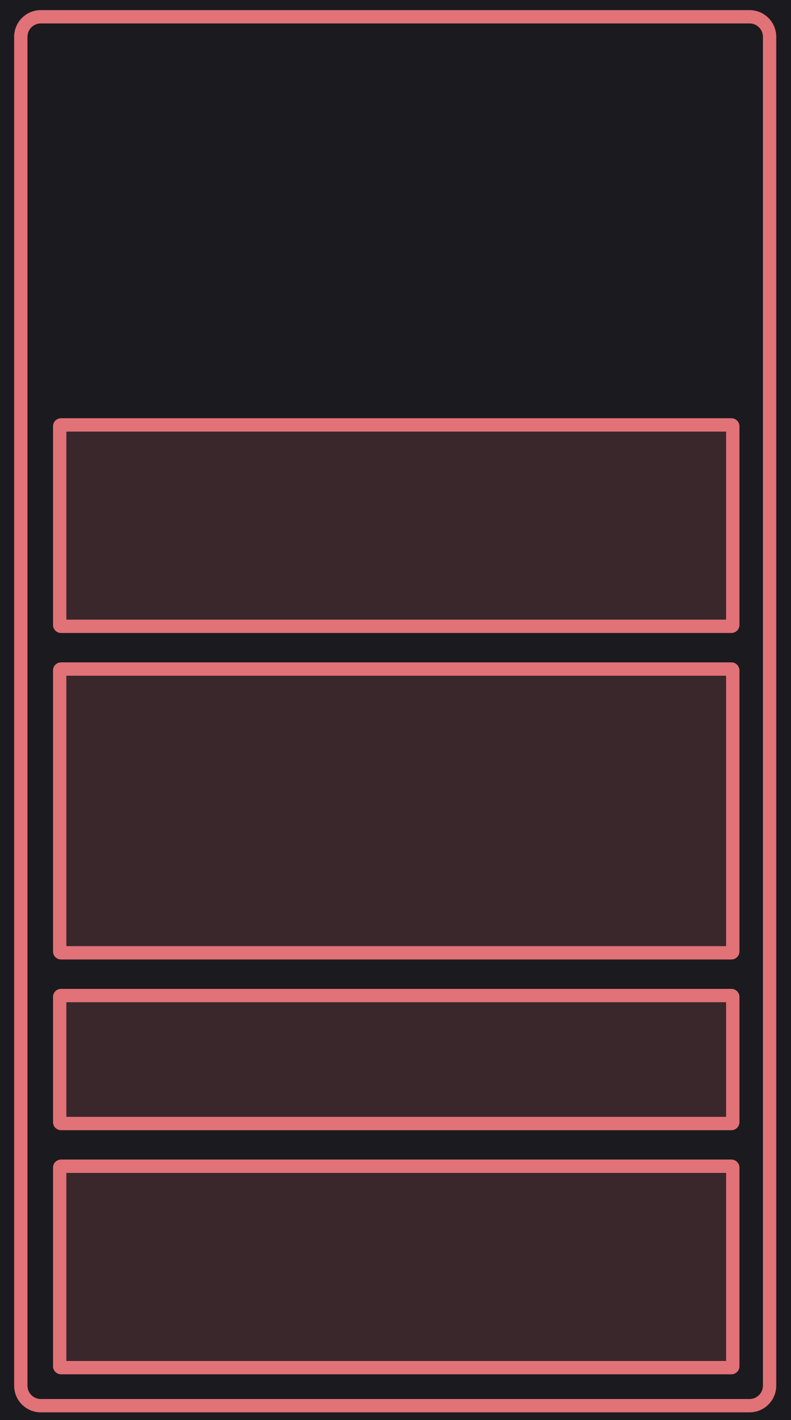
Whenever we need to dynamically allocate memory we need to use the **Heap**

tl;dr

Pointers

Whenever we put something on the **Heap** we need to keep track of its location with a ***pointer***

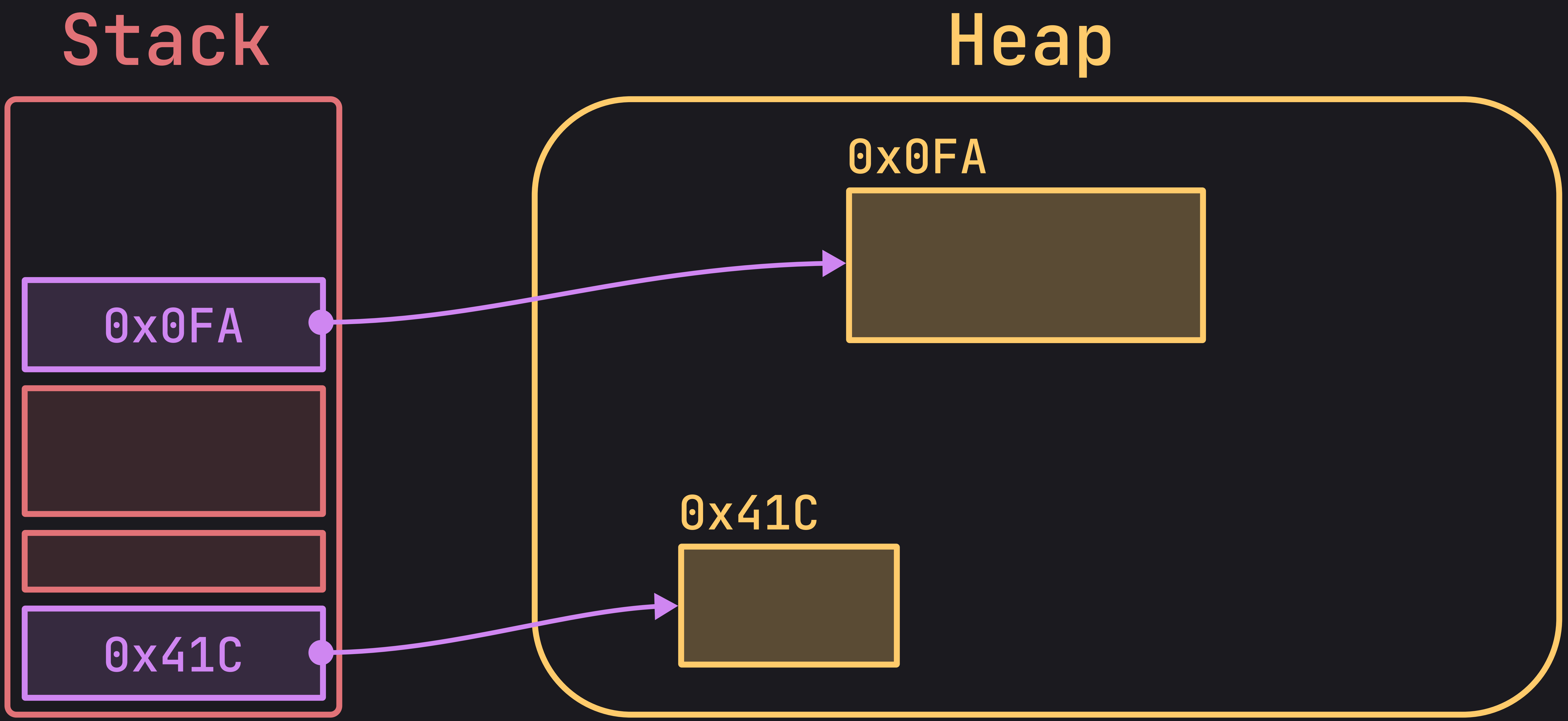
Stack



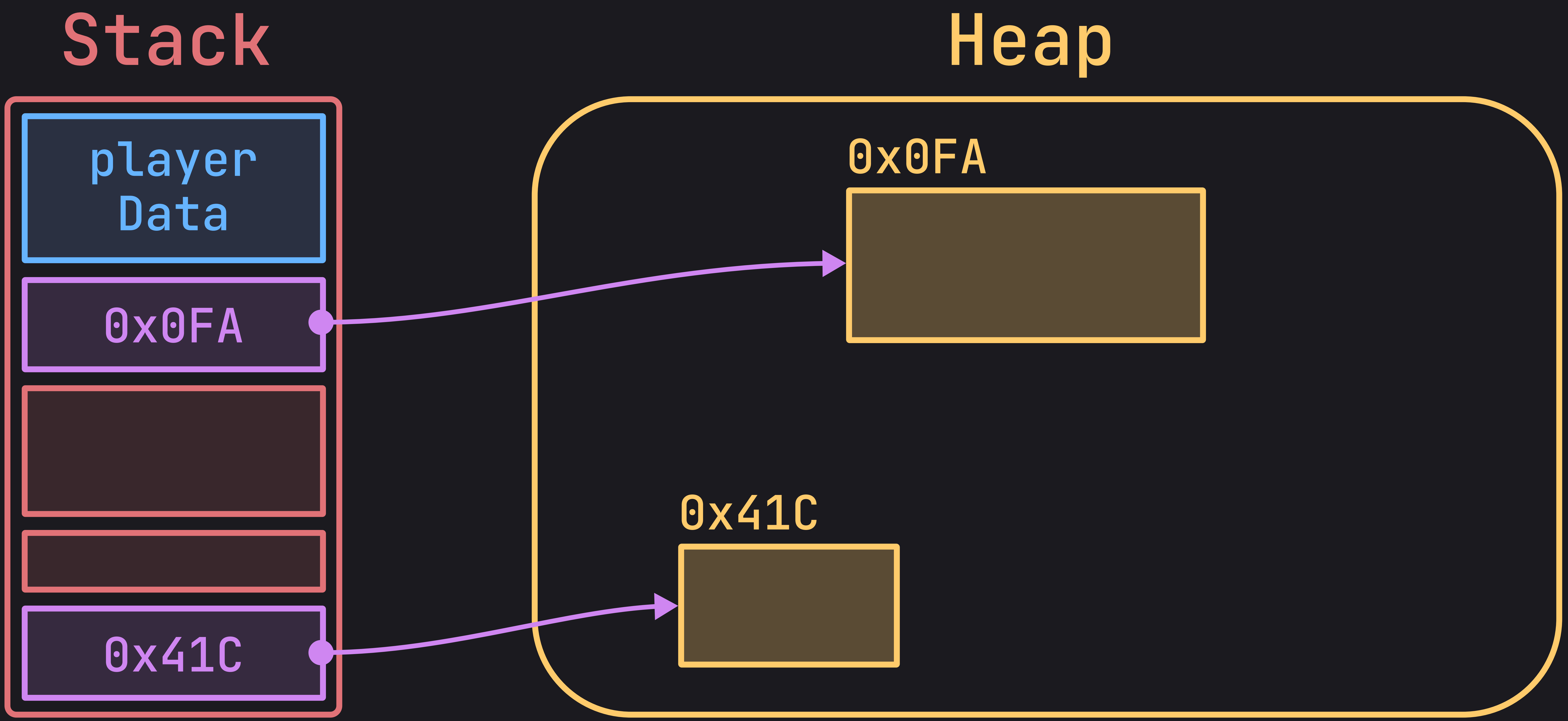
- Ordered and fast
- Used for variables and fixed memory

Heap

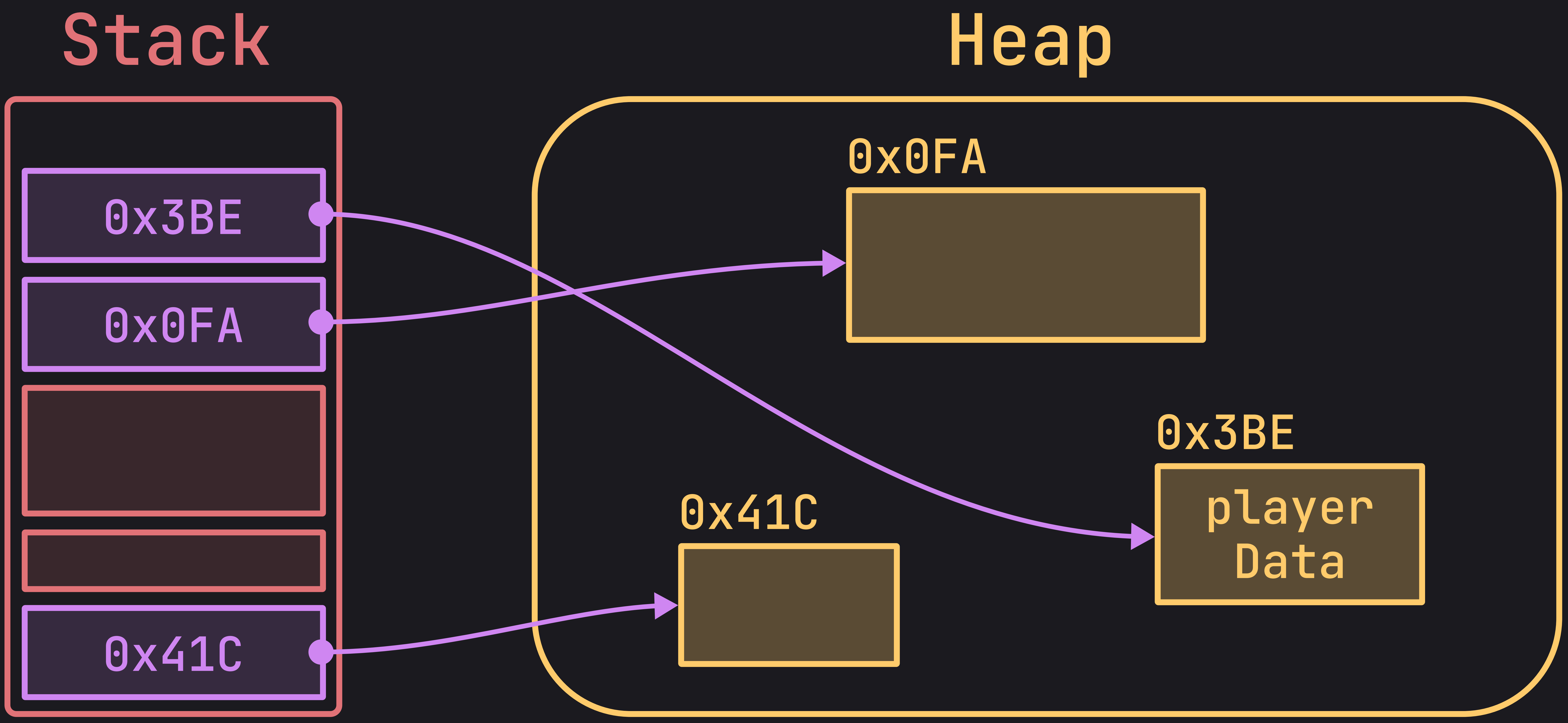
- Unordered and slow
- Used for dynamic memory



We keep track of memory on the **heap** using **pointers** on the **stack**

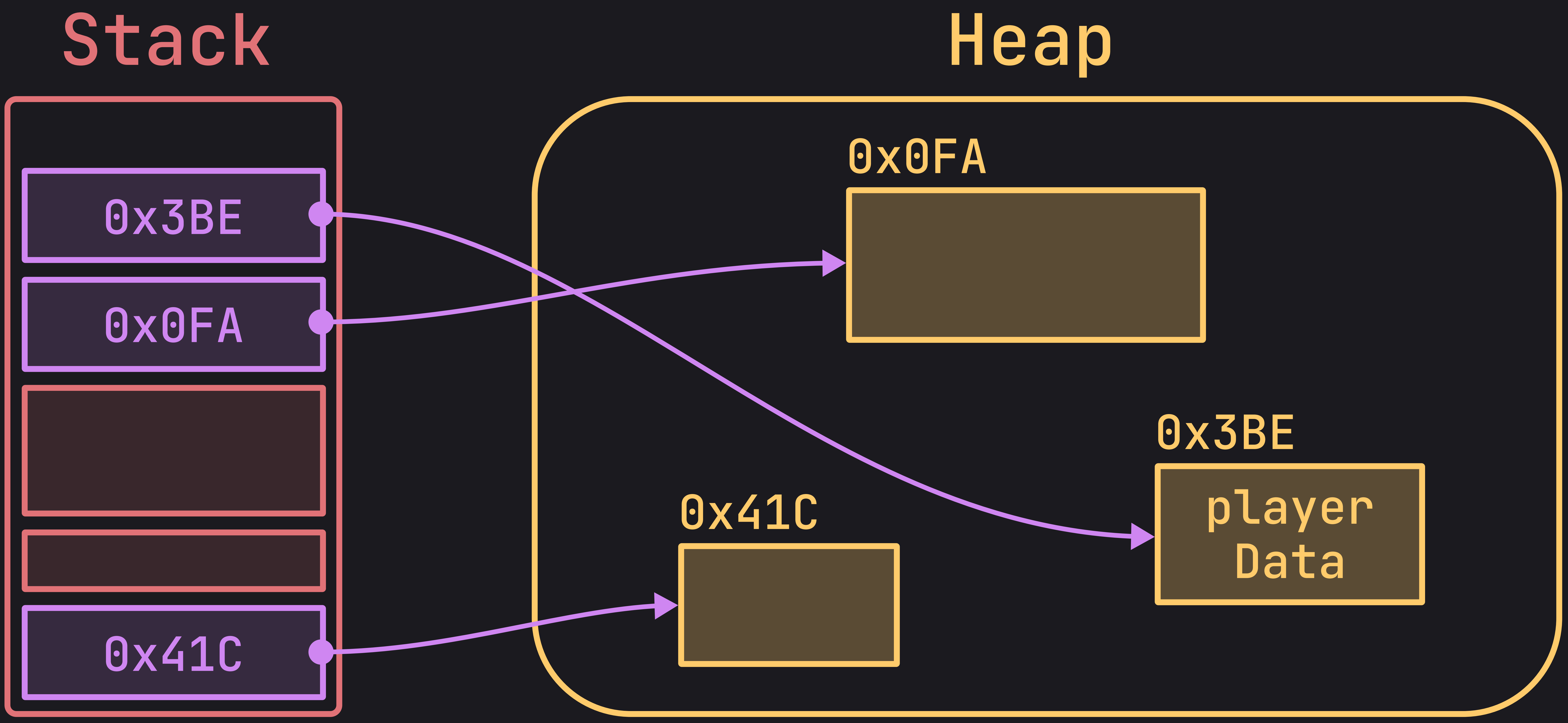



```
Player player1();
```



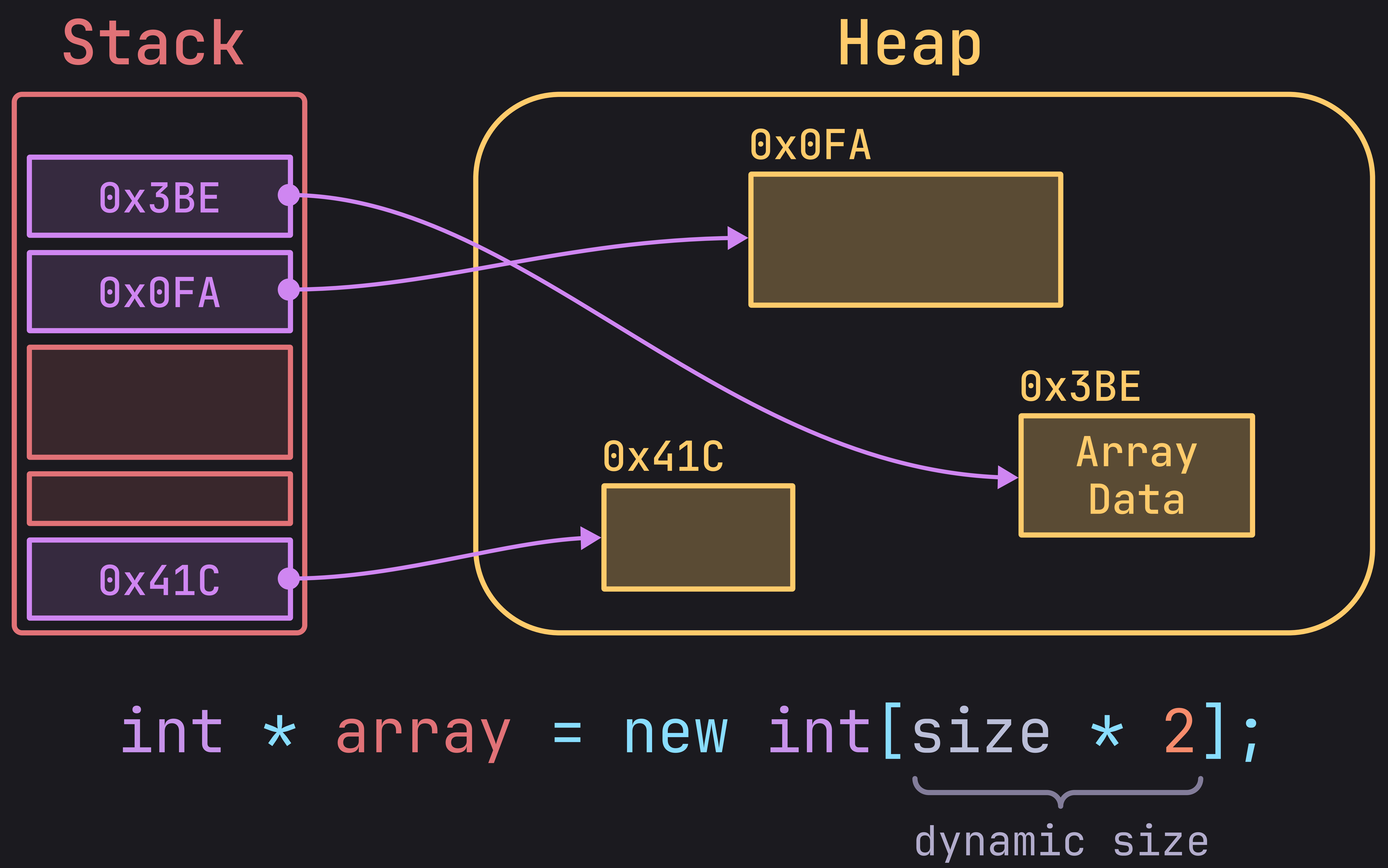

```
Player * player1 = new Player();
```

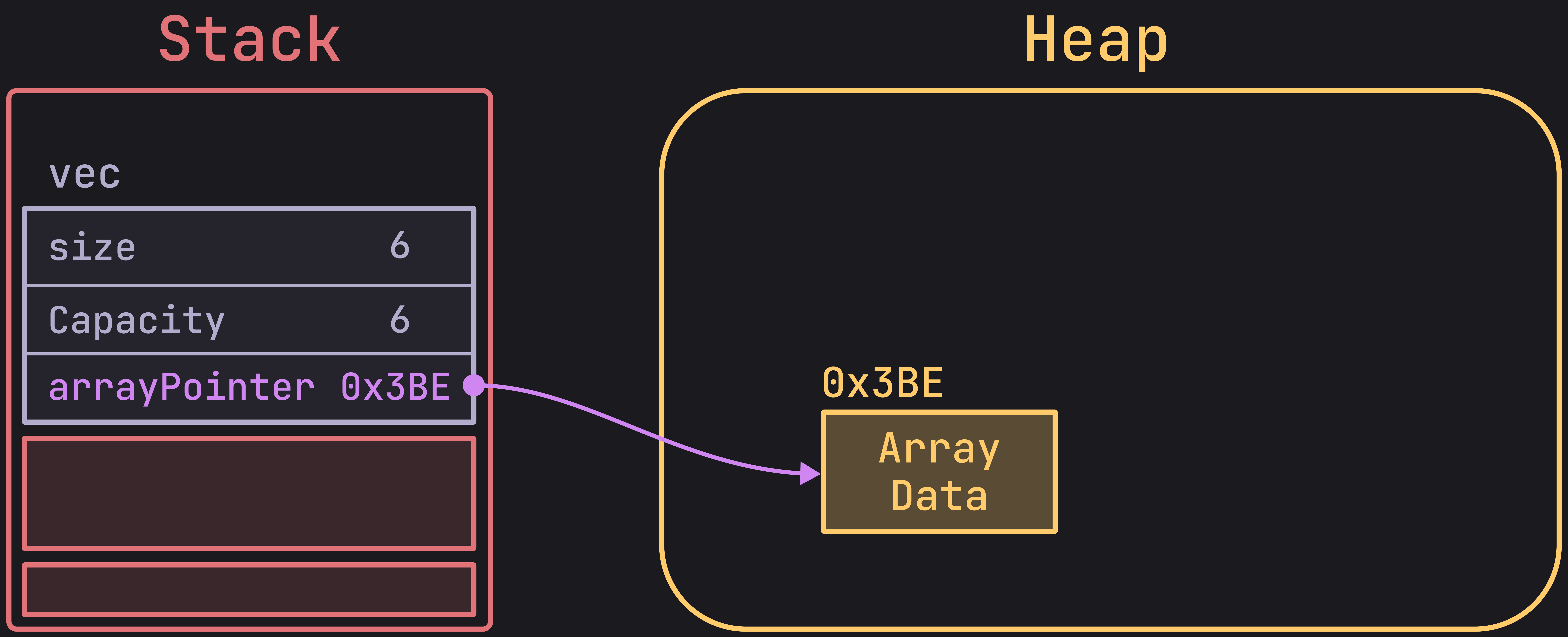
Pointer to a
player object



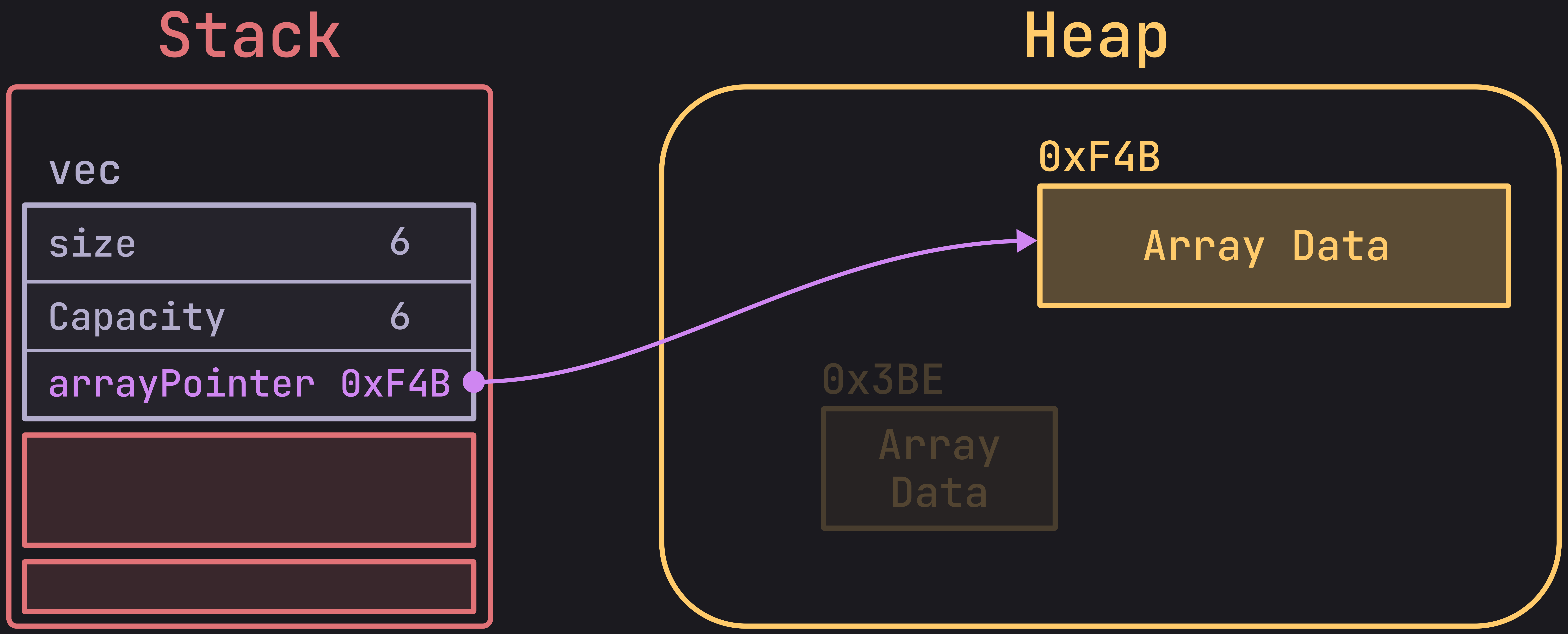

```
Player * player1 = new Player();
```

this allocates memory on the heap and returns a pointer


Stack
Heap
0x3BE
0x0FA
0x41C
0x0FA
0x3BE
Array Data
0x41C
int * array = new int[size * 2];
dynamic size




```
std::vector<int> vec {1, 2, 3, 4, 5, 6};
```




```
std::vector<int> vec {1, 2, 3, 4, 5, 6};
vec.push(7); ⟵ Grows the underlying array
```

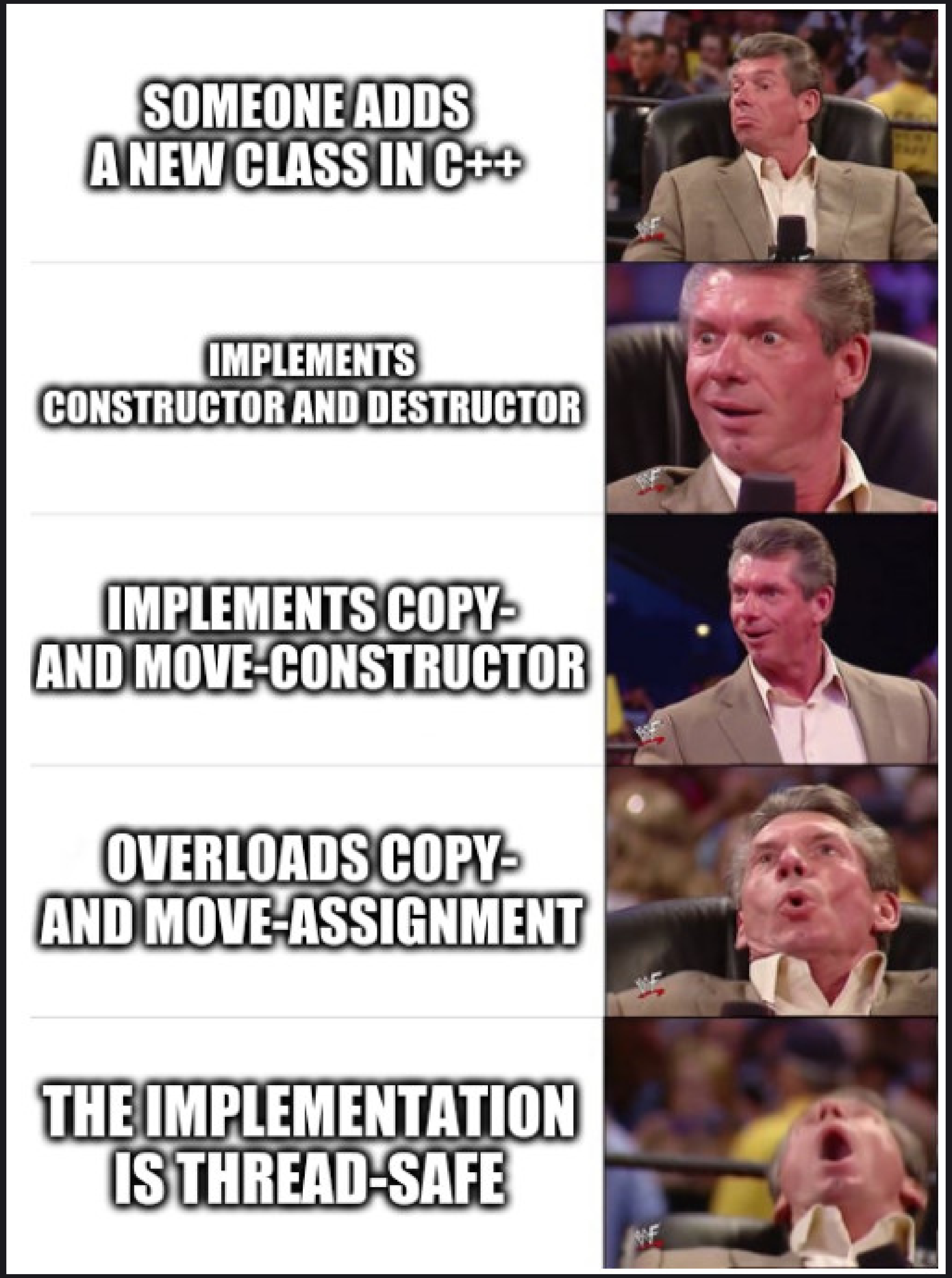
Constructors and Destructors

Constructors and Destructors

There are a ***lot*** of different types of constructors you can create in C++

Today we are learning about:

- Standard Constructor
- Destructor
- Copy Constructor
- Copy Assigment



Constructors

Constructors are all the different ways we can create a certain class

- Allocates heap memory
- sets the properties

Destructors

Destructor is the way we destroy a class

- Frees the heap memory

Default Constructor

```cpp
class Player {
public:
    std::string name;
    int health;
}

Player player; // calls the default constructor
player.name;   // equals: ""
player.health; // equals: 0
```

Default Constructor

```
class Player {
public:
    std::string name;
    int health;
    // next line overrides the default constructor
    Player() {
        name = "Unknown";
        health = 100;
    }
}
```

Parameterized Constructor

```
class Player {
public:
    std::string name;
    int health;
    // Parameterized Constructor
    Player(std::string playerName, int playerHealth) {
        name = playerName;
        health = playerHealth;
    }
}
```

Initialization List

```cpp
class Player {
public:
    std::string name;
    int health;
    // Constructor using an initialization list
    Player(std::string playerName, int playerHealth)
        : name(playerName), health(playerHealth) {}
}
```

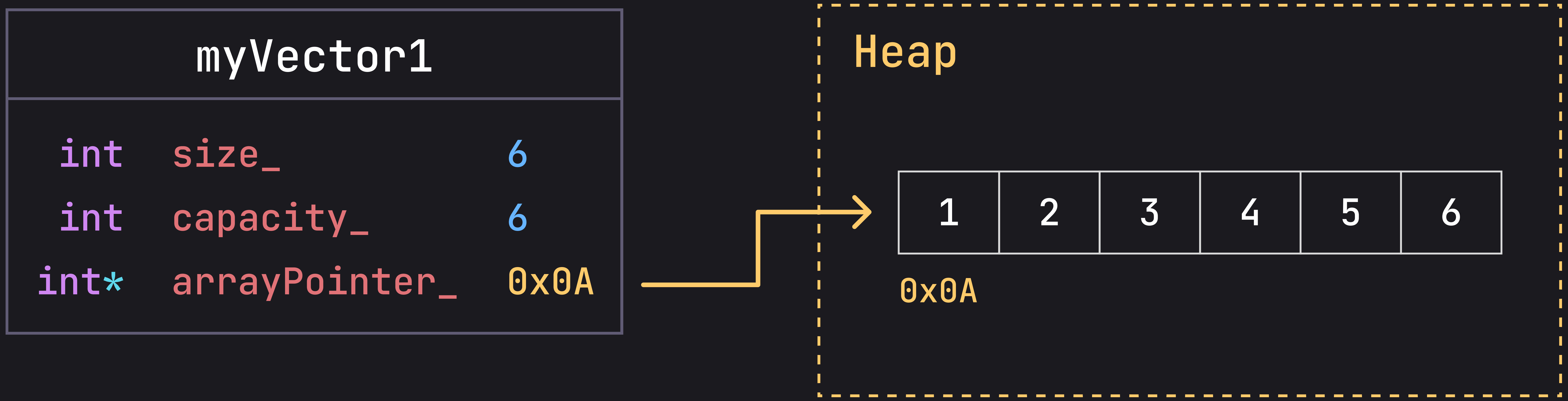
If your class doesn’t deal with **heap memory** this is usually all you need to do

Rule of Three

If your class has a pointer to something on the **heap** ⟹

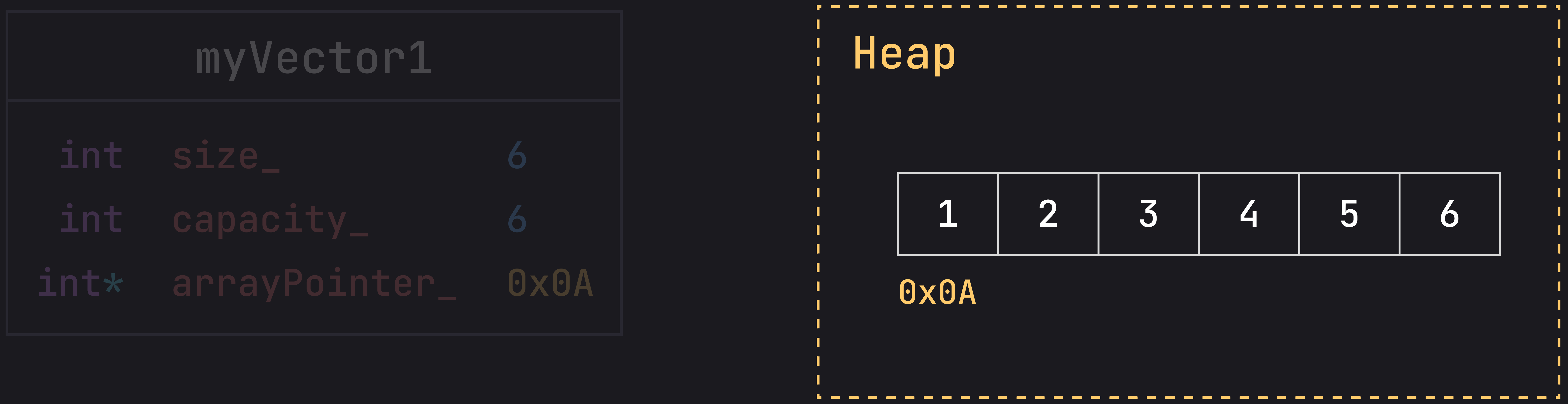
1. Destructor
2. Copy Constructor
3. Copy Assignment Operator

Destructor



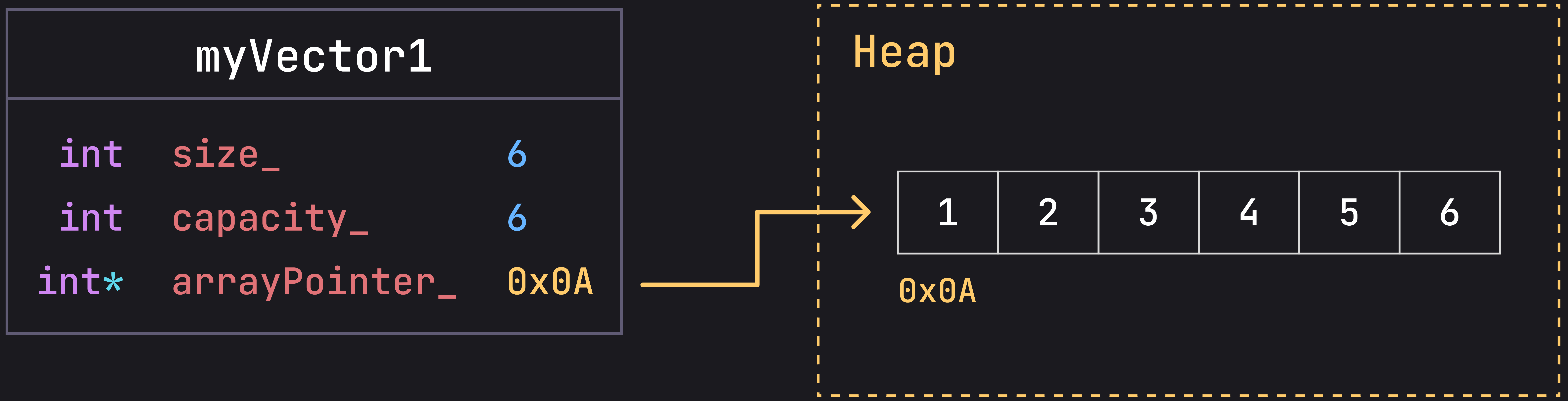
If we use the default destructor on myVector1, the heap memory well remain on the heap. This is called a **memory leak**

Destructor



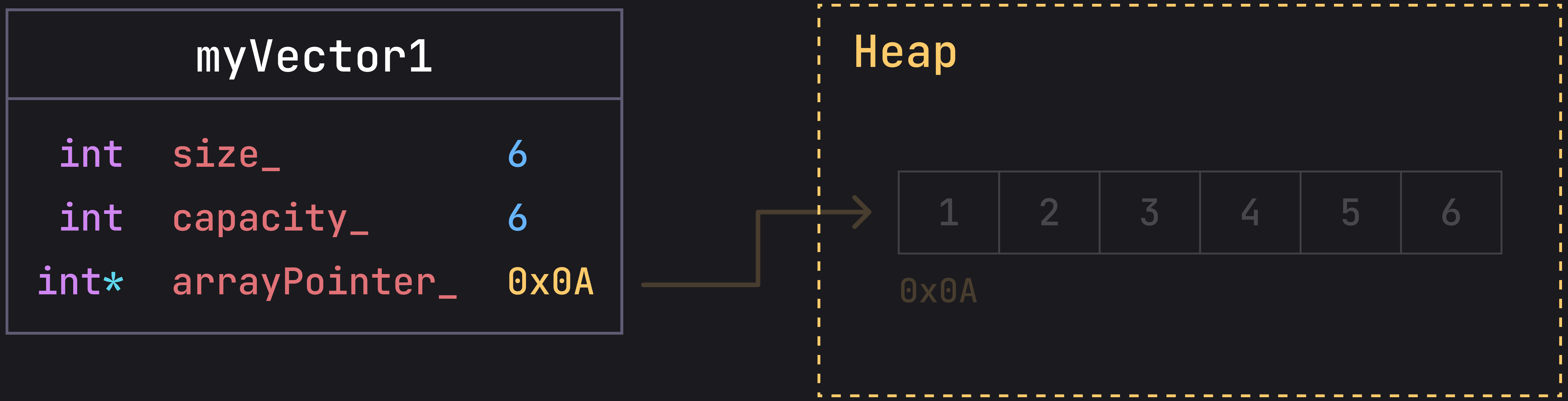
We have now lost all references to the heap memory object, so we can no longer free it.

Destructor



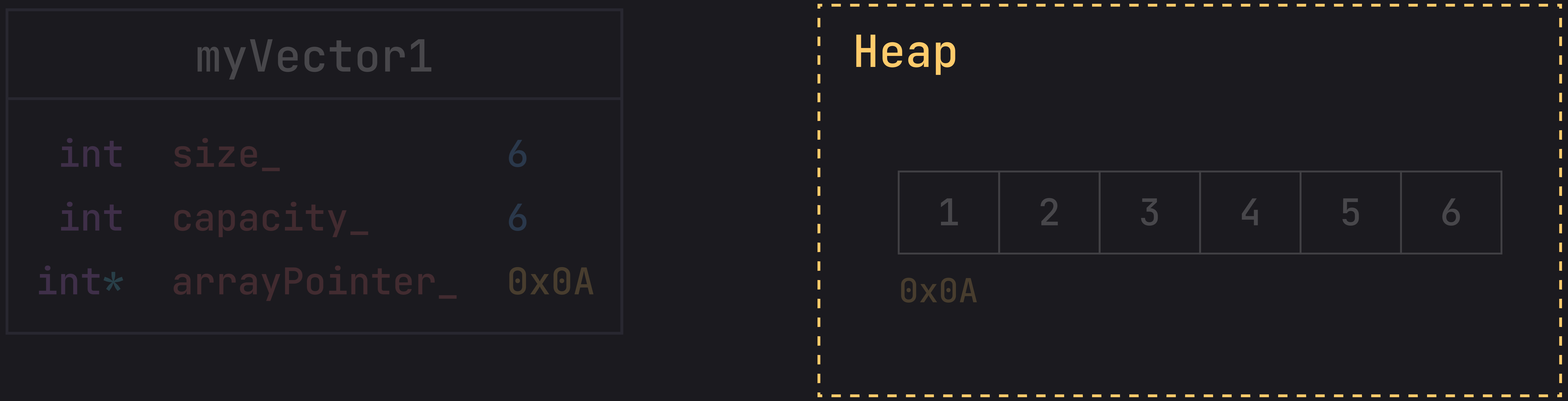
Instead we first need to call delete on the the array

Destructor



And then we can destruct the myVector1 object

Destructor



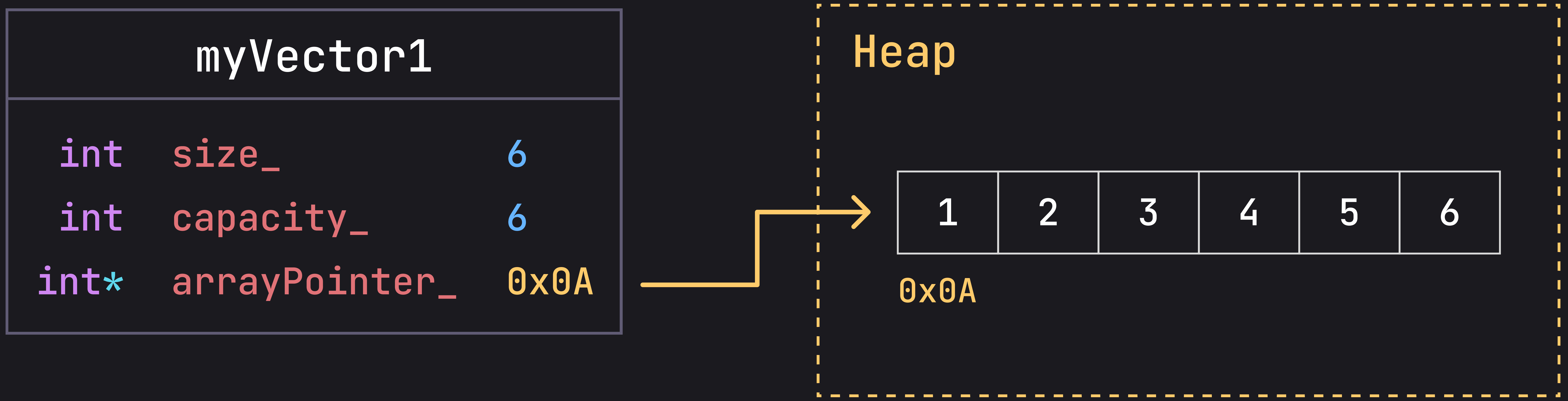
And then we can destruct the myVector1 object

Destructor

```cpp
MyVector::~MyVector() {
   delete[] arrayPointer_;
}
```

We simply need to override the default destructor and in it we delete the heap allocated array

Copy Constructor



Here we have a single instance of our MyVector class. Notice we have three pieces of data. The size, the capacity and the array pointer

Copy Constructor

myVector1

int	size_	6
int	capacity_	6
int*	arrayPointer_	0x0A

myVector2

int	size_	6
int	capacity_	6
int*	arrayPointer_	0x0A

Heap

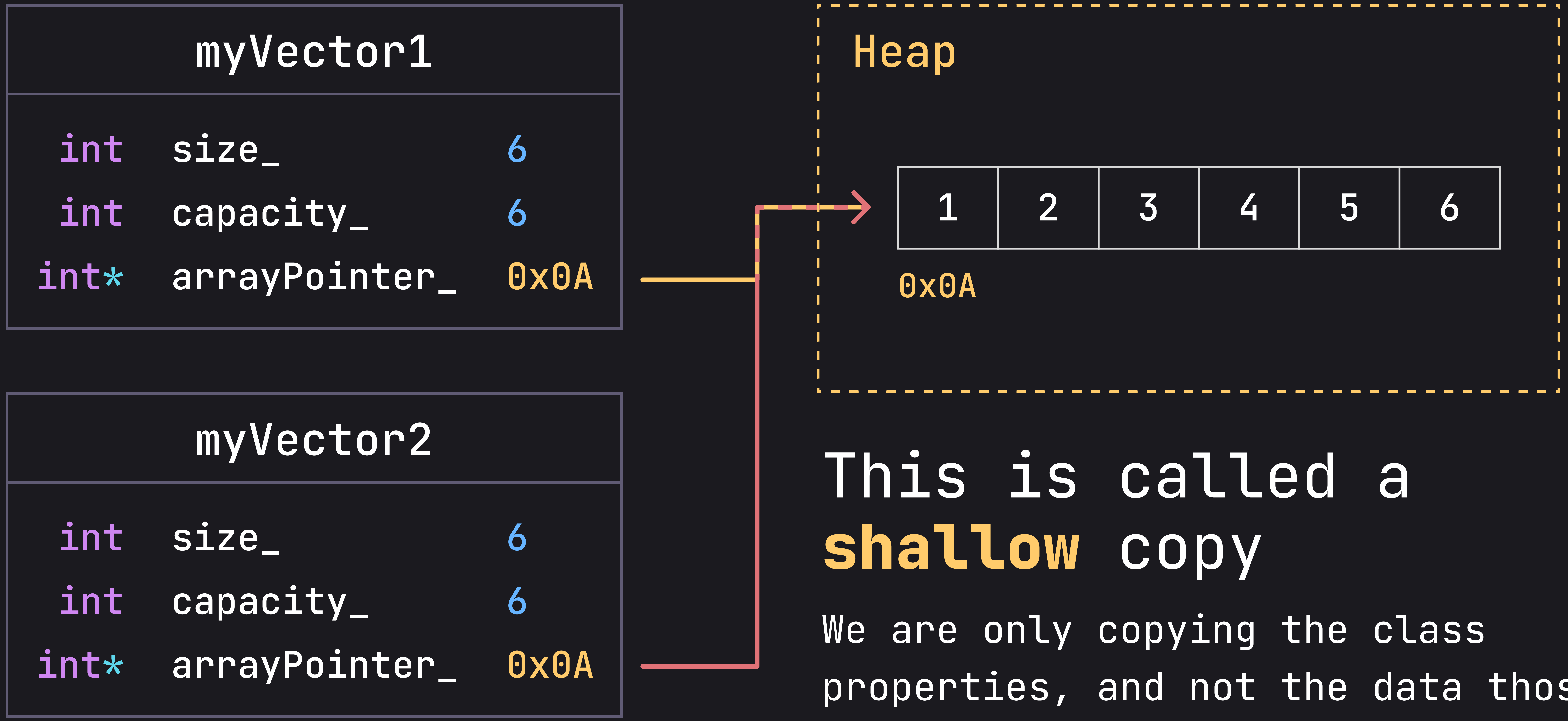
1	2	3	4	5	6

0x0A

If we make a copy using the **default** copy constructor all the **values** get directly copied.

(Remember that a **pointer** is just a value)

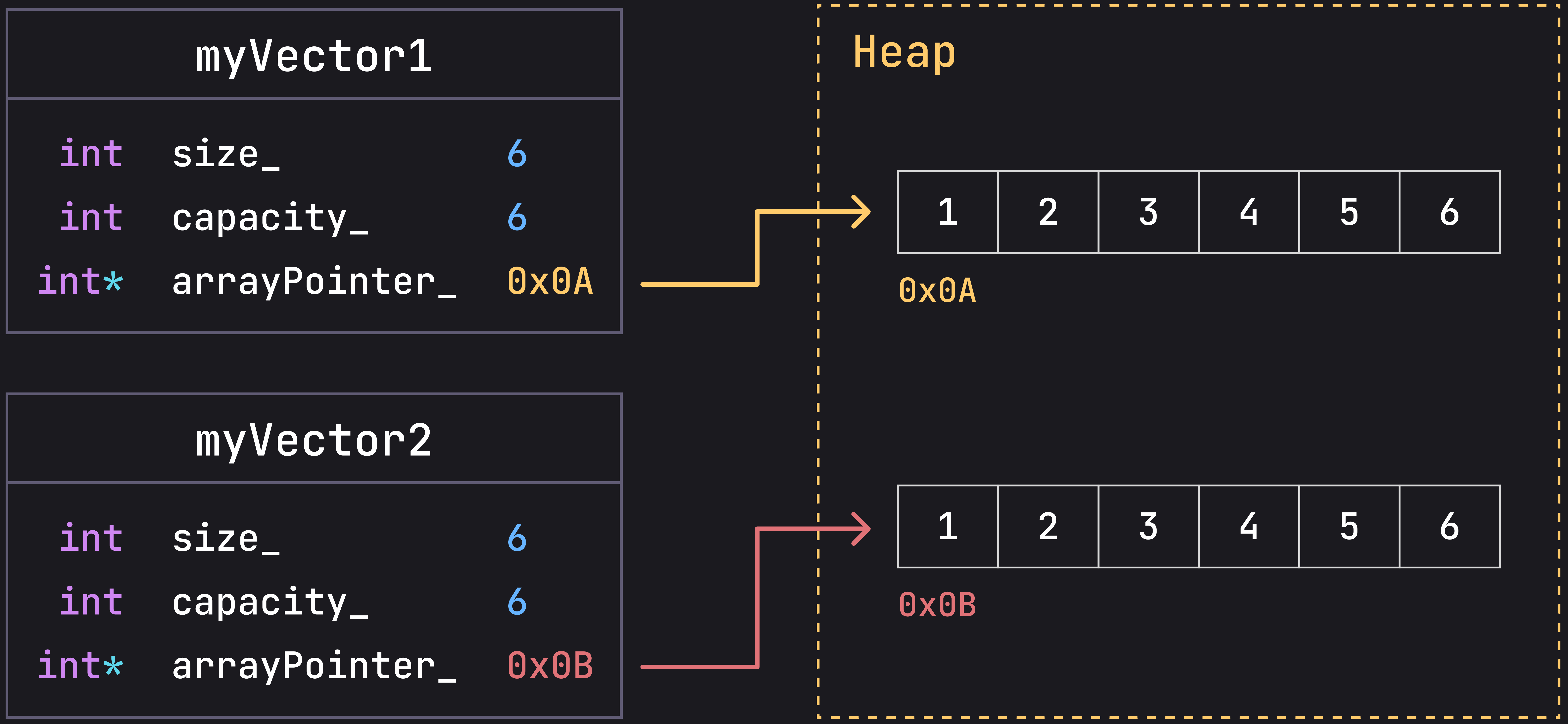
Copy Constructor

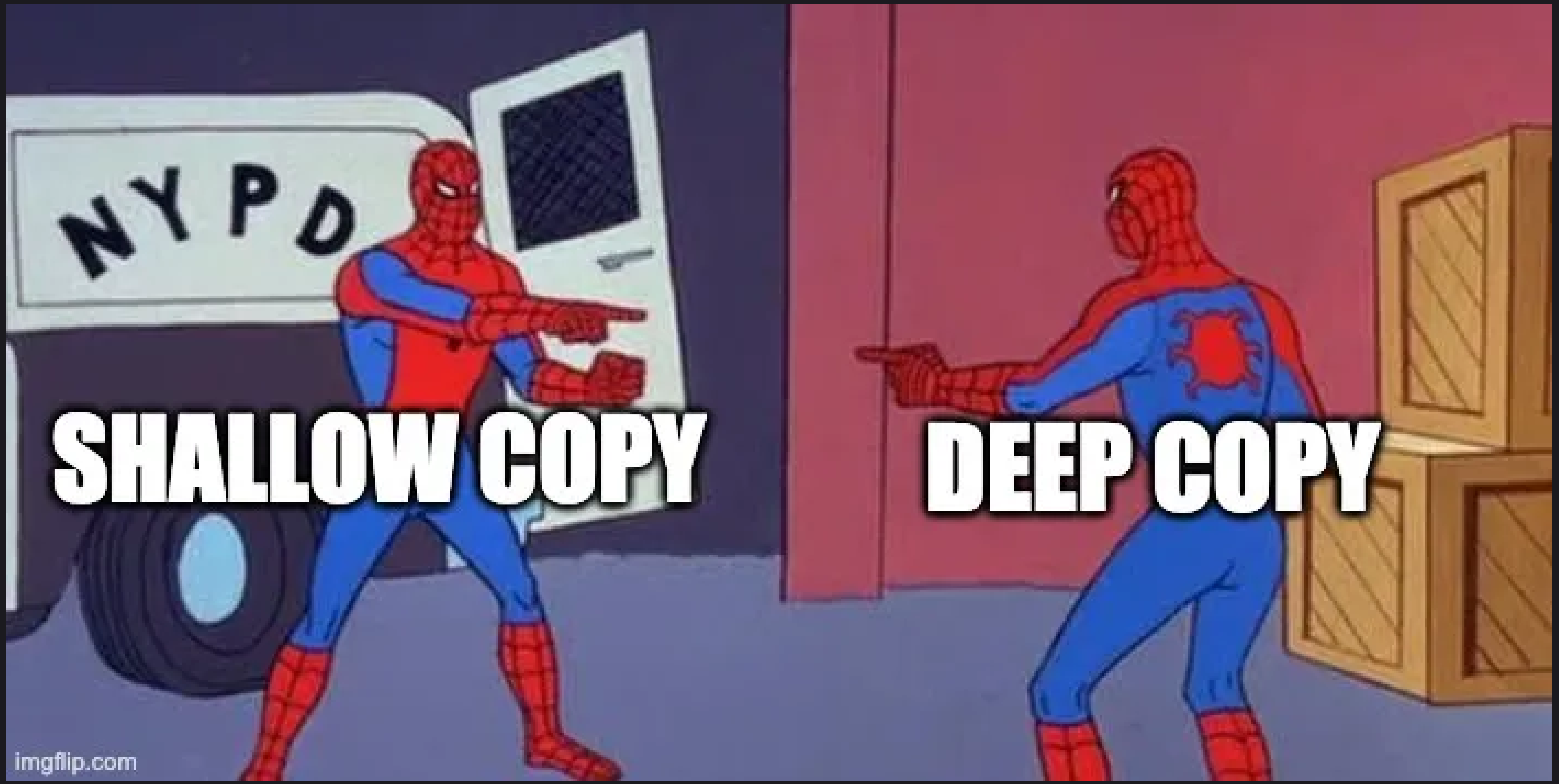


This is called a **shallow** copy

We are only copying the class properties, and not the data those properties point to

But this is what we really want. We want to copy the **underlying array** on the heap!





Has **no pointers** to things on the heap	has a pointer to something on the **heap**

Copy Constructor

```
MyVector::MyVector(const MyVector& other)
    : arrayPointer_ {new int[other.size_]},
      size_ {other.size_},
      capacity_ {other.capacity_} {

  // Deep copy
  for (int i = 0; i < size_; ++i) {
    arrayPointer_[i] = other.arrayPointer_[i];
  }
}
```

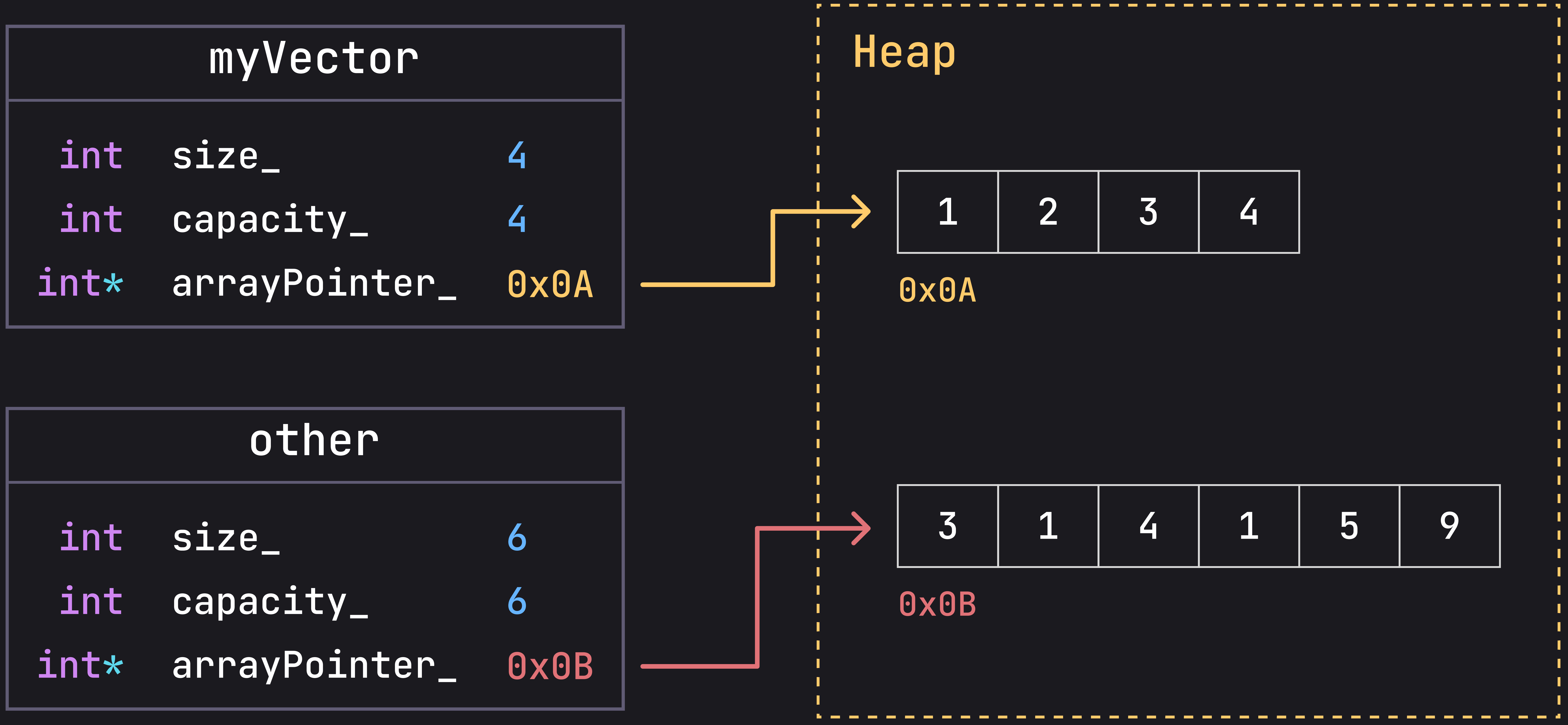
Copy Constructor (improved)

```cpp
MyVector::MyVector(const MyVector& other)
    : arrayPointer_ {other.size_ > 0 ? new int[other.size_] : nullptr},
      size_ {other.size_},
      capacity_ {other.capacity_} {

  // Deep copy
  for (int i = 0; i < size_; ++i) {
    arrayPointer_[i] = other.arrayPointer_[i];
  }
}
```

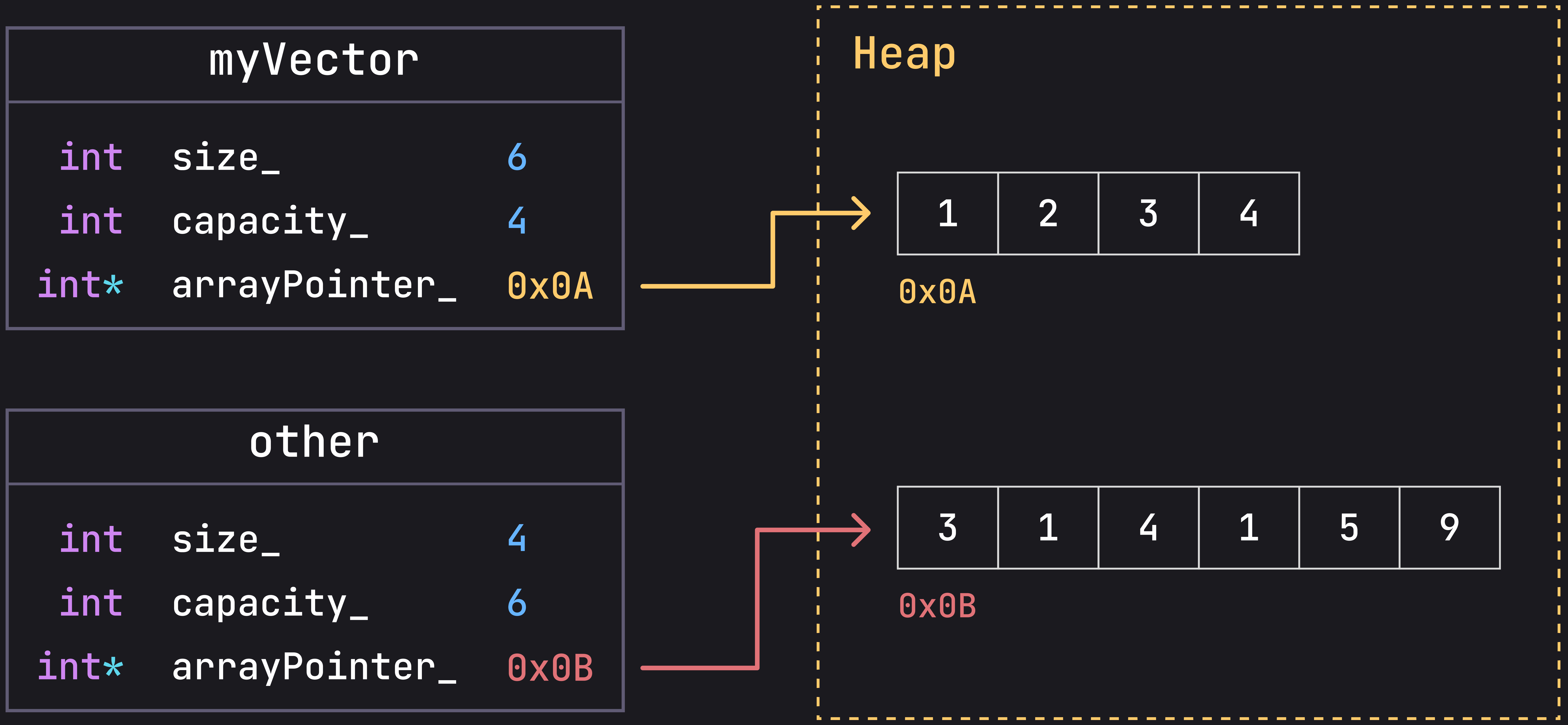
Copy Assignment

```cpp
MyVector& MyVector::operator=(MyVector other) {
  // other is passed by VALUE
  // So it is copied (its copy constructor is called)
  std::swap(arrayPointer_, other.arrayPointer_);
  std::swap(size_, other.size_);
  std::swap(capacity_, other.capacity_);
  return *this;
}
```

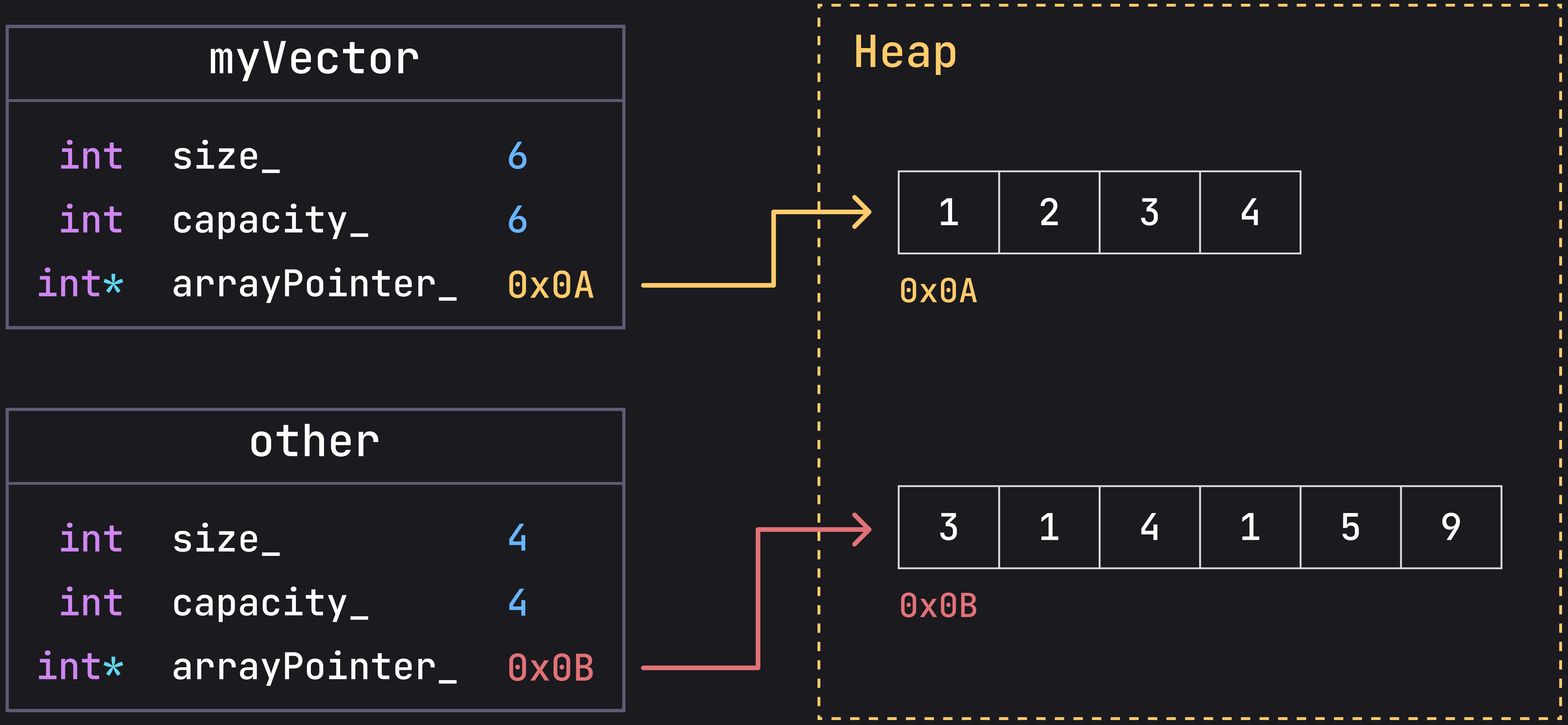
So we first swap all the values of the variables



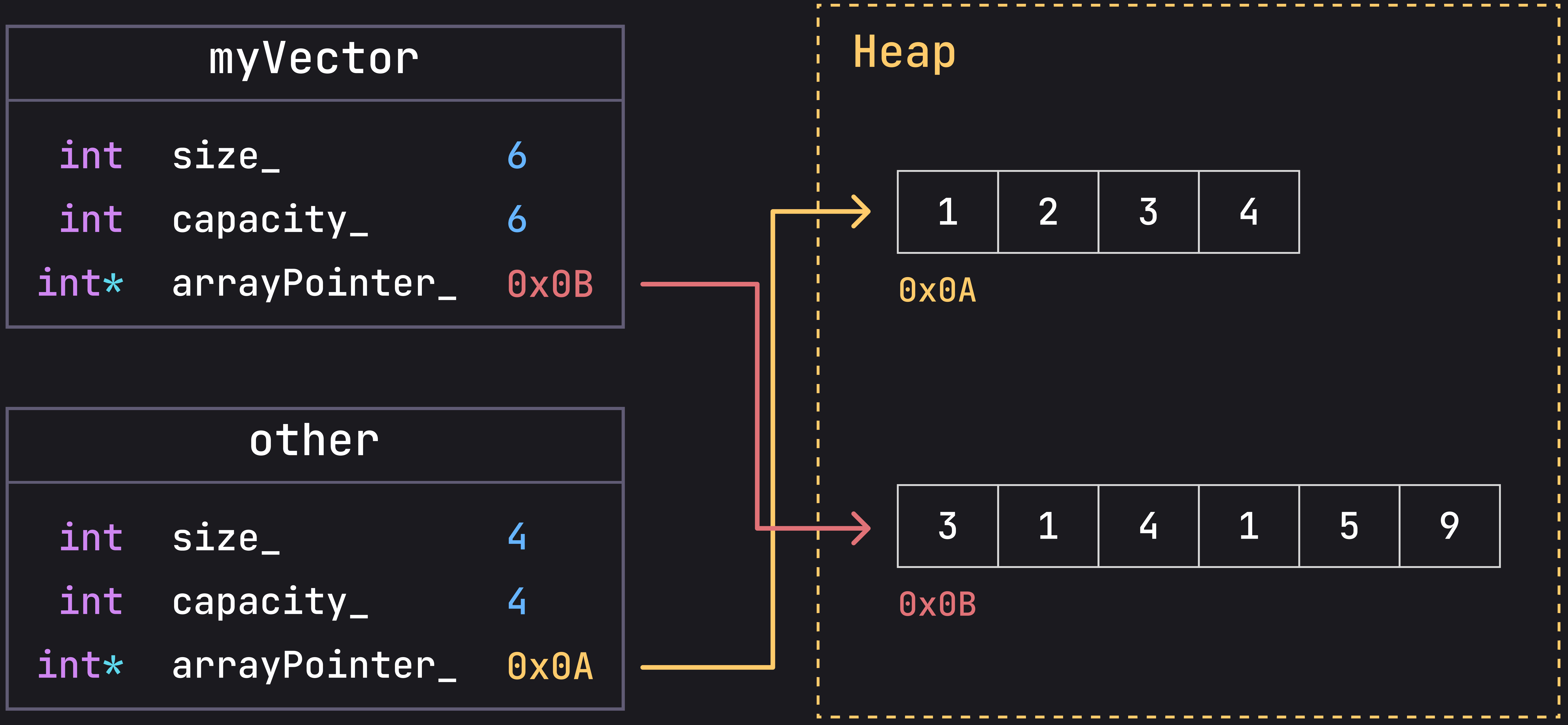
So we first swap all the values of the variables



So we first swap all the values of the variables



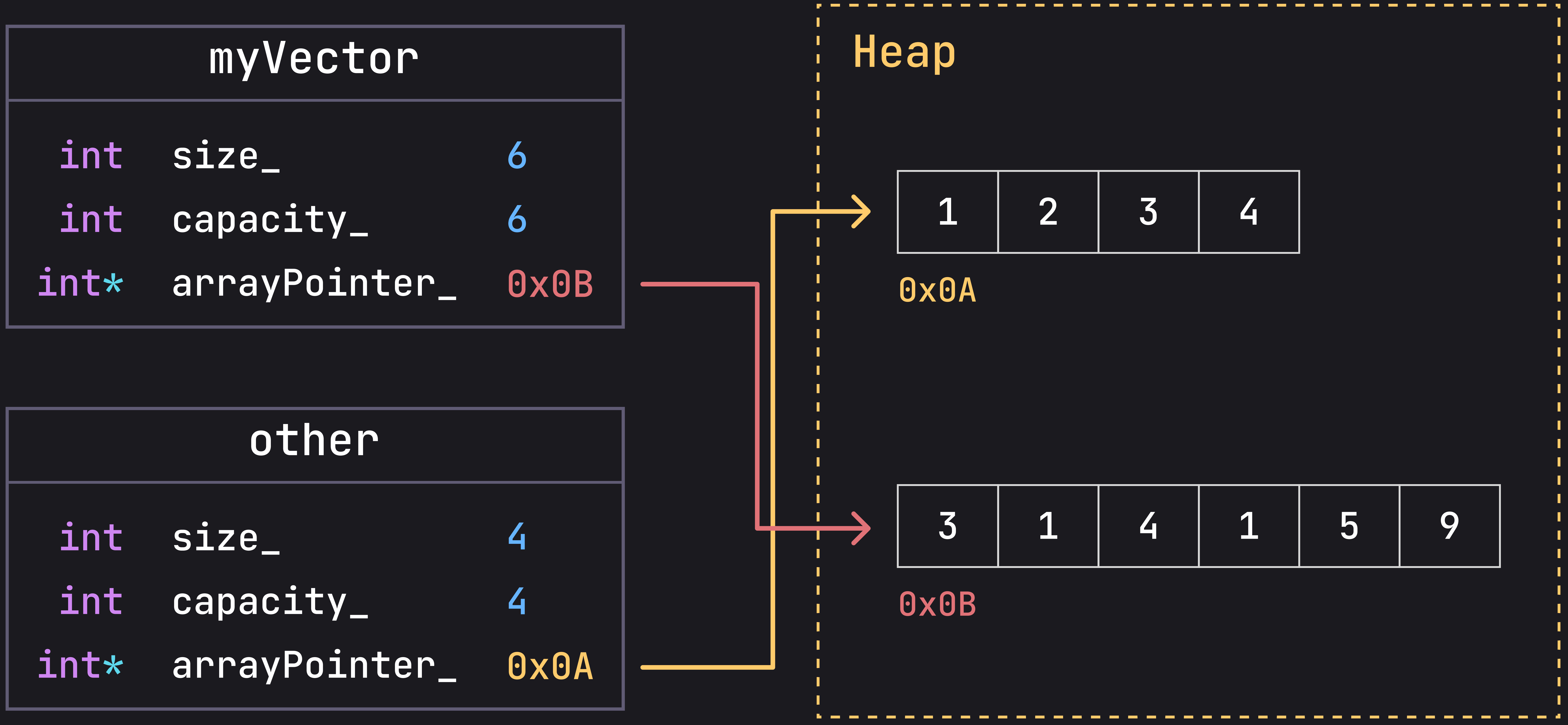
So we first swap all the values of the variables



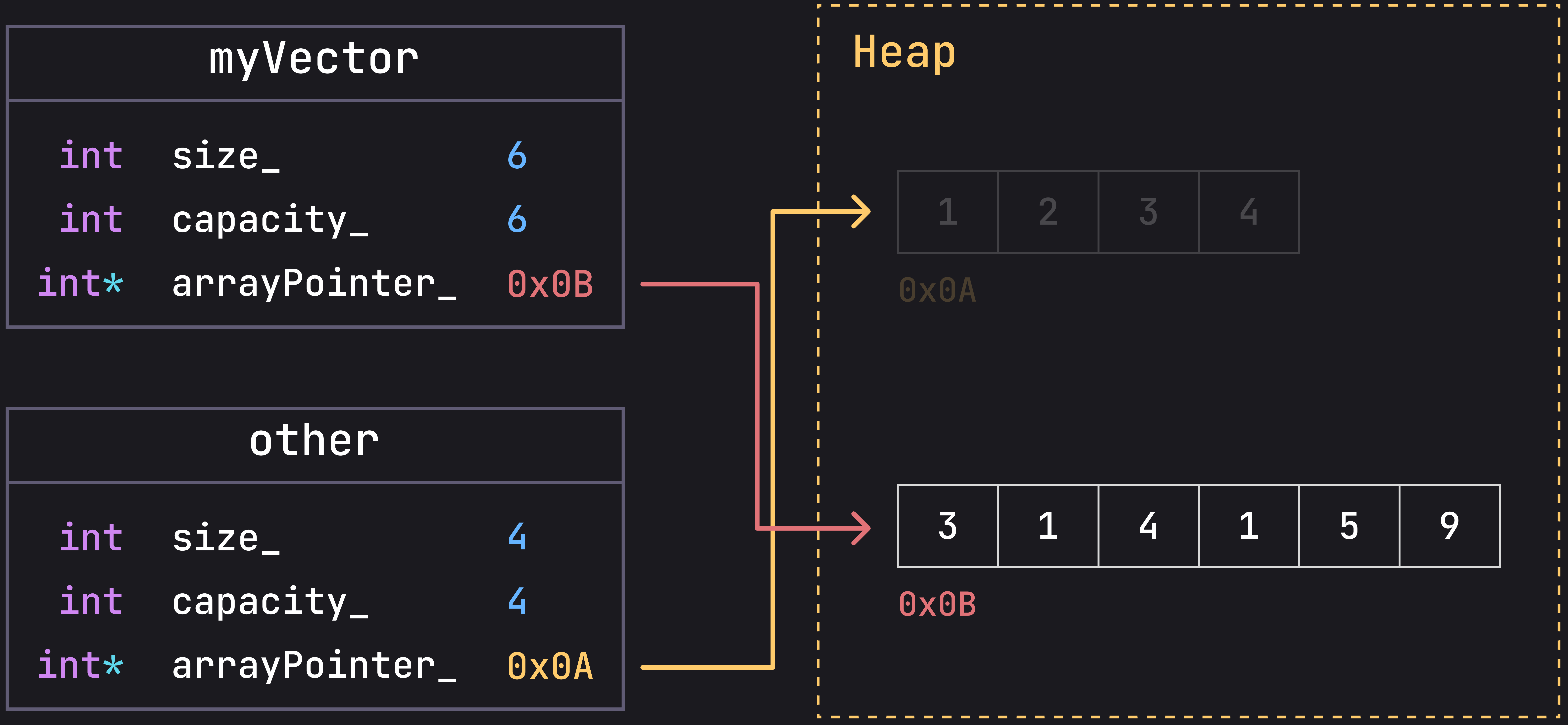
Then the destructor is called on other

```
MyVector& MyVector::operator=(MyVector other) {
  std::swap(arrayPointer_, other.arrayPointer_);
  std::swap(size_, other.size_);
  std::swap(capacity_, other.capacity_);
  return *this;
  // After this line other goes out of scope
  // so its destructor is called
}
```

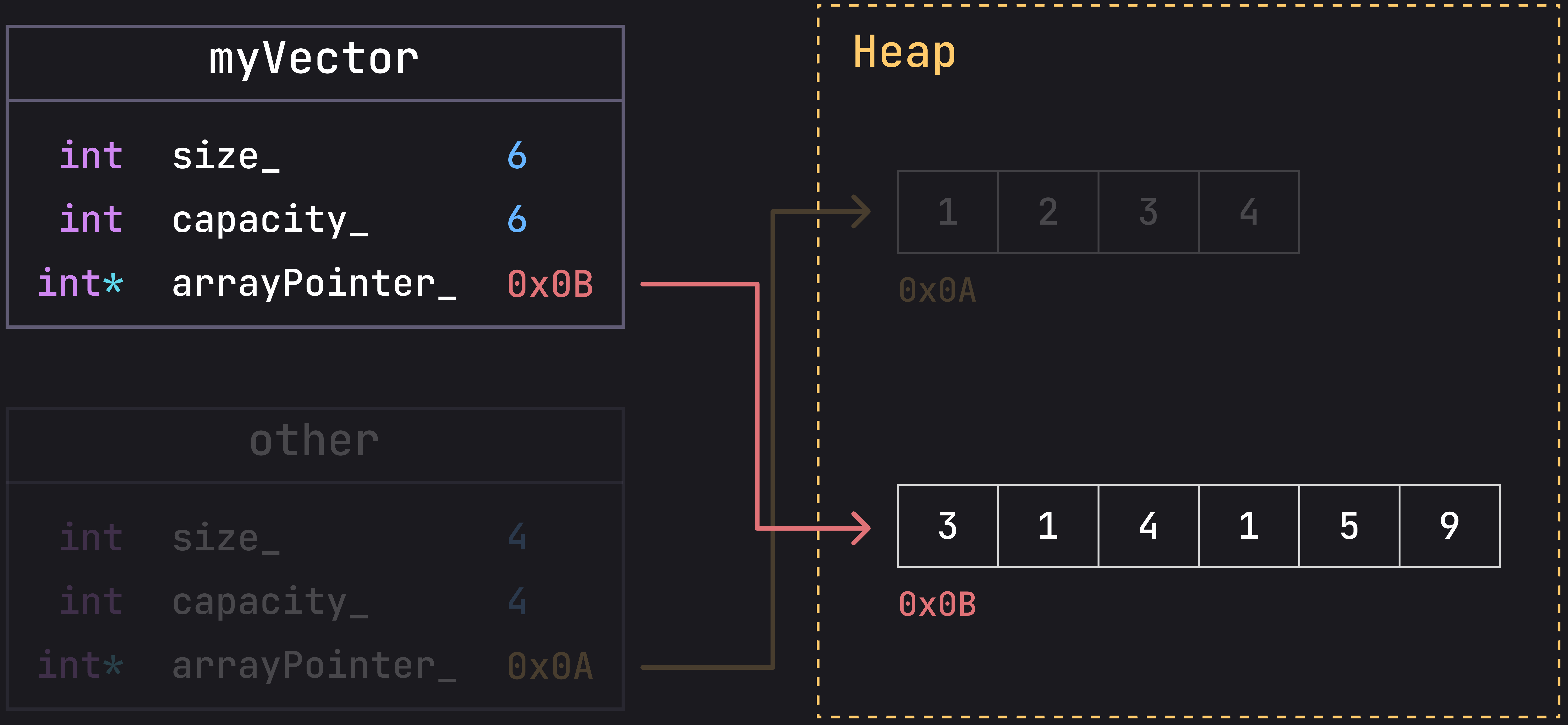
Then the destructor is called on other



Then the destructor is called on other



Then the destructor is called on other



Rule of **Three**

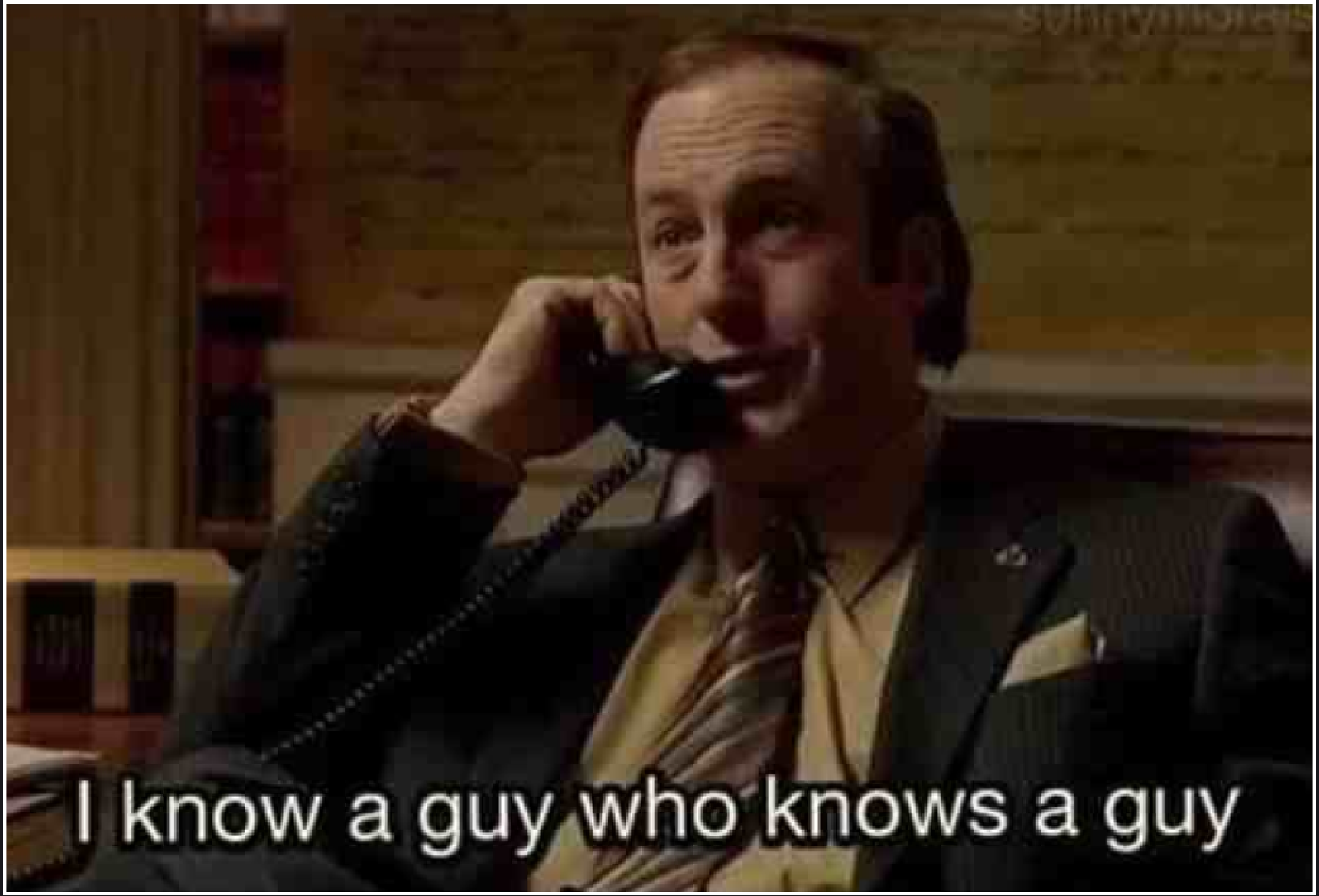
If your class has a pointer to something on the **heap** ⟹

1. Destructor
2. Copy Constructor
3. Copy Assignment Operator

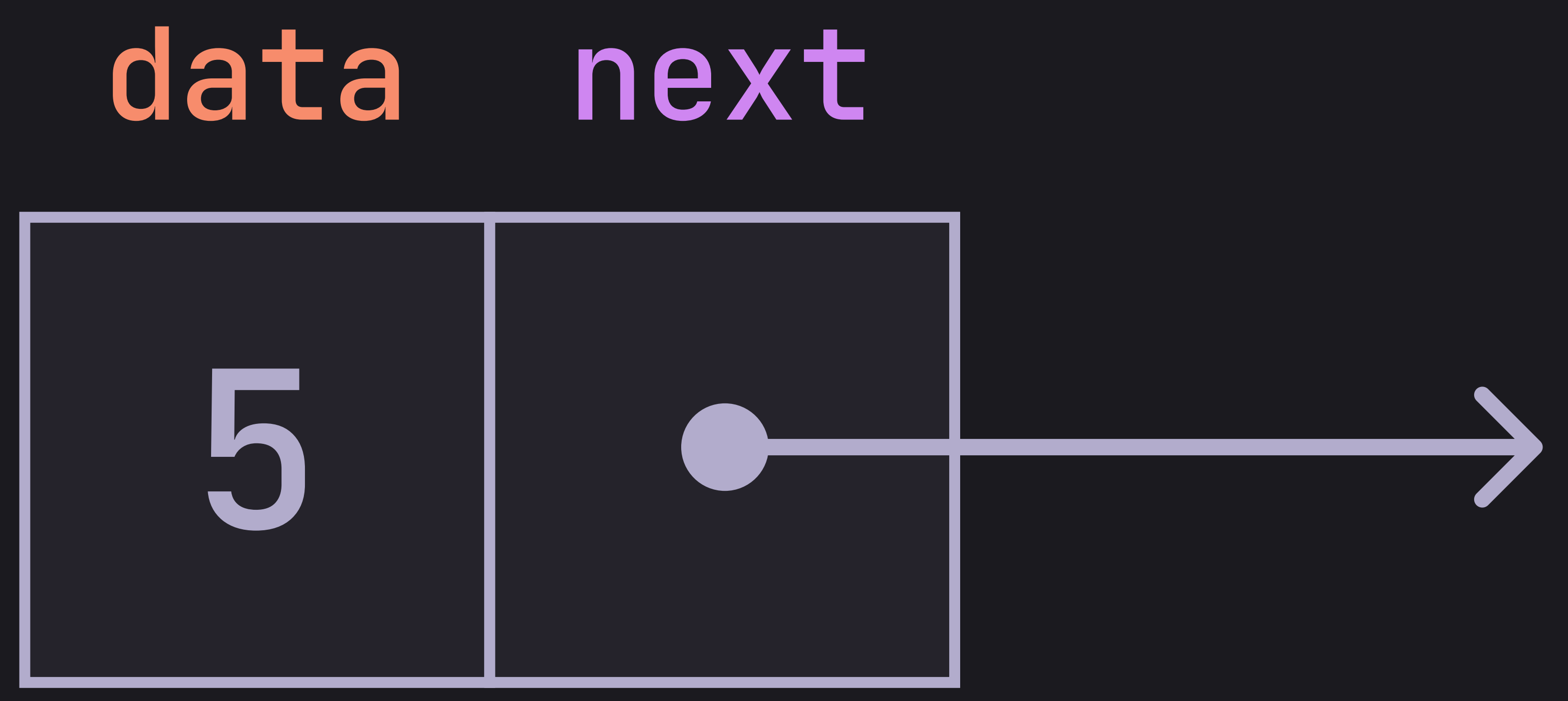
Implementing 1 & 2 makes implementing 3 pretty easy

Let’s try implement the constructors in the activity

Linked List



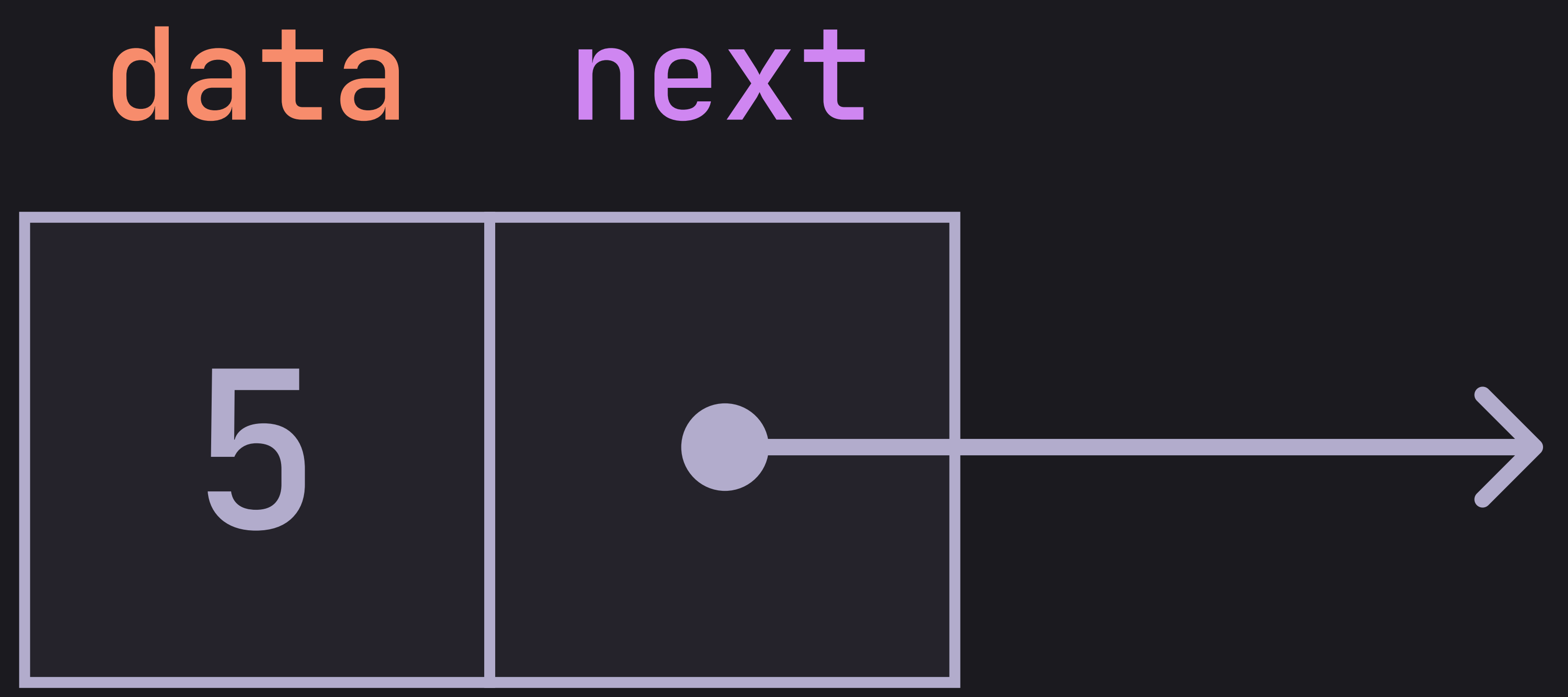
Linked List



Node

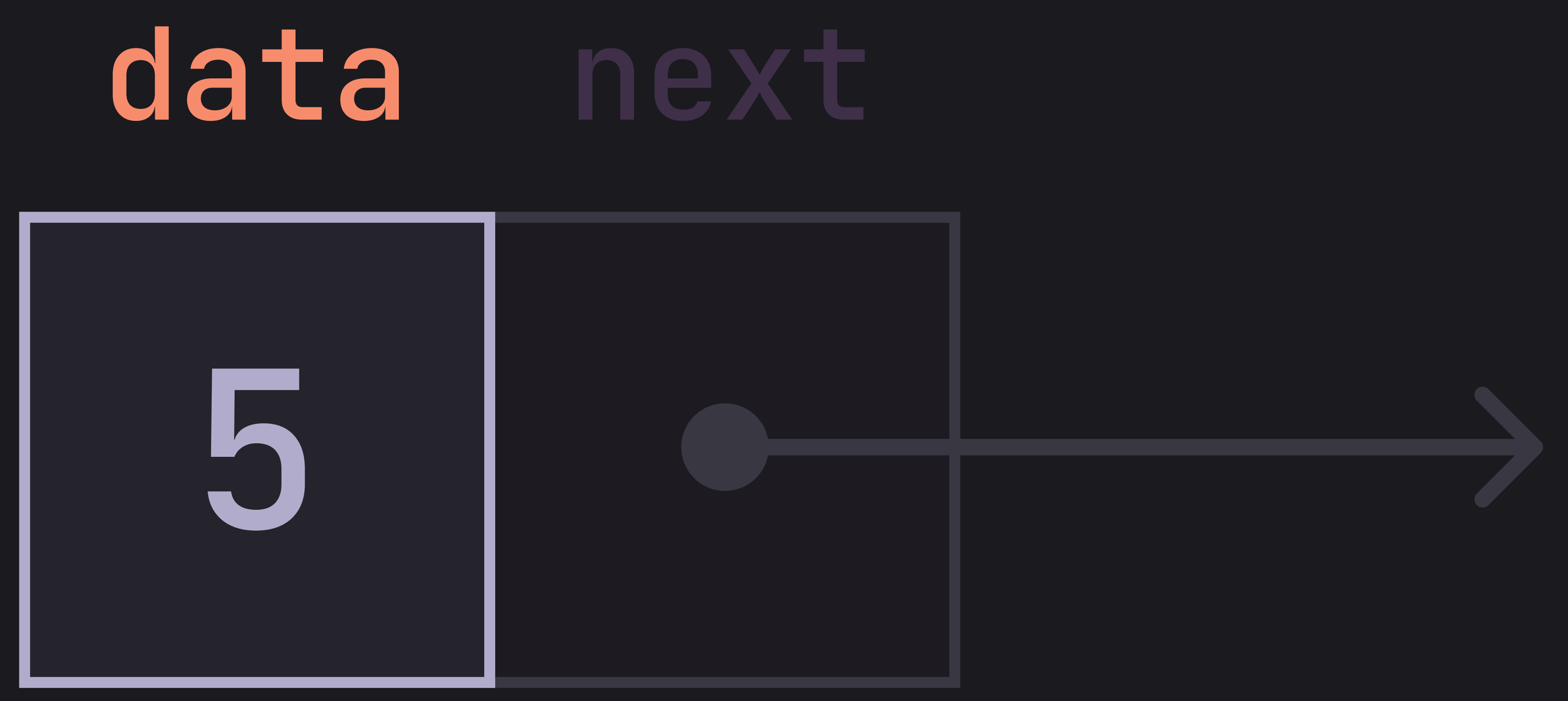
Linked List

```
struct Node {
}
```



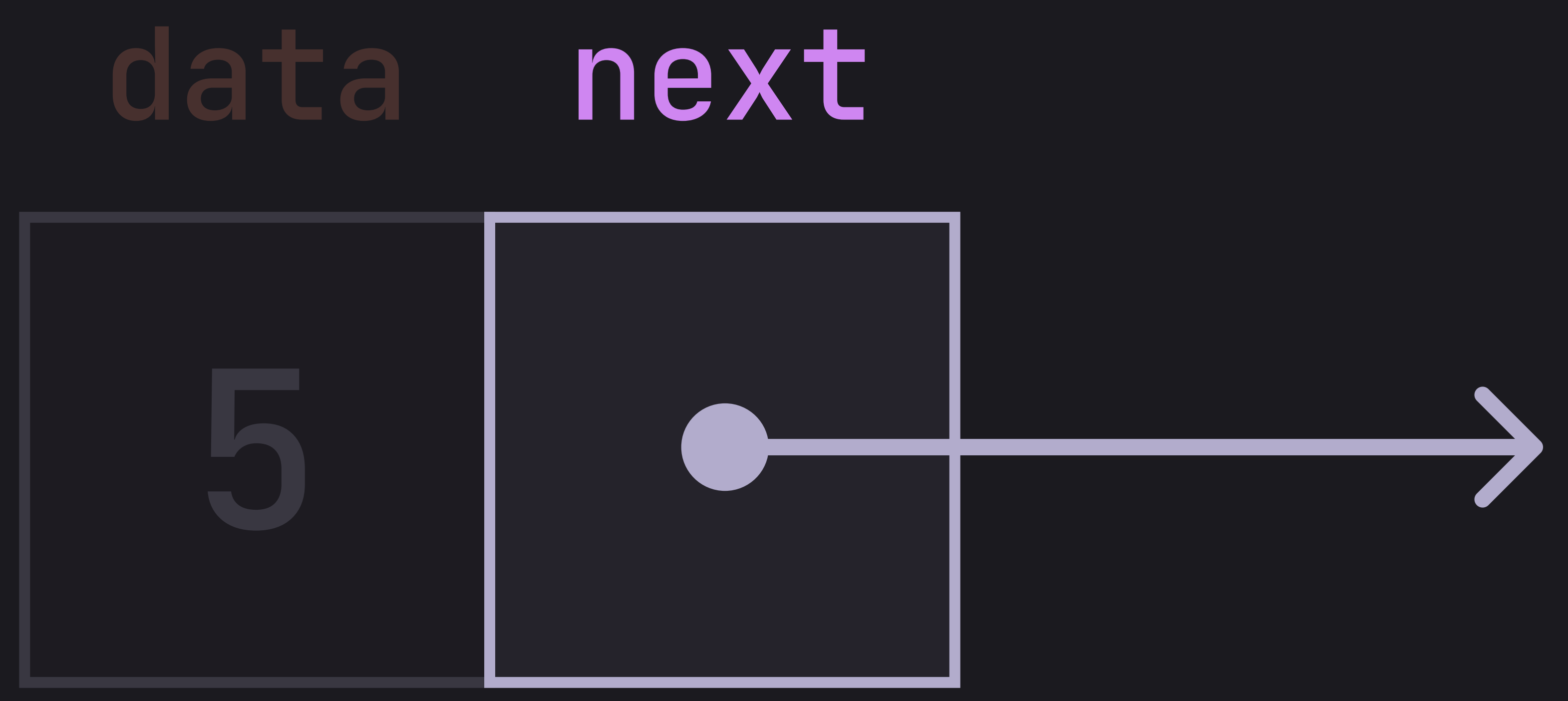
Linked List

```
struct Node {
  int data {};
}
```



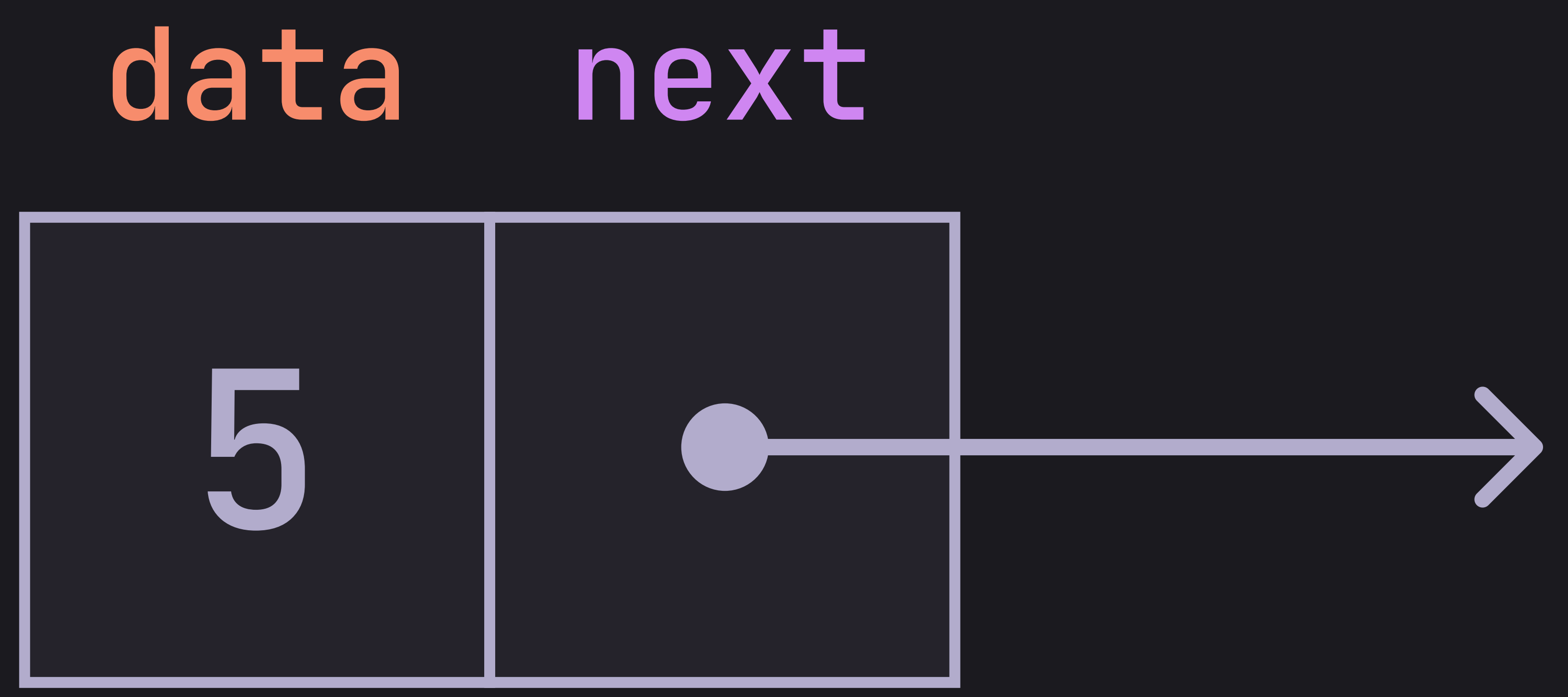
Linked List

```
struct Node {
  int data {};
  Node* next = nullptr;
}
```



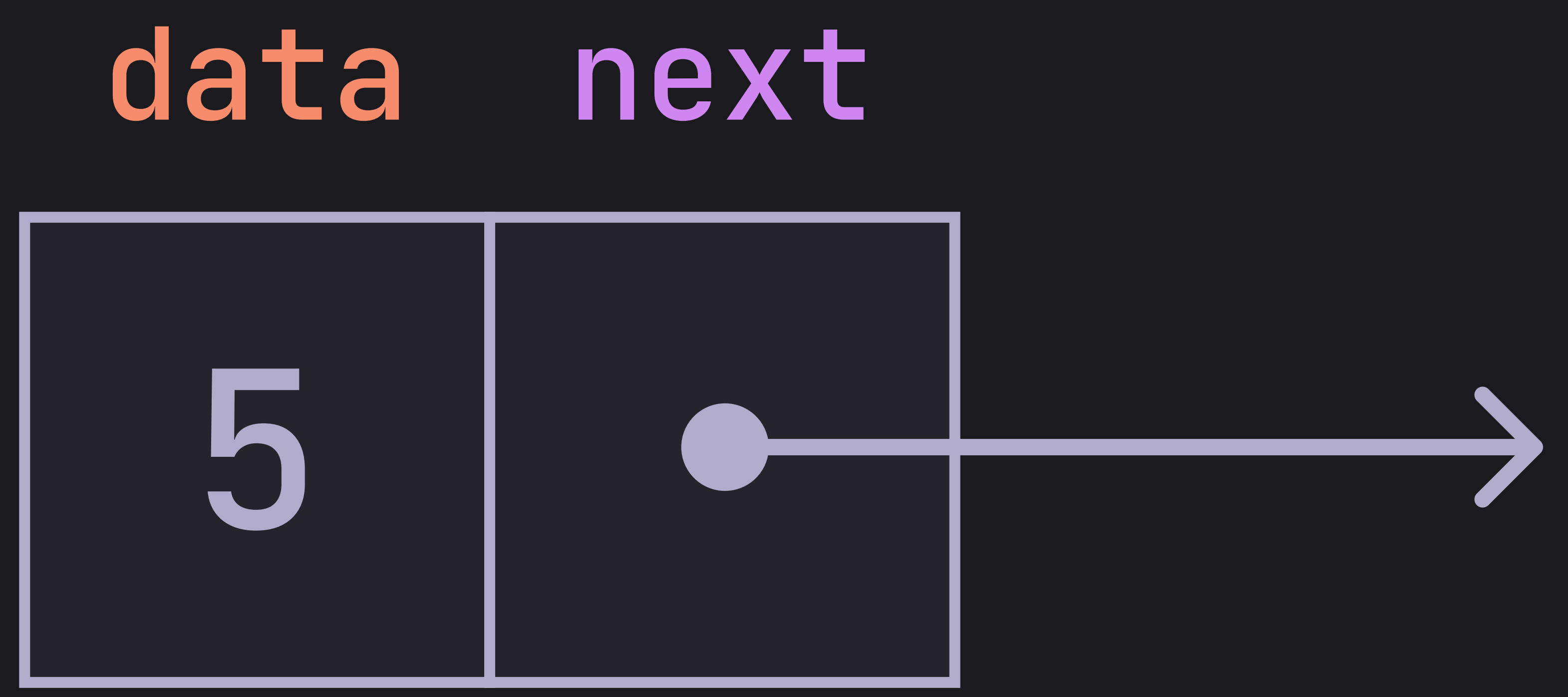
Linked List

```
struct Node {
  int data {};
  Node* next = nullptr;
  Node(){}
}
```



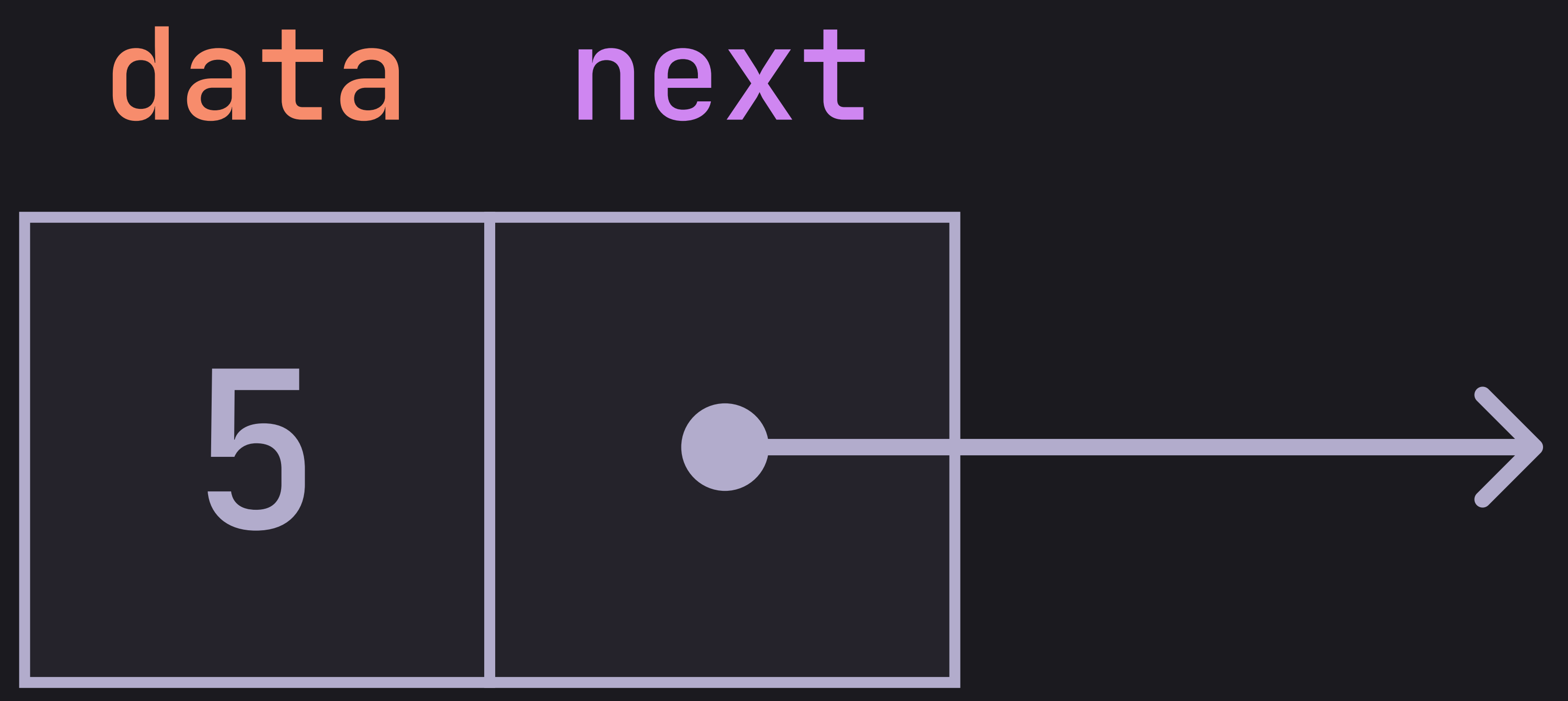
Linked List

```
struct Node {
  int data {};
  Node* next = nullptr;
  Node(){}
  Node(int input_data, Node* next_node= nullptr) :
    data {input_data}, next {next_node} {}
}
```

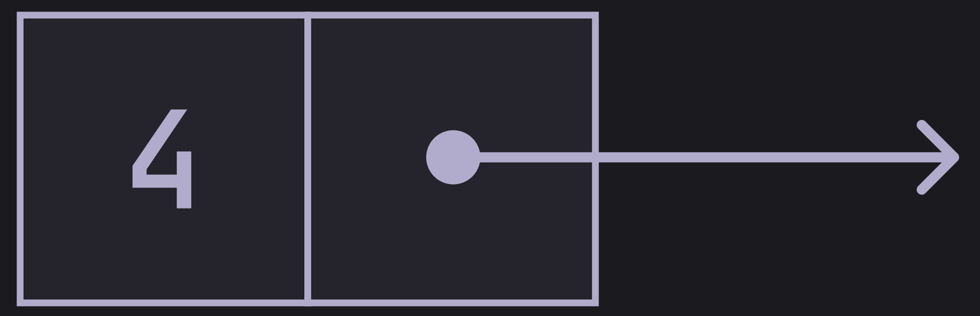


Linked List

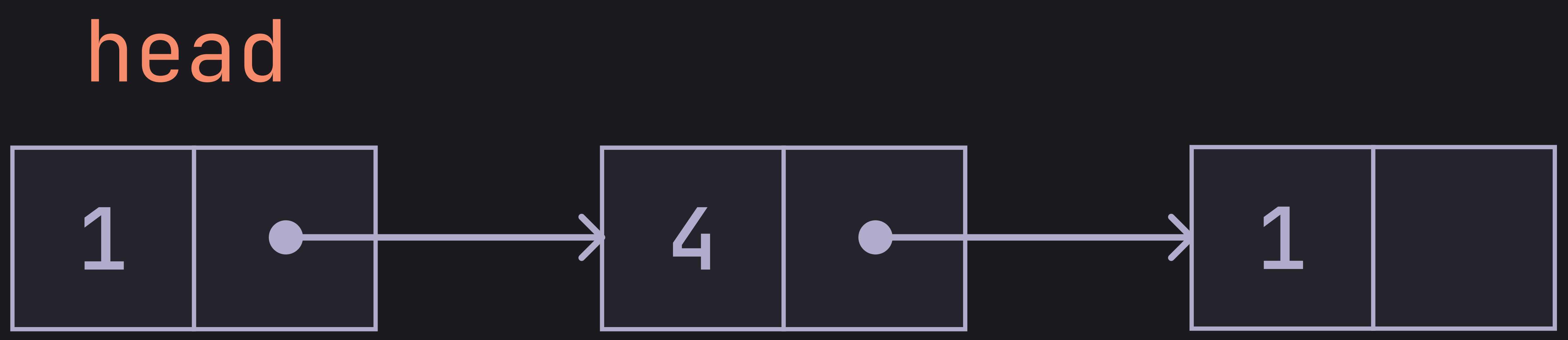
```cpp
template <typename T>
struct Node {
   T data {};
   Node* next = nullptr;
   Node(){}
   Node(T input_data, Node* next_node= nullptr) :
     data {input_data}, next {next_node} {}
}
```



Linked List

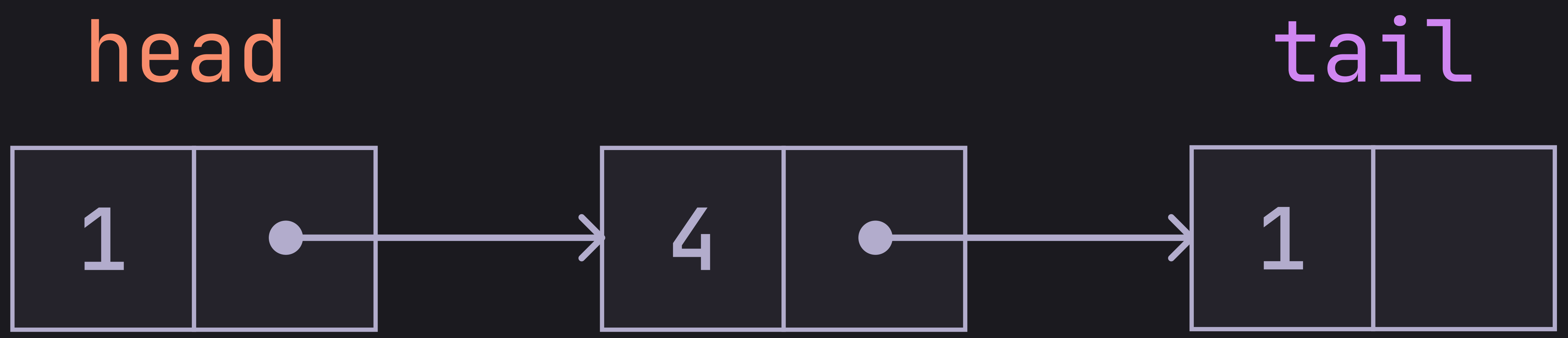


Linked List



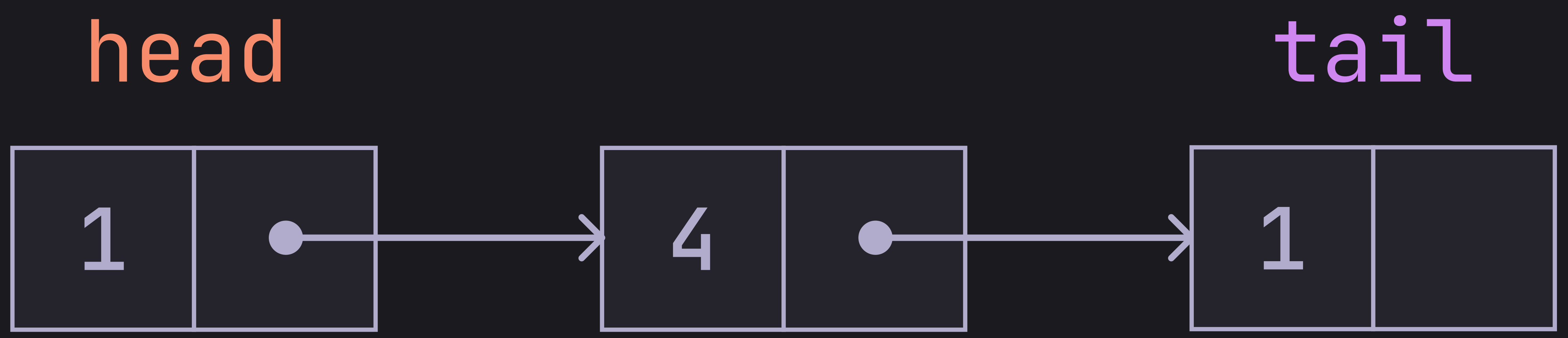
To keep track of the start of our list we define a variable head which points to the first node

Linked List



Optionally we can also define a variable tail which points to the end of our list

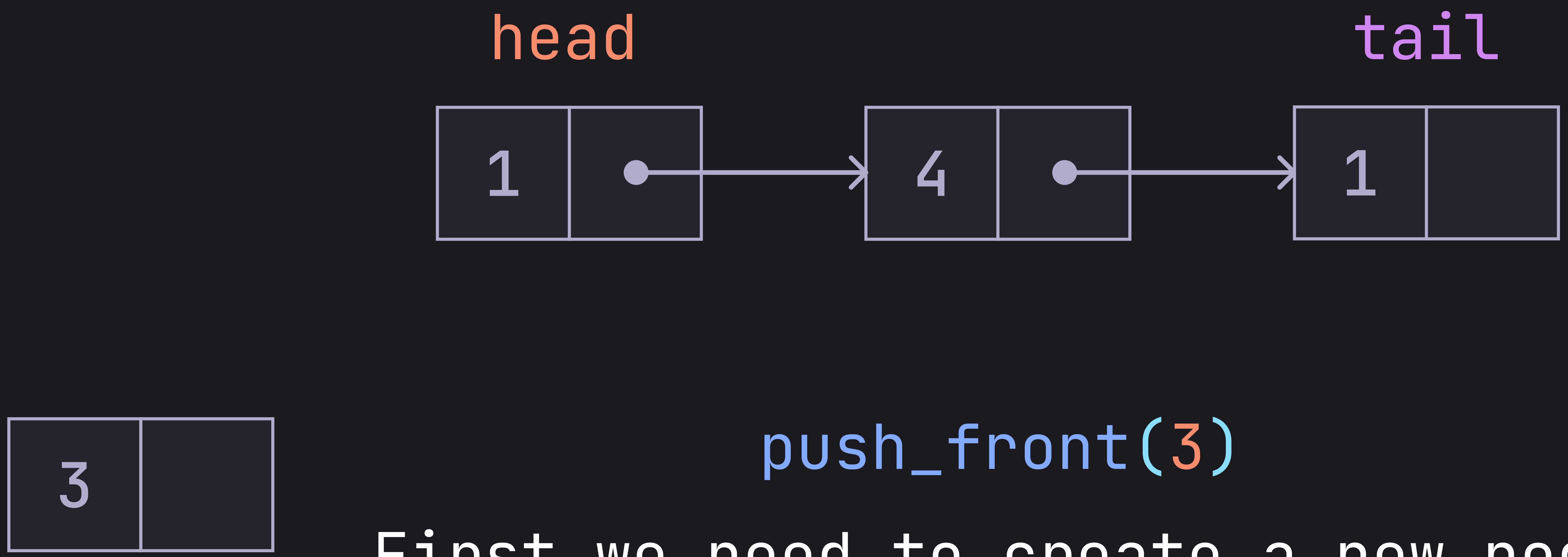
Linked List



`push_front(3)`

Linked lists allow us to quickly push to the front of the list

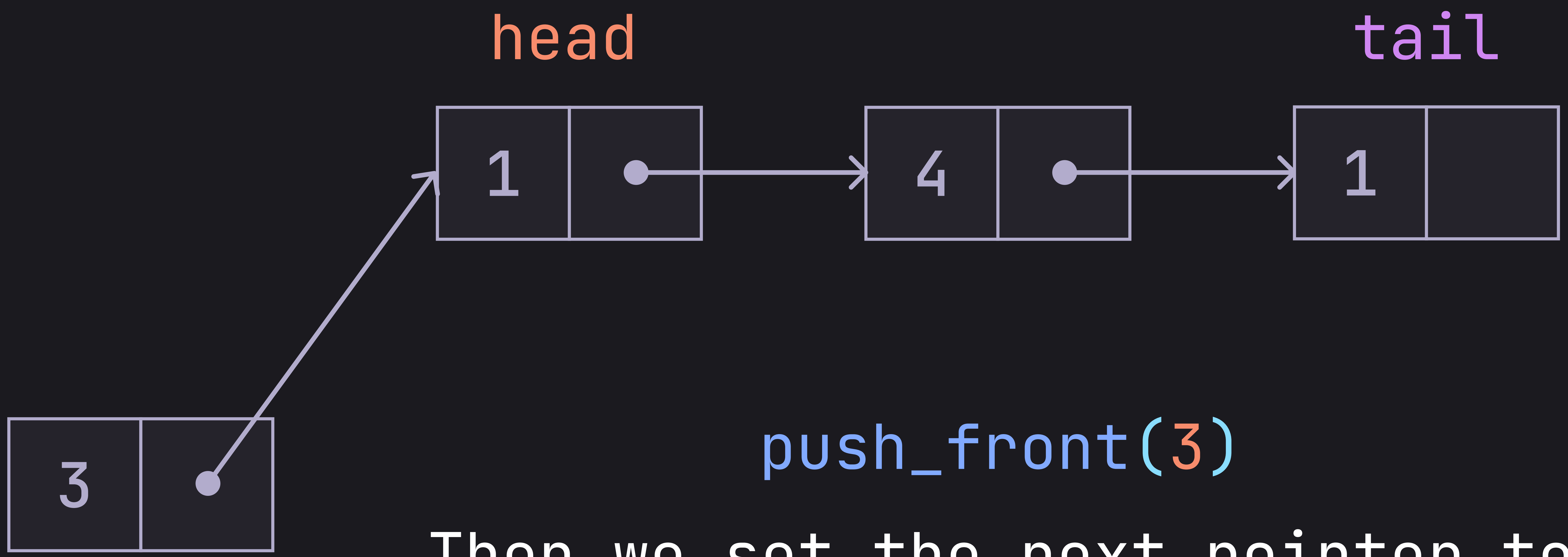
Linked List



push_front(3)

First we need to create a new node

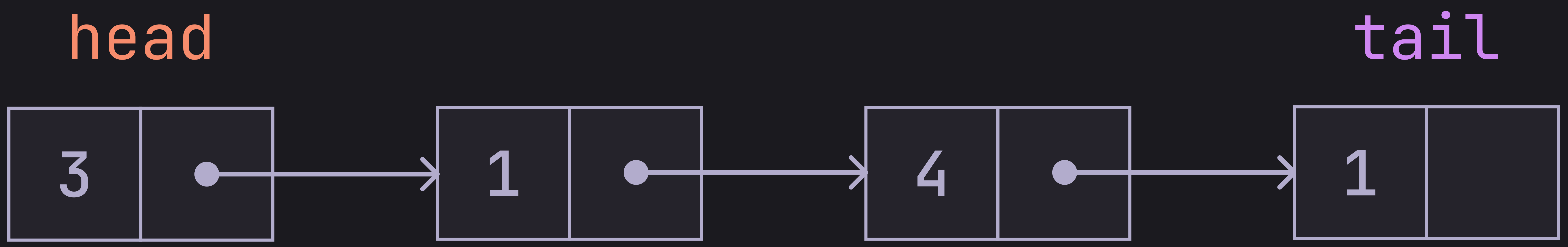
Linked List



push_front(3)

Then we set the next pointer to the current head

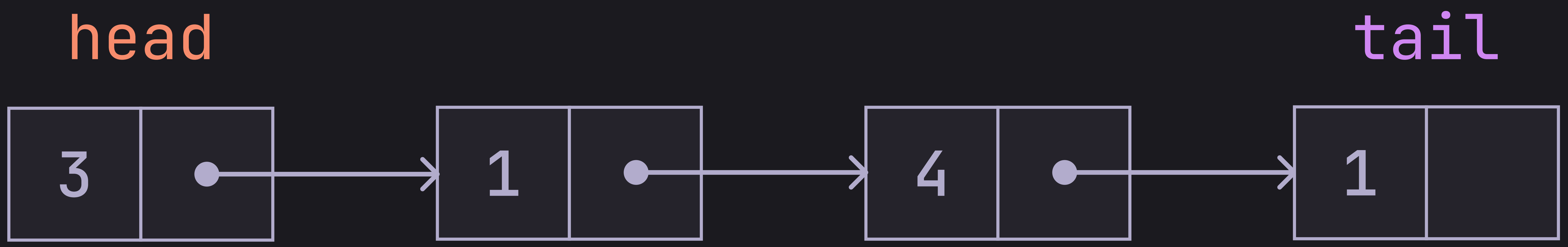
Linked List



`push_front(3)`

Finally we update the head to point to the node we just added

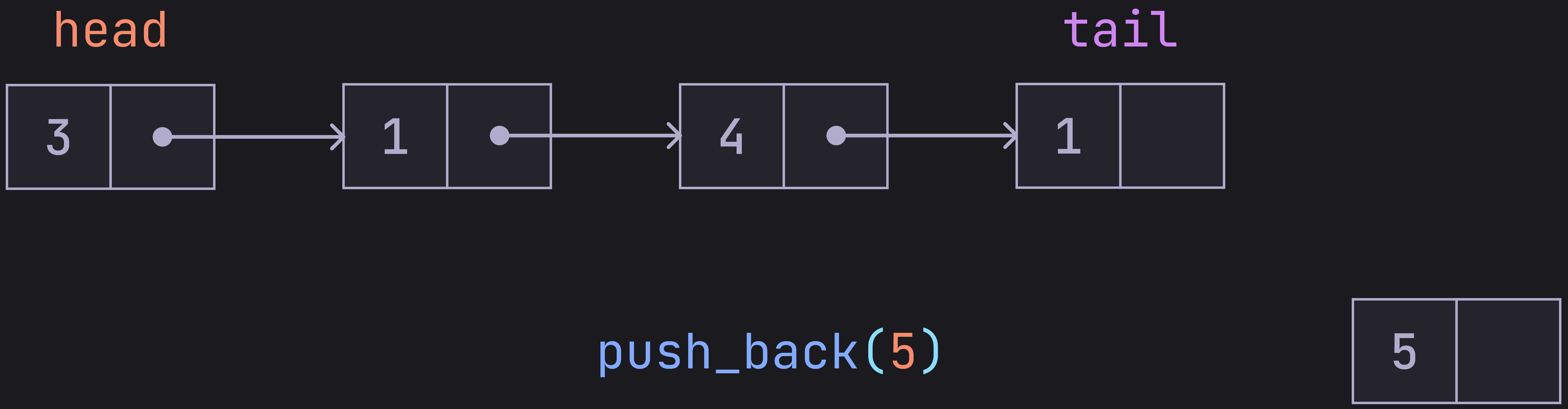
Linked List



push_back(5)

If we are keeping track of the tail Linked lists allow us to quickly push to the back of the list

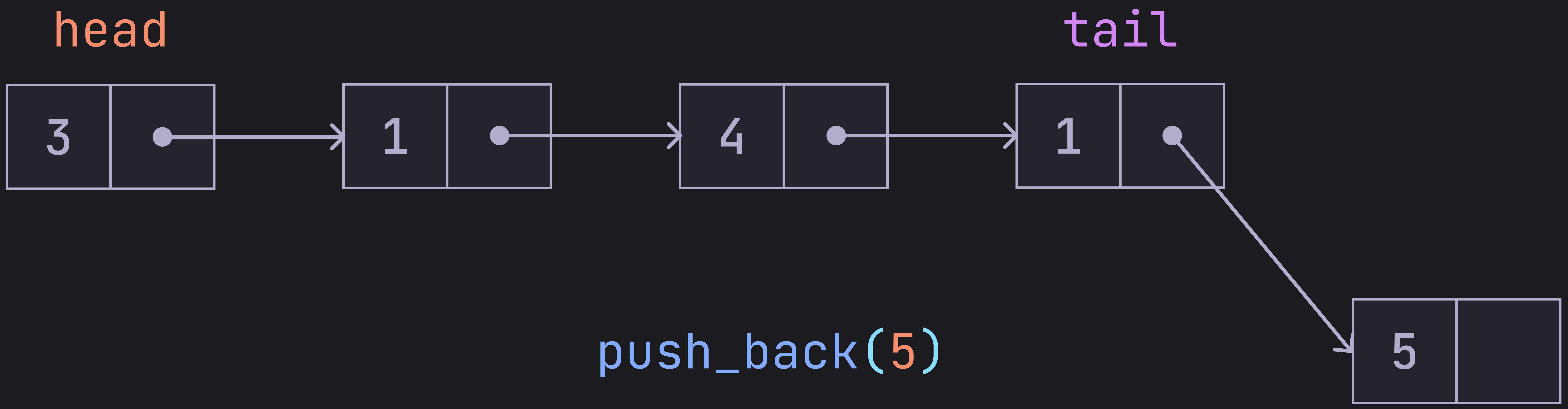
Linked List



push_back(5)

First we create a new node

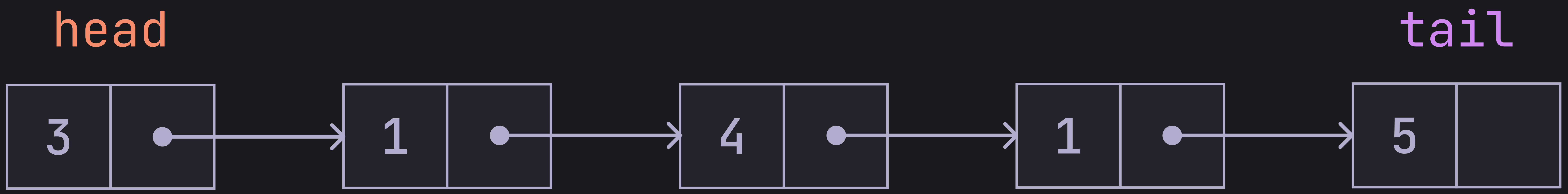
Linked List



push_back(5)

Then we set the next pointer of the tail to the new node we added

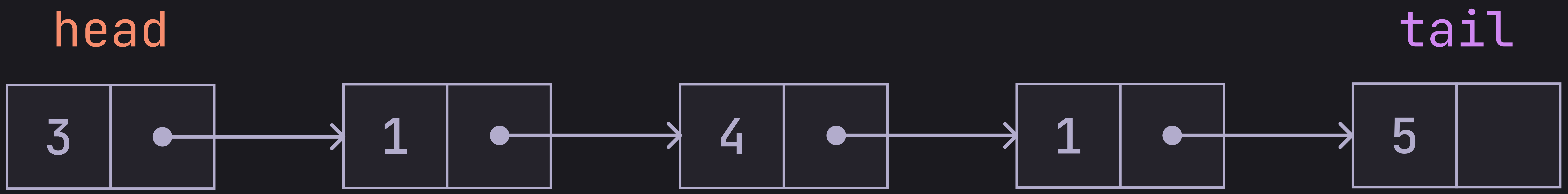
Linked List



push_back(5)

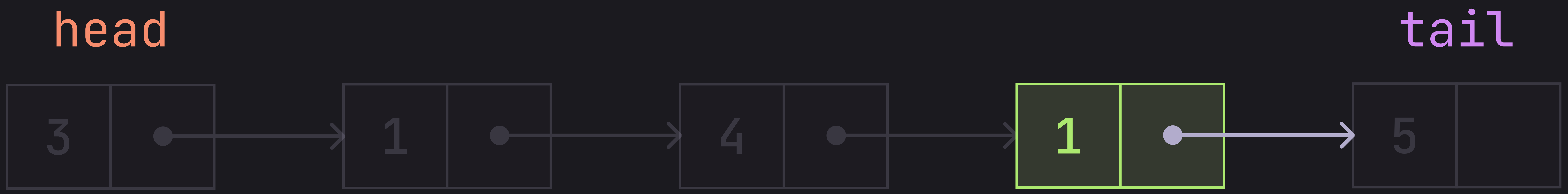
Finally we update the tail to point to the node we just added

Linked List



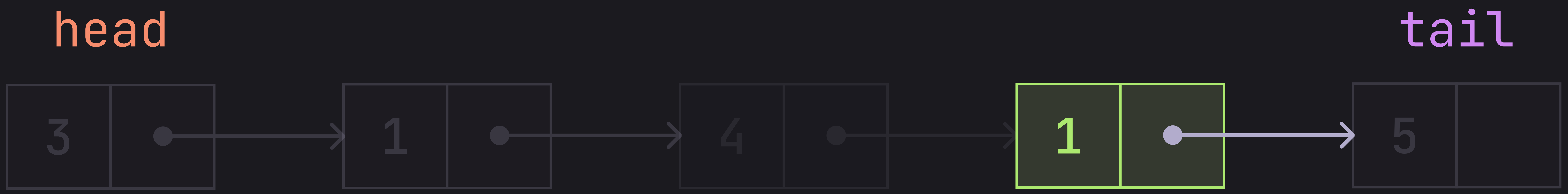
This brings us to a pretty big downside of linked lists which is you cannot randomly access elements

Linked List



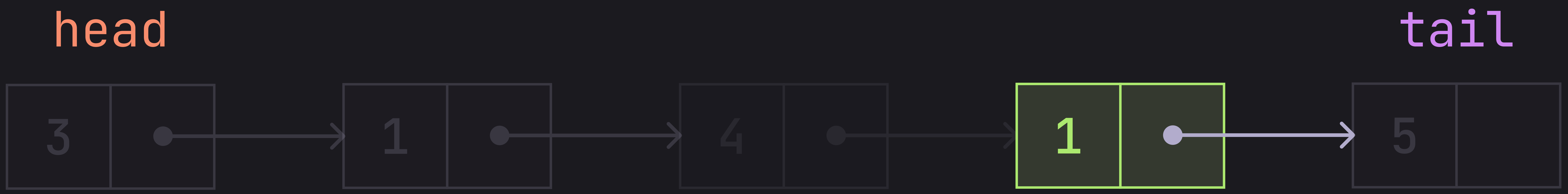
Suppose I want to access the data at the 3rd index

Linked List



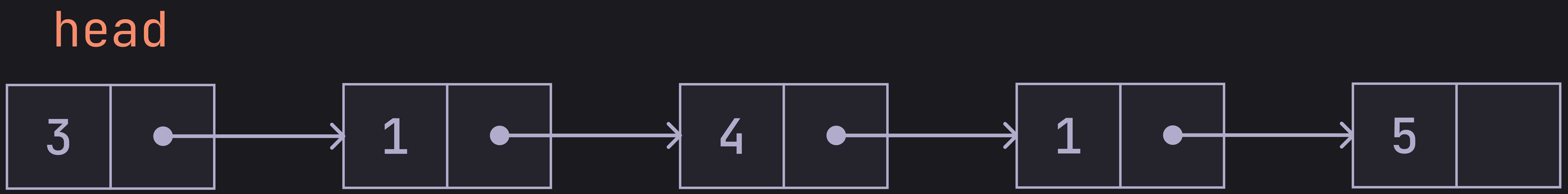
I need to start all the way at the head and keep following the next pointer

Linked List



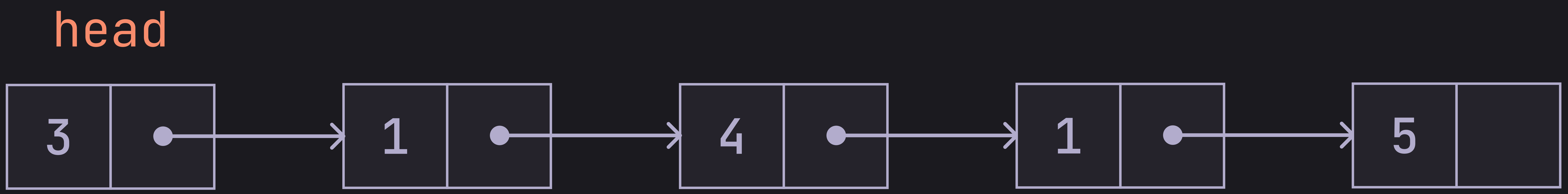
O(n)

Linked List



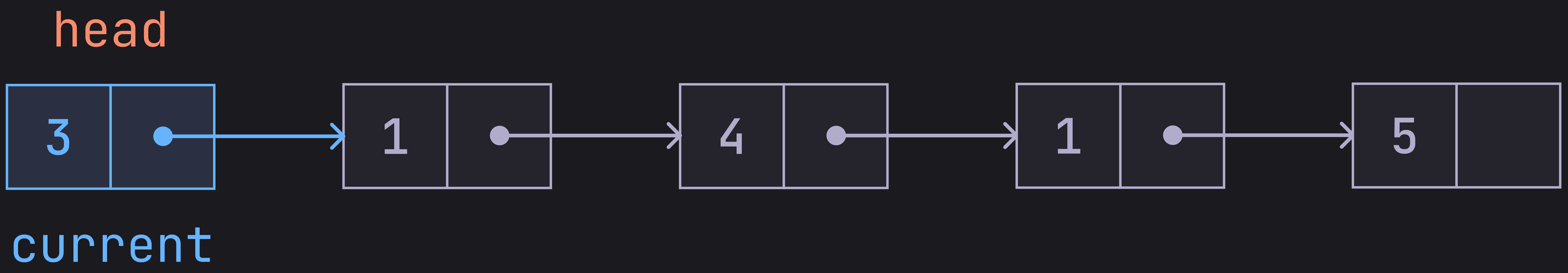
So if we do not keep track of the tail
then we need to first traverse the whole list
before we can add to the end of the list

Linked List



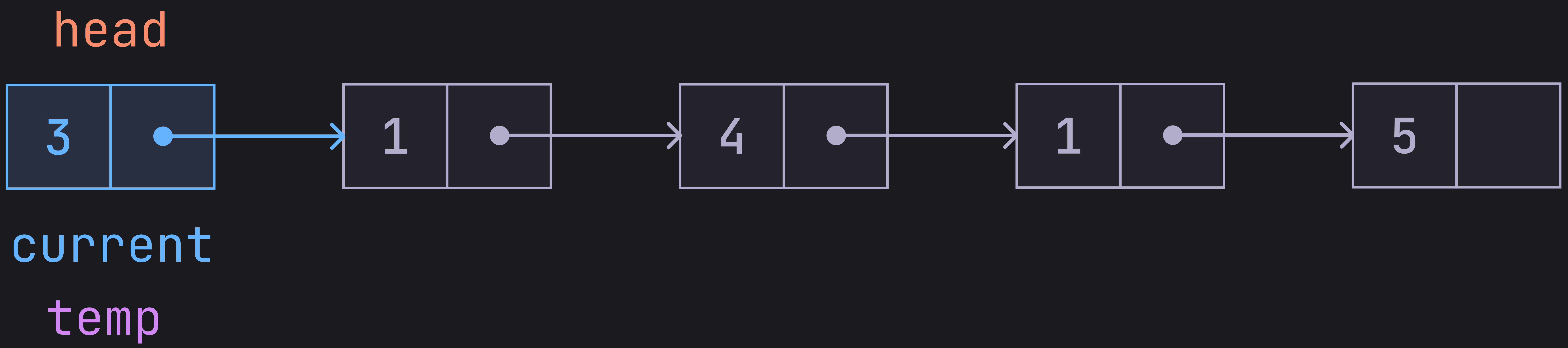
So how do we destruct a linked list?

Linked List



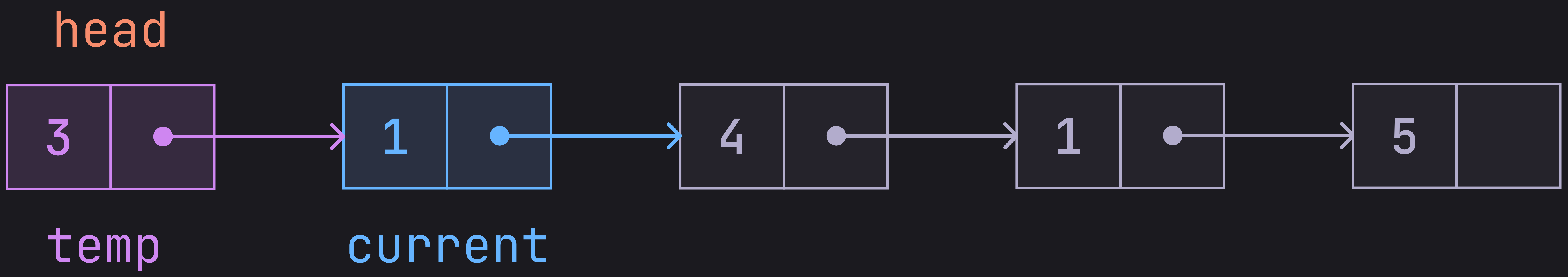
We start from the front and delete each node

Linked List



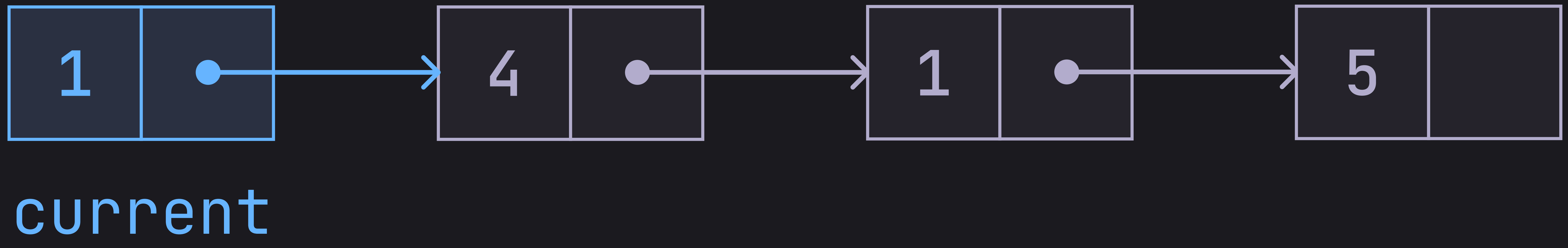
But we need to keep track of the next node before we delete the current node

Linked List



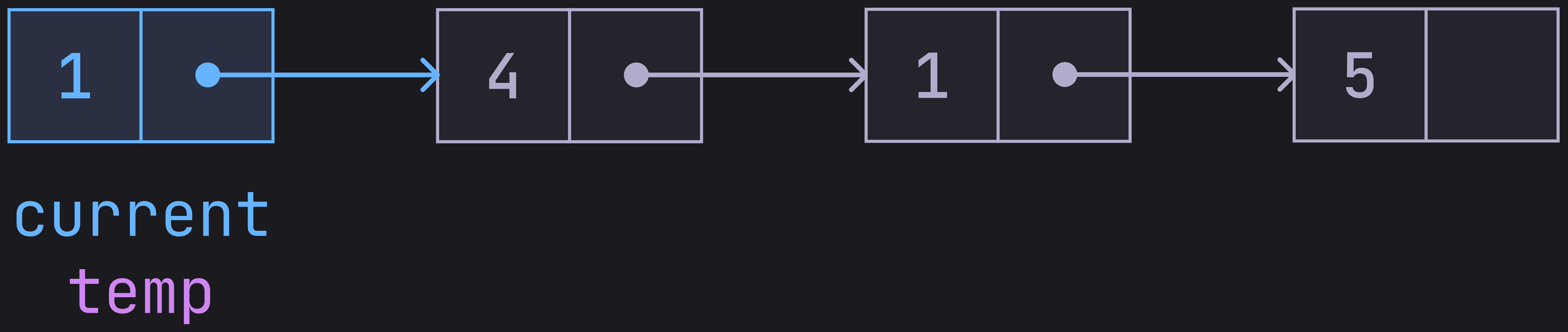
But we need to keep track of the next node before we delete the current node

Linked List



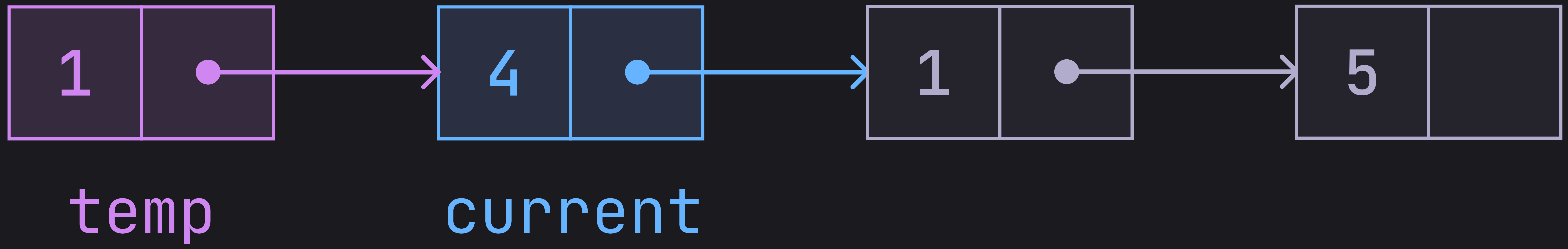
But we need to keep track of the next node before we delete the current node

Linked List



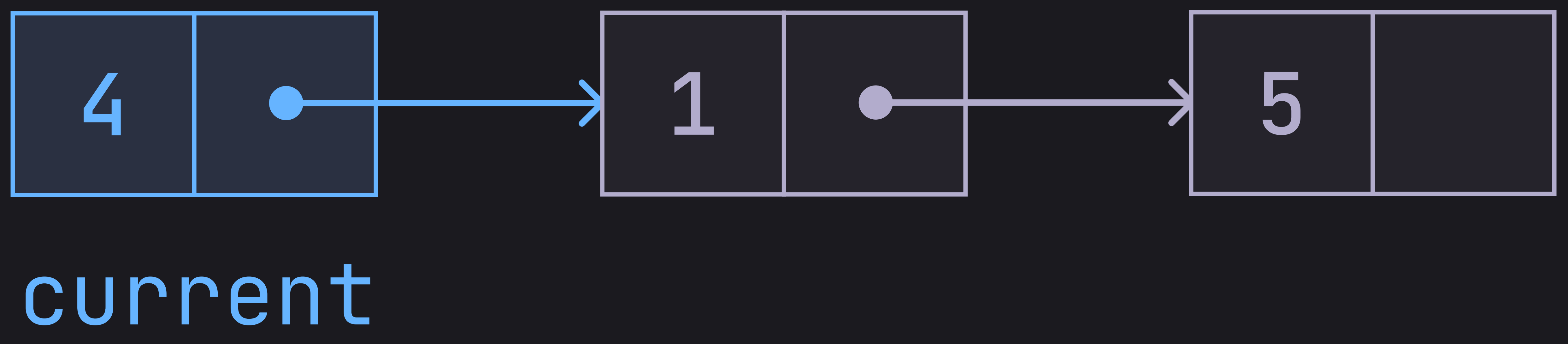
But we need to keep track of the next node before we delete the current node

Linked List



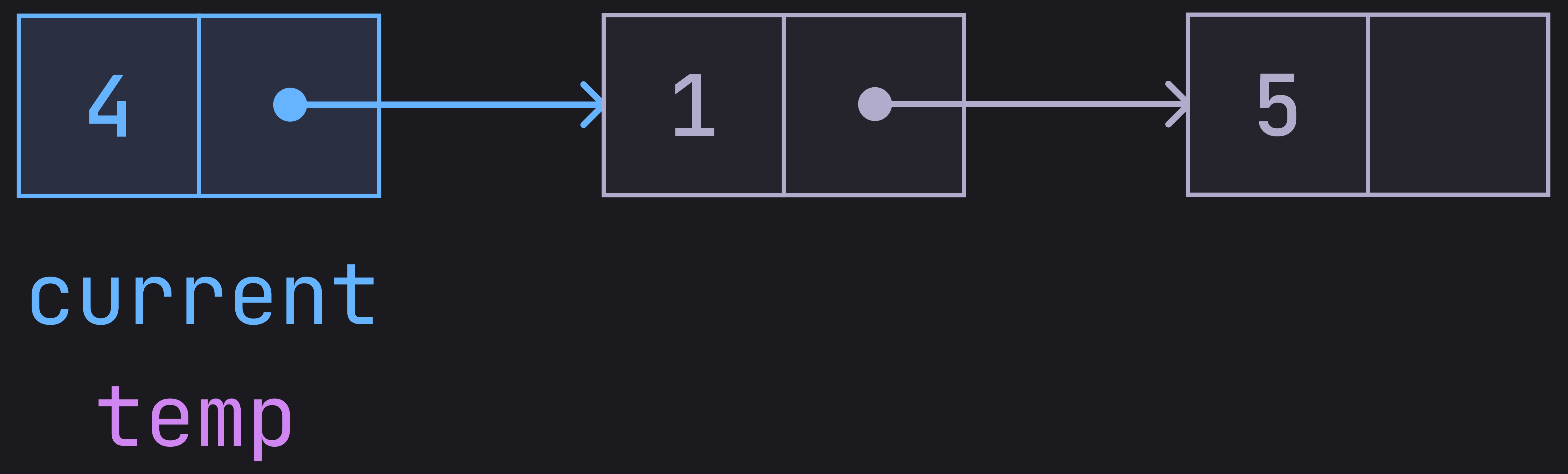
But we need to keep track of the next node before we delete the current node

Linked List



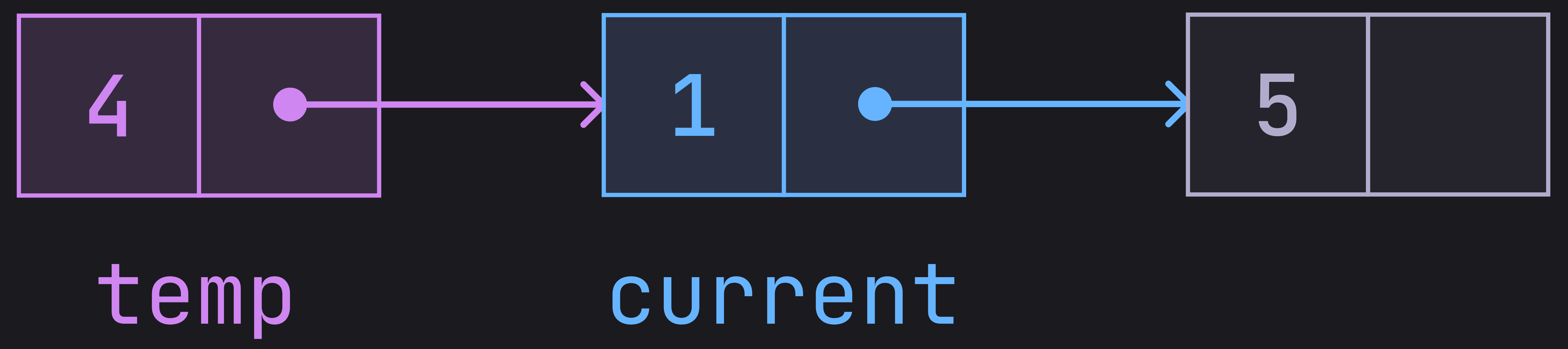
But we need to keep track of the next node before we delete the current node

Linked List



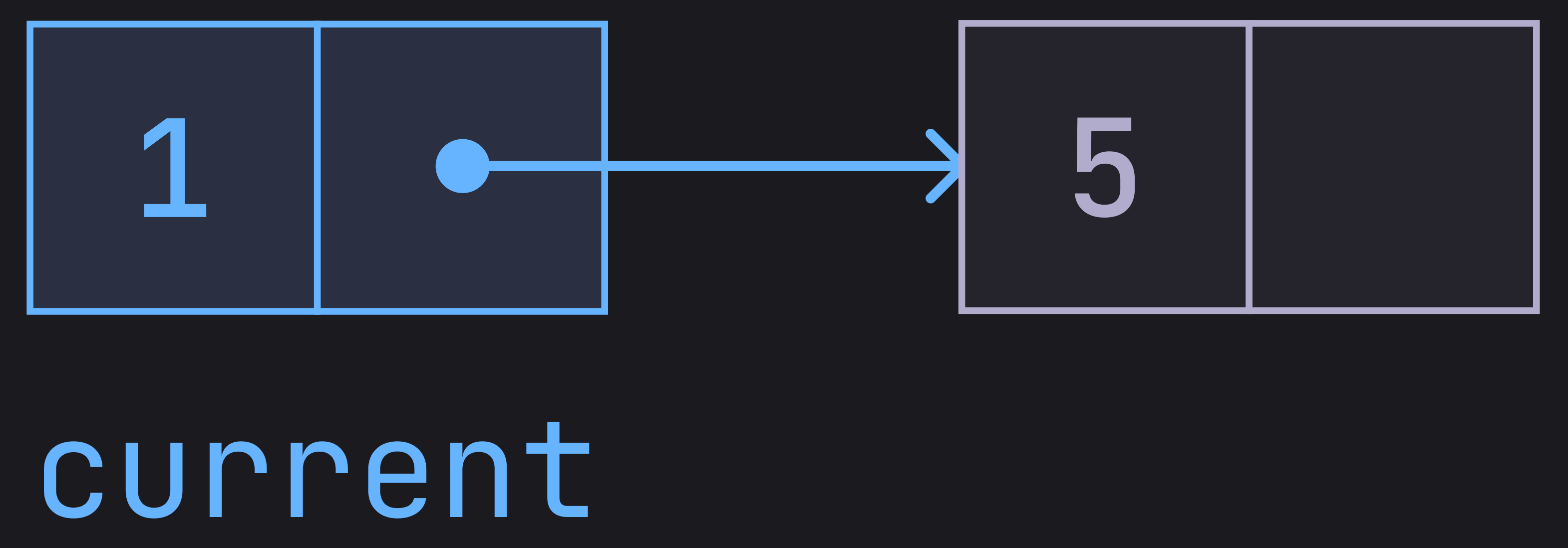
But we need to keep track of the next node before we delete the current node

Linked List



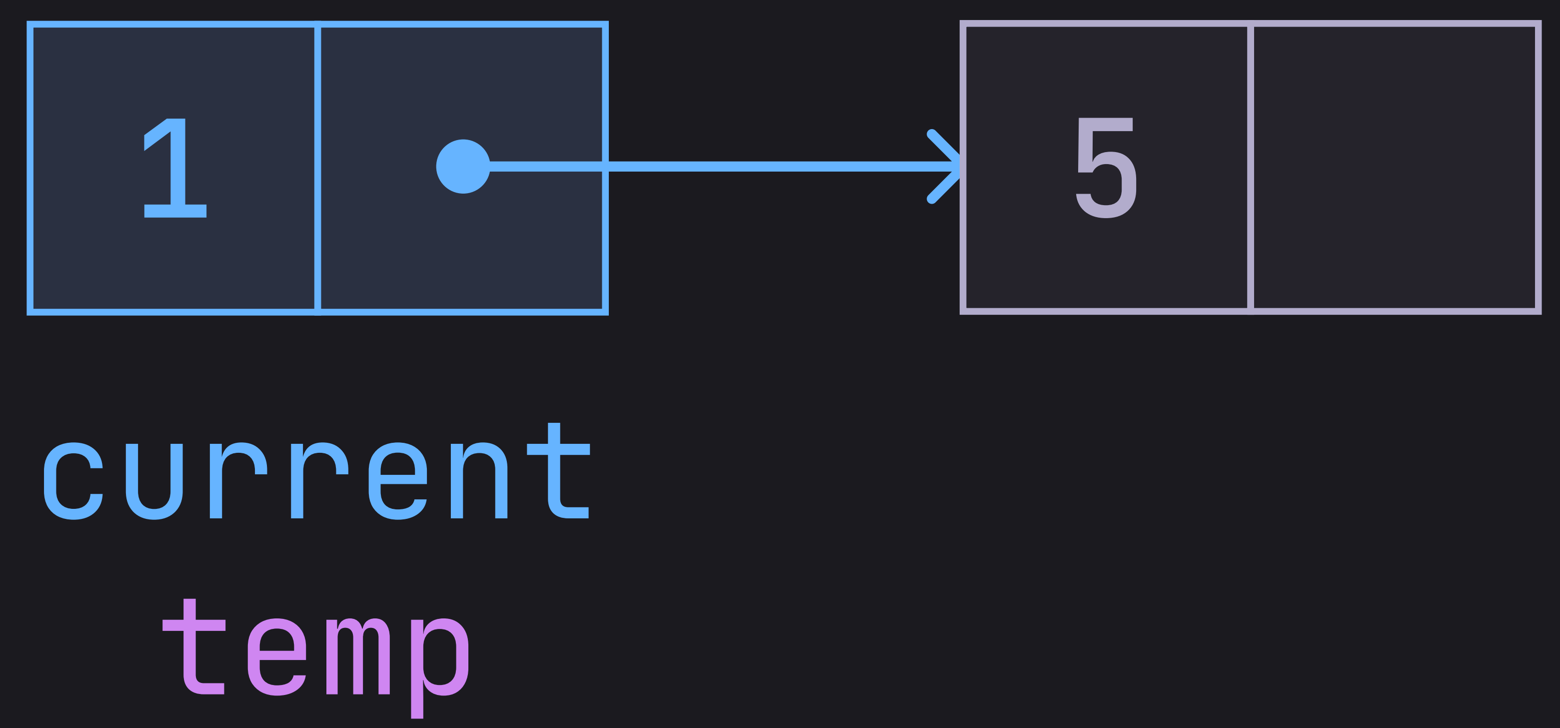
But we need to keep track of the next node before we delete the current node

Linked List



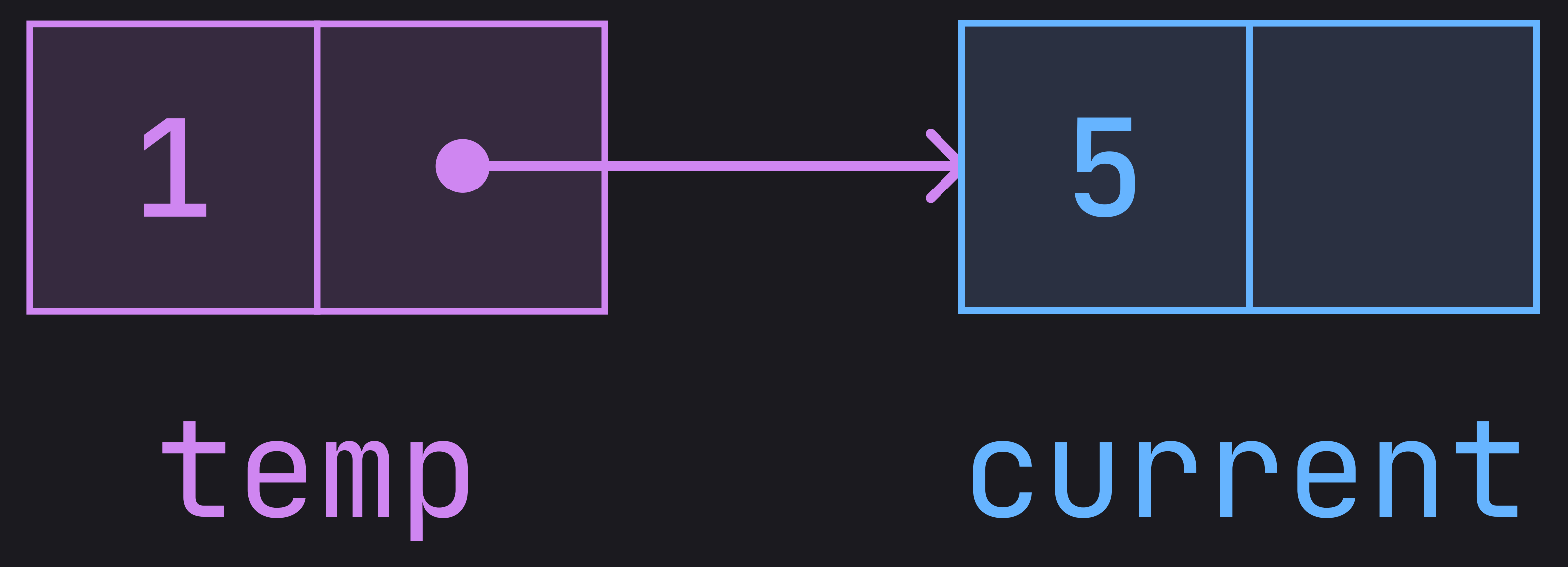
But we need to keep track of the next node before we delete the current node

Linked List



But we need to keep track of the next node before we delete the current node

Linked List



But we need to keep track of the next node before we delete the current node

Linked List

But we need to keep track of the next node before we delete the current node

Linked List

If there is time have a play with the activity

Otherwise thank you all for coming

😊