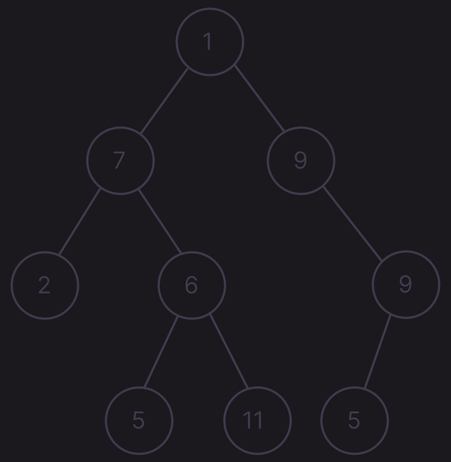
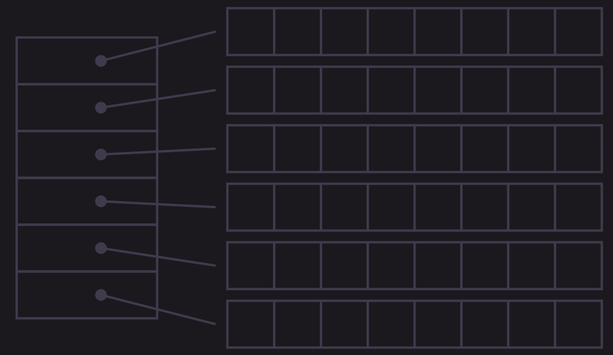
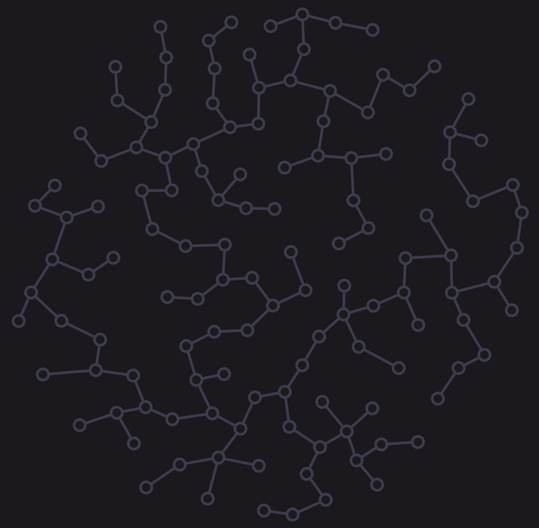
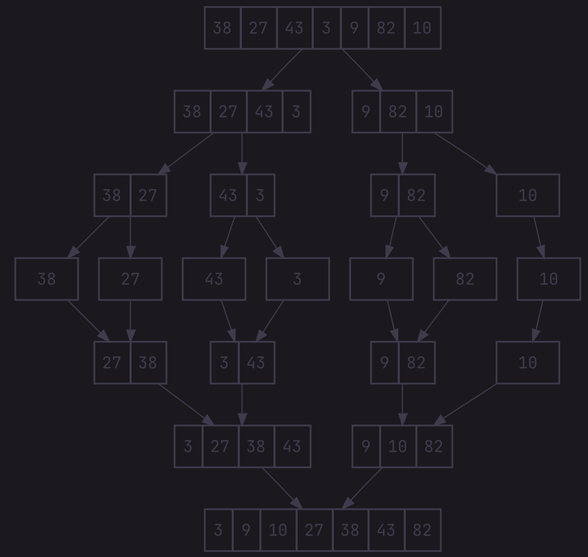




data structures & algorithms

Tutorial 4





Burning questions from last week?

This week’s lab

Today we are learning about a new data structure that will be quite useful for assignment 1 :)

- Review: Constructors
- Implementing iterators
- linked list

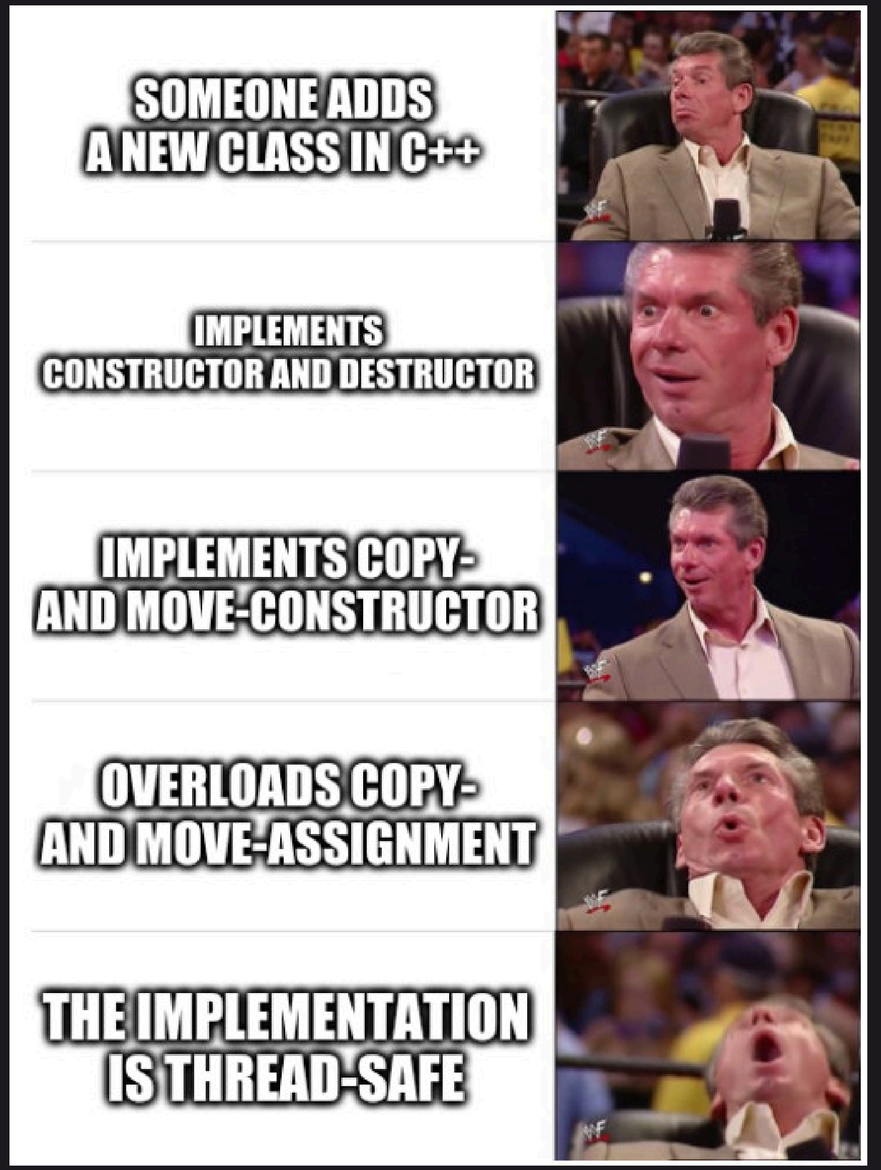
Constructors and Destructors

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

```
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;
    }
}
```

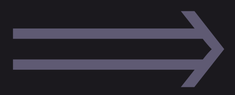
Initialization List

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

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

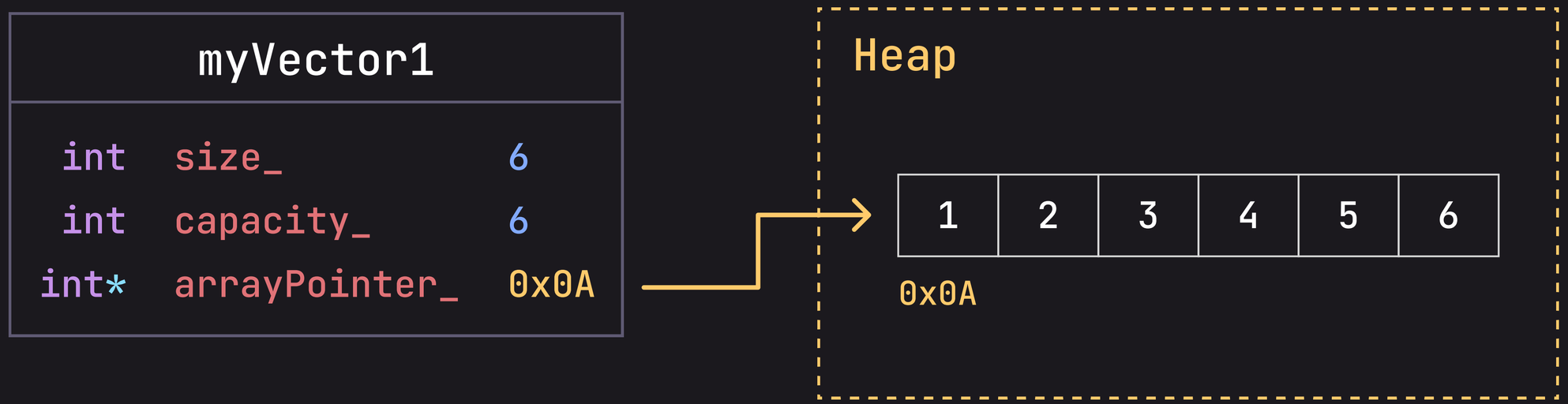
Rule of Three

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



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

Destructor



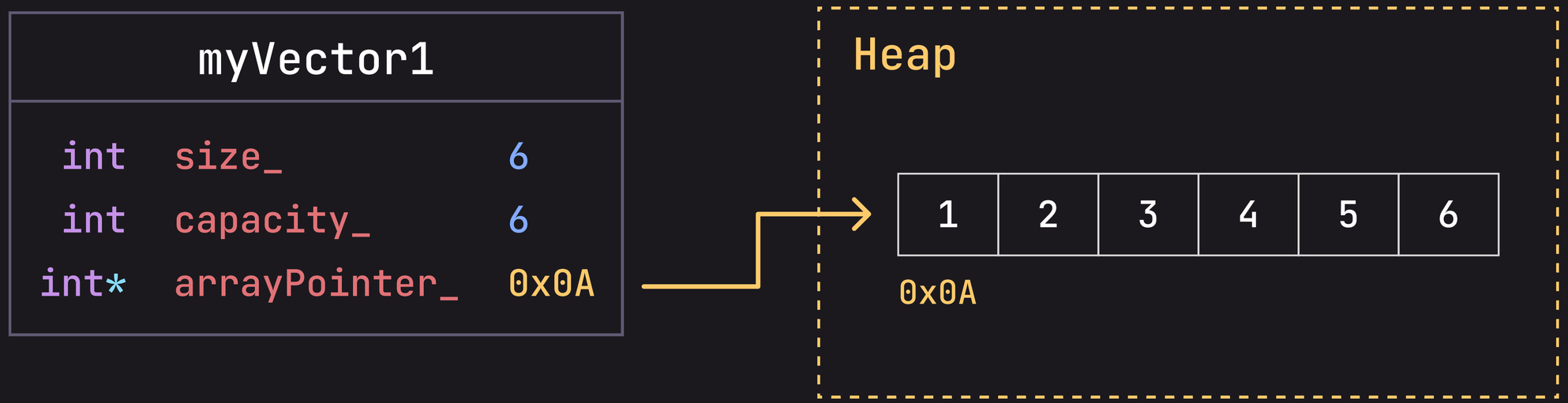
If we use the default destructor on myVector1, the heap memory well remain on the heap. This is called a **memory leak**

Destructor

myVector1		
int	size_	6
int	capacity_	6
int*	arrayPointer_	0x0A

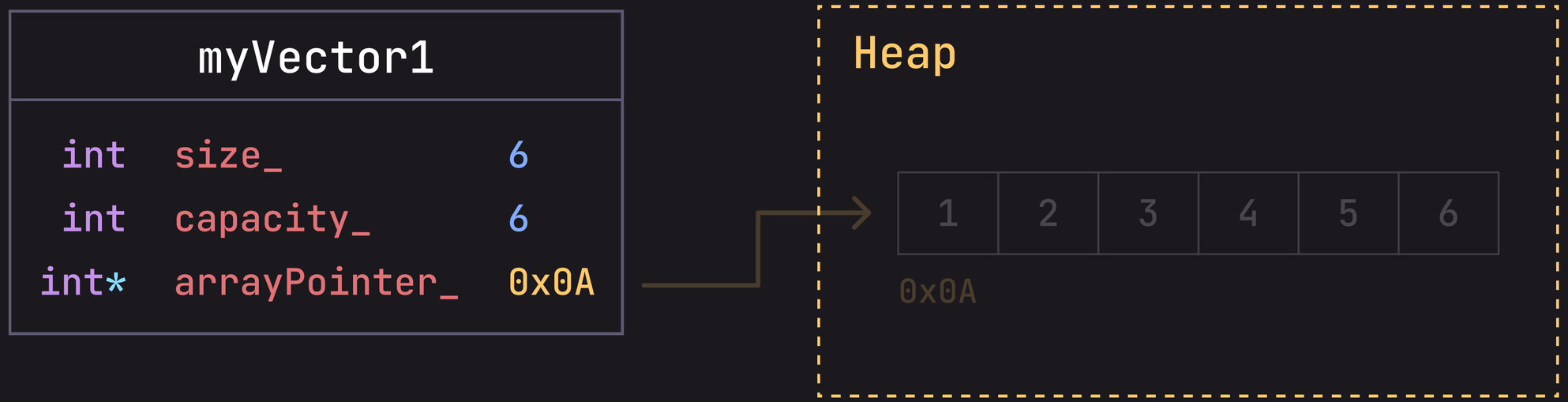
We have now lost all references to the heap memory object, so we can no longer free it.

Destructor



Instead we first need to call delete on the the array

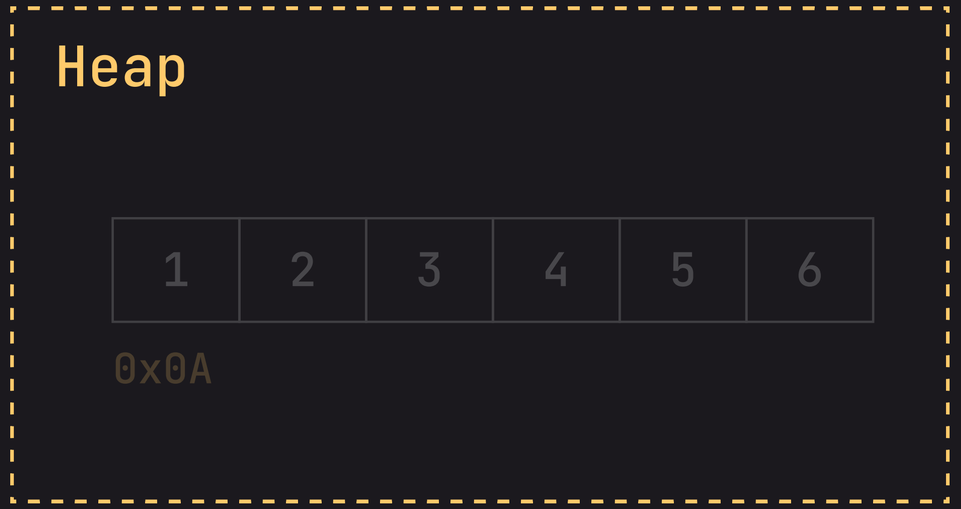
Destructor



And then we can destruct the myVector1 object

Destructor

myVector1		
int	size_	6
int	capacity_	6
int*	arrayPointer_	0x0A



And then we can destruct the myVector1 object

Destructor

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

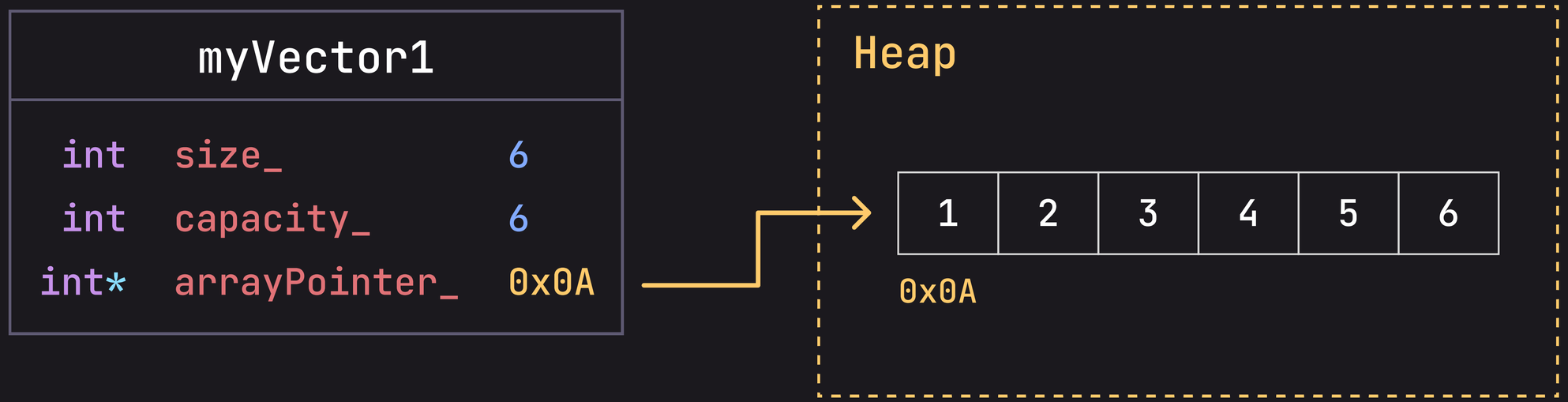
We simply need to override the default destructor and in it we delete the heap allocated array

Destructor (Example)

```cpp
int whoIsMissing(const std::vector<int>& vec) {
  std::vector<bool> present(vec.size() + 1, false); // ⟵ this is a temporary vector

  for (int x : vec) {
    present[x] = true;
  }

  for (std::size_t i = 0; i < present.size(); ++i) {
    if (!present[i]) {
      return i;
    }
  }

  return -1;
}
```

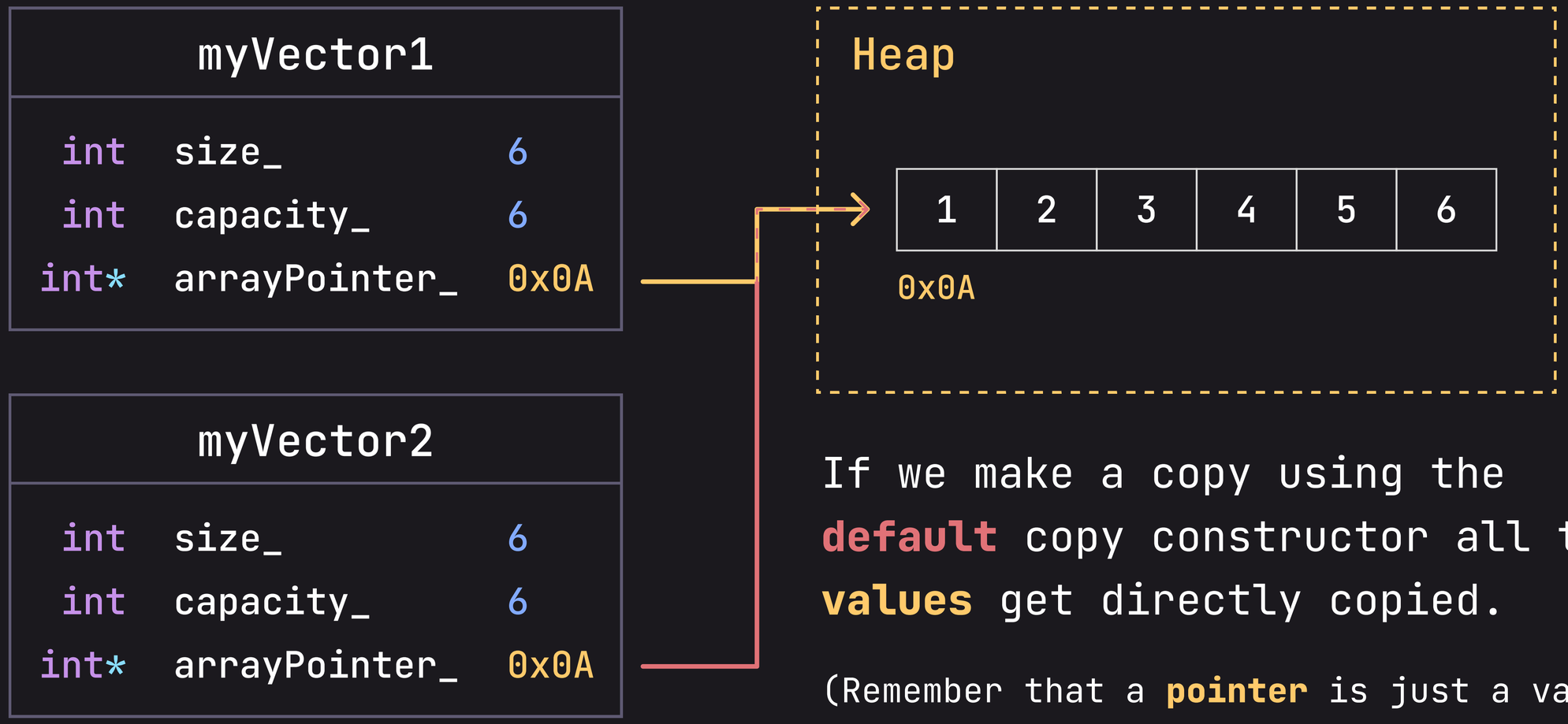
When the function returns, the vector goes out of scope and c++ will then call its destructor

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

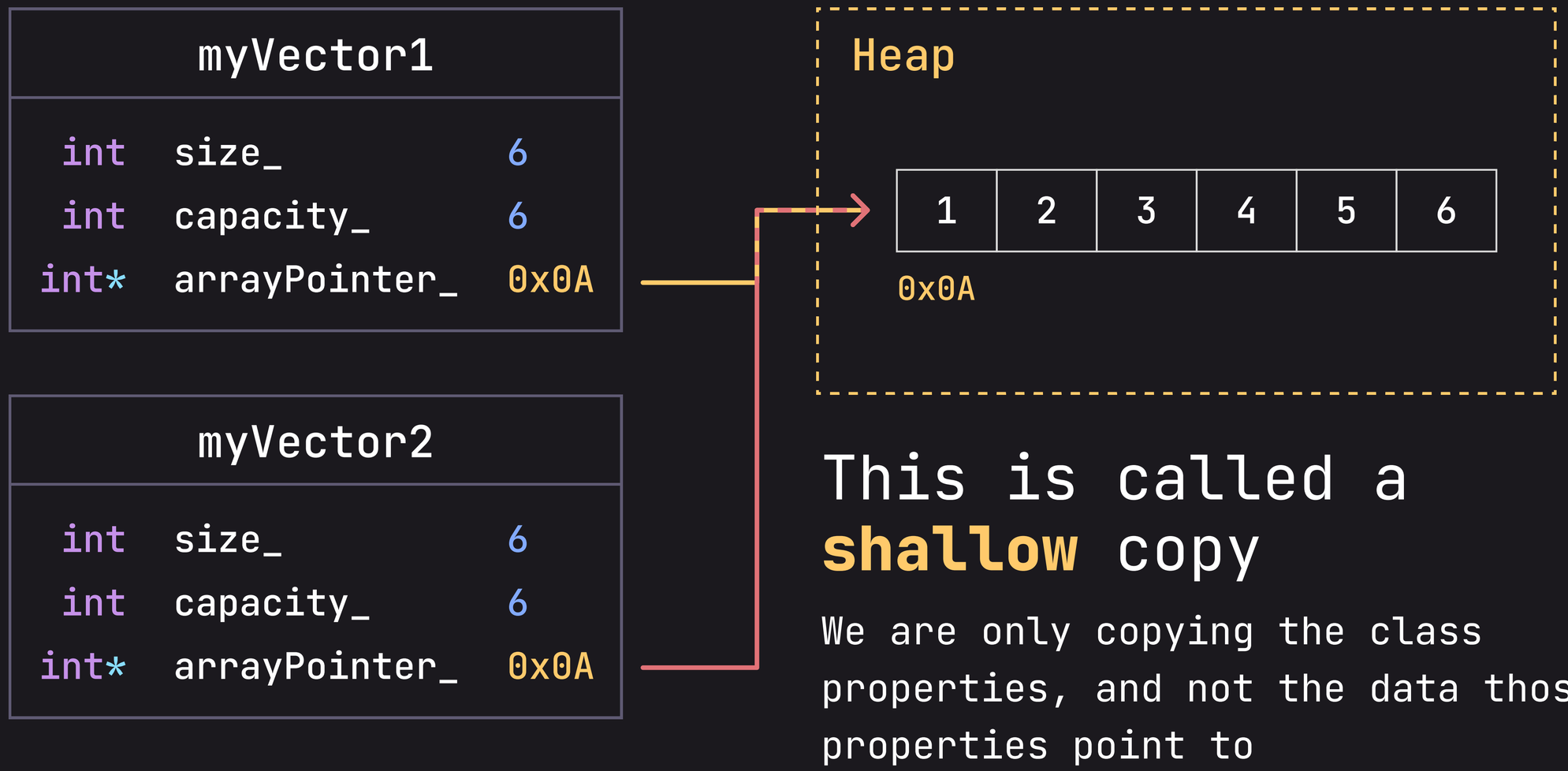
Copy Constructor



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

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

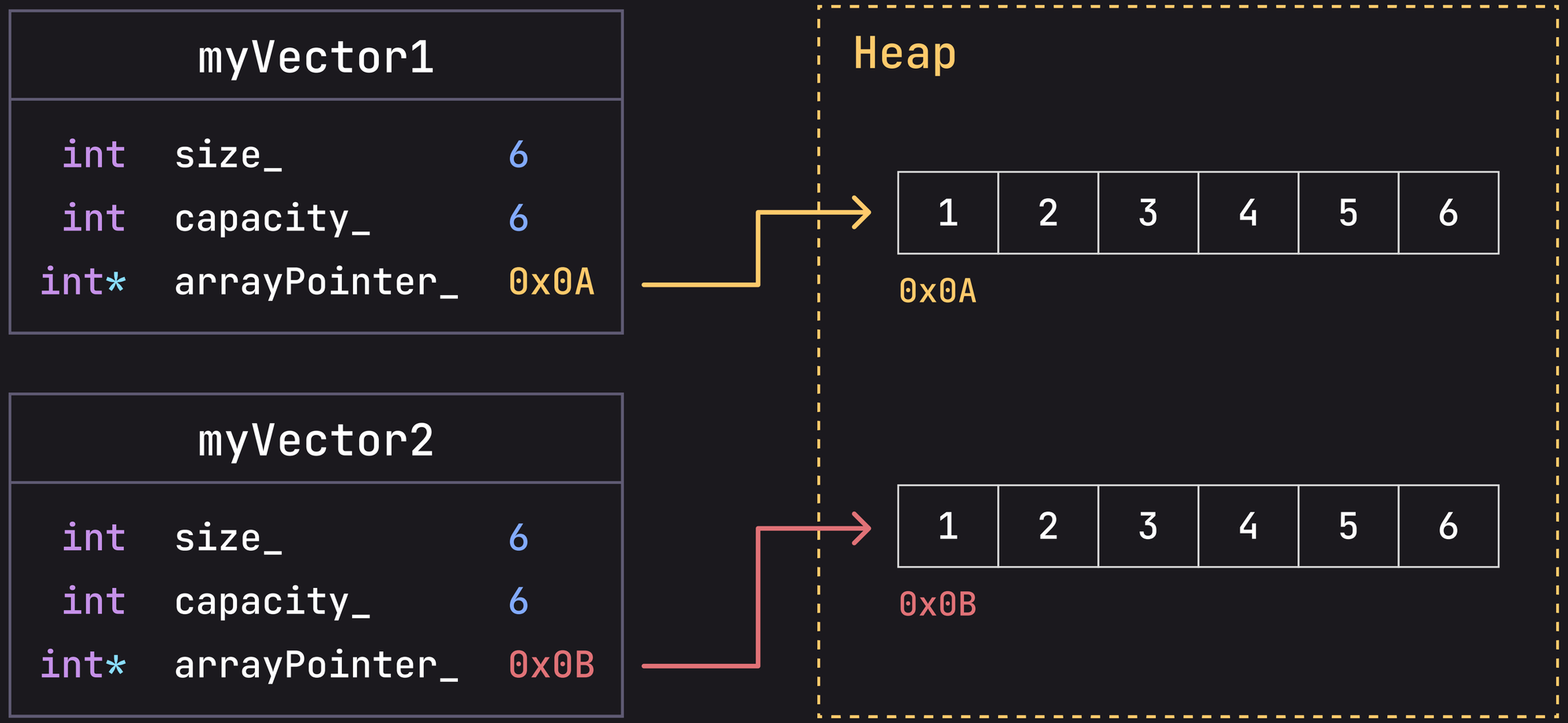
Copy Constructor

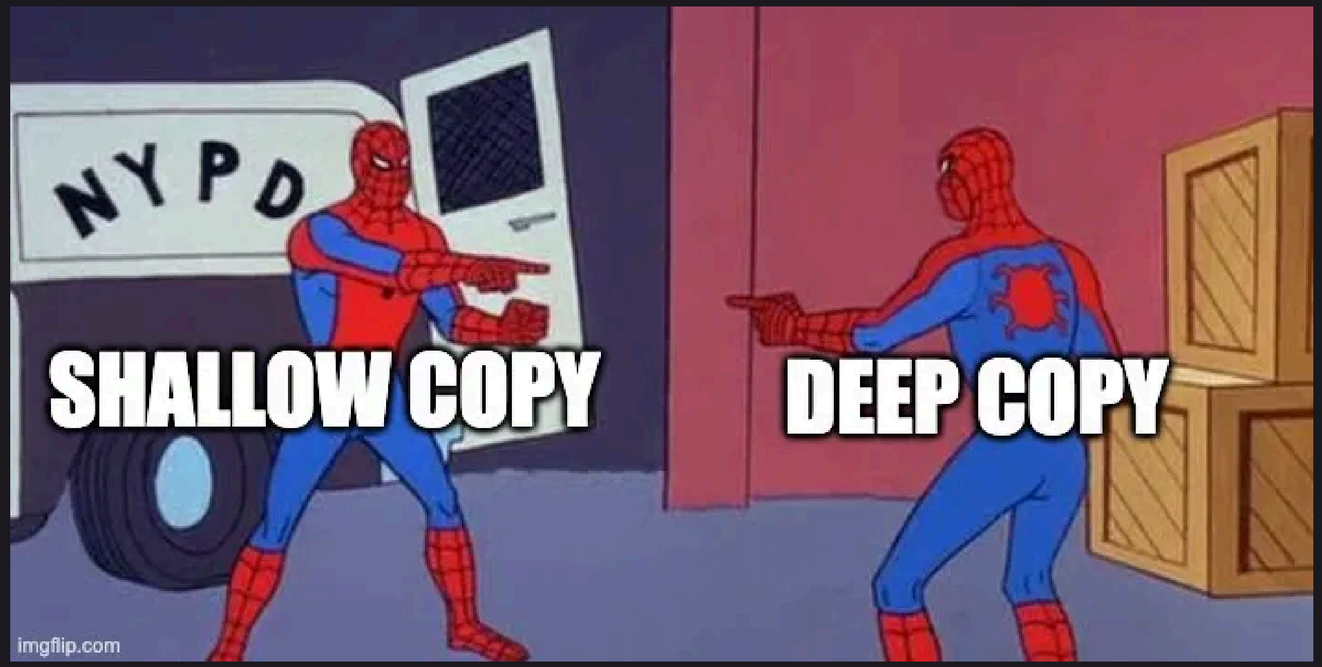


This is called a **shallow** copy

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)

```
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 Constructor (improved)

```
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];
  }
}
```

Safely handles the case where the other vector is empty

When is the Copy Constructor called?

```
void passByValueExample(MyVector vec) {
  // Does the MyVector copy constructor get called?
}

void passByRefernceExample(MyVector & vec) {
  // Does the MyVector copy constructor get called?
}

void passByPointerExample(MyVector * vec) {
  // Does the MyVector copy constructor get called?
}
```

When is the Copy Constructor called?

```
void passByValueExample(MyVector vec) {
  // Does the MyVector copy constructor get called?
}
```

Yes

```
void passByRefernceExample(MyVector & vec) {
  // Does the MyVector copy constructor get called?
}
```

No

```
void passByPointerExample(MyVector * vec) {
  // Does the MyVector copy constructor get called?
}
```

No

When is the Copy Constructor called?

```
MyVector myVec1();
myVec1.push(1);
myVec1.push(2);
myVec1.push(3);

MyVector myVec2(myVec1) // ⟵ makes a copy
```

We can also explicitly call the copy constructor

Copy Assignment

```
MyVector myVec1();
myVec1.push(1);
myVec1.push(2);
myVec1.push(3);

MyVector myVec2 = myVec1 // ⟵ makes a copy
```

It might feel more natural to make a copy by using the assignment operator

Copy Assignment

```
MyVector myVec1();
myVec1.push(1);
myVec1.push(2);
myVec1.push(3);

MyVector myVec2 = myVec1 // ⟵ makes a copy
```

Notice this is very similar to the copy constructor, so if possible we do not want to have to repeat that same code logic

Copy Assignment

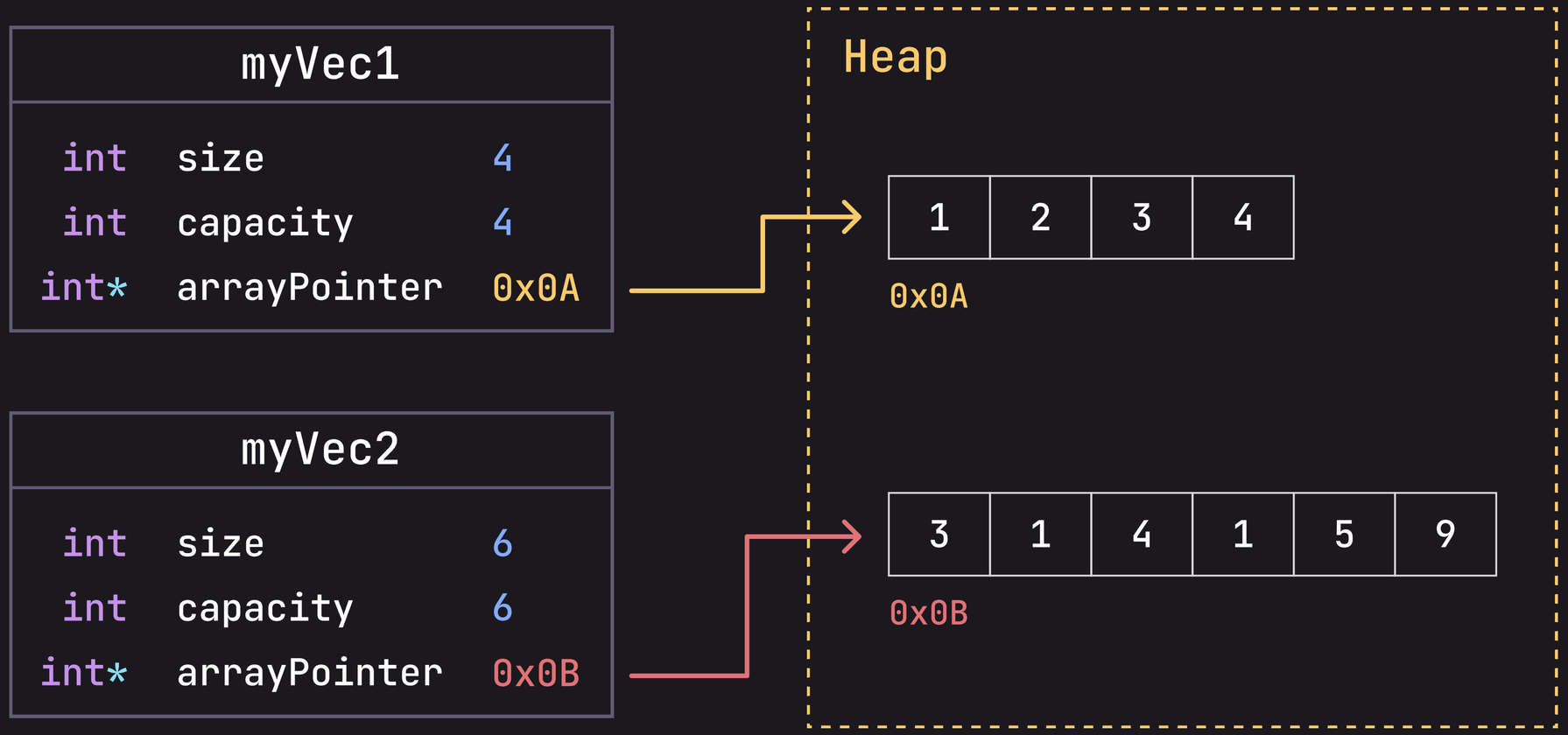
```
MyVector myVec1();

myVec1.push(1);
myVec1.push(2);

MyVector myVec2();
myVec2.push(1);

myVec2 = myVec1 // ⟵ makes a copy and overwrites myVec2
```

But the copy assignment operator does more than the copy constructor because we can also use it to overwrite existing variables

Copy Swap Idiom

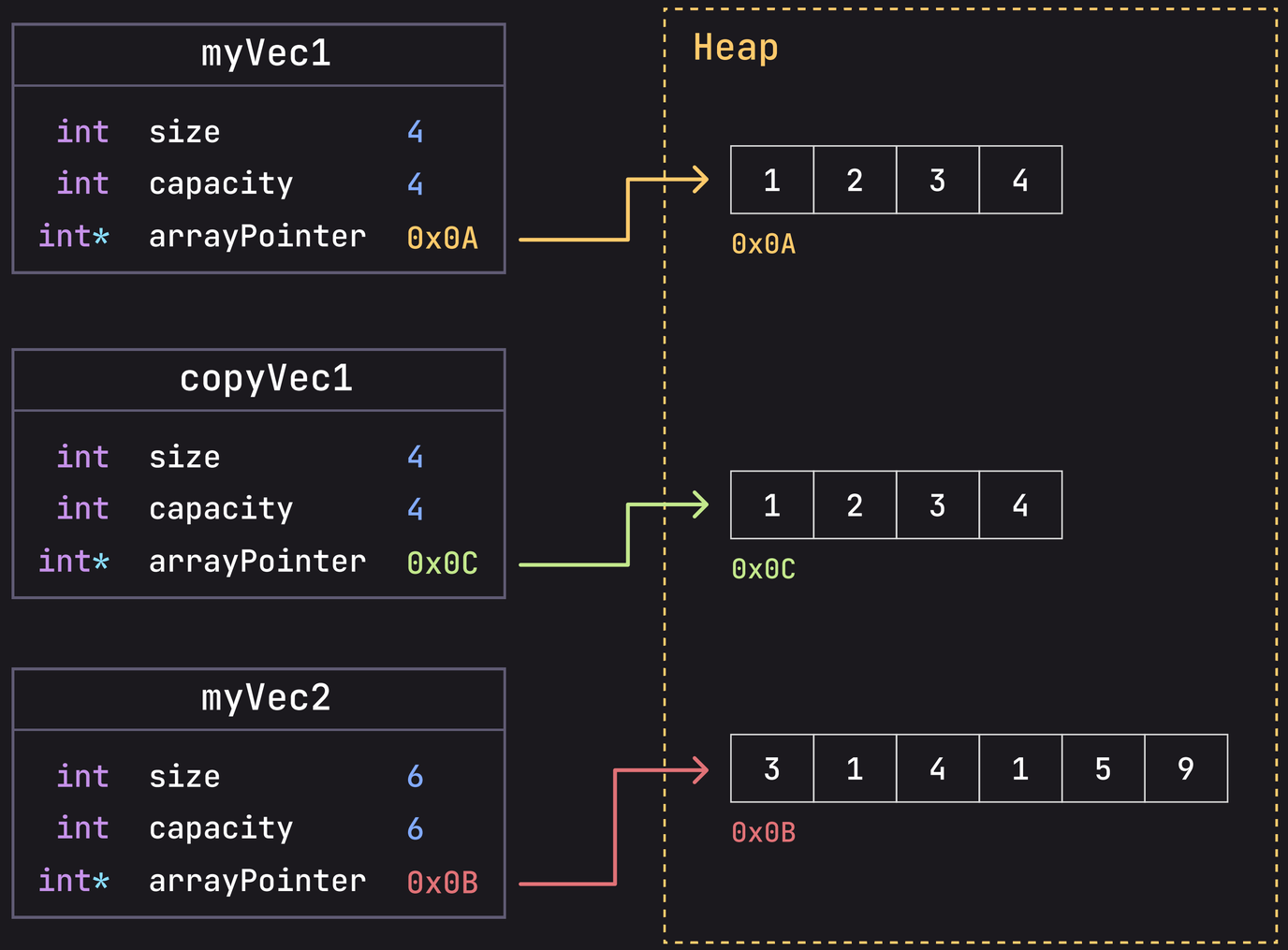
```
MyVector& MyVector::operator=(MyVector other) {
  // other is passed by VALUE
  // So it is copied (its copy constructor is called)
  std::swap(arrayPointer_, other.arrayPointer_);
  std::swap(size_, other.size_);
  std::swap(capacity_, other.capacity_);
  return *this;
}
```

Copy Swap Idiom

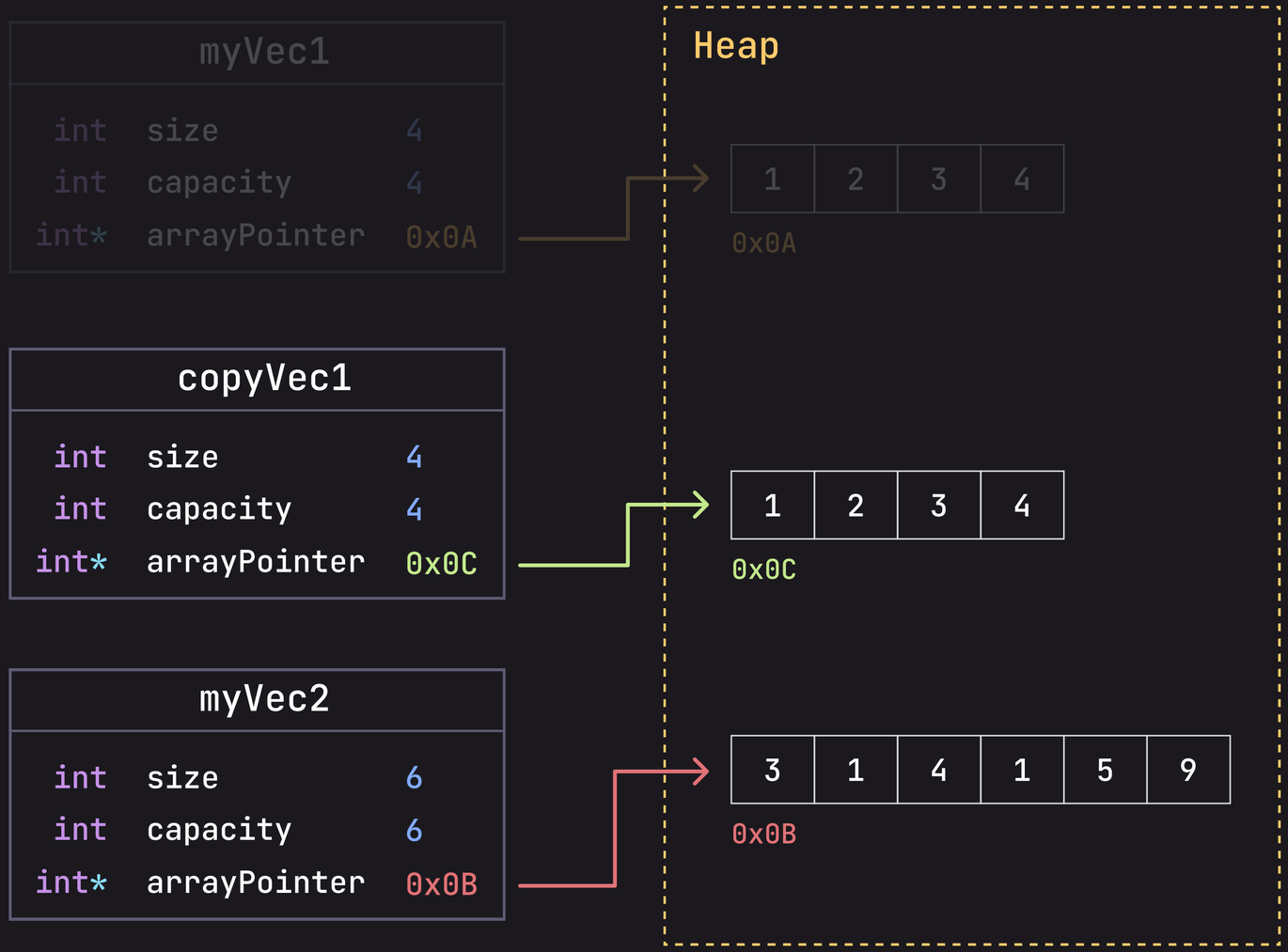



```
myVec2 = myVec1 // ⟵ makes a copy and overwrites myVec2
```

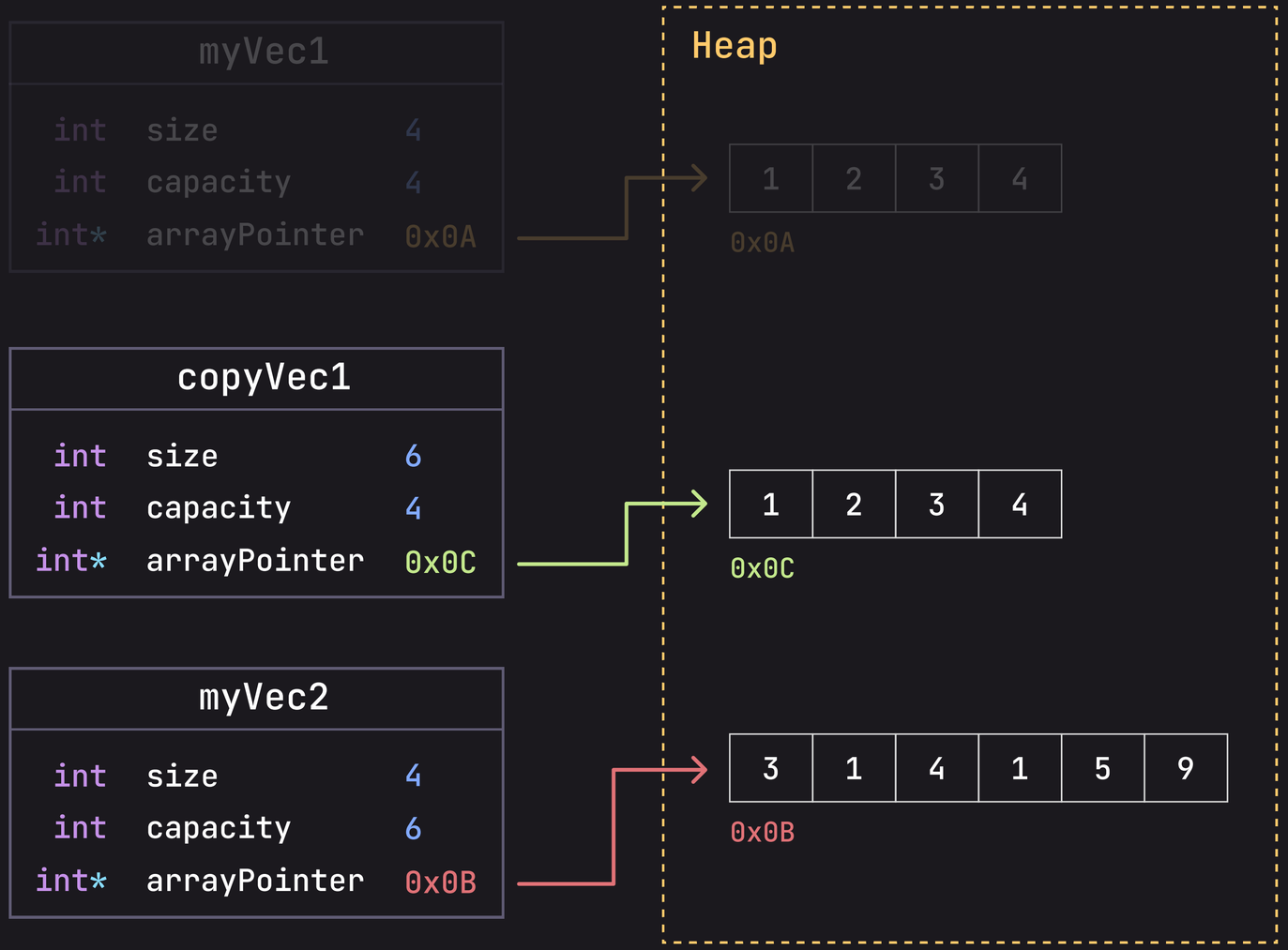
Copy Swap Idiom



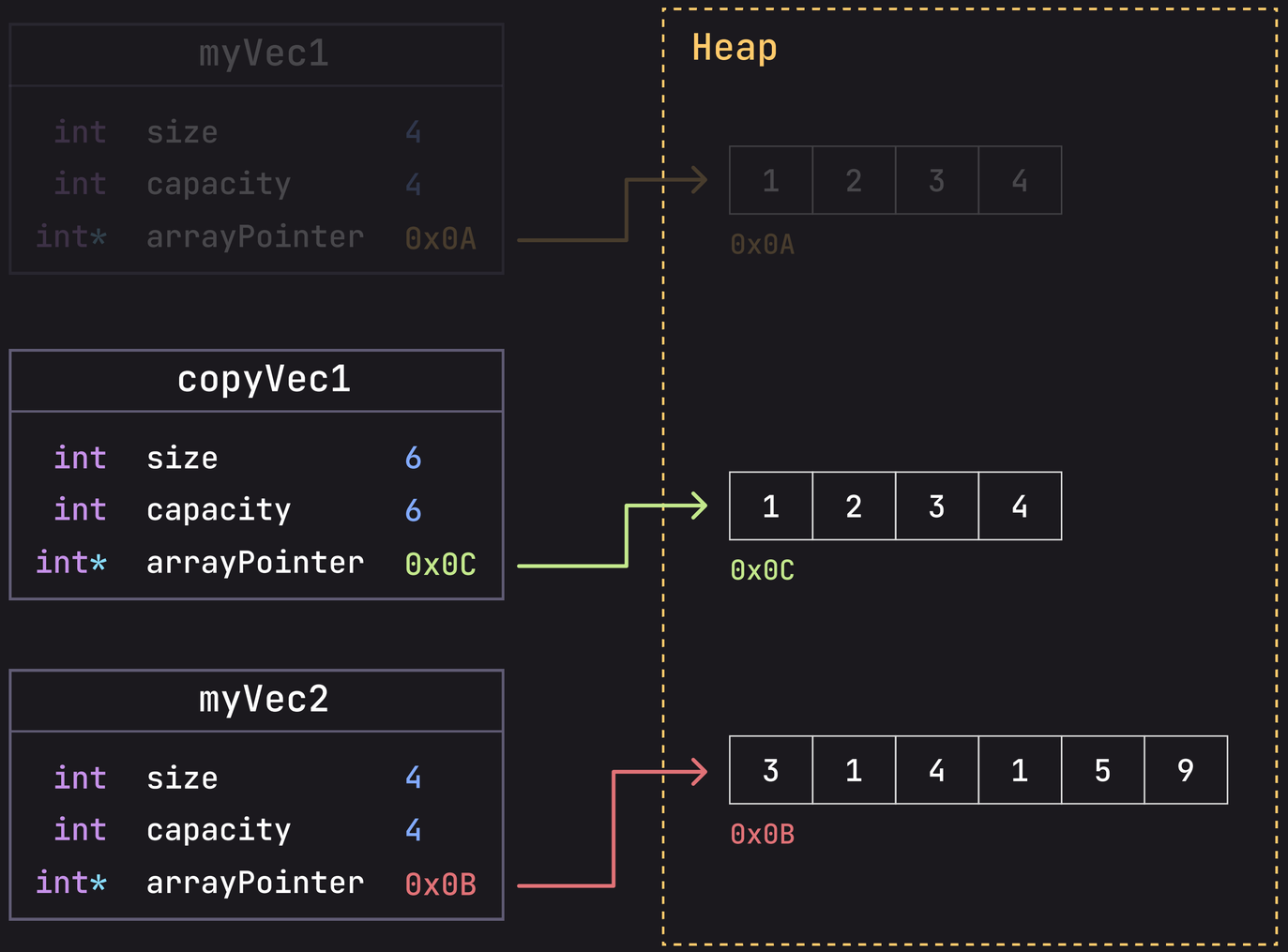
Copy Swap Idiom



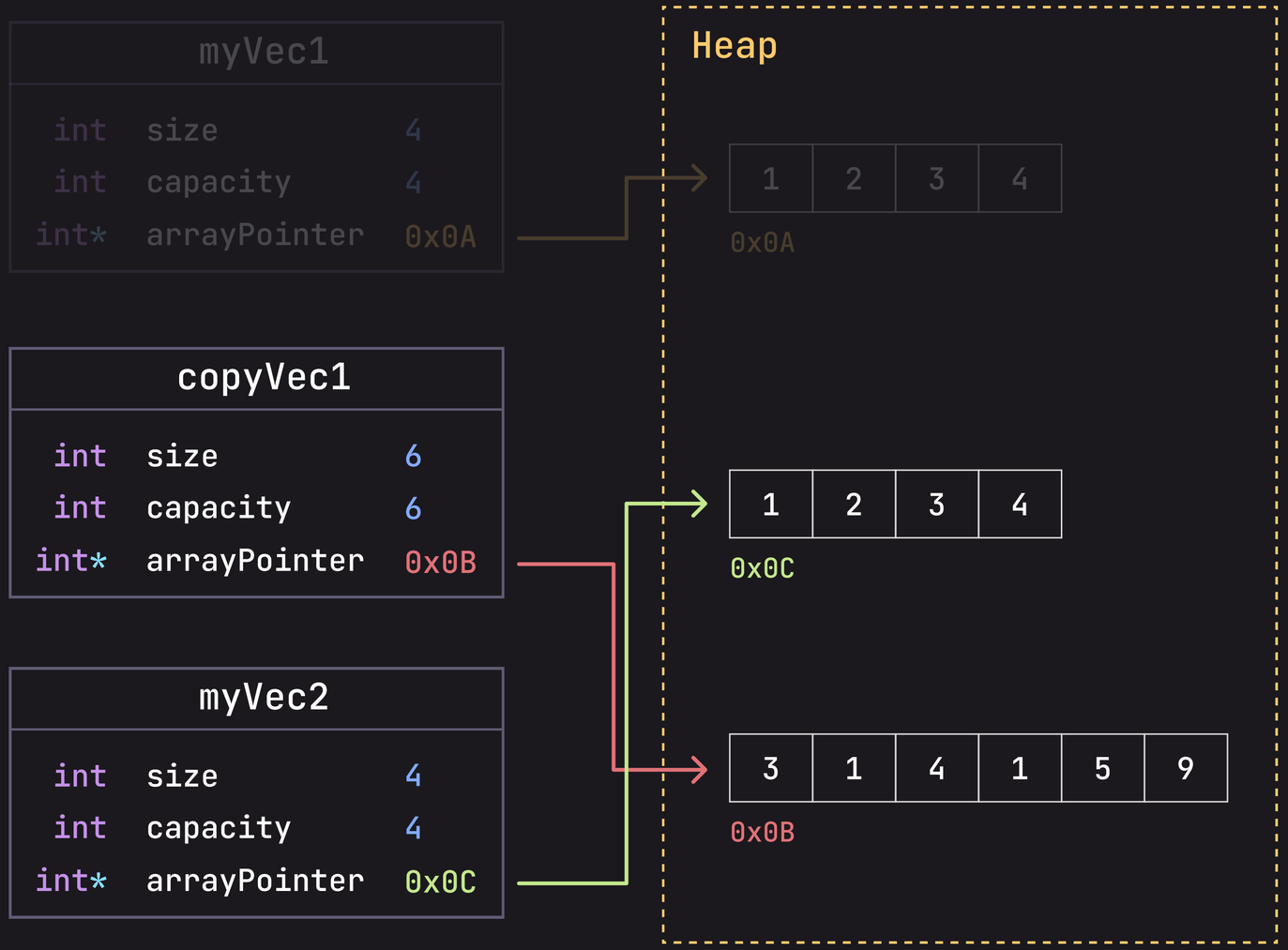
Copy Swap Idiom



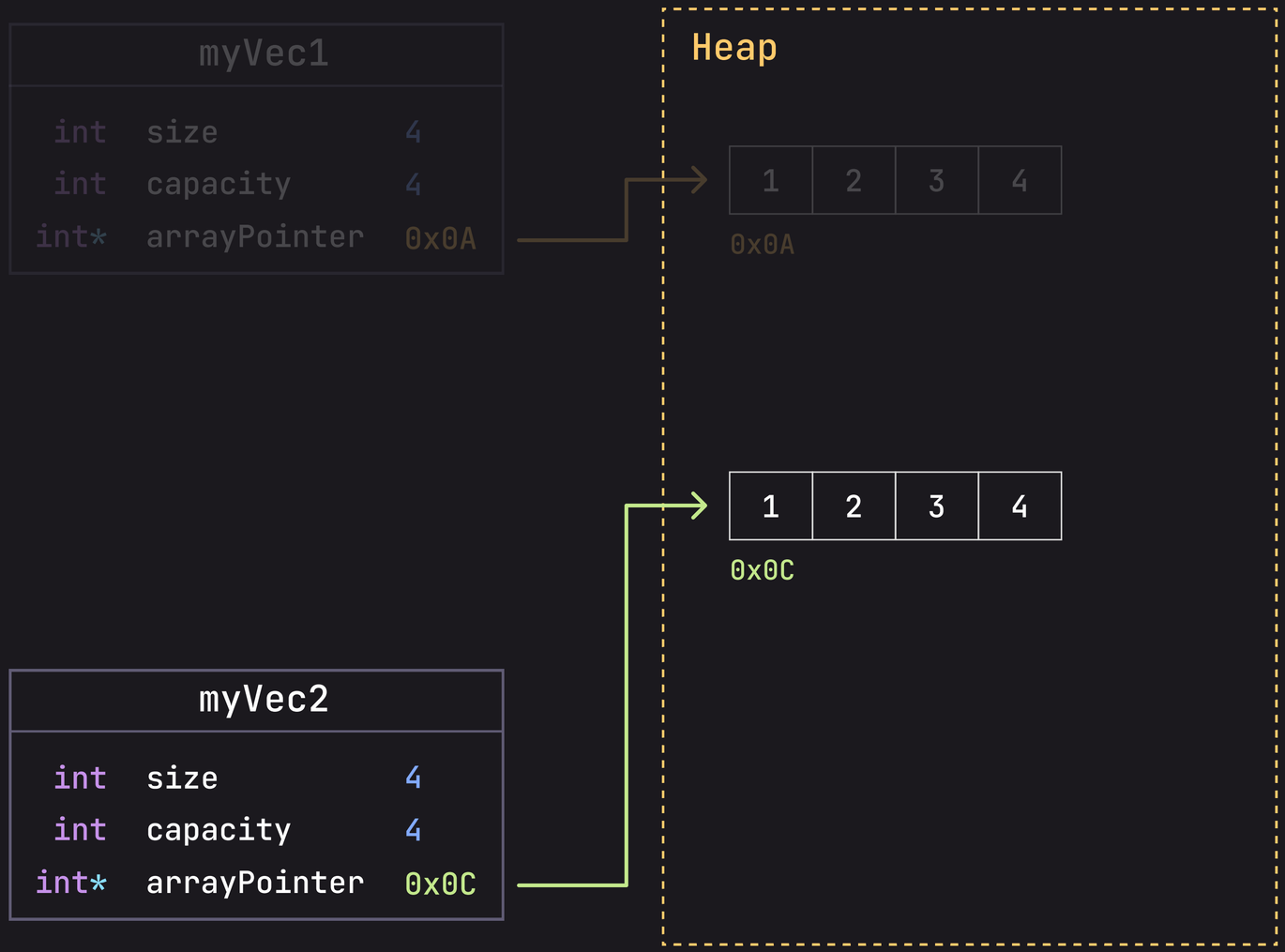
Copy Swap Idiom



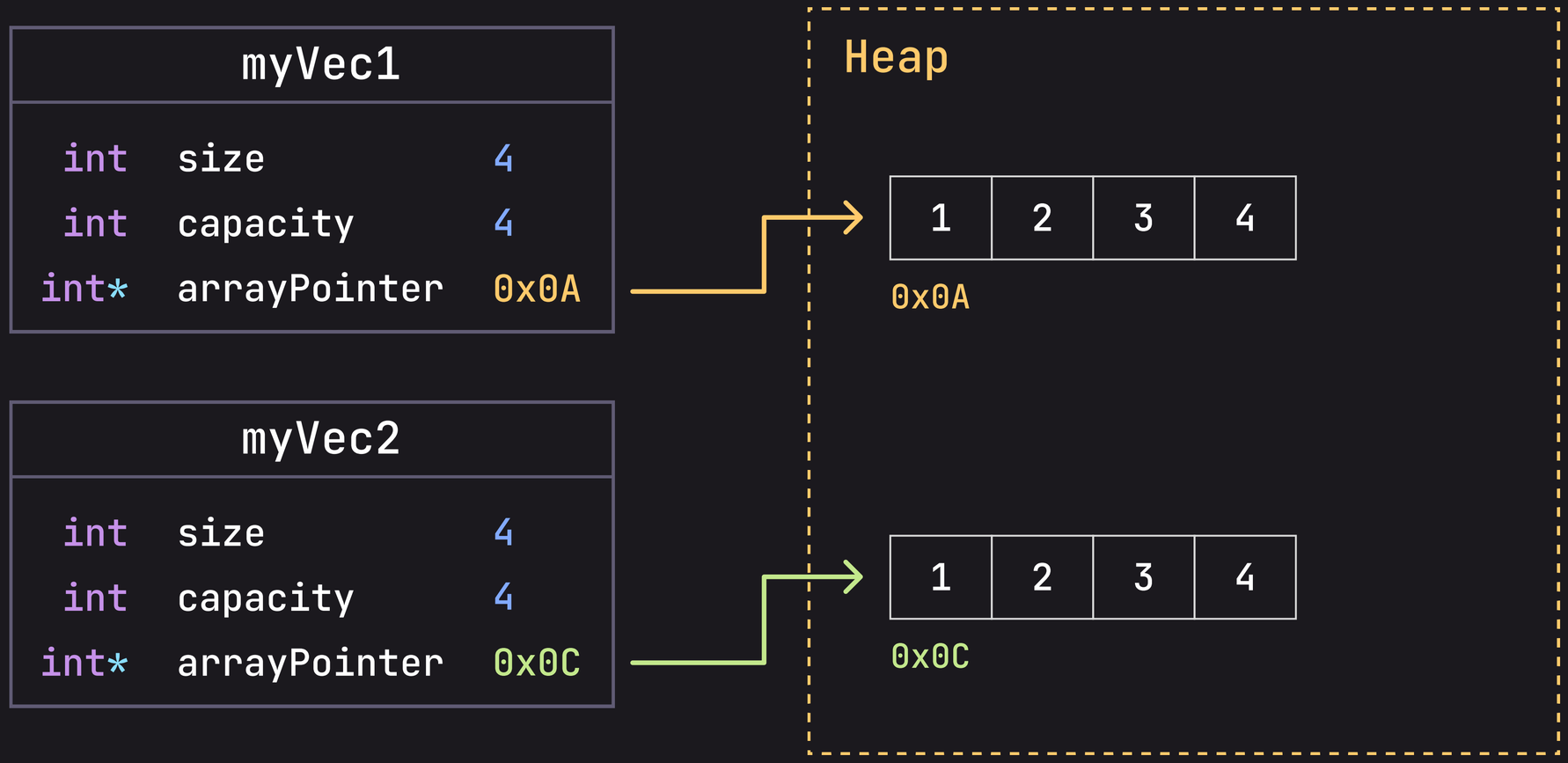
Copy Swap Idiom



Copy Swap Idiom



Copy Swap Idiom




```
myVec2 = myVec1 // ⟵ makes a copy and overwrites myVec2
```

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
   // implementation
}
```

```
myVec2 = myVec1 // ⟵ in this case myVec1 is passed in as other
```

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  // implementation
}
```

How do we first create a copy?

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  // other is a copied through pass by value
}
```

Trick question! We already have made
a copy by passing by value

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  // other is a copied through pass by value
}
```

How do we swap all the properties of other with this instance?

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  std::swap(size, other.size);
  std::swap(capacity, other.capacity);
  std::swap(arrayPointer, other.arrayPointer);
}
```

We use the std::swap function 😊

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  std::swap(size, other.size);
  std::swap(capacity, other.capacity);
  std::swap(arrayPointer, other.arrayPointer);
}
```

Ok now we do not need *other* anymore...

So how do we delete it?

Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  std::swap(size, other.size);
  std::swap(capacity, other.capacity);
  std::swap(arrayPointer, other.arrayPointer);
  // After this line other goes out of scope
  // so its destructor is called
}
```

Trick question!

Because *other* is a local variable, it will go out of scope at the end of the method and C++ will call its destructor

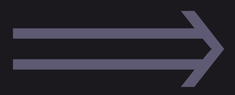
Copy Swap Idiom

```
MyVector& MyVector::operator=(MyVector other) {
  std::swap(size, other.size);
  std::swap(capacity, other.capacity);
  std::swap(arrayPointer, other.arrayPointer);
  return *this
}
```

The last thing to do is to return the correct type.
We expect a reference to the current instance of MyVector

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 easy

Let’s try implement the constructors in the activity

Implementing Iterators

```
class Iterator {
public:
    T& operator*();
    Iterator& operator++();
    Iterator& operator--();
    bool operator==(const Iterator& x, const Iterator& y)
    bool operator≠(const Iterator& x, const Iterator& y)
}
```

How you implement an iterator is up to you
as long as it has these public methods

Implementing Iterators

```
class Iterator {
public:
    T& operator*();
    Iterator& operator++();
    Iterator& operator--();
    bool operator==(const Iterator& x, const Iterator& y)
    bool operator≠(const Iterator& x, const Iterator& y)
}
```

You will need to store some information that keeps track of where in the collection you are

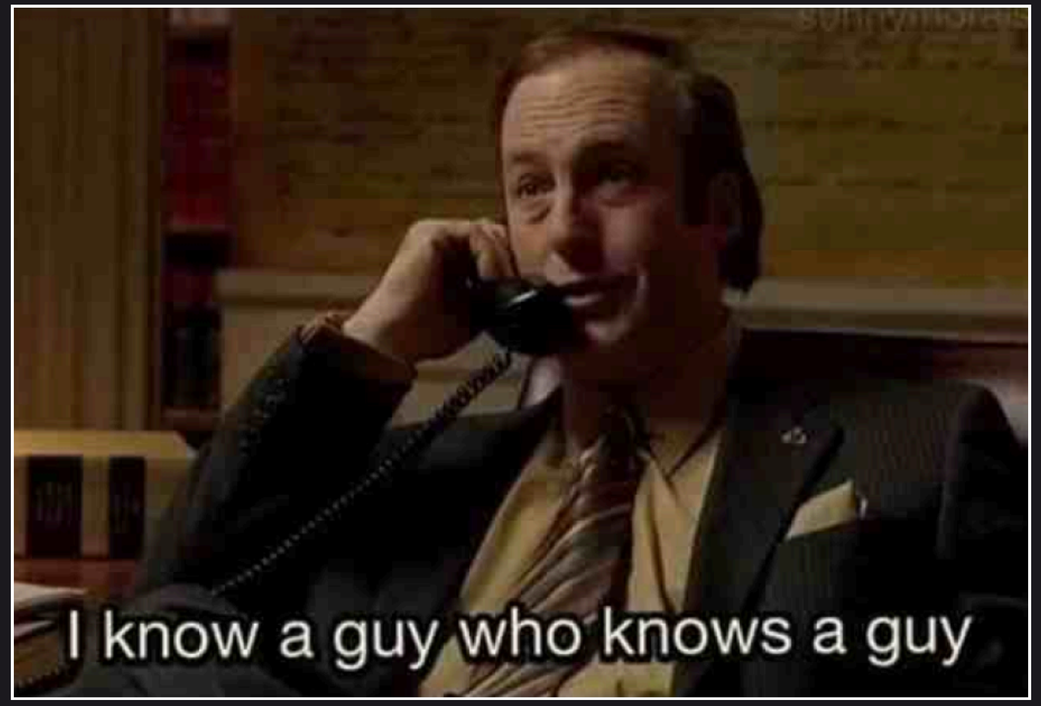
Implementing Iterators

```
Iterator MyVector<T>::begin()
Iterator MyVector<T>::end() const
```

implement the Iterator class and add the following methods to the MyVector class

Let’s have a go on Ed

Linked List



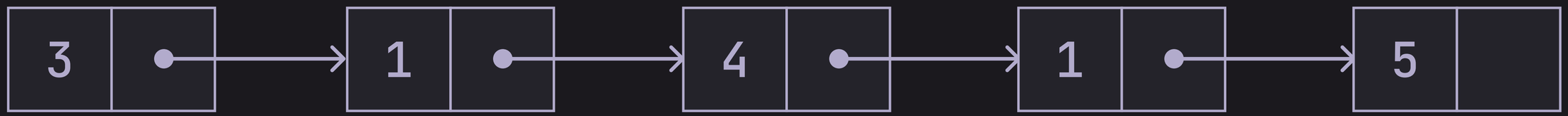
Linked List

Suppose we have an array like so

Linked List

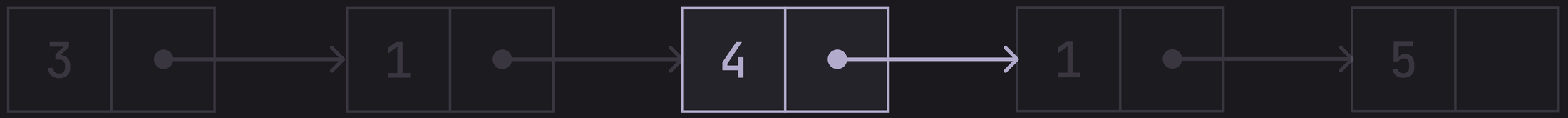
But the data is spread out in memory

Linked List

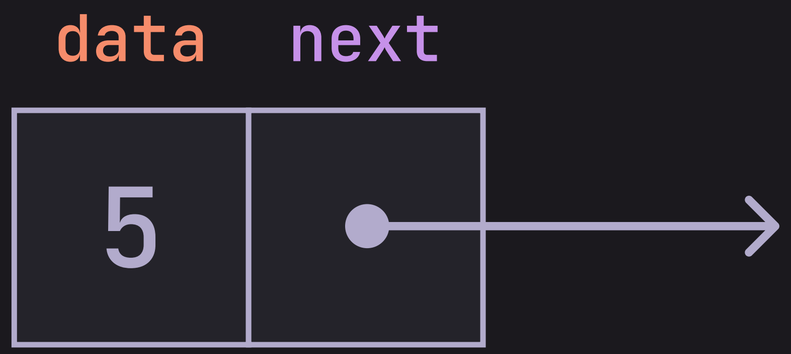


Then instead of using indices to impose an order
we can use pointers

Linked List



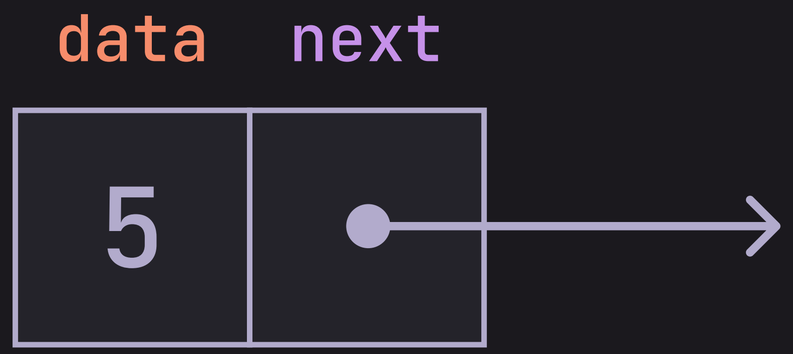
Linked List



Node

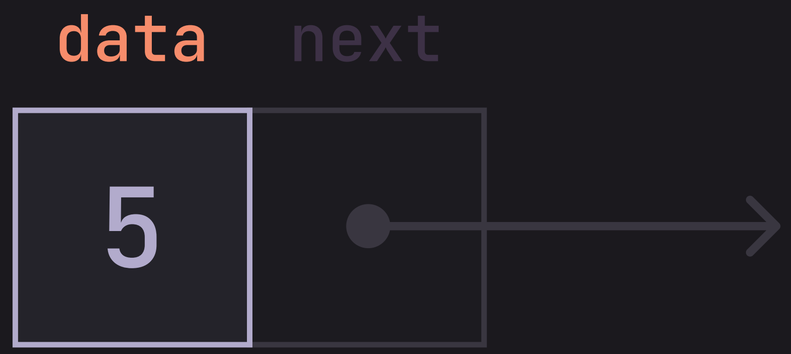
Linked List

```
struct Node {
}
```



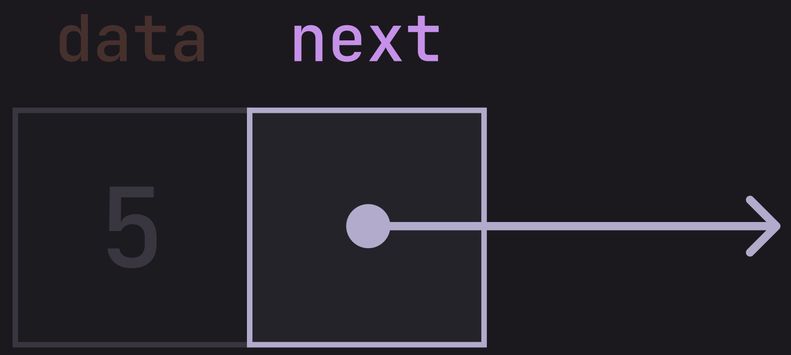
Linked List

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



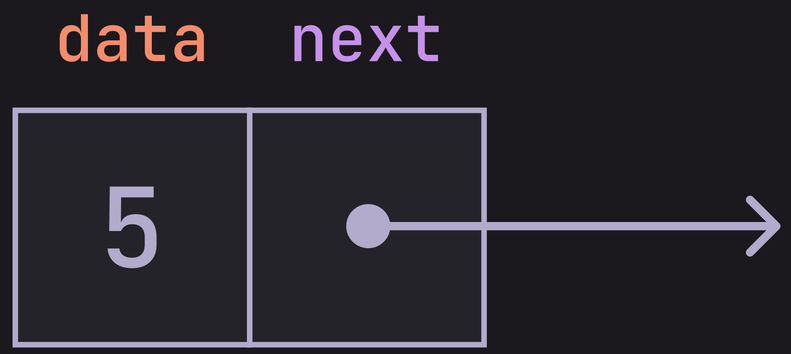
Linked List

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



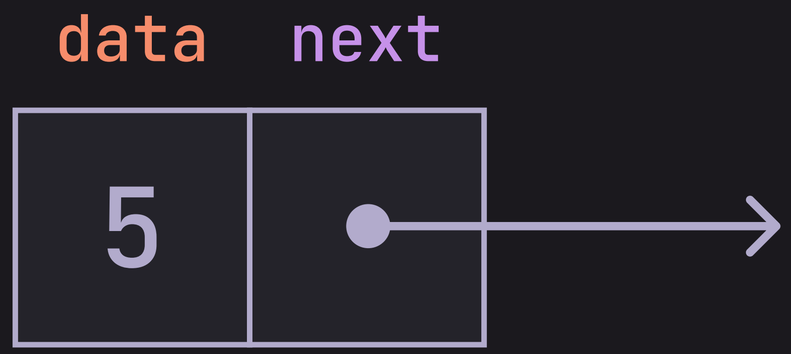
Linked List

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



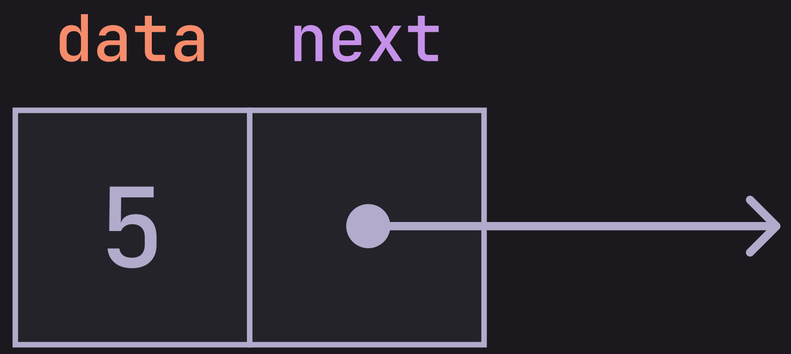
Linked List

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

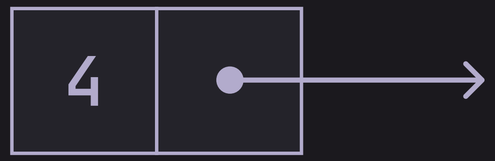


Linked List

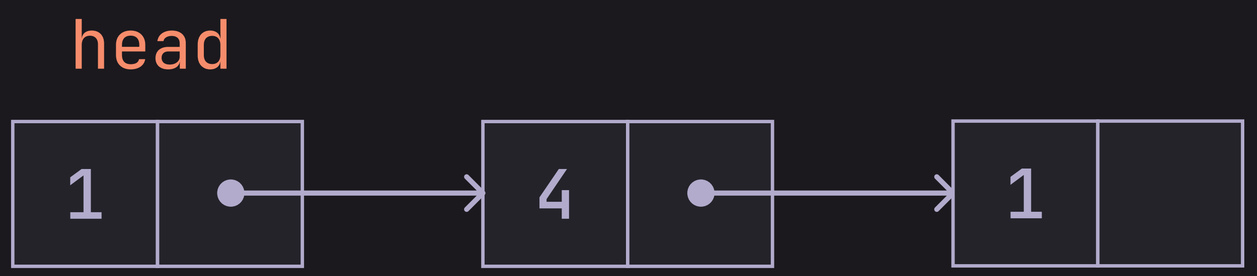
```
template <typename T>
struct Node {
  T data {};
  Node* next = nullptr;
  Node(){}
  Node(T input_data, Node* next_node= nullptr) :
    data {input_data}, next {next_node} {}
}
```



Linked List

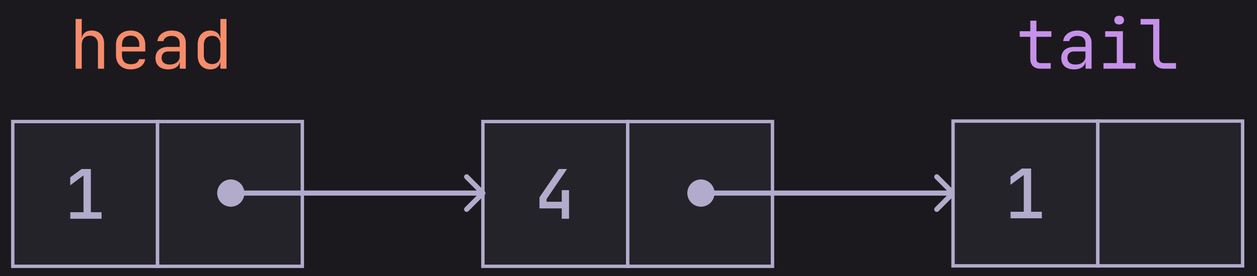


Linked List



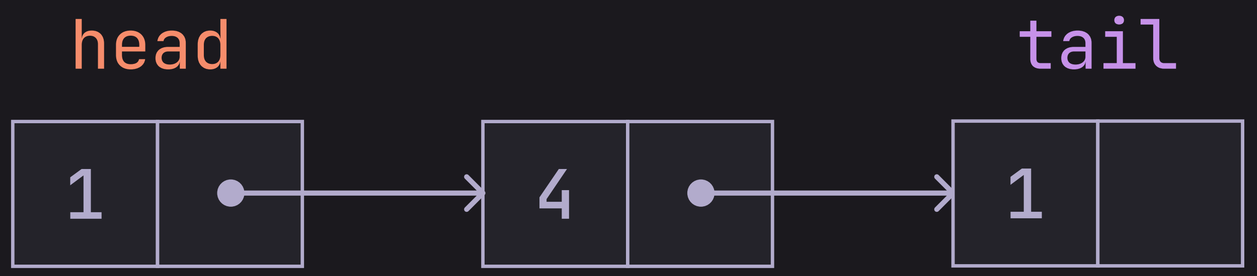
To keep track of the start of our list we define a variable head which points to the first node

Linked List



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

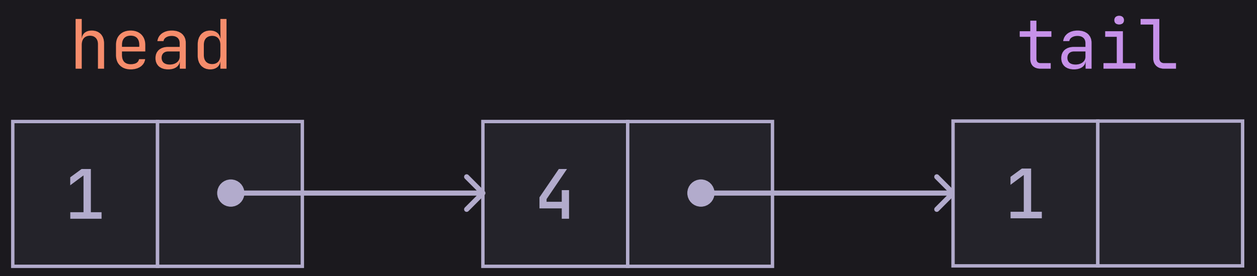
Linked List



`push_front(3)`

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

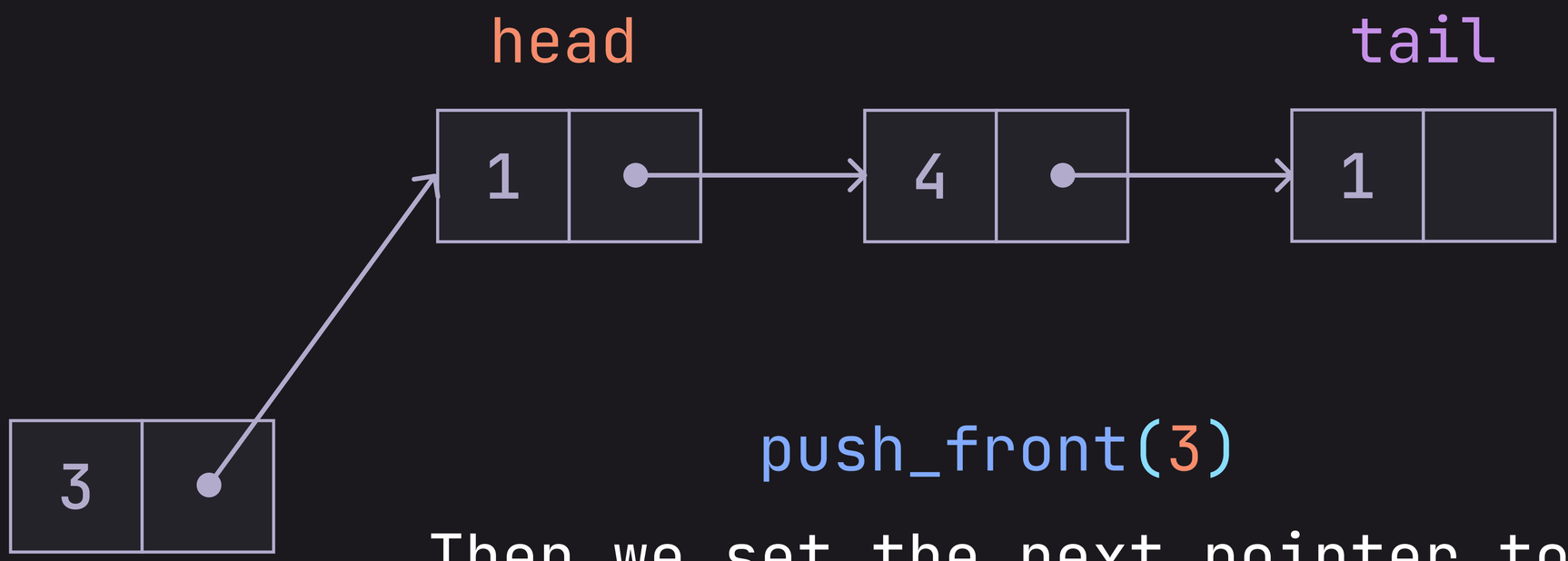
Linked List



`push_front(3)`

First we need to create a new node

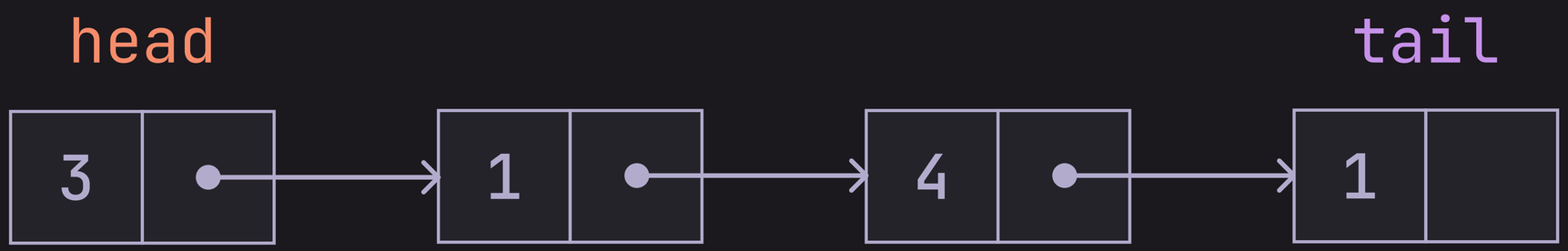
Linked List



`push_front(3)`

Then we set the next pointer to the current `head`

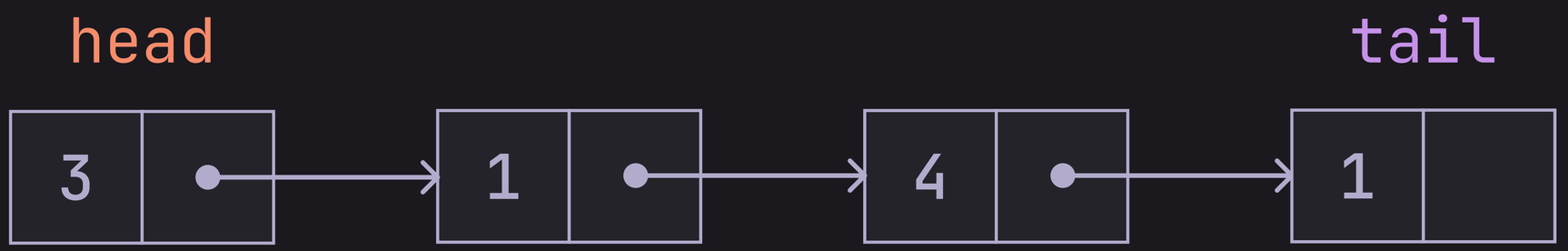
Linked List



`push_front(3)`

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

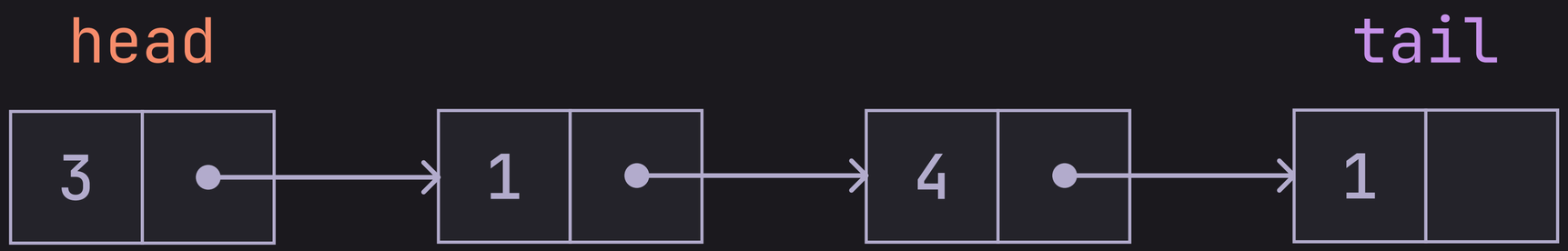
Linked List



`push_back(5)`

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

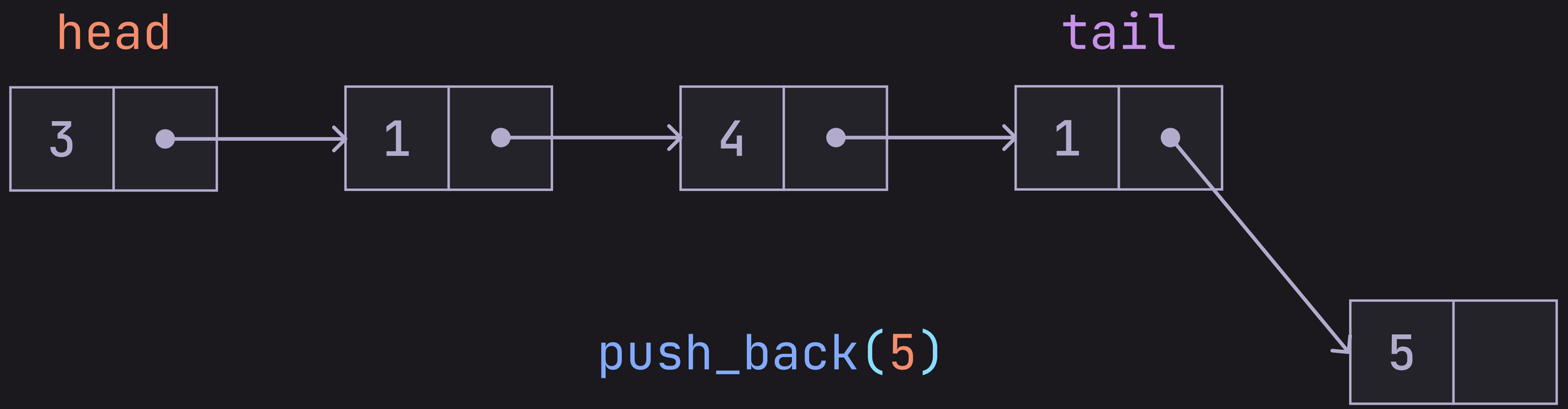
Linked List



`push_back(5)`

First we create a new node

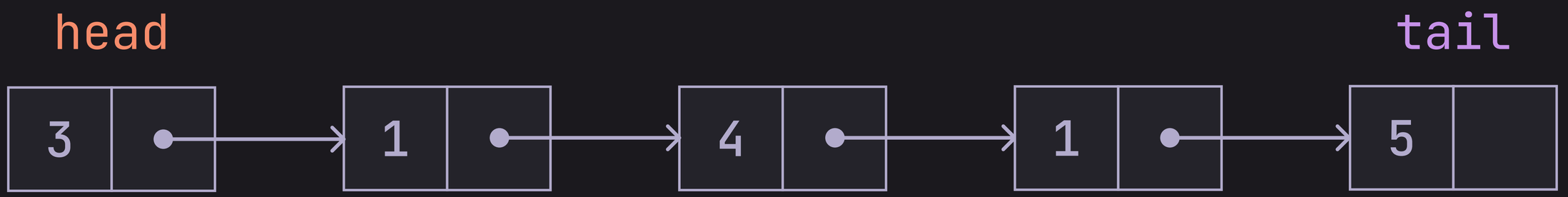
Linked List



`push_back(5)`

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

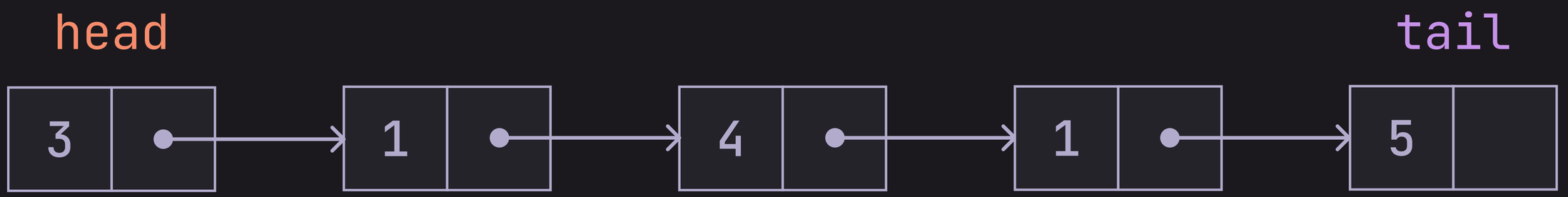
Linked List



`push_back(5)`

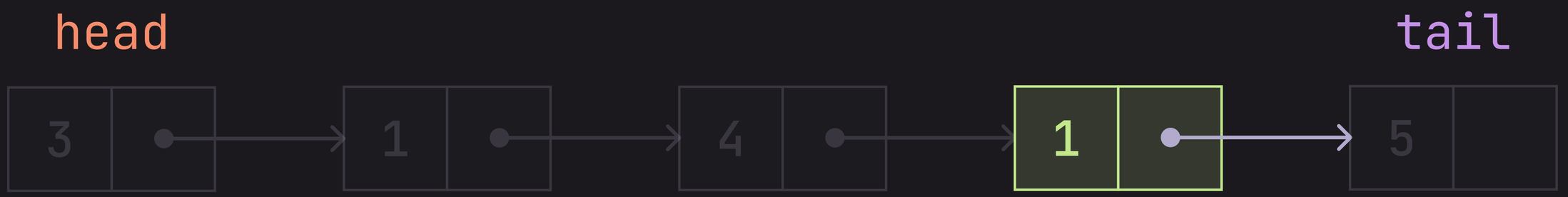
Finally we update the `tail` to point to the node we just added

Linked List



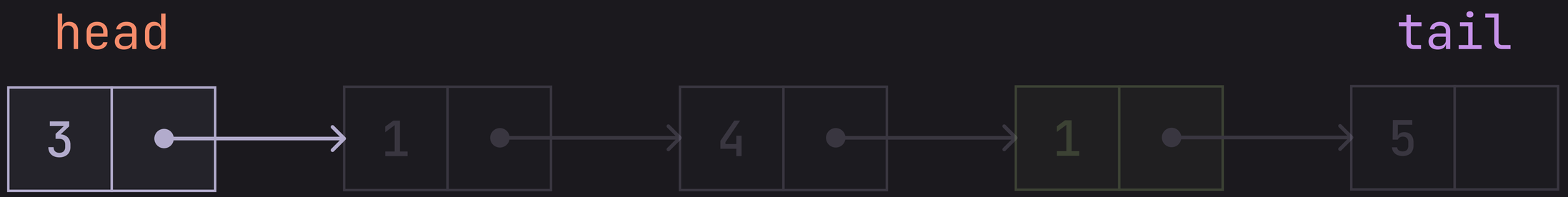
This brings us to a pretty big downside of linked lists which is you cannot randomly access elements

Linked List



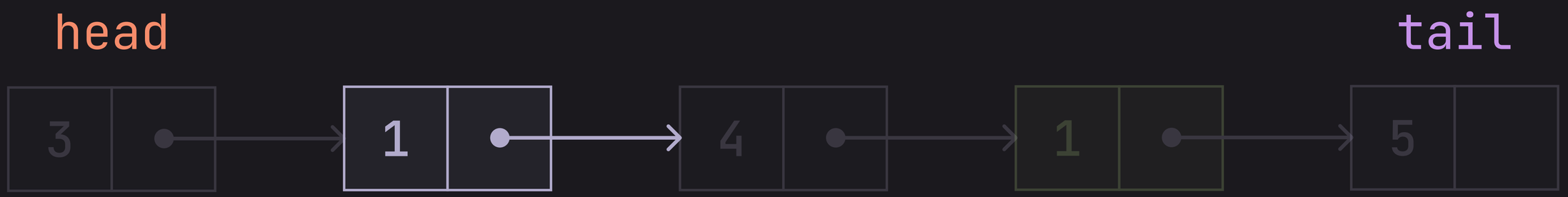
Suppose I want to access the data at the 3rd index

Linked List



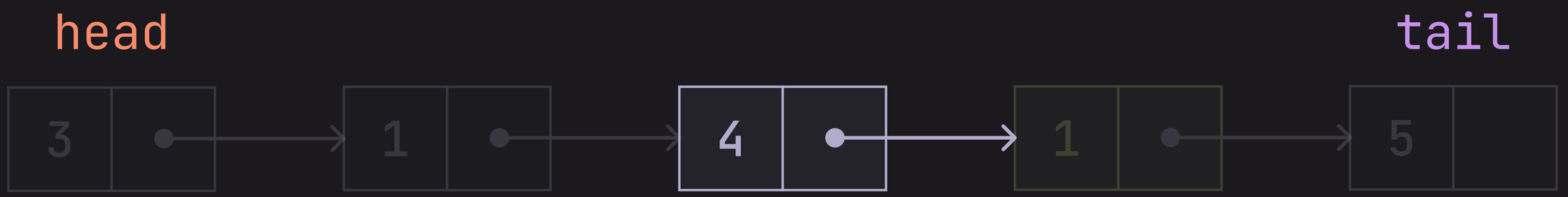
I need to start all the way at the head and keep following the next pointer

Linked List



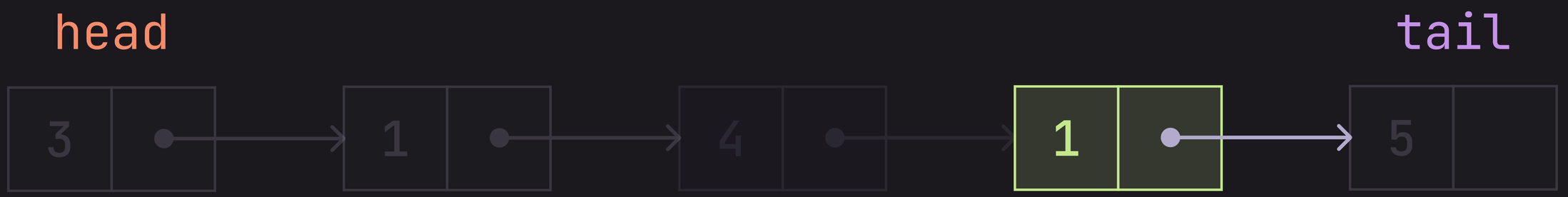
I need to start all the way at the head and keep following the next pointer

Linked List



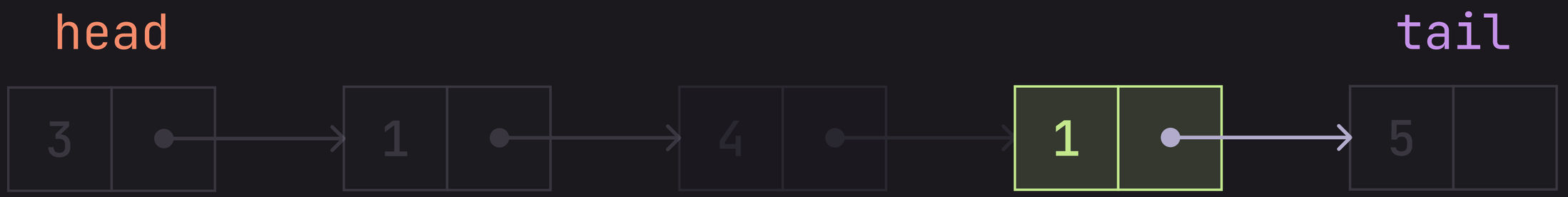
I need to start all the way at the head and keep following the next pointer

Linked List



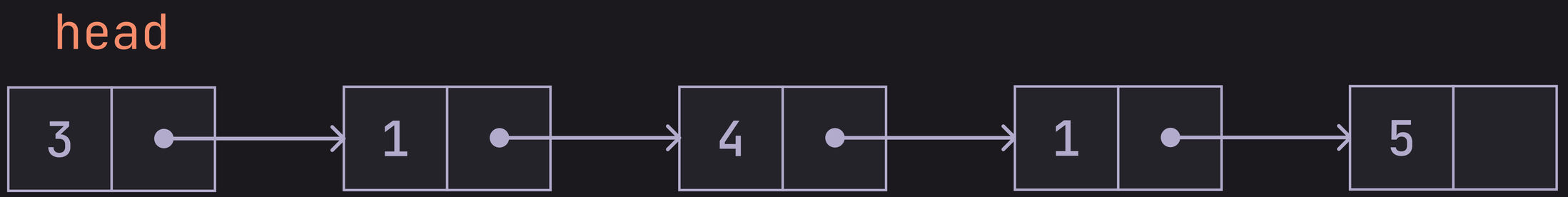
I need to start all the way at the head and keep following the next pointer

Linked List



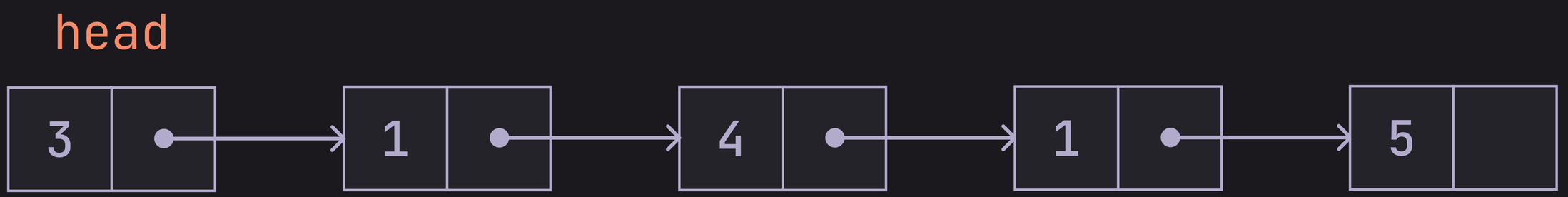
O(n)

Linked List



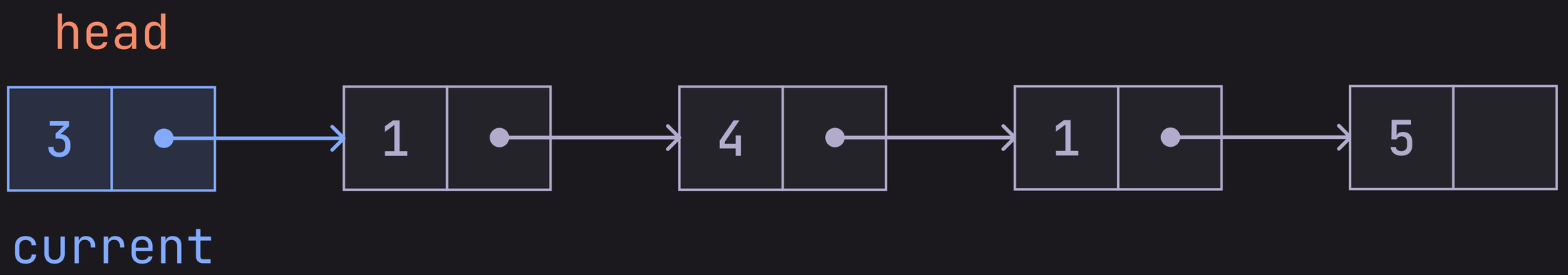
So if we do not keep track of the tail
then we need to first traverse the whole list
before we can add to the end of the list

Linked List



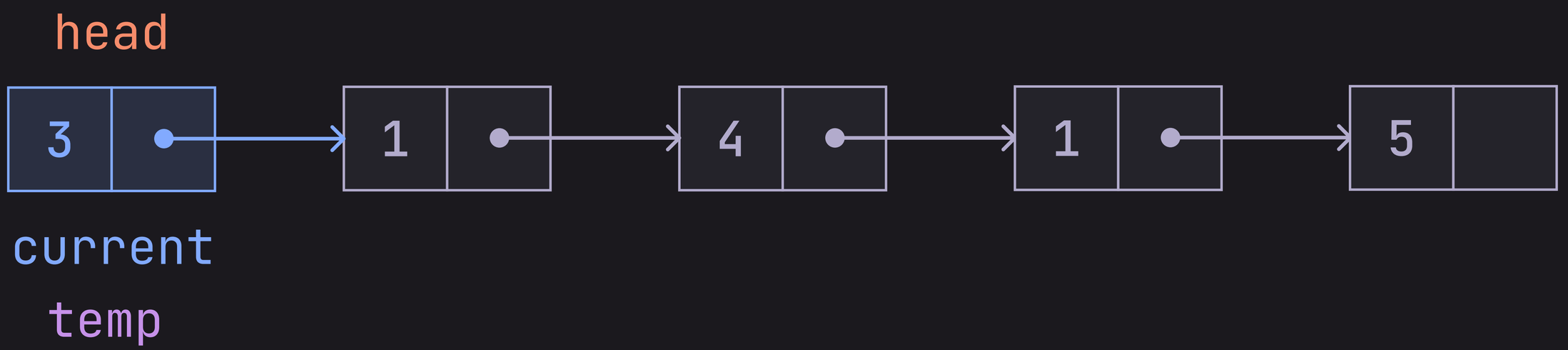
So how do we destruct a linked list?

Linked List



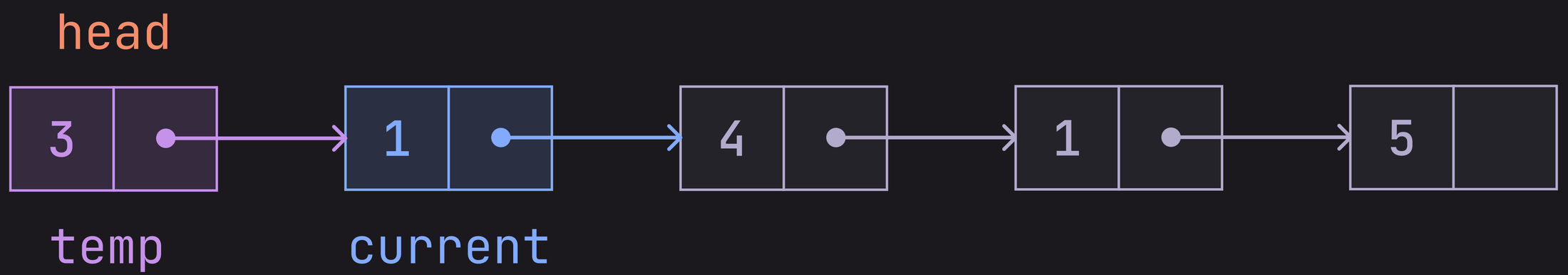
We start from the front and delete each node

Linked List



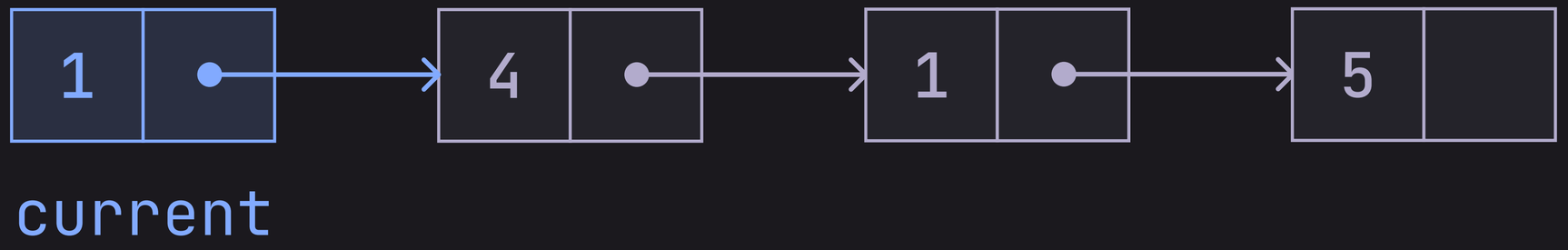
But we need to keep track of the next node before we delete the current node

Linked List



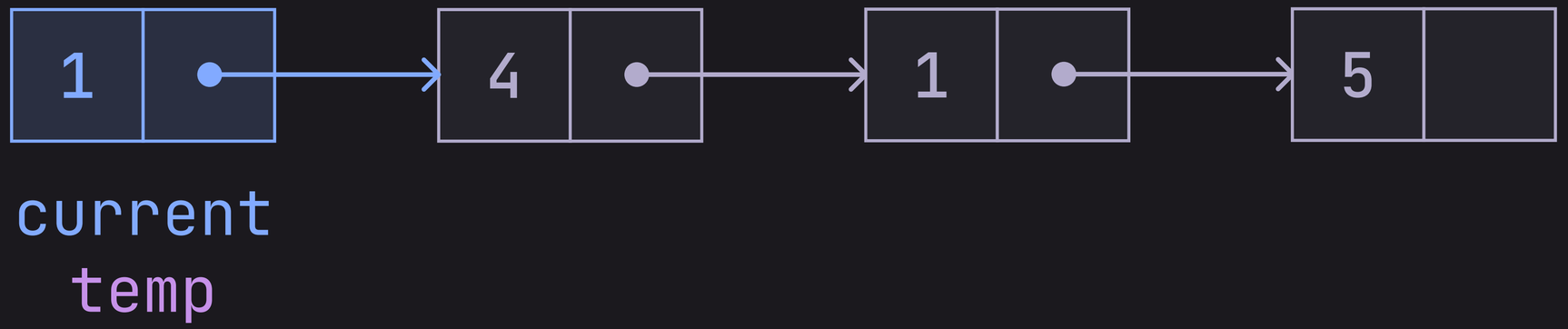
But we need to keep track of the next node before we delete the current node

Linked List



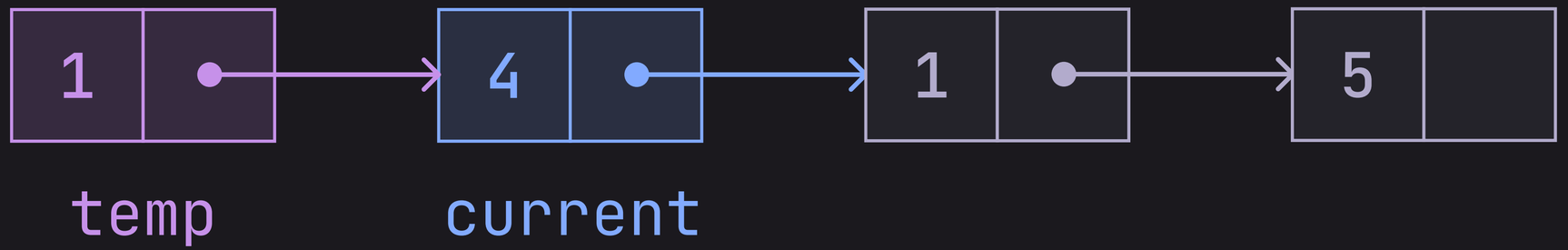
But we need to keep track of the next node before we delete the current node

Linked List



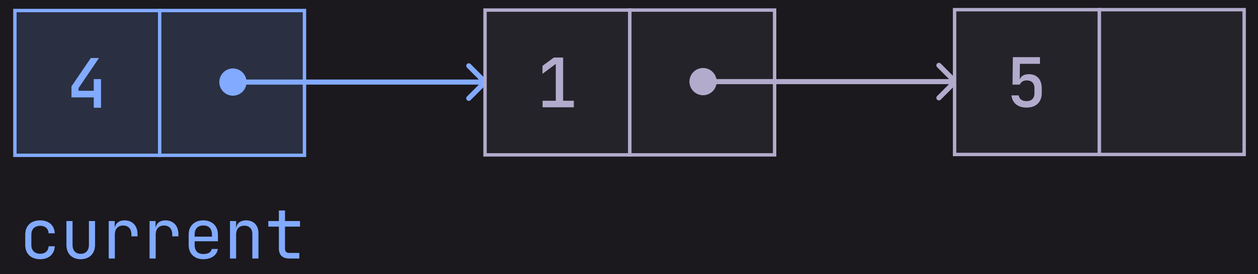
But we need to keep track of the next node before we delete the current node

Linked List



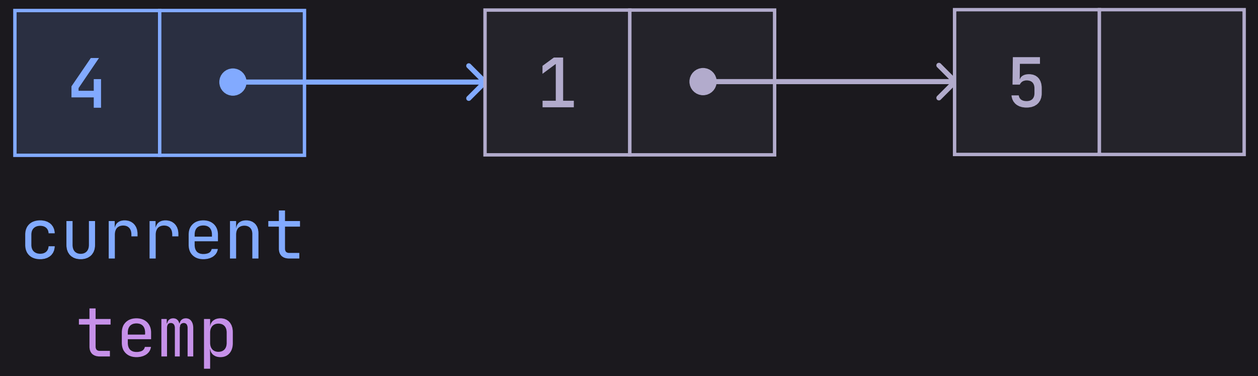
But we need to keep track of the next node before we delete the current node

Linked List



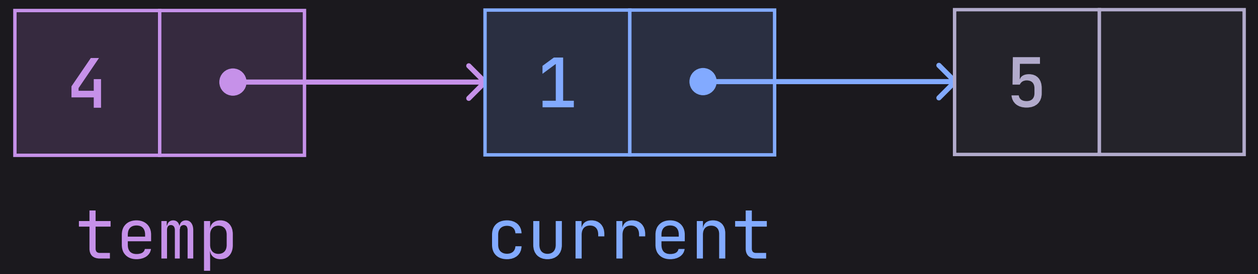
But we need to keep track of the next node before we delete the current node

Linked List



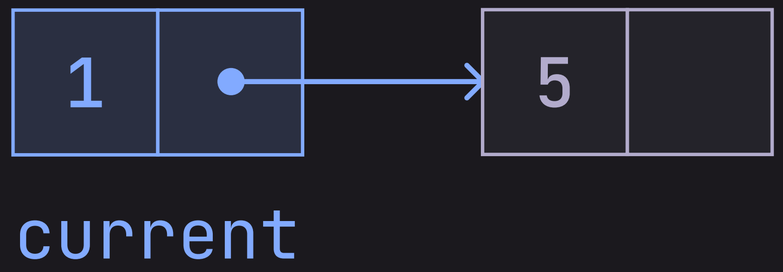
But we need to keep track of the next node before we delete the current node

Linked List



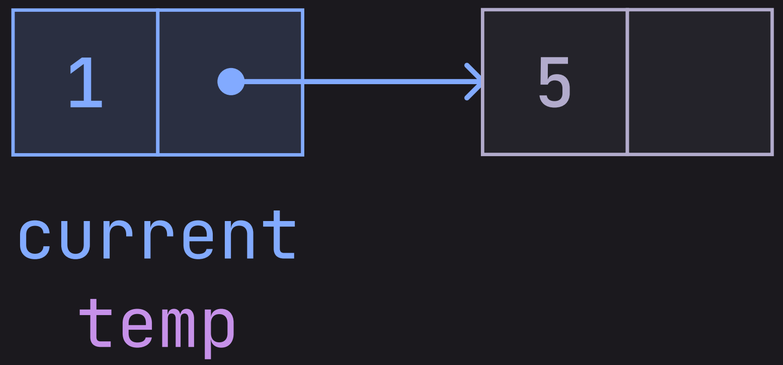
But we need to keep track of the next node before we delete the current node

Linked List



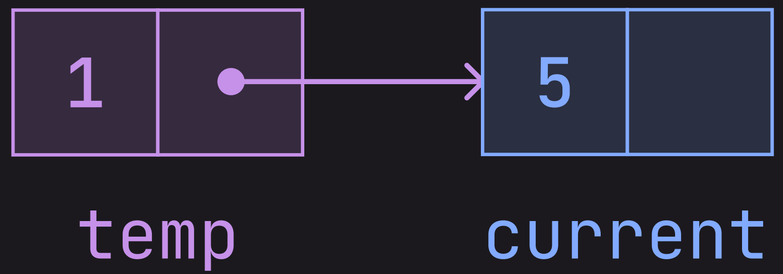
But we need to keep track of the next node before we delete the current node

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

current

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

Linked List