Data structures & algorithms

Tutorial 4

Lesson overview

- Recap of important topics from last week
- Function call and return statement breakdown
- Factorial problem revisited with recursion explained
- Brief call stack explanation
- Fibonacci numbers
- Adding iterators to MyVector
- Working on the assignment and questions if we have time

A quick recap of important things from last week

Templates

```
template <typename T>
T doubleValue(T a) {
   return a * 2;
}
```

- Templates are similar to Java's generics except better, they allow you to define a blueprint for a class or function in which the user can use any type
- We can make functions or classes templated, doing so will allow it to accept a generic type as a parameter
- The `T` is the generic type, and it can be called anything you want it to be… `cat`, `guitar`, `dsa`…etc.
- You are also allowed to include as many generic types as you want! Which can become extremely useful when creating a templated class
- When calling the `doubleValue` function, you can explicitly specify you type you would like…
  `int myValueDoubled = doubleValue<int>(10)`

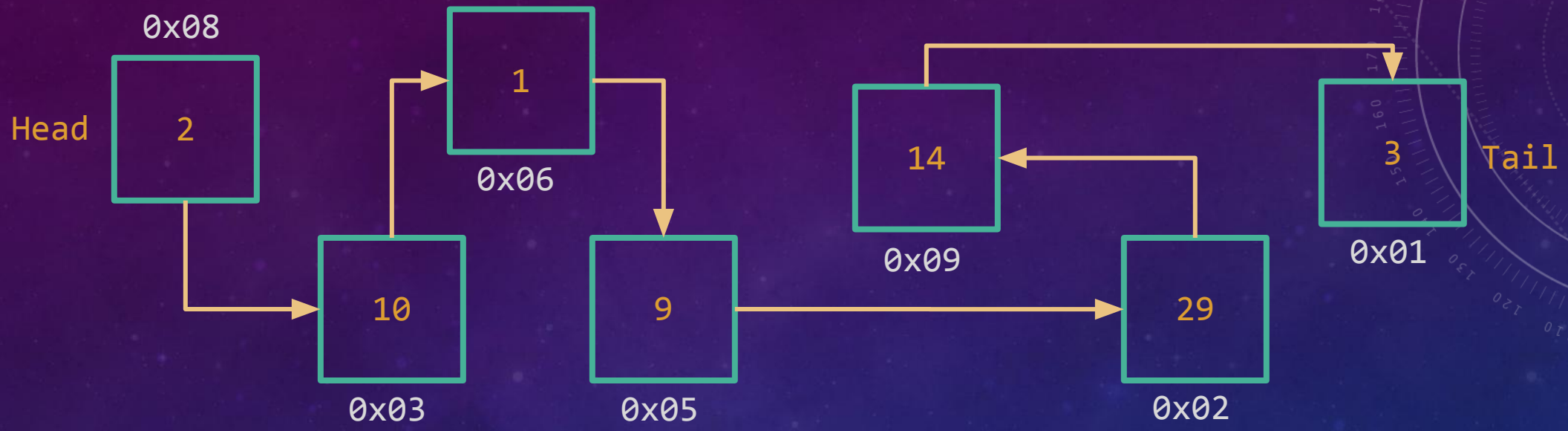
Copy constructor

```
int main() {
    MyVector myVector{1, 2, 3, 4};
    MyVector shallowCopy{myVector};
    shallowCopy[0] = 10;
    myVector[0]; // Returns 10
    shallowCopy[0]; // Returns 10
    return 0;
}
```

```
int main() {
    MyVector myVector{1, 2, 3, 4};
    MyVector deepCopy{myVector};
    deepCopy[0] = 10;
    myVector[0]; // Returns 1
    deepCopy[0]; // Returns 10
    return 0;
}
```

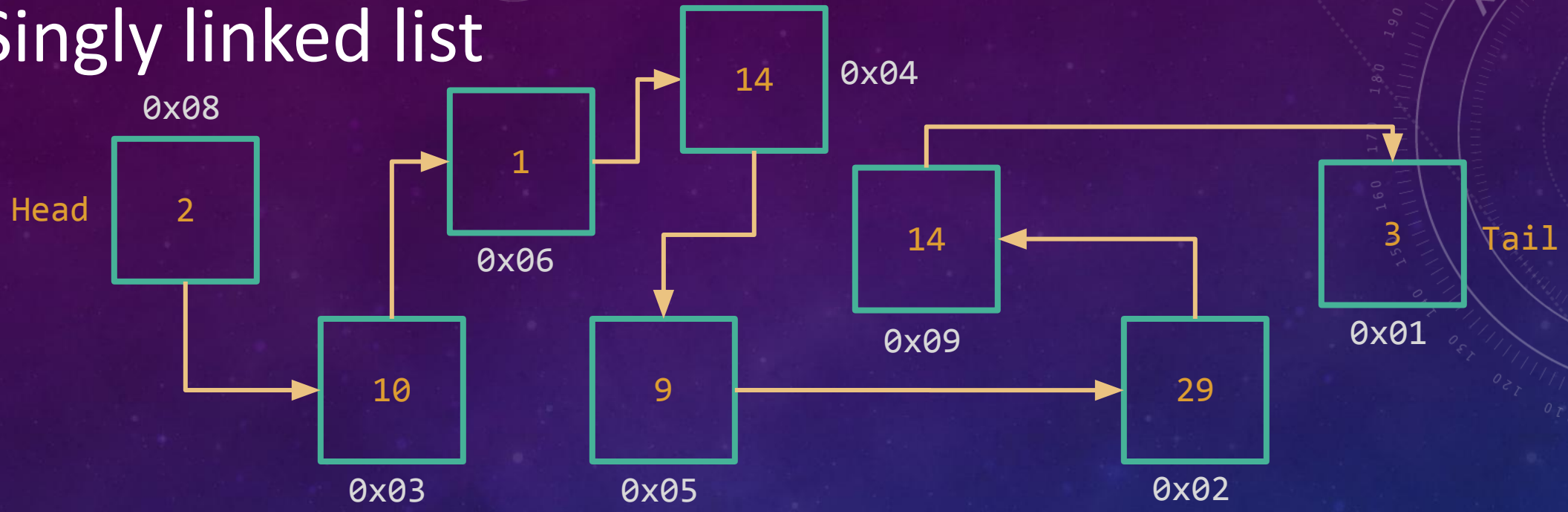
- There is a primary difference between a shallow copy and a deep copy. A shallow copy is very similar to a reference, in the fact that the if you alter the shallow copy… the original gets altered as well. The only difference is a shallow copy creates a completely new object, whilst a reference creates an alias to an existing one.
- A deep copy also creates a new object but copies all the data over into a new location that won't affect the original variable.

Singly linked list



- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data… each node is then linked together, with the previous node pointing at the next one.
- Since each piece of data is stored in its own node, they aren't stored in a contiguous line. Rather in random areas on the RAM. Because of this, this means singly linked lists are not accessible via an index like vectors. Which makes element access O(n) time complexity, rather than O(1)
- Sometimes linked lists will also have a variable for the head, and tail of the linked list. But not always! Which will allow O(1) time complexity to retrieve either value and insert or delete at either end.

Singly linked list

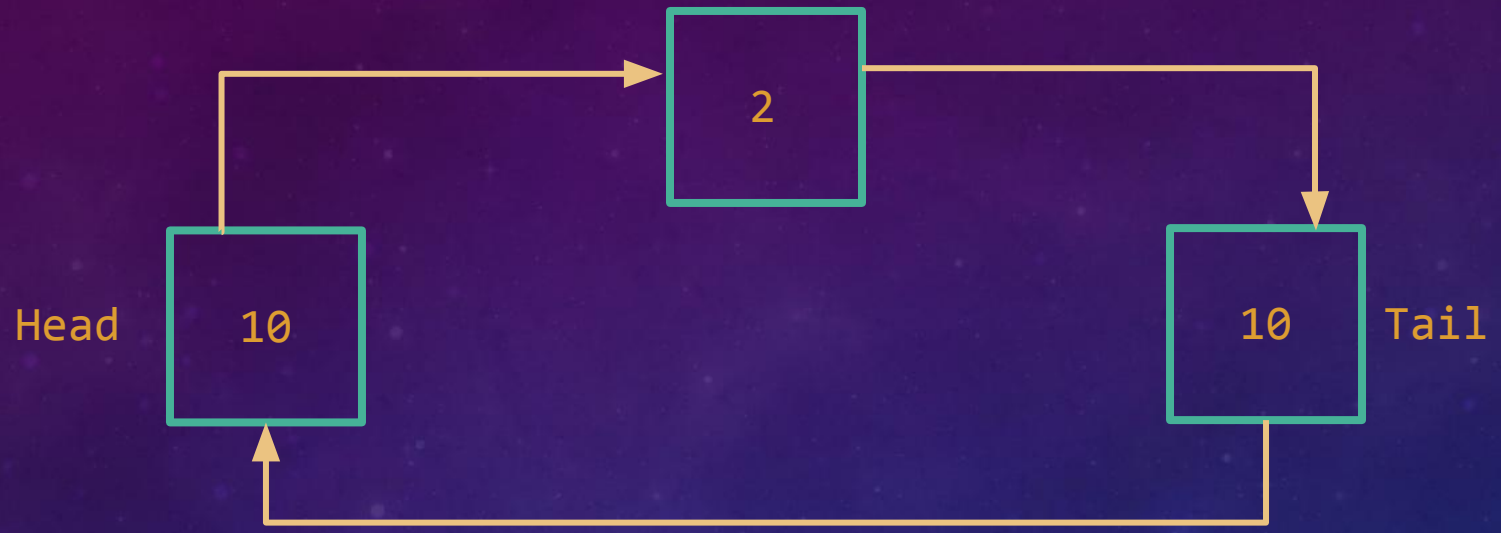


- If I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists… And tell the node containing 1 to now point to the node containing 14… then tell the node containing 14 to point to the node containing 9.
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple

Types of Linked Lists

- We looked at the different types of linked lists, like the singly linked lists

Types of Linked Lists



- We looked at the different types of linked lists, like the singly linked lists, the circular singly linked list

Types of Linked Lists

- We looked at the different types of linked lists, like the singly linked lists, the circular singly linked list, the doubly linked list

Types of Linked Lists

- We looked at the different types of linked lists, like the singly linked lists, the circular singly linked list, the doubly linked list, and the circular doubly linked list

Lab time 😎

Function calls and return statements

```
int main() {

   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem

Function calls and return statements

```
int main() {

   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement

Function calls and return statements

```
int main() {
   int num = addTen(4);
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`…

Function calls and return statements

```
int main() {
   int num = addTen(4);
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into…

Function calls and return statements

```
int main() {
   int num = addTen(4);
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into… is the return statement from that function

Function calls and return statements

```
int main() {
   int num = addTen(4);
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into… is the return statement from that function
- So what the compiler does when it reaches the return statement…

Function calls and return statements

```
int main() {
   int num = a + 10;
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into… is the return statement from that function
- So what the compiler does when it reaches the return statement… is it replaces that function call…

Function calls and return statements

```
int main() {
   int num = 4 + 10;
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into… is the return statement from that function
- So what the compiler does when it reaches the return statement… is it replaces that function call… fills in necessary values with its parameters…

Function calls and return statements

```
int main() {
   int num = 14;
   return 0;
}
```

```
int addTen(int a) {
   return a + 10;
}
```

- This is how I think of function calls, it will hopefully help with understanding recursion which I will demonstrate shortly with a recognizable problem
- Whenever you call a function that returns a value, that function call instinctively turns into whatever is in the return statement
- So for example, if I made a function call like `addTen(4)`… what this call really turns into… is the return statement from that function
- So what the compiler does when it reaches the return statement… is it replaces that function call… fills in necessary values with its parameters… and does the calculations if needed

Recursion for the factorial problem

```
int main() {
    int factorial = factorial(5);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`

```
int factorial(int n) {
    if(n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = factorial(5);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `5` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return

```
int factorial(5) {
    if(5 == 0) {
        return 1;
    }
    return 5 * factorial(5 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * factorial(5 - 1);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `5` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return

```
int factorial(5) {
    if(5 == 0) {
        return 1;
    }
    return 5 * factorial(5 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * factorial(5 - 1);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `5` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return
- Since we returned another function call, we continue to recursively call it

```
int factorial(5) {
    if(5 == 0) {
        return 1;
    }
    return 5 * factorial(5 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * factorial(4);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `4` doesn't equal `0` so we skip the base case

```
int factorial(4) {
    if(4 == 0) {
        return 1;
    }
    return 4 * factorial(4 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * factorial(3);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `4` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return
- Since we returned another function call again, we continue to recursively call it

```
int factorial(4) {
    if(4 == 0) {
        return 1;
    }
    return 4 * factorial(4 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * factorial(3);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `3` doesn't equal `0` so we skip the base case

```
int factorial(3) {
    if(3 == 0) {
        return 1;
    }
    return 3 * factorial(3 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * factorial(2);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `3` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return
- Since we returned another function call again, we continue to recursively call it

```
int factorial(3) {
    if(3 == 0) {
        return 1;
    }
    return 3 * factorial(3 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * factorial(2);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `2` doesn't equal `0` so we skip the base case

```
int factorial(2) {
    if(2 == 0) {
        return 1;
    }
    return 2 * factorial(2 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * factorial(1);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `2` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return
- Since we returned another function call again, we continue to recursively call it

```
int factorial(2) {
    if(2 == 0) {
        return 1;
    }
    return 2 * factorial(2 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * factorial(1);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `1` doesn't equal `0` so we skip the base case

```
int factorial(1) {
    if(1 == 0) {
        return 1;
    }
    return 1 * factorial(1 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * 1 * factorial(0);
    return 0;
}
```

- So we'll put what I just showed you into practice, with `factorial(5)`
- `1` doesn't equal `0` so we skip the base case
- After the condition, we hit the `return` statement, so we can replace our function call with that return
- Since we returned another function call again, we continue to recursively call it

```
int factorial(1) {
    if(1 == 0) {
        return 1;
    }
    return 1 * factorial(1 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * 1 * factorial(0);
    return 0;
}
```

- Now this time, we have a different case. Our base case gets triggered because `0` is equal to `0`
- This means the return statement inside the condition will be returned

```
int factorial(0) {
    if(0 == 0) {
        return 1;
    }
    return 0 * factorial(0 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * 1 * 1;
    return 0;
}
```

- Now this time, we have a different case. Our base case gets triggered because `0` is equal to `0`
- This means the return statement inside the condition will be returned
- There is no recursive call in this return, which is important for a base case

```
int factorial(0) {
    if(0 == 0) {
        return 1;
    }
    return 0 * factorial(0 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 5 * 4 * 3 * 2 * 1 * 1;
    return 0;
}
```

- Now this time, we have a different case. Our base case gets triggered because `0` is equal to `0`
- This means the return statement inside the condition will be returned
- There is no recursive call in this return, which is important for a base case
- Then the compiler does some calculations…

```
int factorial(0) {
    if(0 == 0) {
        return 1;
    }
    return 0 * factorial(0 - 1);
}
```

Recursion for the factorial problem

```
int main() {
    int factorial = 120;
    return 0;
}
```

- Now this time, we have a different case. Our base case gets triggered because `0` is equal to `0`
- This means the return statement inside the condition will be returned
- There is no recursive call in this return, which is important for a base case
- Then the compiler does some calculations… and you get the value of `factorial(5)`

```
int factorial(0) {
    if(0 == 0) {
        return 1;
    }
    return 0 * factorial(0 - 1);
}
```

The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

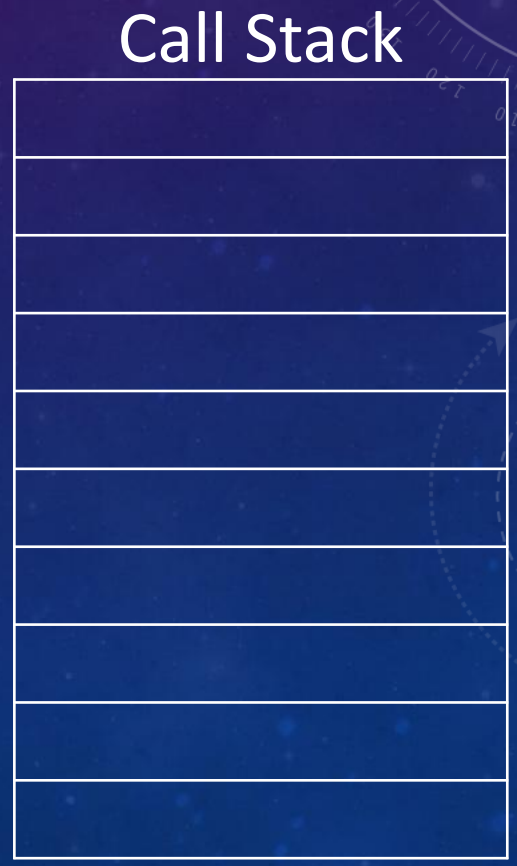
- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack

The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack…

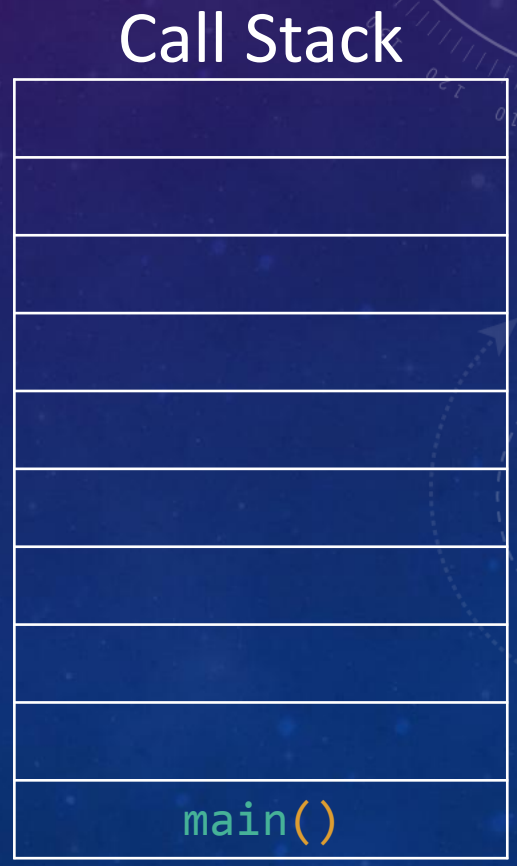


The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack…

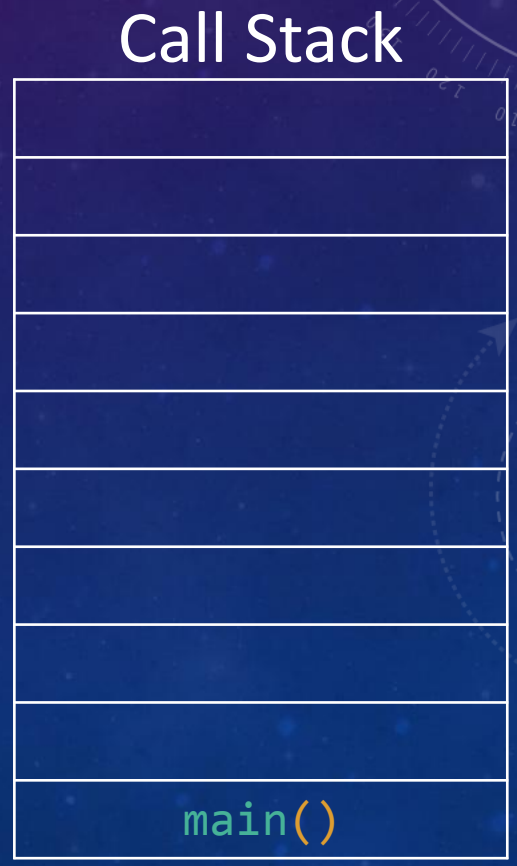


The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call…

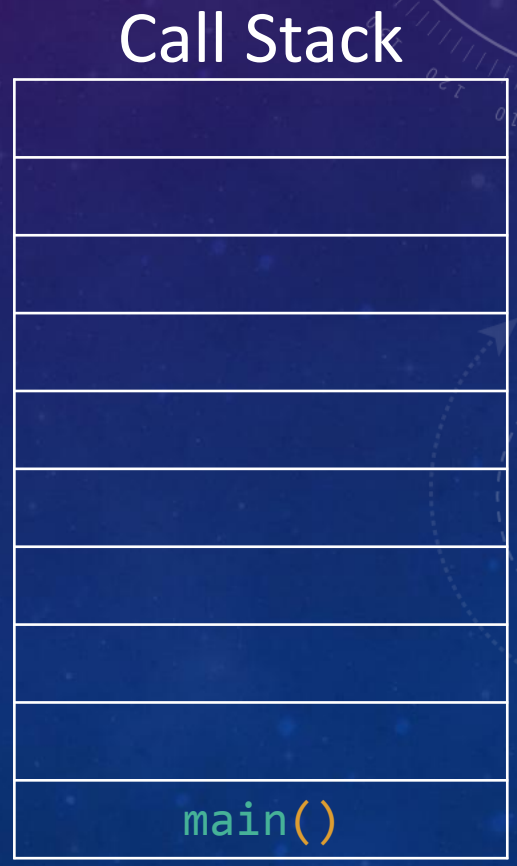


The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call…

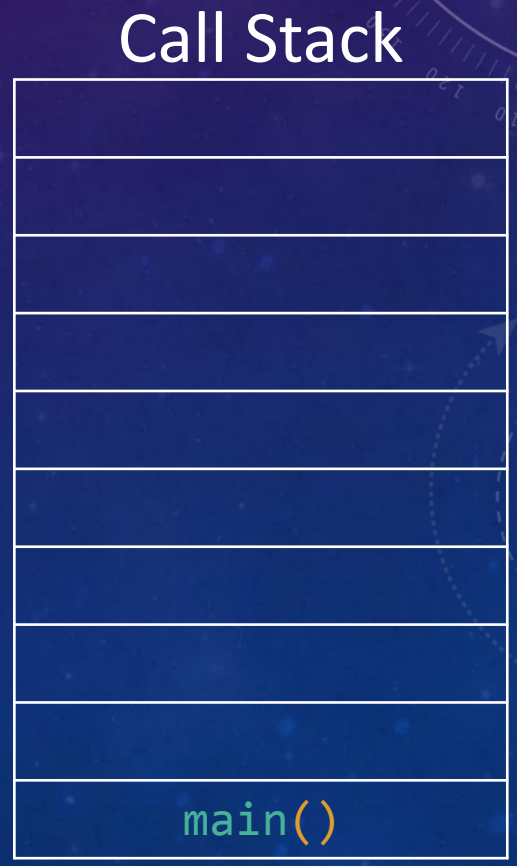


The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope

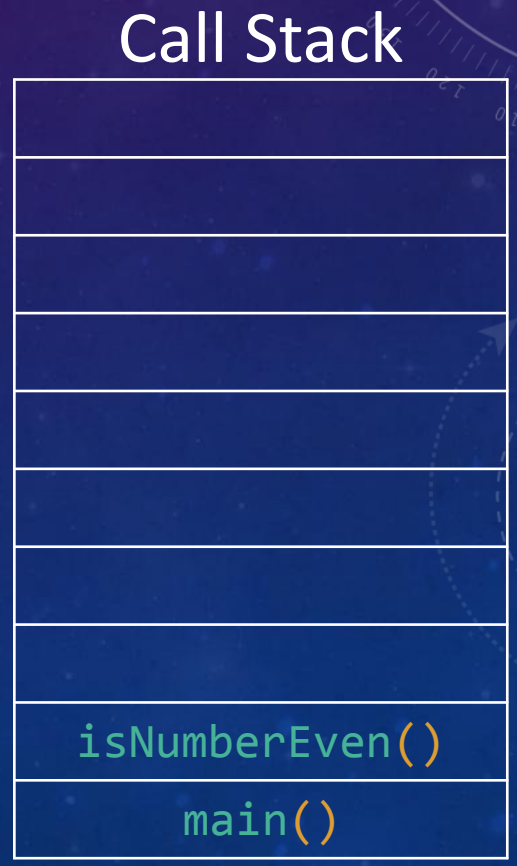


The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope



The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope
- The program now begins to execute each line within that function…

The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

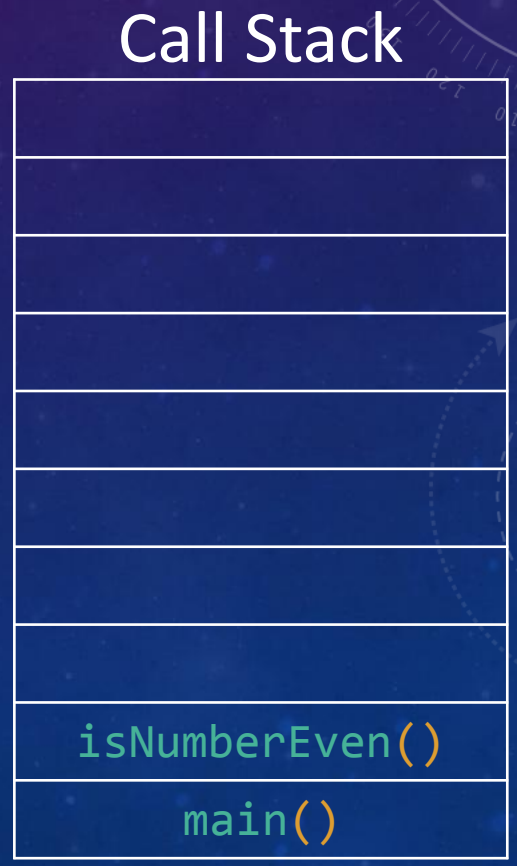
- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope
- The program now begins to execute each line within that function…

The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope
- The program now begins to execute each line within that function… it doesn't enter the condition as `19 % 2 != 0`



The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

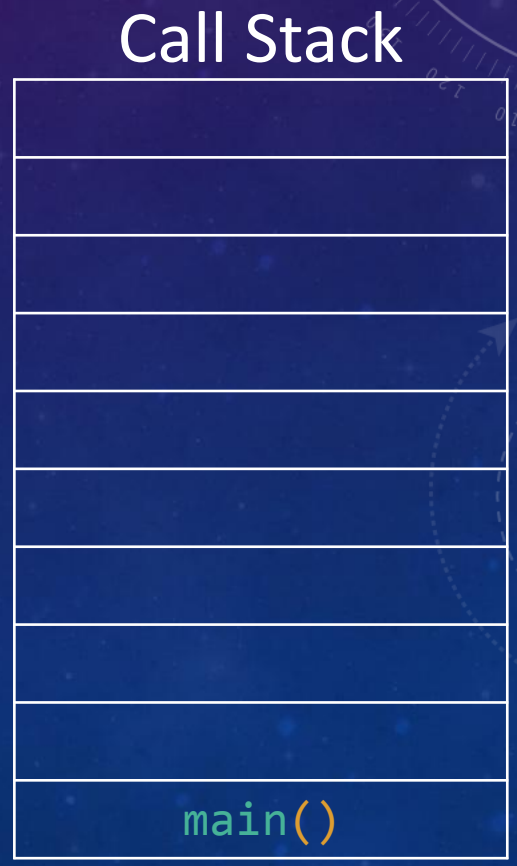
- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope
- The program now begins to execute each line within that function… it doesn't enter the condition as `19 % 2 != 0`
- Finally hits the return statement and re-enters the scope of main, so it can pop that function call from the stack

The call stack

```
int main() {
    int num = 19;
    bool isEven = isNumberEven(num);
    return 0;
}
```

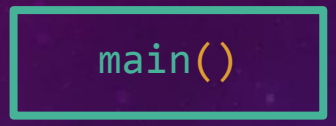
```
bool isNumberEven(bool n) {
    if(n % 2 == 0) {
        return true;
    }
    return false;
}
```

- The call stack is what our programs use to manage function calls
- Whenever a function call is made, it is pushed onto the call stack, and when the program reaches the end of that function scope, it gets popped from the stack
- So the program starts in main, and pushes that to the stack… the program will go through each line… and when it sees a function call… it pushes it to the stack and enters its scope
- The program now begins to execute each line within that function… it doesn't enter the condition as `19 % 2 != 0`
- Finally hits the return statement and re-enters the scope of main, so it can pop that function call from the stack



Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

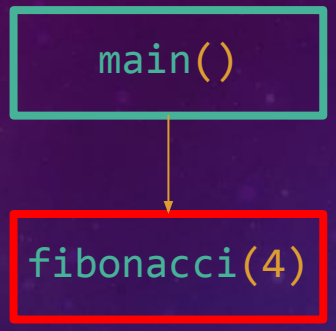


Call Stack

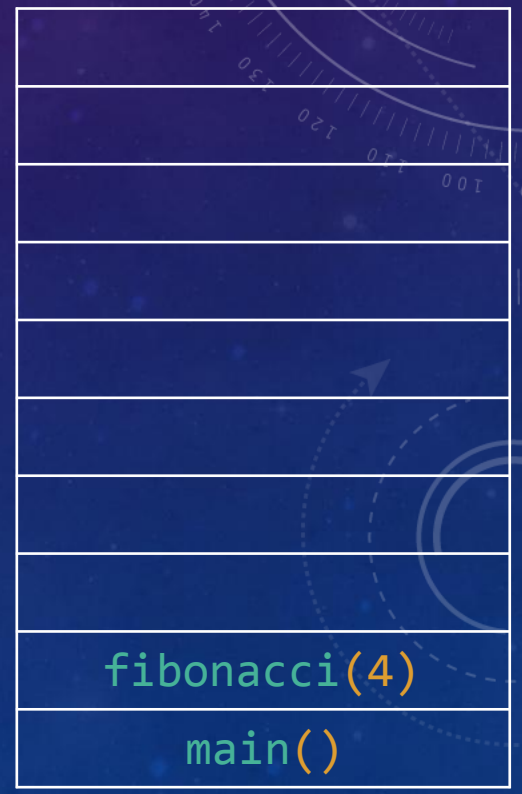
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The program enters `fibonacci(4)` and will proceed to recursively call until its base case is triggered, adding each call to the call stack



Call Stack

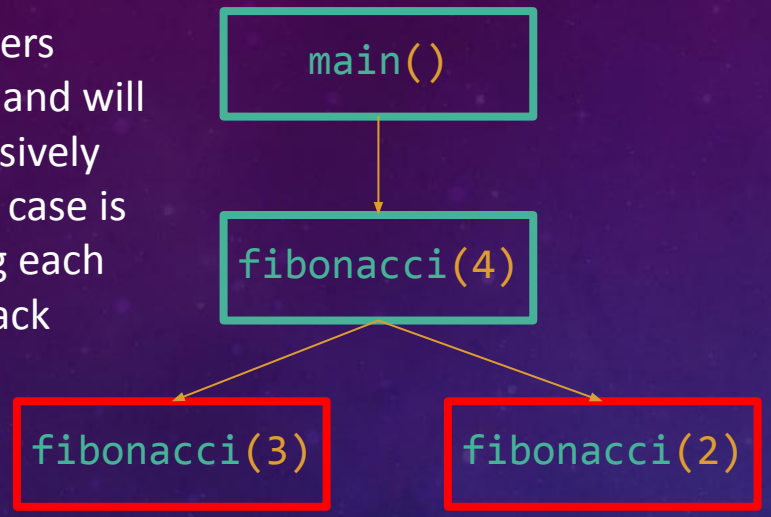



```
return fibonacci(4);
```

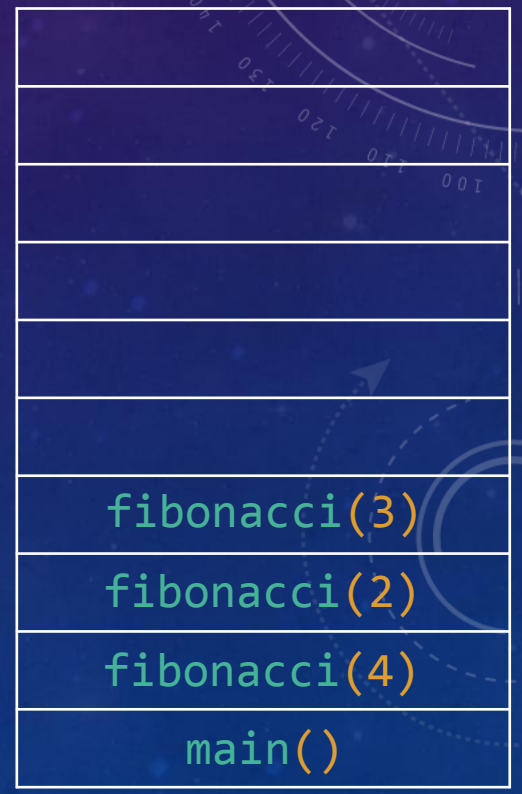
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The program enters `fibonacci(4)` and will proceed to recursively call until its base case is triggered, adding each call to the call stack



Call Stack

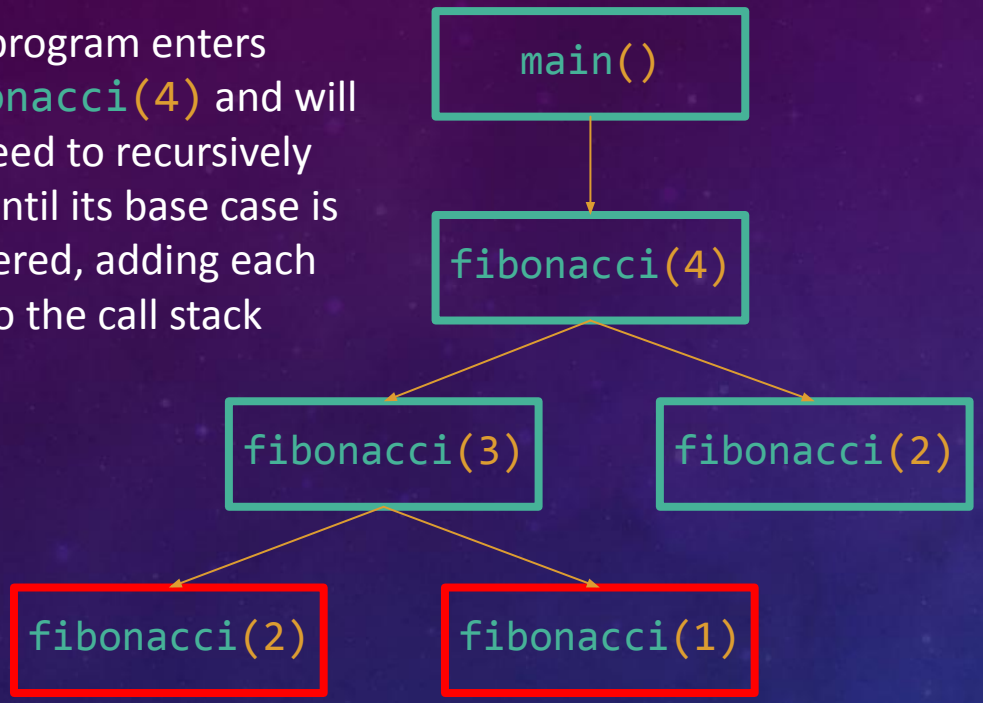



```
return fibonacci(3) + fibonacci(2);
```

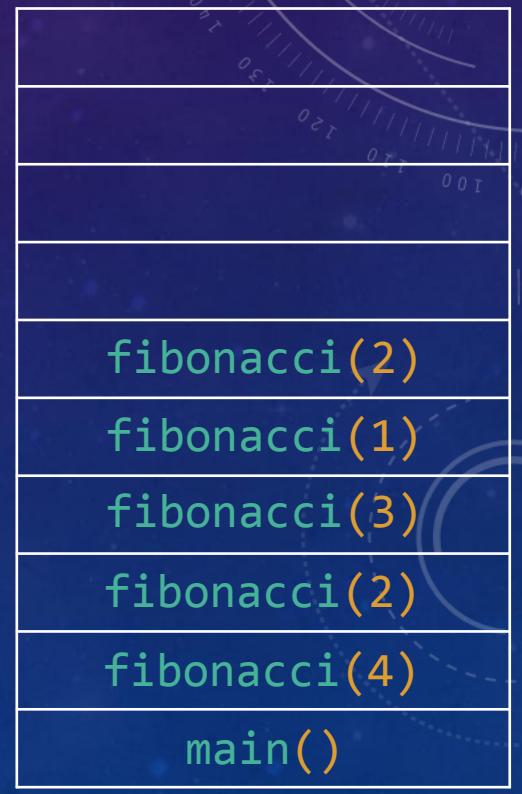
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The program enters `fibonacci(4)` and will proceed to recursively call until its base case is triggered, adding each call to the call stack



Call Stack

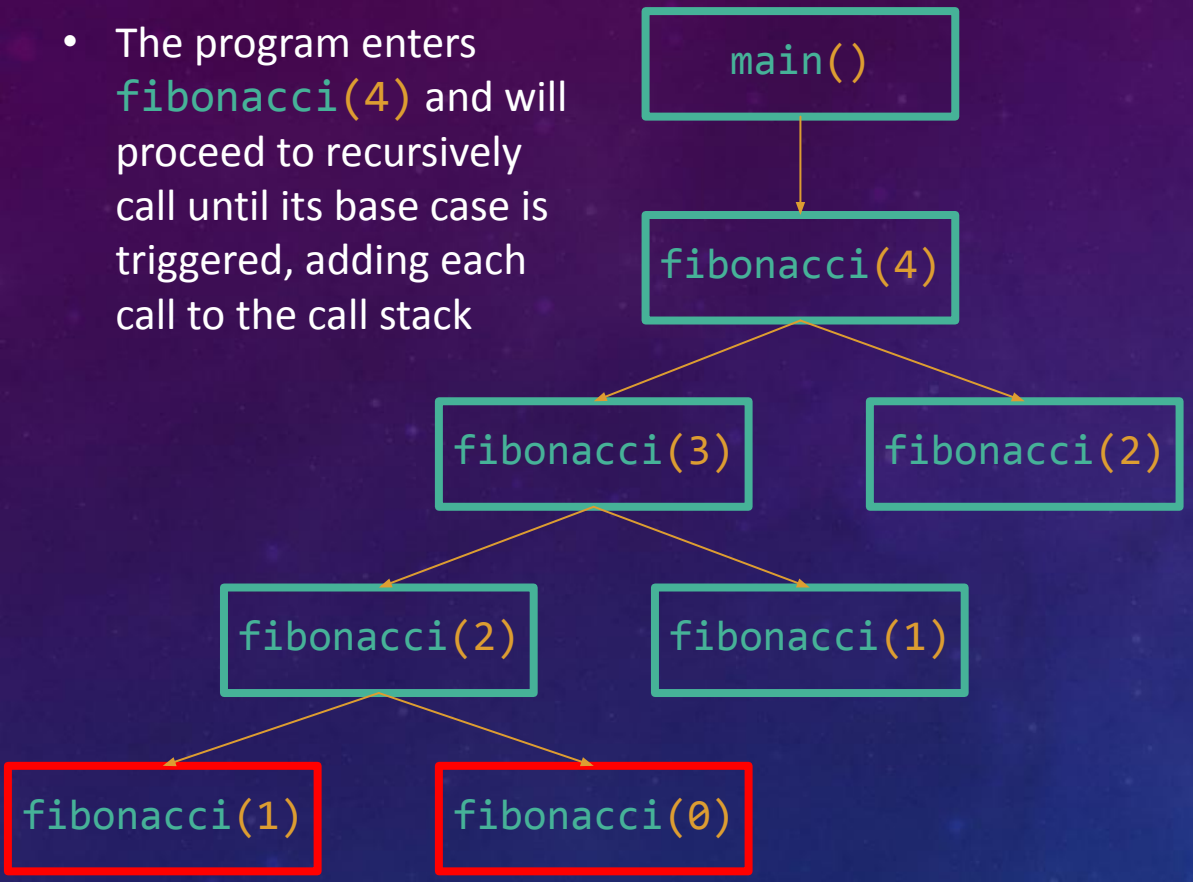



```
Return (fibonacci(2) + fibonacci(1)) + fibonacci(2);
```

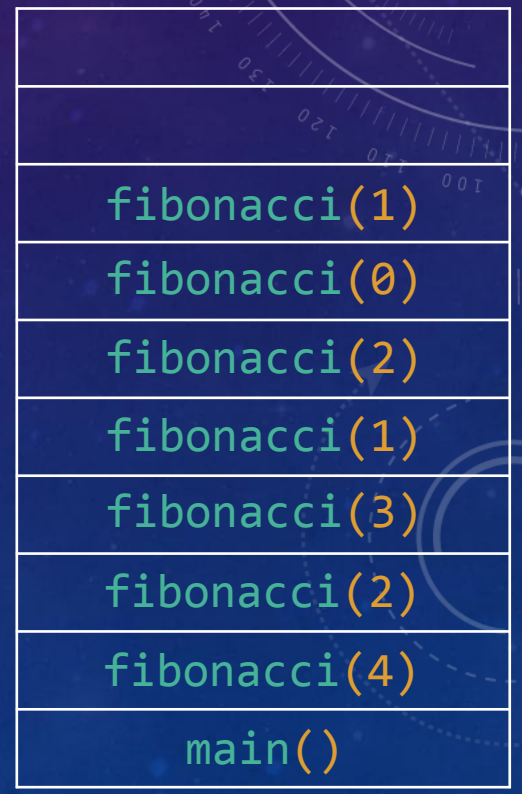
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The program enters `fibonacci(4)` and will proceed to recursively call until its base case is triggered, adding each call to the call stack



Call Stack

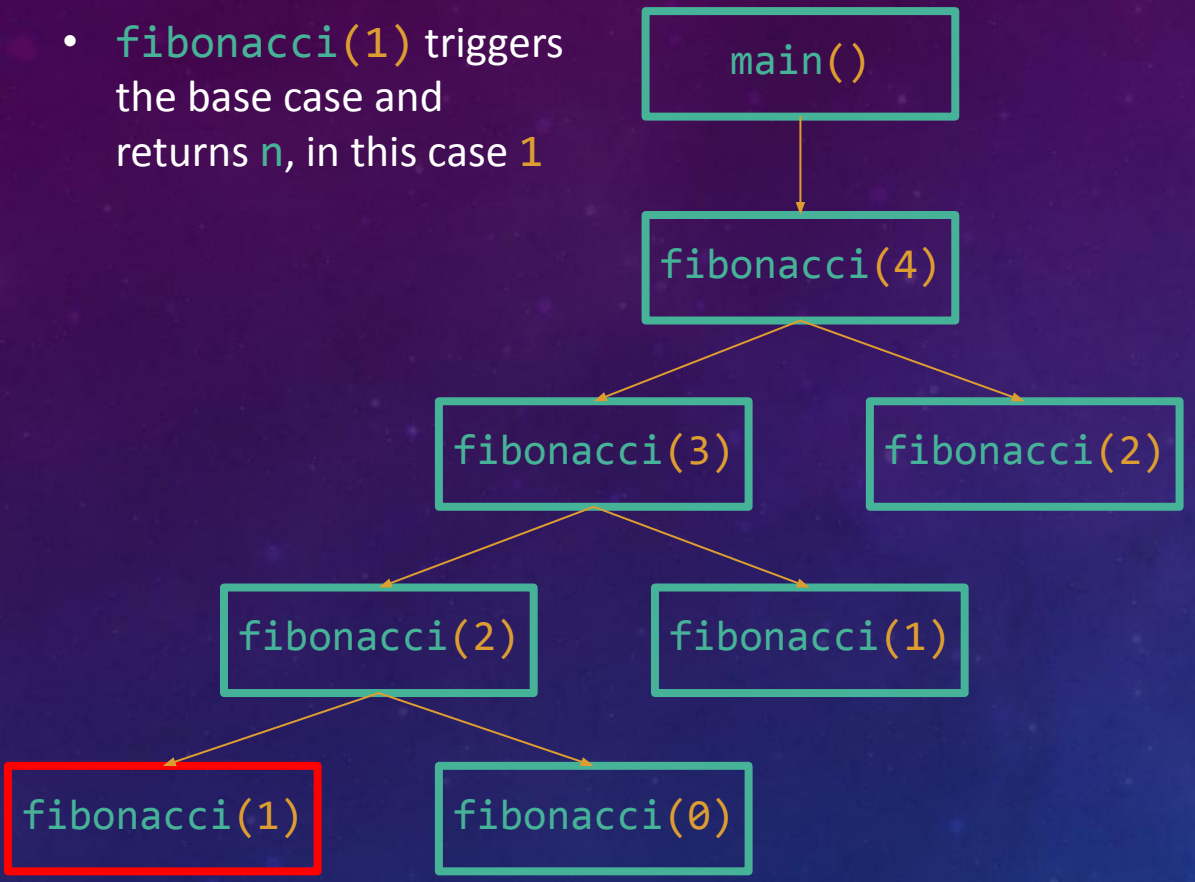



```
Return ((fibonacci(1) + fibonacci(0)) + fibonacci(1)) + fibonacci(2);
```

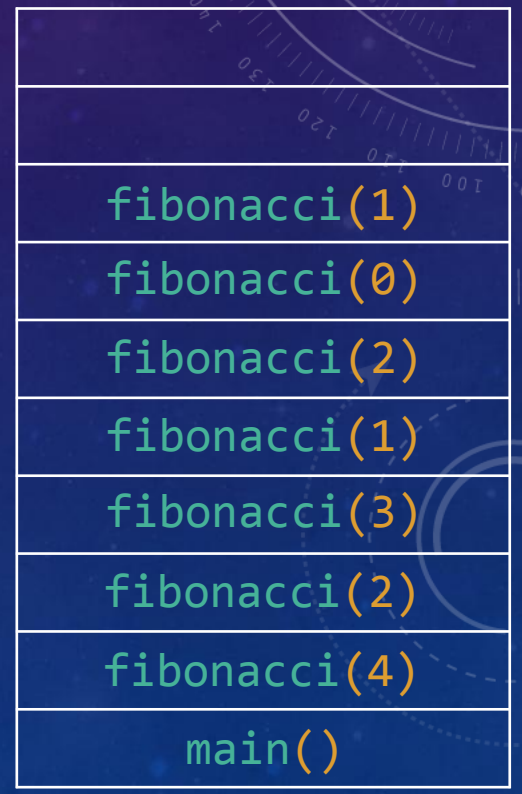
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`



Call Stack

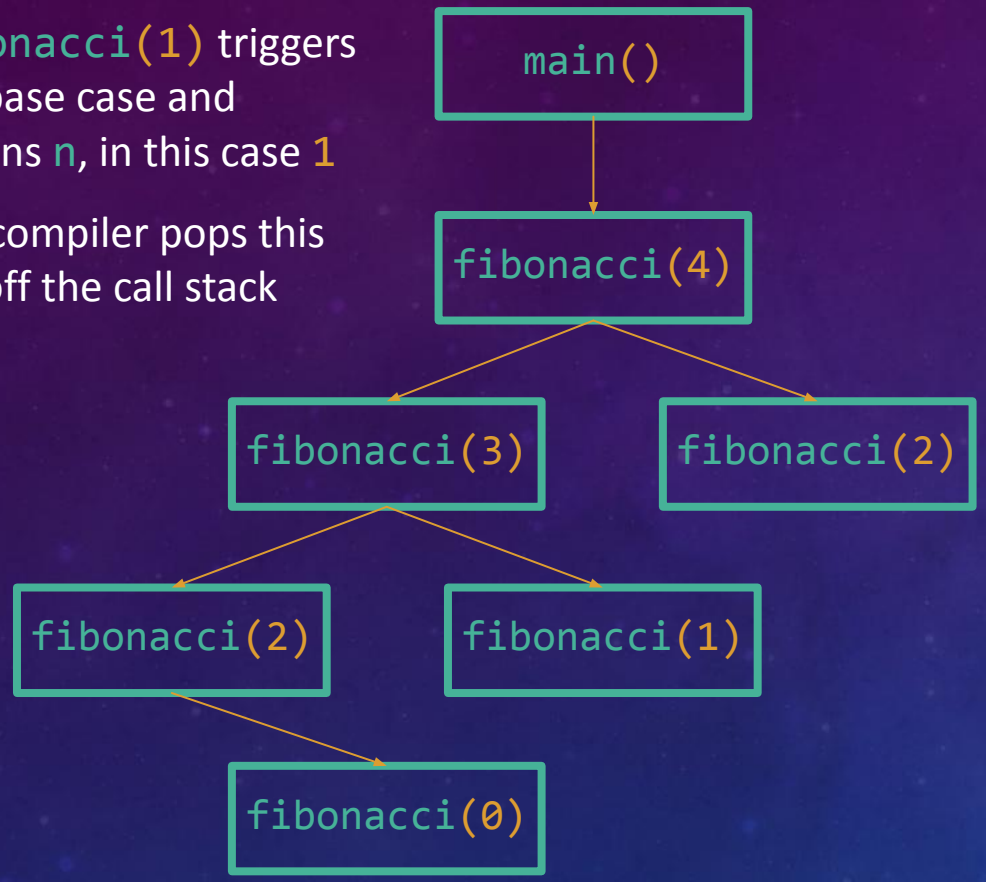



```
Return ((1 + fibonacci(0)) + fibonacci(1)) + fibonacci(2);
```

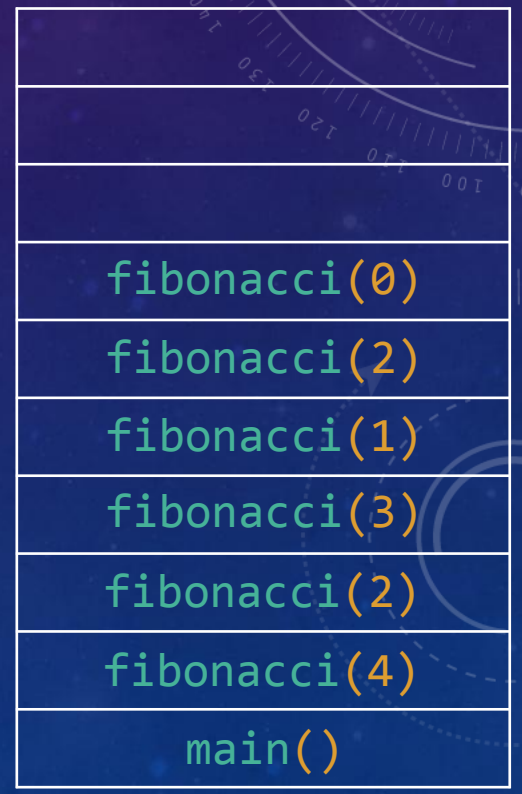
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`
- The compiler pops this call off the call stack



Call Stack

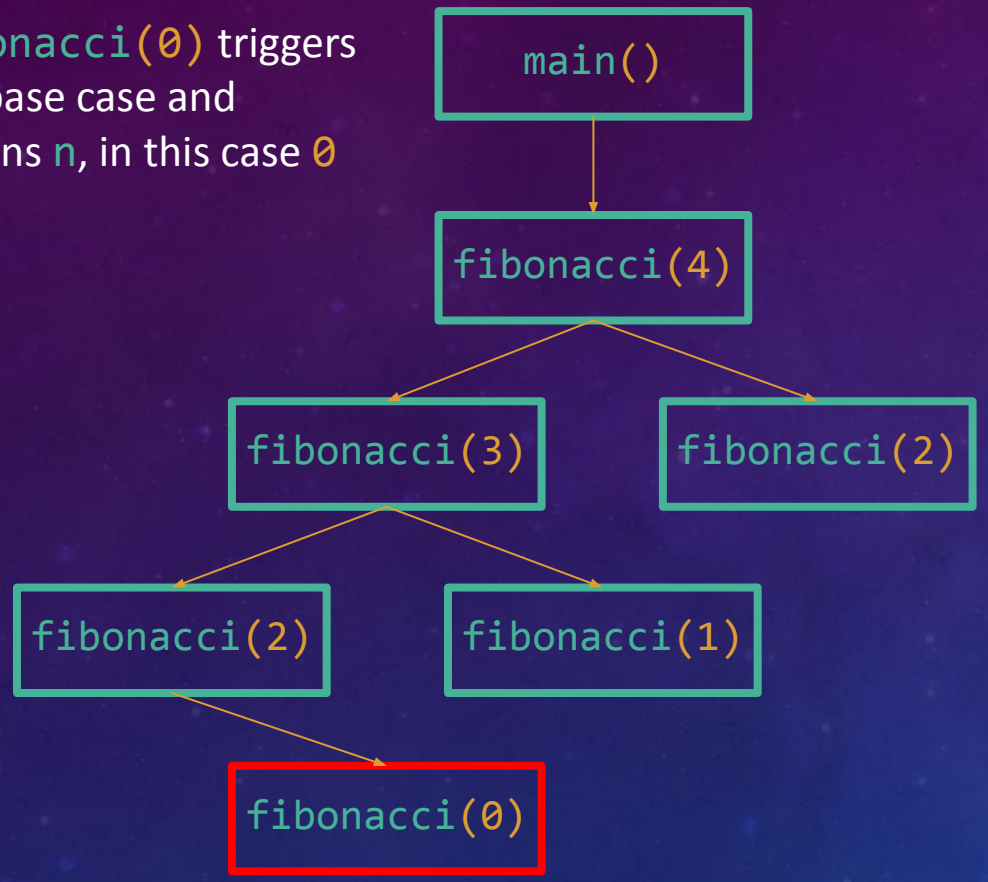



```
Return ((1 + fibonacci(0)) + fibonacci(1)) + fibonacci(2);
```

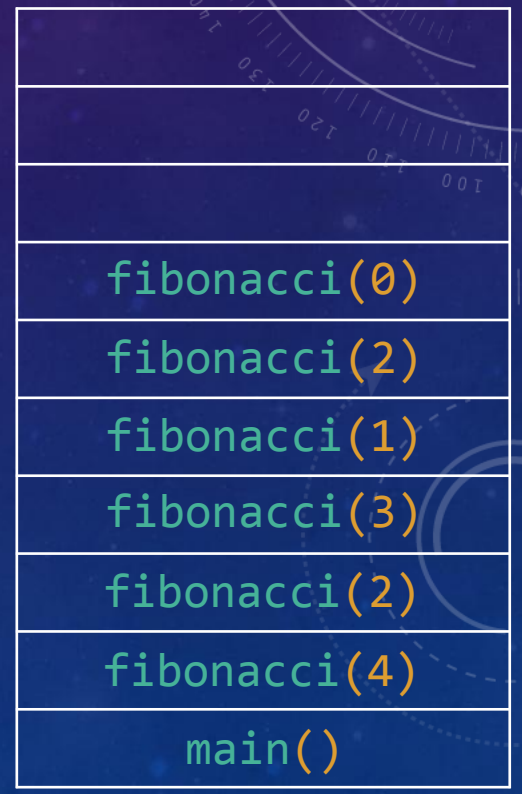
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(0)` triggers the base case and returns `n`, in this case `0`



Call Stack

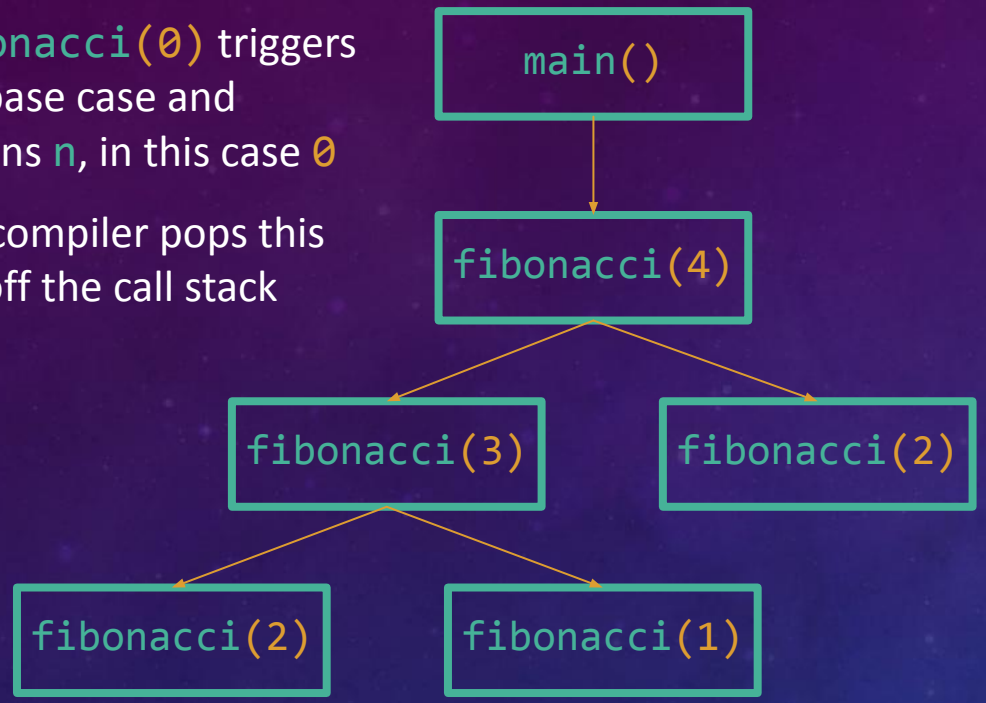



```
Return ((1 + 0) + fibonacci(1)) + fibonacci(2);
```

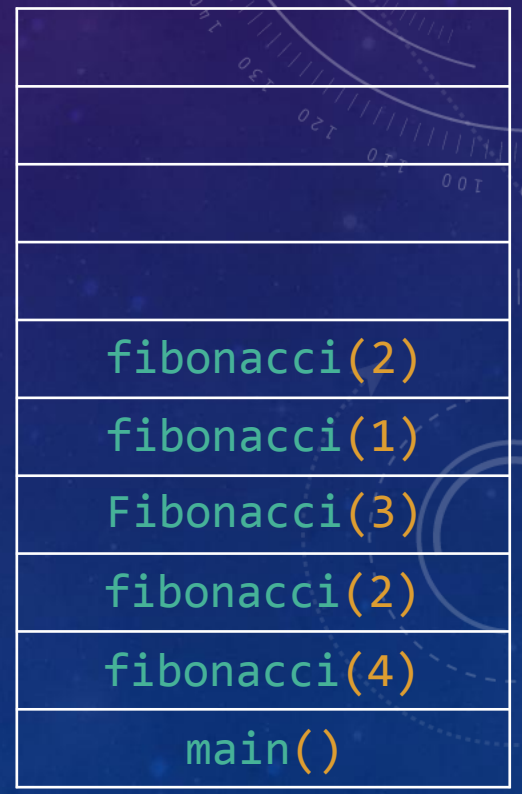
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(0)` triggers the base case and returns `n`, in this case `0`
- The compiler pops this call off the call stack



Call Stack

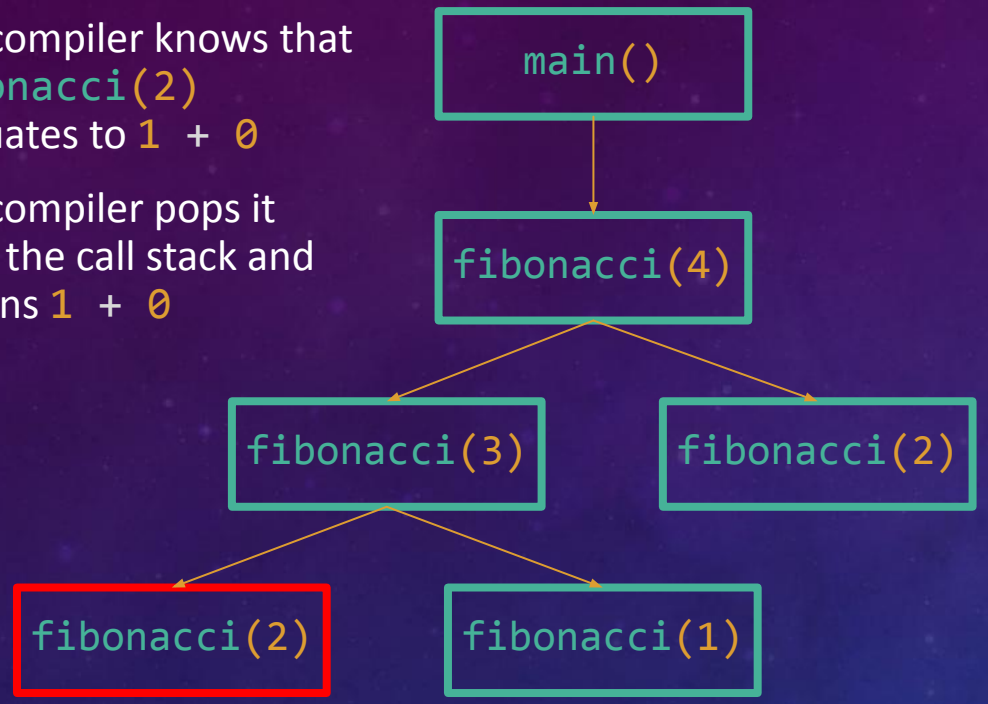



```
Return ((1 + 0) + fibonacci(1)) + fibonacci(2);
```

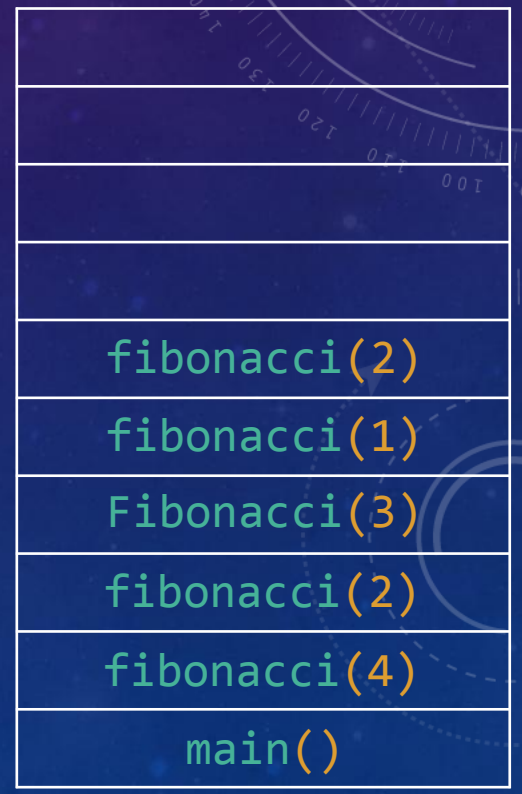
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(2)` evaluates to `1 + 0`
- The compiler pops it from the call stack and returns `1 + 0`



Call Stack

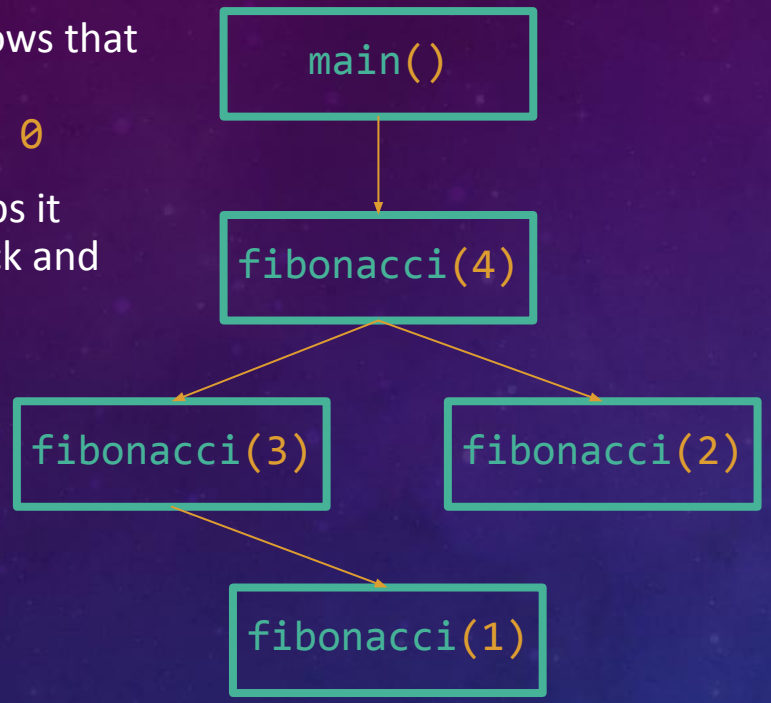



```
Return ((1 + 0) + fibonacci(1)) + fibonacci(2);
```

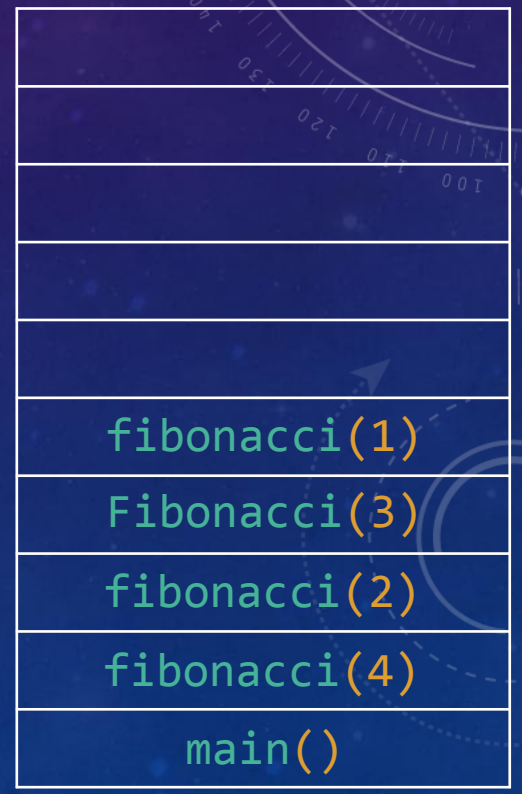
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(2)` evaluates to `1 + 0`
- The compiler pops it from the call stack and returns `1 + 0`



Call Stack

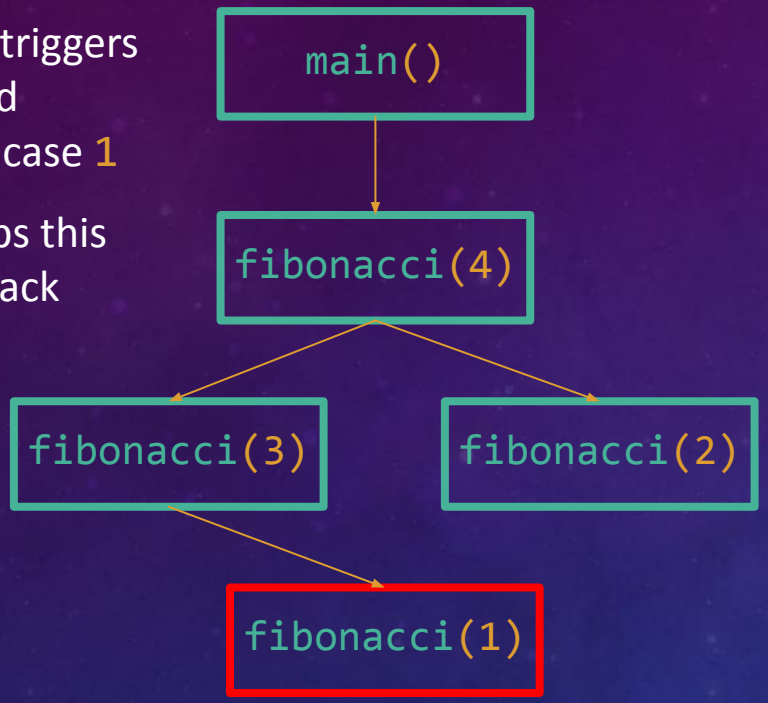



```
Return (1 + fibonacci(1)) + fibonacci(2);
```

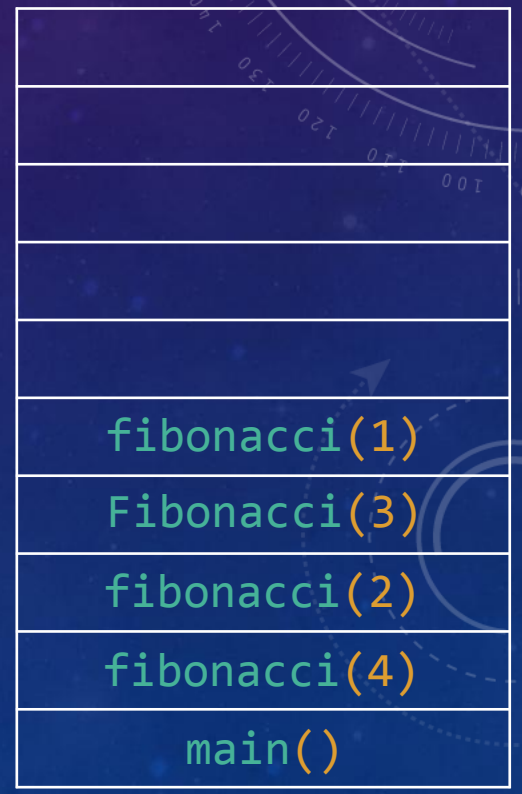
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`
- The compiler pops this call off the call stack



Call Stack

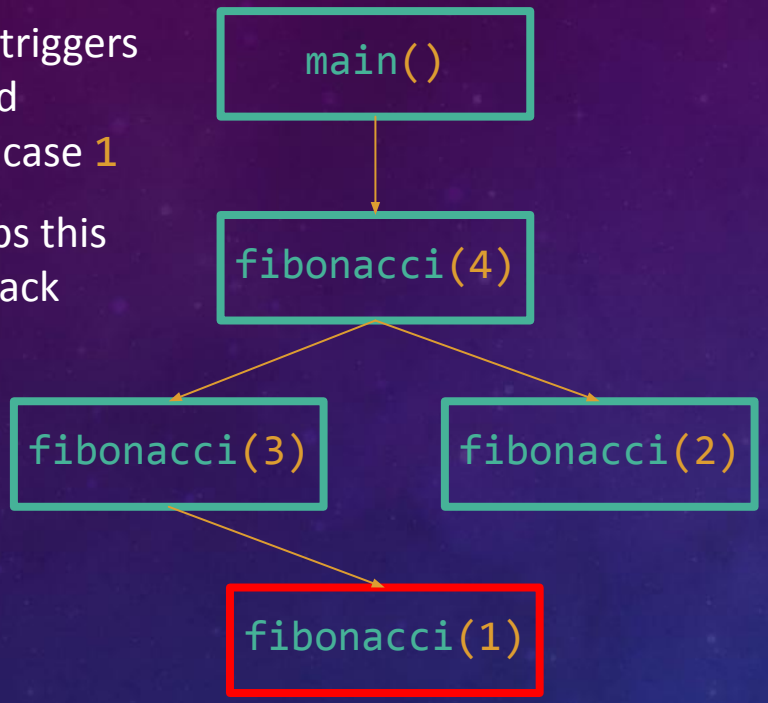



```
Return (1 + fibonacci(1)) + fibonacci(2);
```

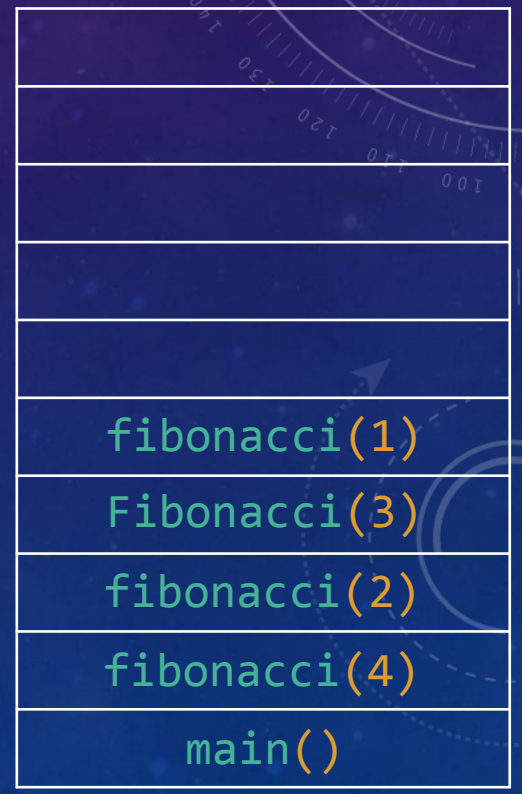
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`
- The compiler pops this call off the call stack



Call Stack

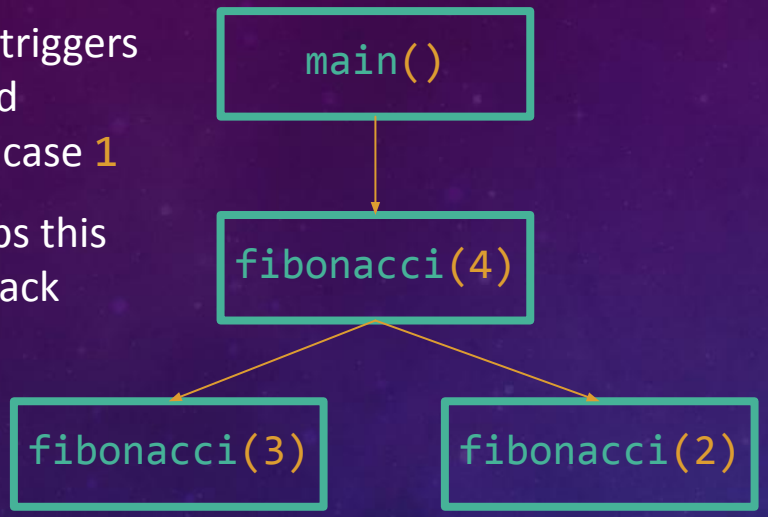



```
Return (1 + 1) + fibonacci(2);
```

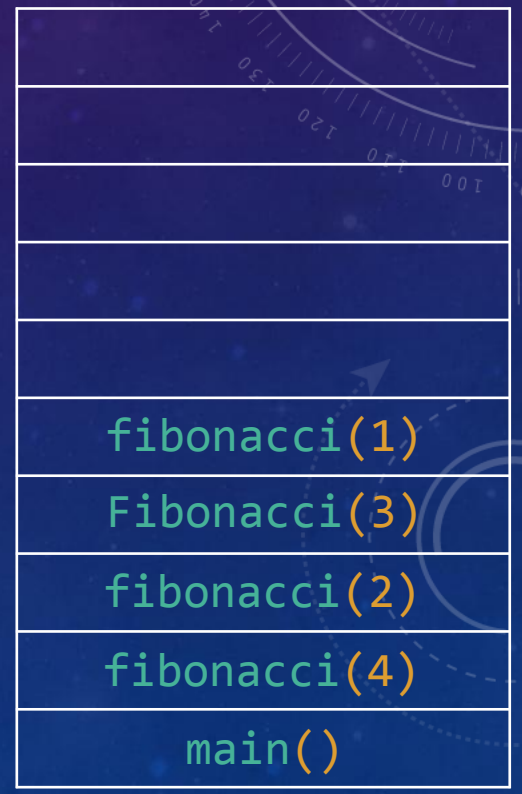
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`
- The compiler pops this call off the call stack



Call Stack

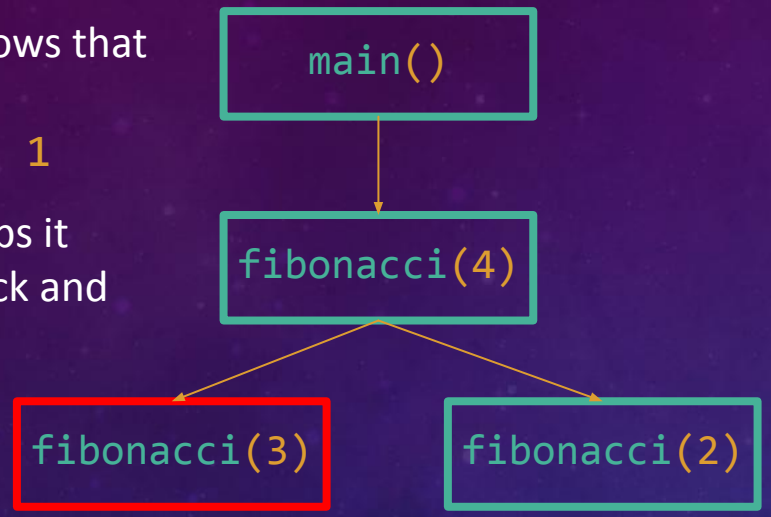



```
Return (1 + 1) + fibonacci(2);
```

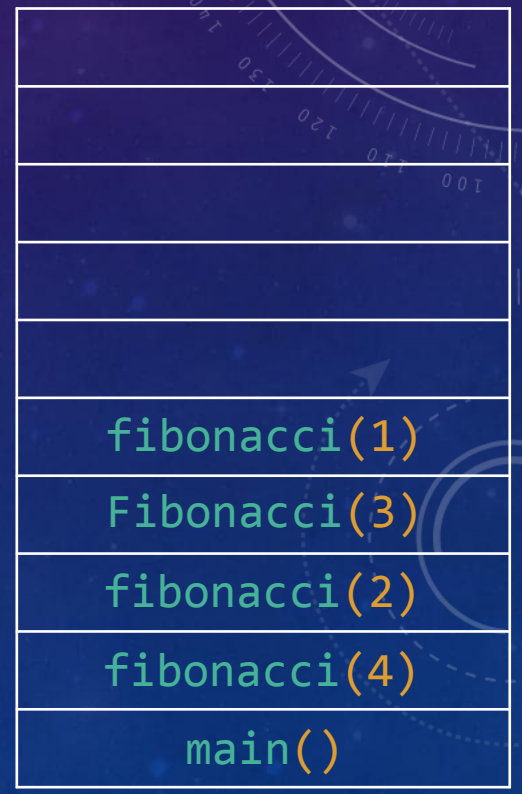
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(3)` evaluates to `1 + 1`
- The compiler pops it from the call stack and returns `1 + 1`



Call Stack

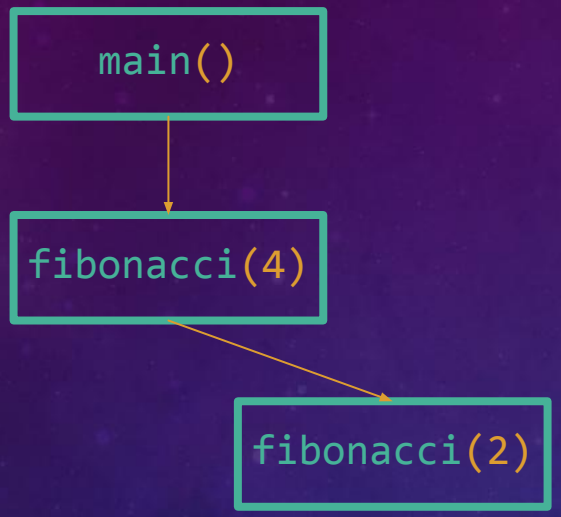



```
Return (1 + 1) + fibonacci(2);
```

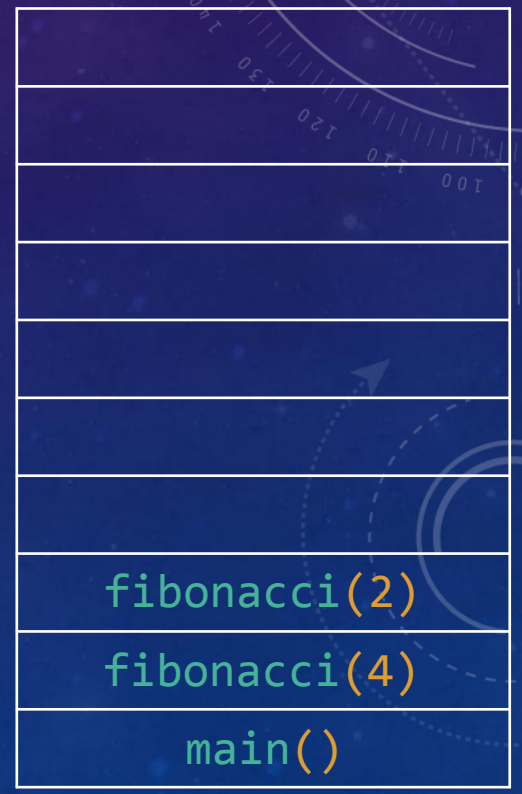
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(3)` evaluates to `1 + 1`
- The compiler pops it from the call stack and returns `1 + 1`



Call Stack

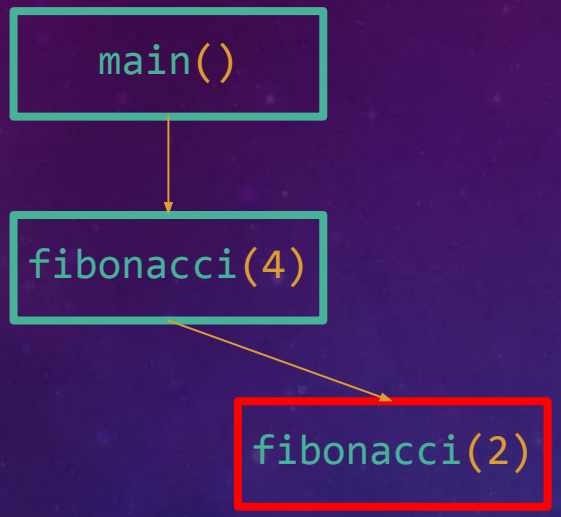



```
return 2 + fibonacci(2);
```

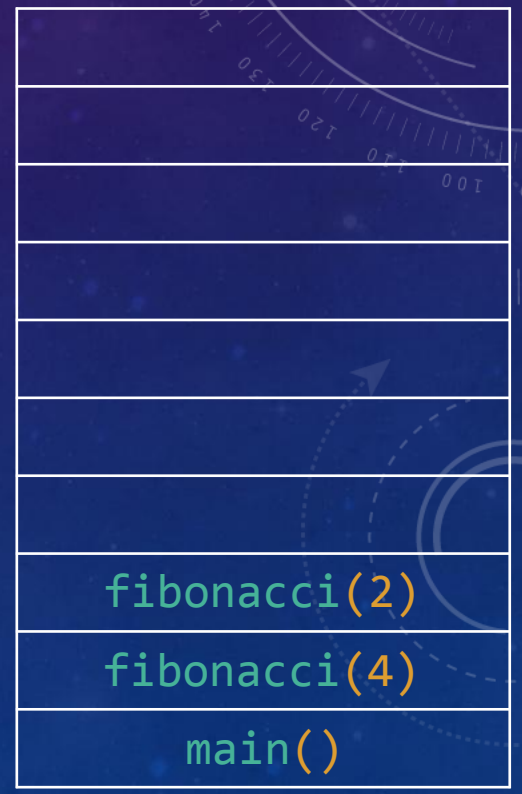
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(2)` doesn't trigger the base case, and further gets recursively called



Call Stack

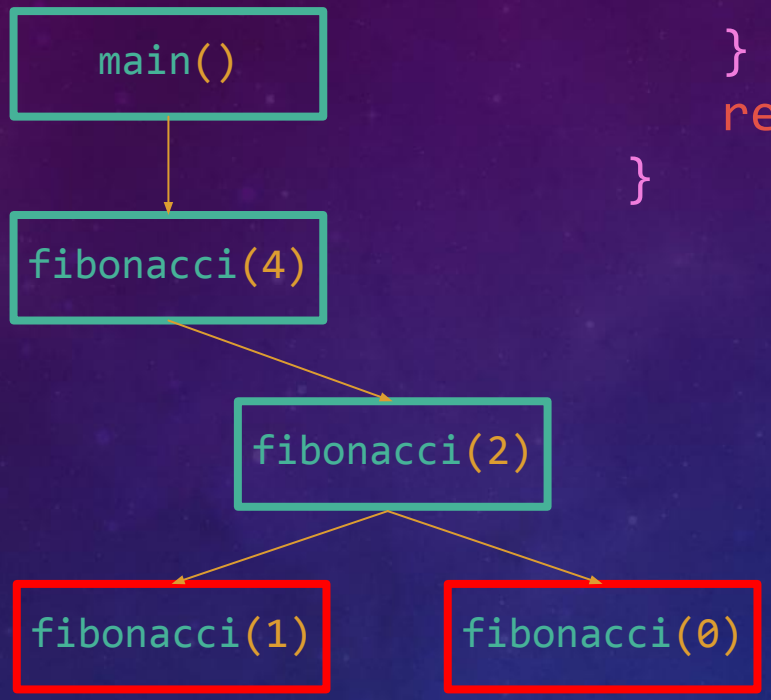



```
return 2 + fibonacci(2);
```

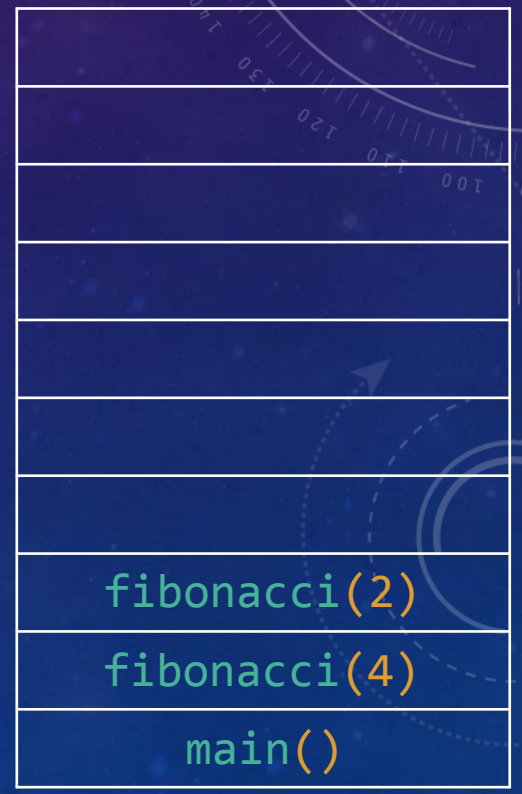
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(2)` doesn't trigger the base case, and further gets recursively called
- These get pushed to the call stack



Call Stack

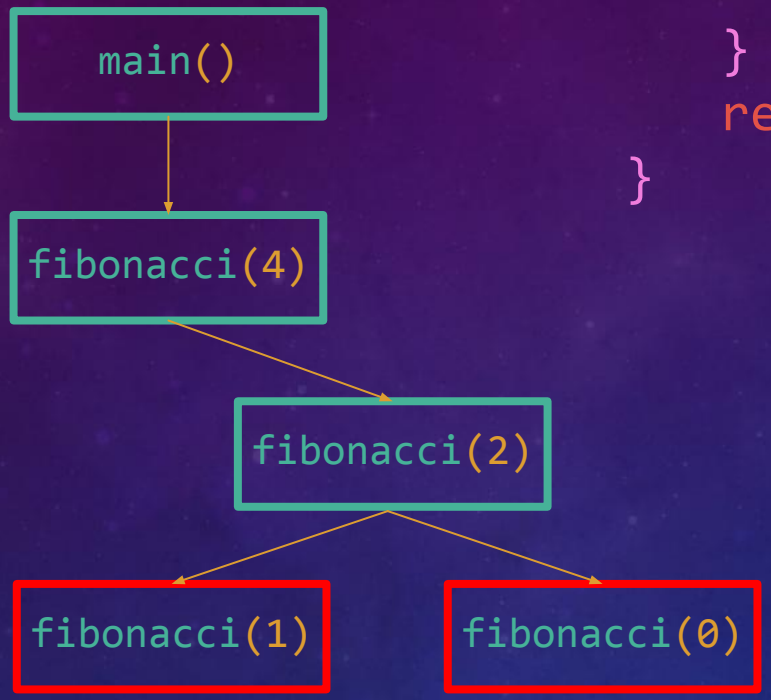



```
return 2 + fibonacci(2);
```

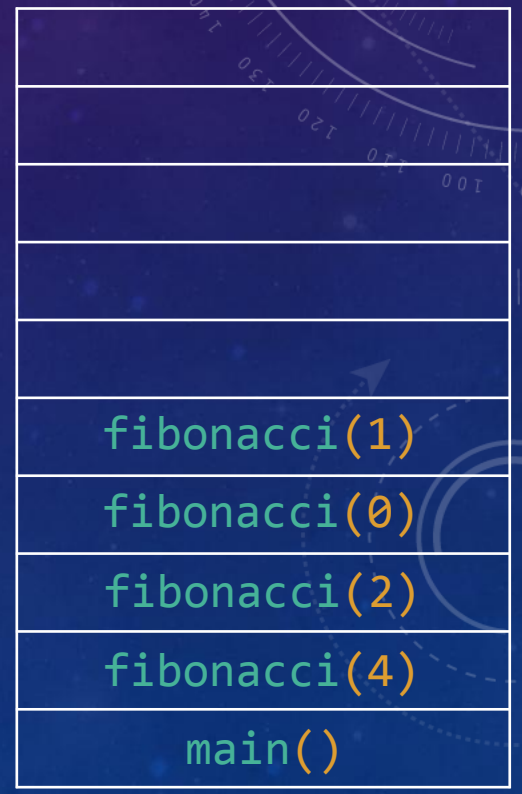
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(2)` doesn't trigger the base case, and further gets recursively called
- These get pushed to the call stack



Call Stack

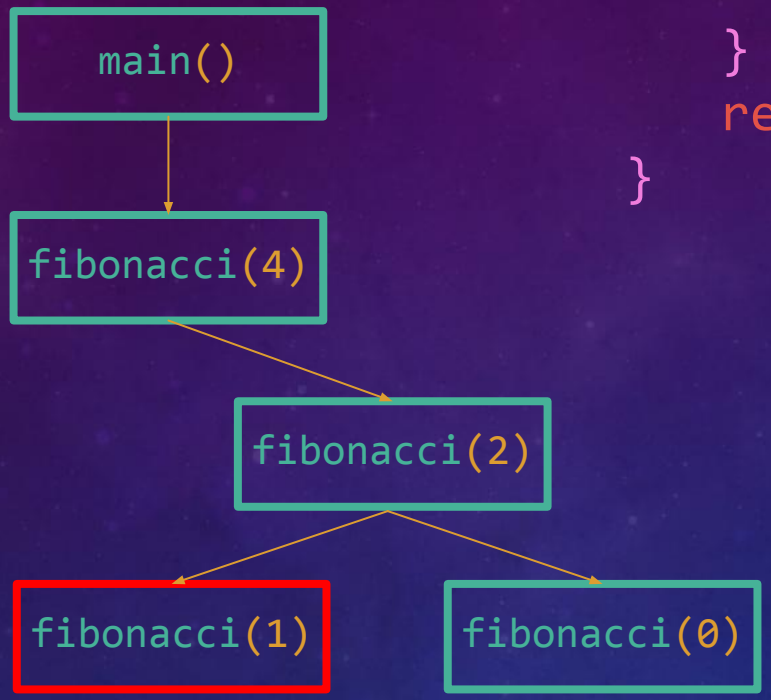



```
return 2 + (fibonacci(1) + fibonacci(0);
```

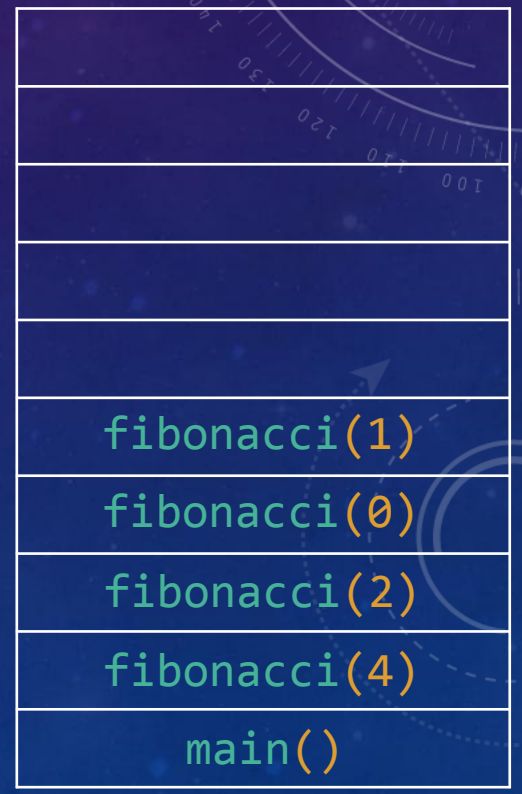
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`



Call Stack

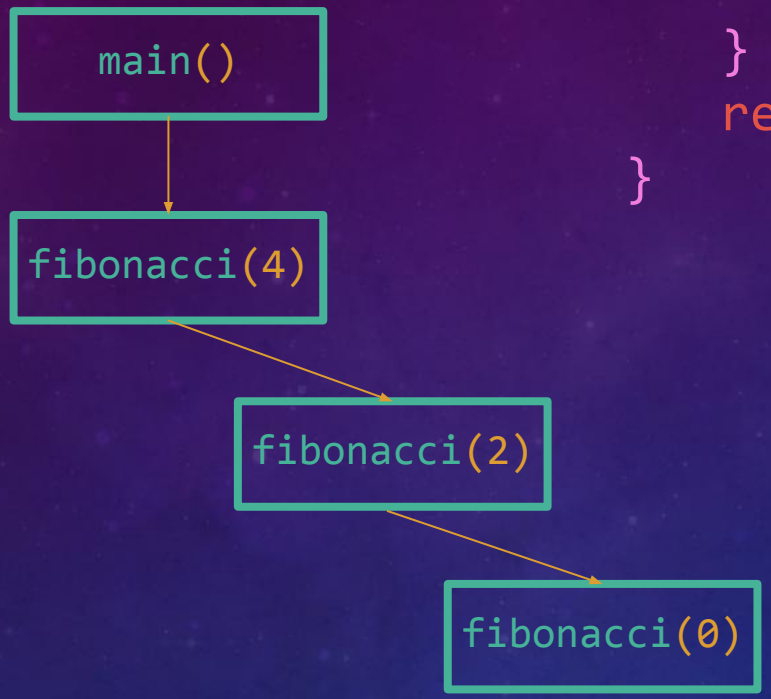



```
return 2 + (fibonacci(1) + fibonacci(0);
```

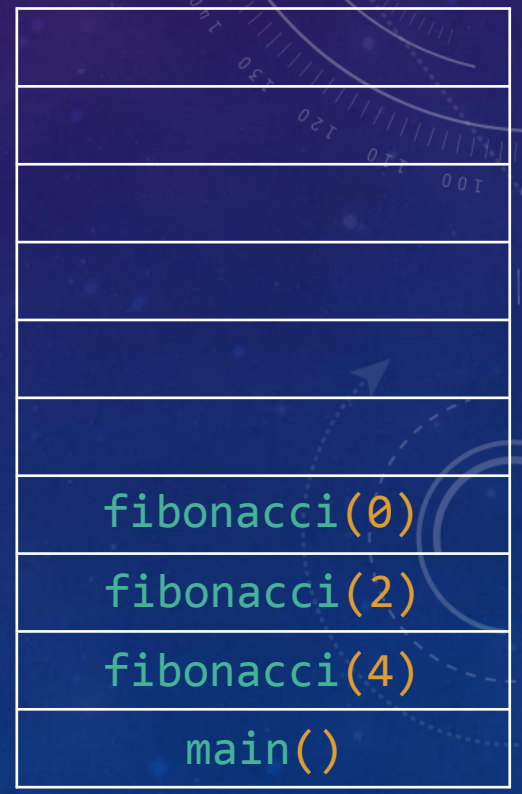
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(1)` triggers the base case and returns `n`, in this case `1`
- The compiler pops this call off the call stack



Call Stack

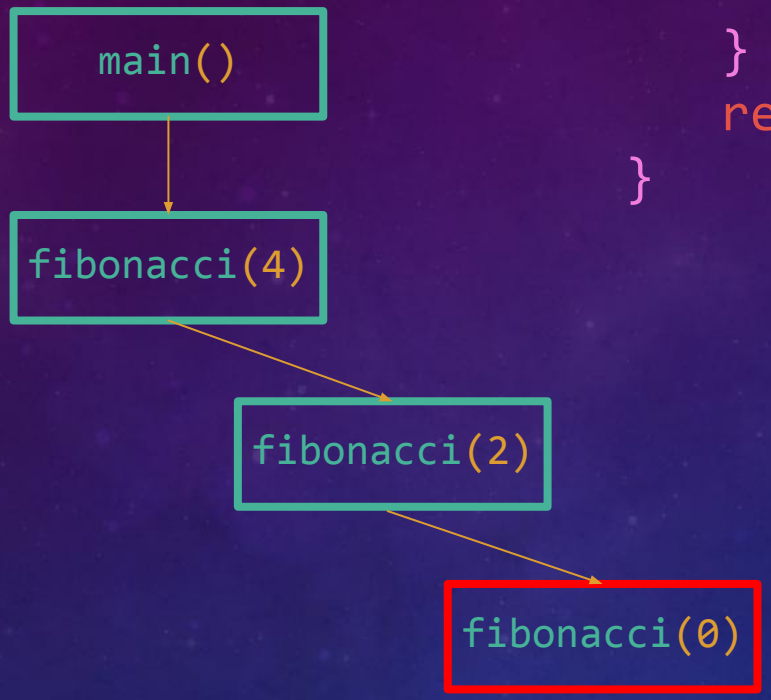



```
return 2 + (1 + fibonacci(0);
```

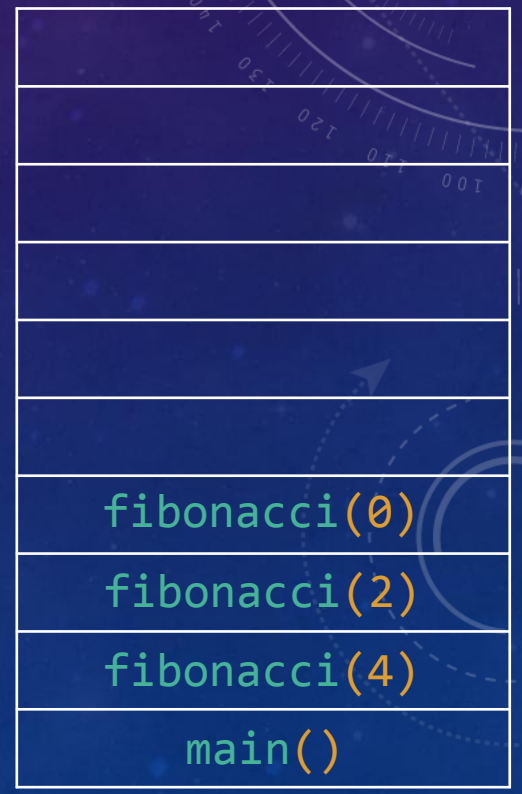
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(0)` triggers the base case and returns `n`, in this case `0`



Call Stack

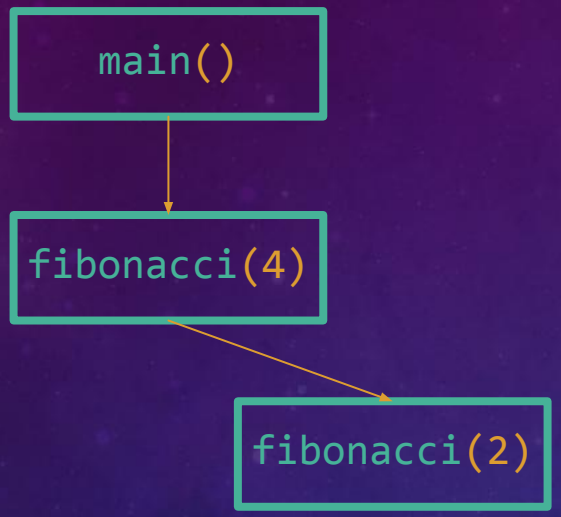



```
return 2 + (1 + fibonacci(0);
```

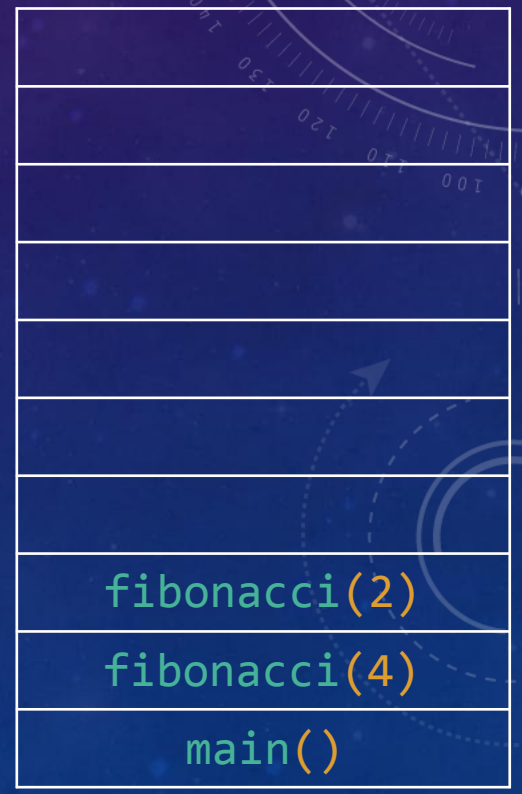
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- `fibonacci(0)` triggers the base case and returns `n`, in this case `0`
- The compiler pops this call off the call stack



Call Stack

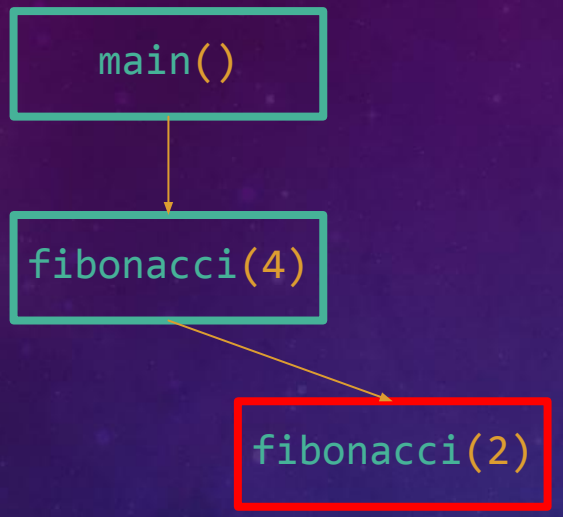



```
return 2 + (1 + fibonacci(0);
```

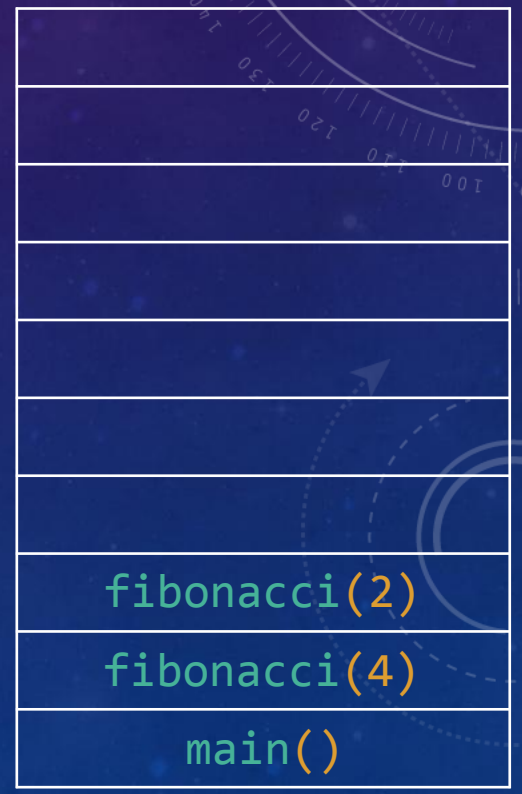
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(2)` evaluates to `1 + 0`
- The compiler pops it from the call stack and returns `1 + 0`



Call Stack

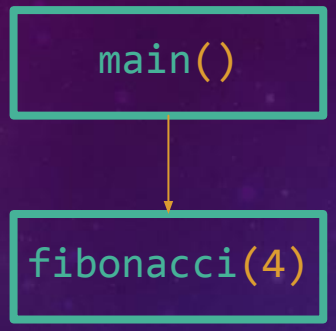



```
return 2 + (1 + 0);
```

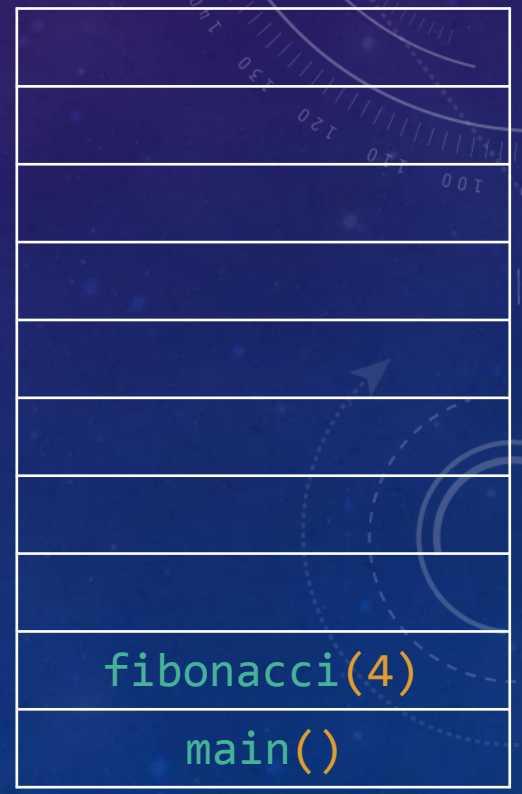
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(2)` evaluates to `1 + 0`
- The compiler pops it from the call stack and returns `1 + 0`



Call Stack

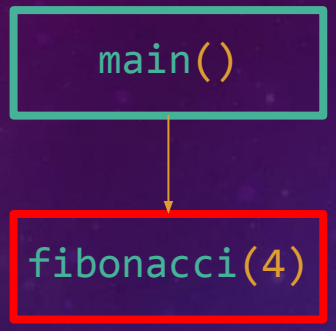



```
return 2 + 1;
```

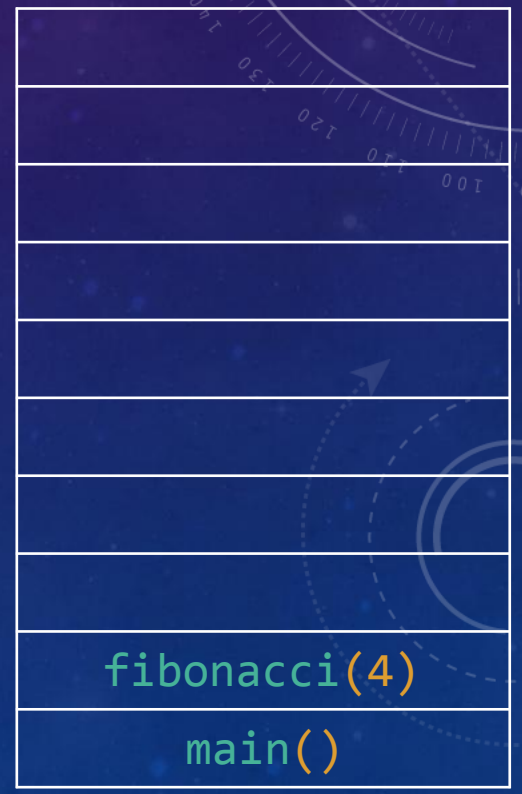
Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(4)` evaluates to `2 + 1`
- The compiler pops it from the call stack and returns `2 + 1`



Call Stack




```
return 2 + 1;
```

Recursion with Fibonacci

```
int fibonacci(int n) {
    if(n == 0 n || n == 1) {
        return n;
    }
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

- The compiler knows that `fibonacci(4)` evaluates to `2 + 1`
- The compiler pops it from the call stack and returns `2 + 1`

`main()`

Call Stack


```
return 3;
```

Give “Fibonacci numbers” and try and implement a solution to the problem!

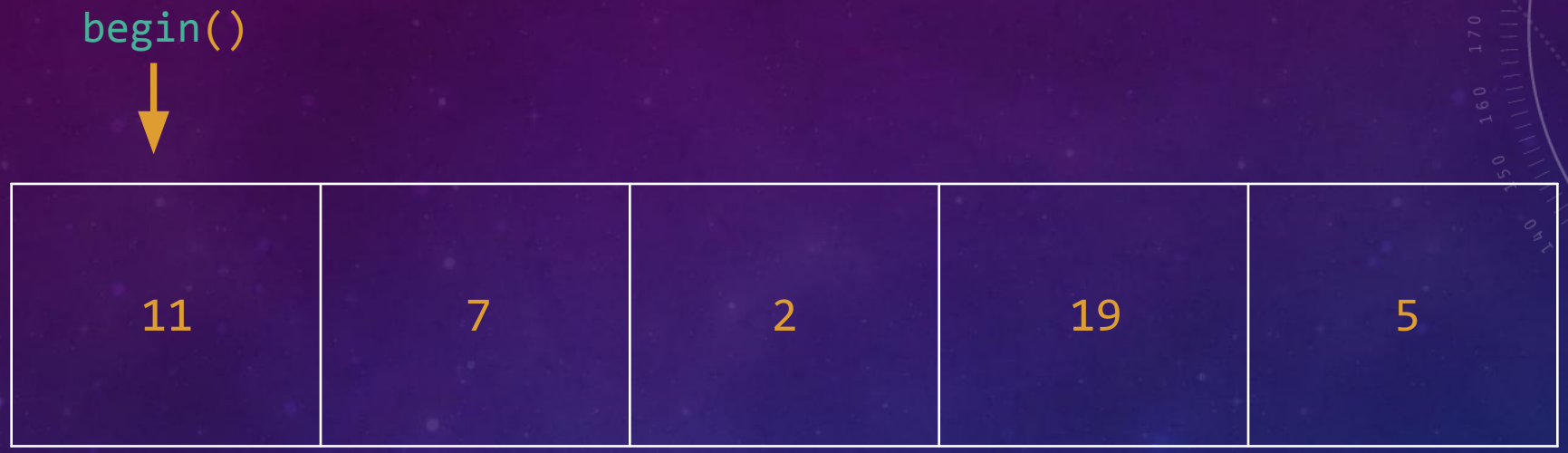
Adding iterators to MyVector

- An iterator is essentially a high-level abstraction of a pointer, that you can you use to traverse through a container like a vector or linked list

Adding iterators to MyVector

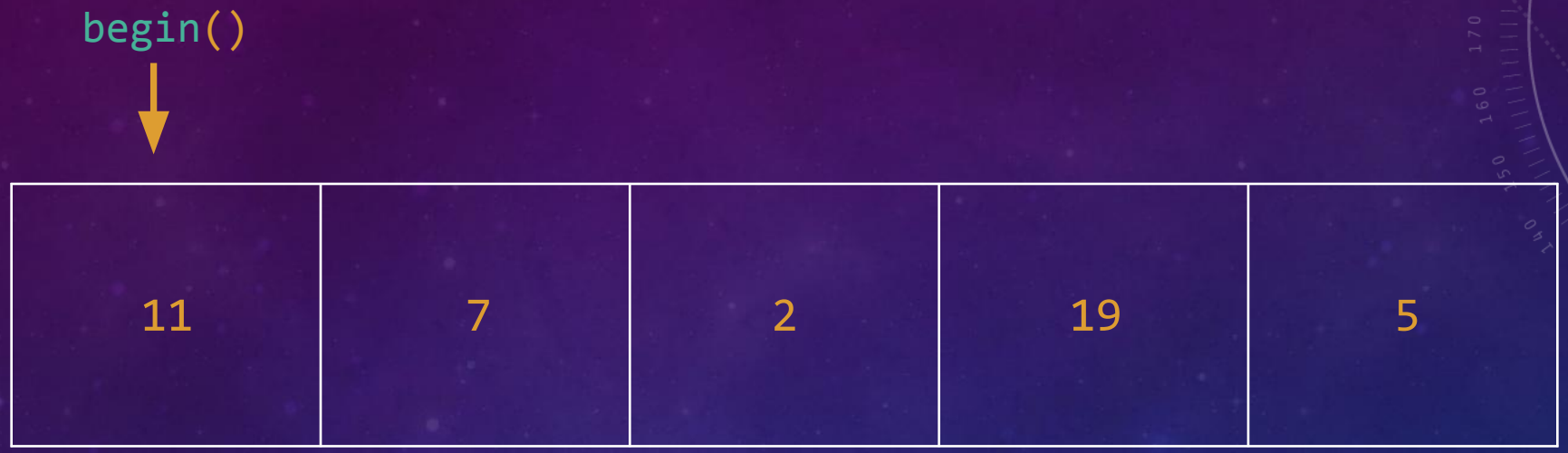
- An iterator is essentially a high-level abstraction of a pointer, that you can you use to traverse through a container like a vector or linked list
- We have looked briefly at the common operations that are in most C++ containers like `begin()` and `end()` which both will return an iterator

Adding iterators to MyVector



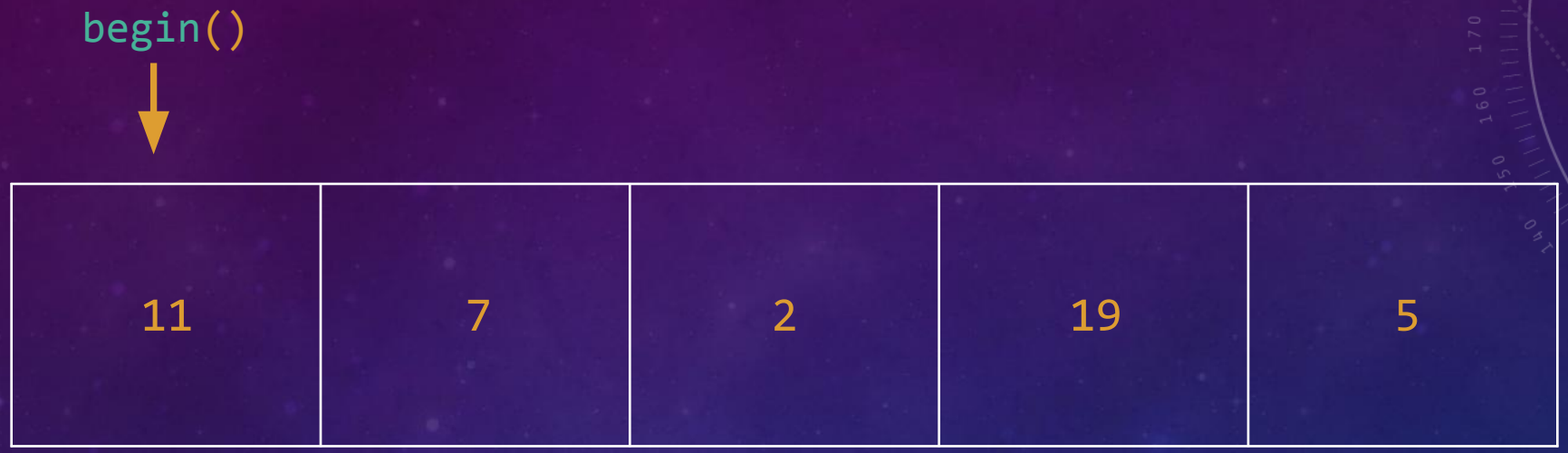
- An iterator is essentially a high-level abstraction of a pointer, that you can you use to traverse through a container like a vector or linked list
- We have looked briefly at the common operations that are in most C++ containers like `begin()` and `end()` which both will return an iterator
- `begin()` will return an iterator that points to the first element in this container

Adding iterators to MyVector



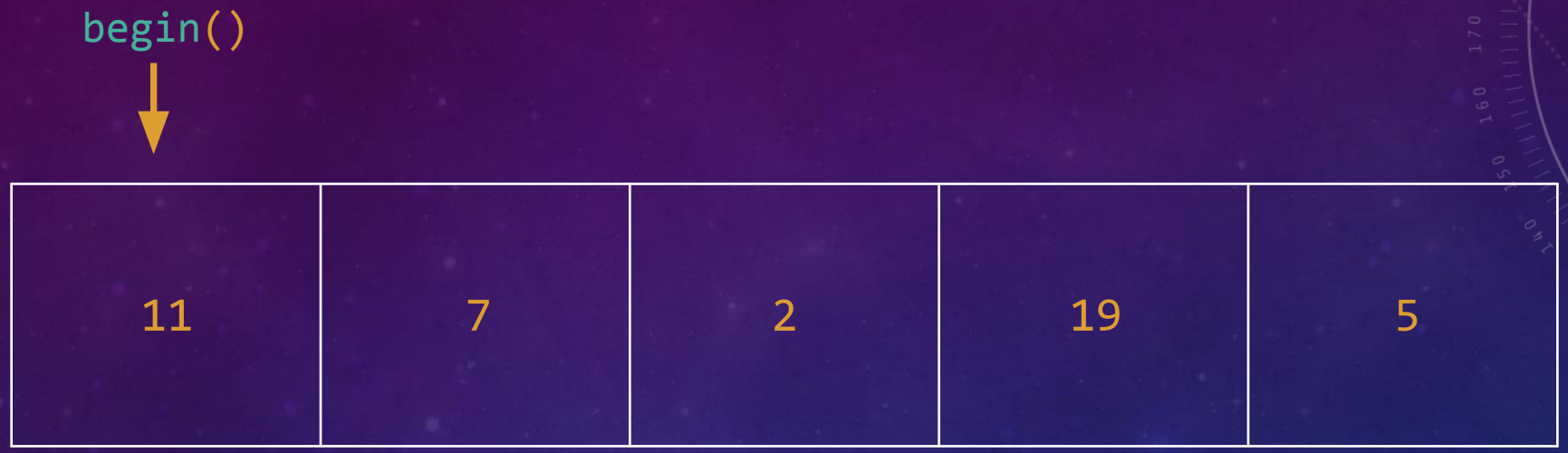
- An iterator is essentially a high-level abstraction of a pointer, that you can you use to traverse through a container like a vector or linked list
- We have looked briefly at the common operations that are in most C++ container like `begin()` and `end()` which both will return an iterator
- `begin()` will return an iterator that points to the first element in this container
- Whilst `end()` will return an iterator that points one past the last element in the container

Adding iterators to MyVector



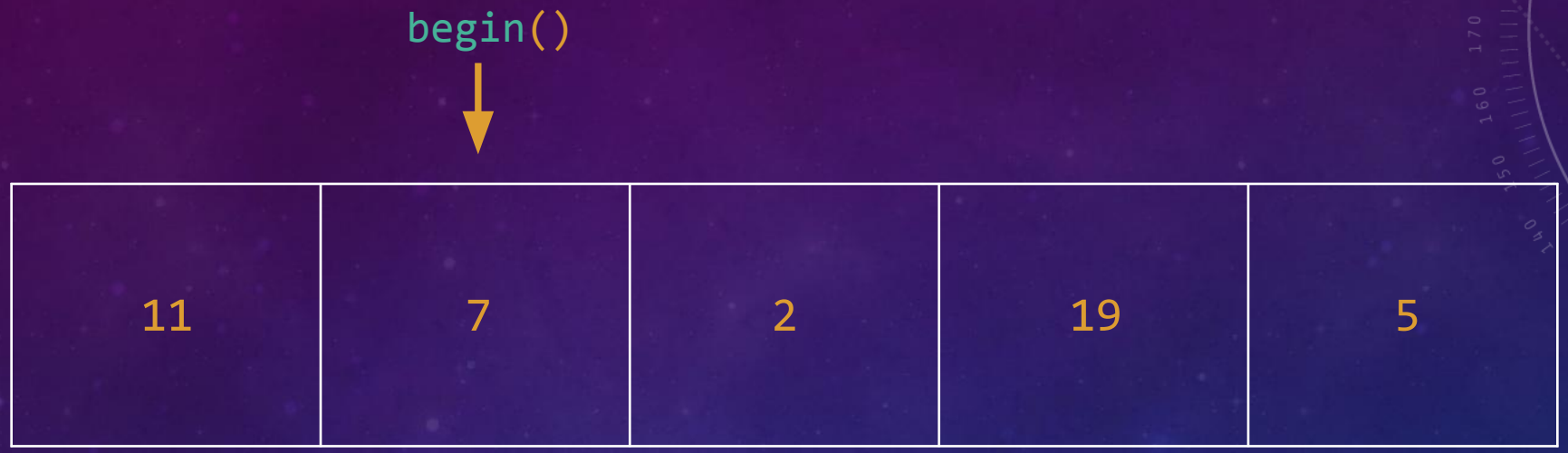
- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator

Adding iterators to MyVector



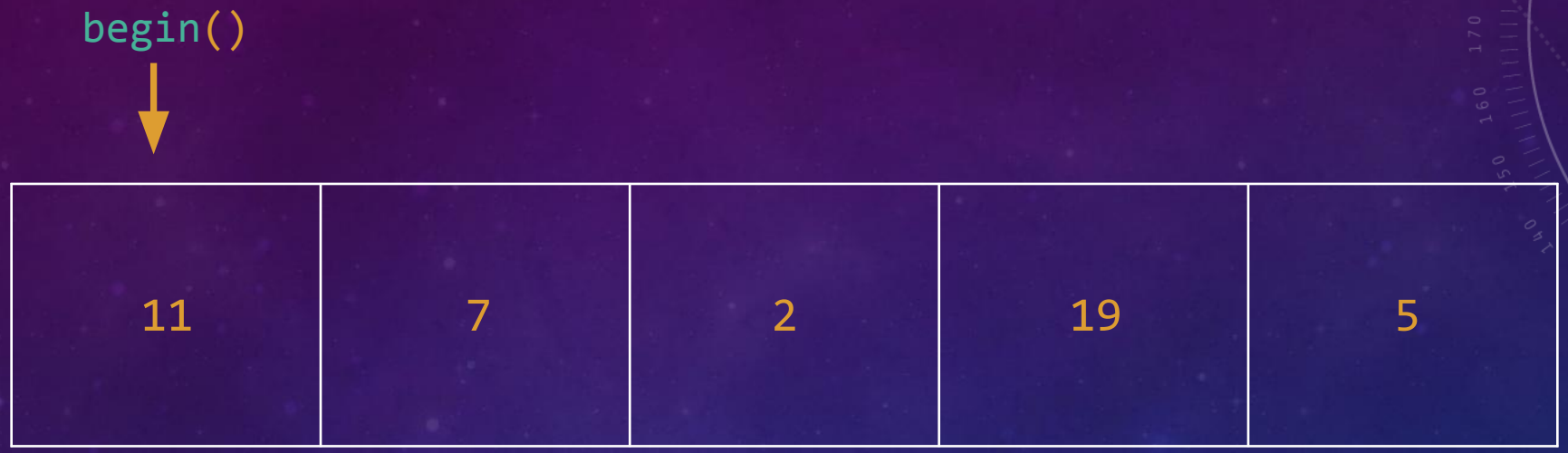
- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator
- You will also need to implement the operator functions to…

Adding iterators to MyVector



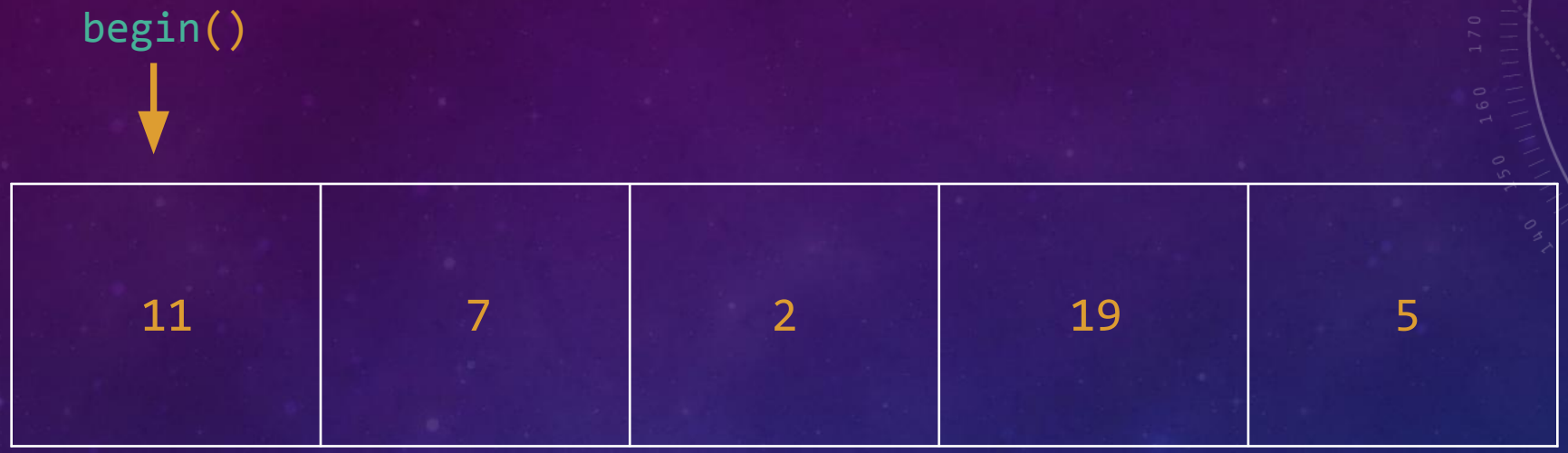
- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator
- You will also need to implement the operator functions to… increment the iterator…

Adding iterators to MyVector



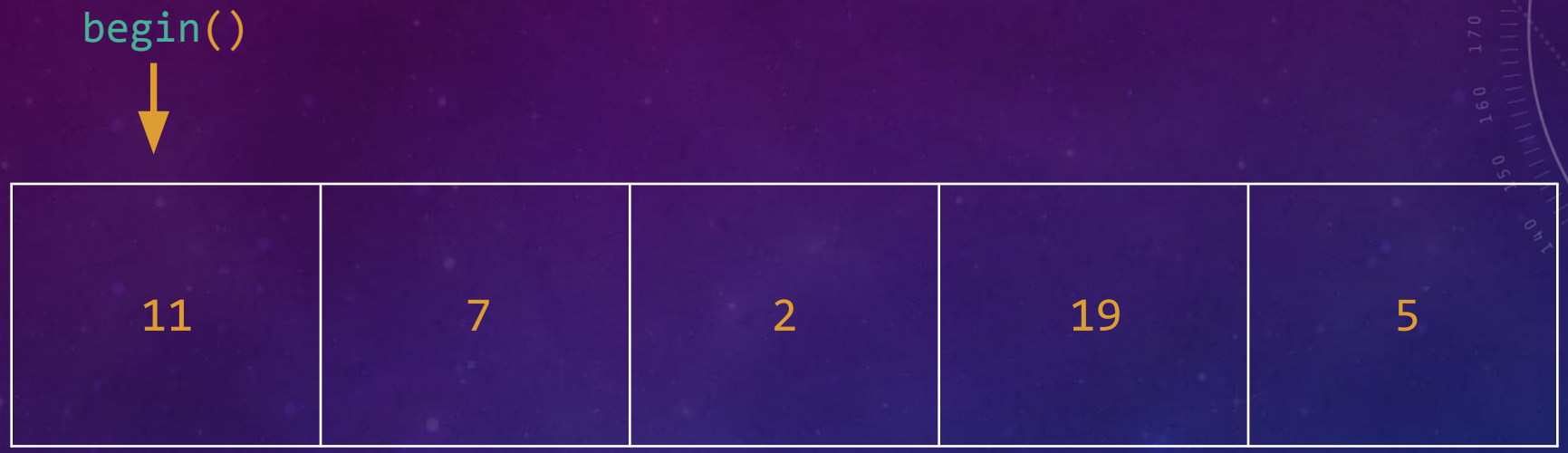
- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator
- You will also need to implement the operator functions to… increment the iterator… and decrement the iterator to allow you to traverse through it

Adding iterators to MyVector



- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator
- You will also need to implement the operator functions to… increment the iterator… and decrement the iterator to allow you to traverse through it
- As well as the operator function to use `*` to de-reference your iterator to get the value stored. So when you use `*(MyVector.begin())` it will return the value `12`

Adding iterators to MyVector



- The goal for this exercise is to implement the methods to return both the `begin()` and `end()` iterators of MyVector, as well as the constructor for Iterator
- You will also need to implement the operator functions to… increment the iterator… and decrement the iterator to allow you to traverse through it
- As well as the operator function to use `*` to de-reference your iterator to get the value stored. So when you use `*(MyVector.begin())` it will return the value `11`
- This will be extremely useful when completing your first assignment, which was released yesterday

Give “Adding iterators to MyVector” a go to try and implement what we need!

Access to google drive

- I will upload slides to the Google Drive after every class
- https://drive.google.com/drive/folders/1H5psebndM_YVyoJE-BJ_ODNJOfgq9-uI

Contact: Thomas.golding@uts.edu.au