Nitish Dhinaharan

005036295

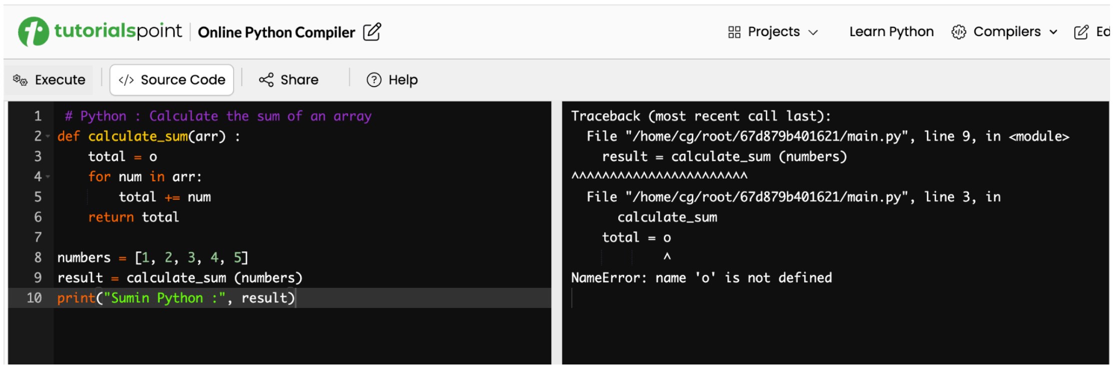
Assignment 2: Syntax, Semantics, and Memory Management

Github: https://github.com/ndhinaharan36295/MSCS-632_Assignment2

Part 1: Analyzing Syntax and Semantics

Section 1:

Python:



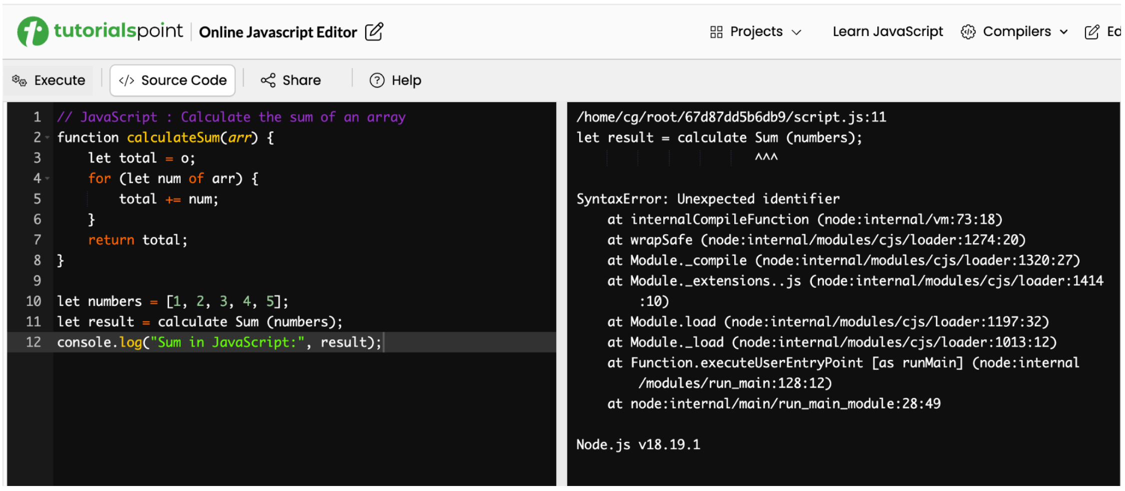
```python
# Python : Calculate the sum of an array
def calculate_sum(arr) :
    total = o
    for num in arr:
        total += num
    return total

numbers = [1, 2, 3, 4, 5]
result = calculate_sum (numbers)
print("Sumin Python :", result)
```

```
Traceback (most recent call last):
  File "/home/cg/root/67d879b401621/main.py", line 9, in <module>
    result = calculate_sum (numbers)
^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/cg/root/67d879b401621/main.py", line 3, in
      calculate_sum
    total = o
            ^
NameError: name 'o' is not defined
```

The given python code snippet produces a syntax error - **NameError: name 'o' is not defined** in line #3 (total = o). This is because the character 'o' is not defined in scope here.

Python detects syntax errors at runtime. If the compiler runs into an invalid syntax while running the program, it stops execution (at the first occurrence of an error) and throws a syntax error

JavaScript:



```javascript
// JavaScript : Calculate the sum of an array
function calculateSum(arr) {
    let total = o;
    for (let num of arr) {
        total += num;
    }
    return total;
}

let numbers = [1, 2, 3, 4, 5];
let result = calculate Sum (numbers);
console.log("Sum in JavaScript:", result);
```

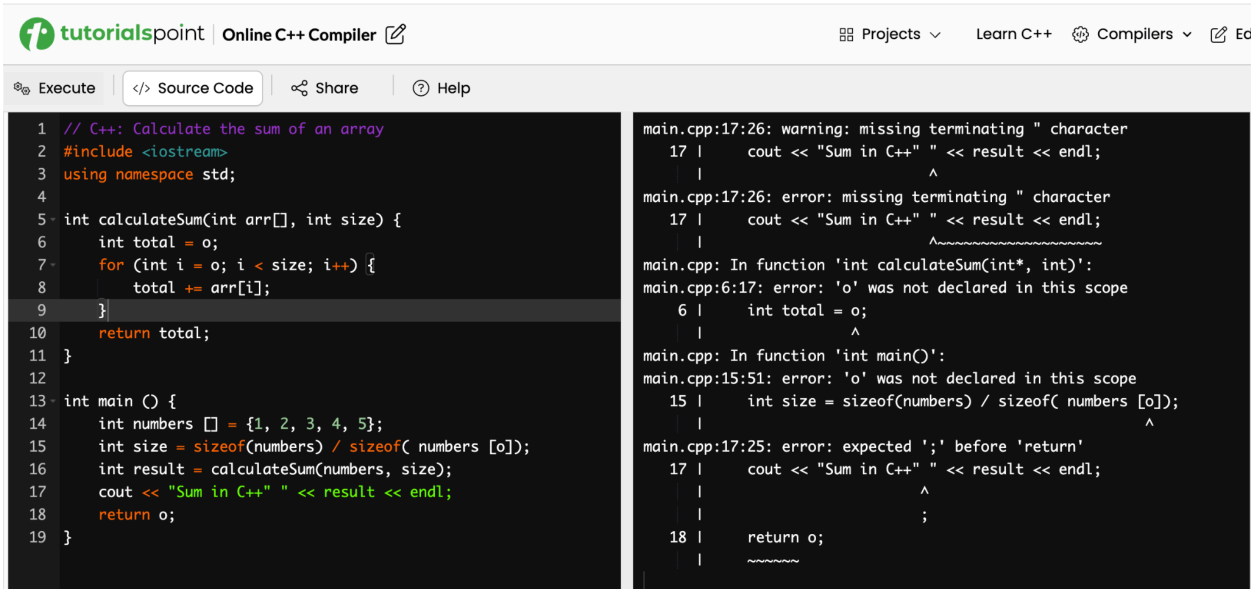
```
/home/cg/root/67d87dd5b6db9/script.js:11
let result = calculate Sum (numbers);
                       ^^^

SyntaxError: Unexpected identifier
    at internalCompileFunction (node:internal/vm:73:18)
    at wrapSafe (node:internal/modules/cjs/loader:1274:20)
    at Module._compile (node:internal/modules/cjs/loader:1320:27)
    at Module._extensions..js (node:internal/modules/cjs/loader:1414
        :10)
    at Module.load (node:internal/modules/cjs/loader:1197:32)
    at Module._load (node:internal/modules/cjs/loader:1013:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal
        /modules/run_main:128:12)
    at node:internal/main/run_main_module:28:49

Node.js v18.19.1
```

The above given JavaScript code snippet runs into a **SyntaxError** in the naming convention of the function (**calculate Sum()**) in line #11.

Once, correcting the SyntaxError in the function name (changing to **calculateSum()**), this program then later runs into a **ReferenceError** in line #3 (let total = o), since ‘o’ is not defined.

Javascript also handles syntax errors similar to Python. The compiler stops execution of the program right upon encountering an error. (Reed, 2024)

C++:



```
1  // C++: Calculate the sum of an array
2  #include <iostream>
3  using namespace std;
4
5  int calculateSum(int arr[], int size) {
6      int total = o;
7      for (int i = o; i < size; i++) {
8          total += arr[i];
9      }
10     return total;
11 }
12
13 int main () {
14     int numbers [] = {1, 2, 3, 4, 5};
15     int size = sizeof(numbers) / sizeof( numbers [o]);
16     int result = calculateSum(numbers, size);
17     cout << "Sum in C++" " << result << endl;
18     return o;
19 }
```

```
main.cpp:17:26: warning: missing terminating " character
   17 |     cout << "Sum in C++" " << result << endl;
      |                          ^
main.cpp:17:26: error: missing terminating " character
   17 |     cout << "Sum in C++" " << result << endl;
      |                          ^~~~~~~~~~~~~~~~~~~~
main.cpp: In function 'int calculateSum(int*, int)':
main.cpp:6:17: error: 'o' was not declared in this scope
    6 |     int total = o;
      |                 ^
main.cpp: In function 'int main()':
main.cpp:15:51: error: 'o' was not declared in this scope
   15 |     int size = sizeof(numbers) / sizeof( numbers [o]);
      |                                                   ^
main.cpp:17:25: error: expected ';' before 'return'
   17 |     cout << "Sum in C++" " << result << endl;
      |                         ^
      |                         ;
   18 |     return o;
      |     ~~~~~~
```

The above given C++ code snippet, runs into multiple errors.

- Firstly, compilation starts in **main()** in line #13
- In line #17, there’s a **SyntaxError** for missing termination “ character - since there’s an extra “ in the line.
- And due to this syntax error, the line isn’t compiled properly and hence, runs into another **SyntaxError** for **expected ‘;’ before return**
- In line #15, runs into a SyntaxError of ‘o’ not declared in scope
- In line #6, **total = o**, runs into the same SyntaxError of ‘o’ isn’t declared in the scope

C++ is a compiled language, so syntax errors are detected at compile-time before execution. The compiler provides all the errors throughout the entire program and points to specific lines of errors. (Mejia Alvarez et al., 2024)

The key differences in how syntax errors are handled in these languages are that:-

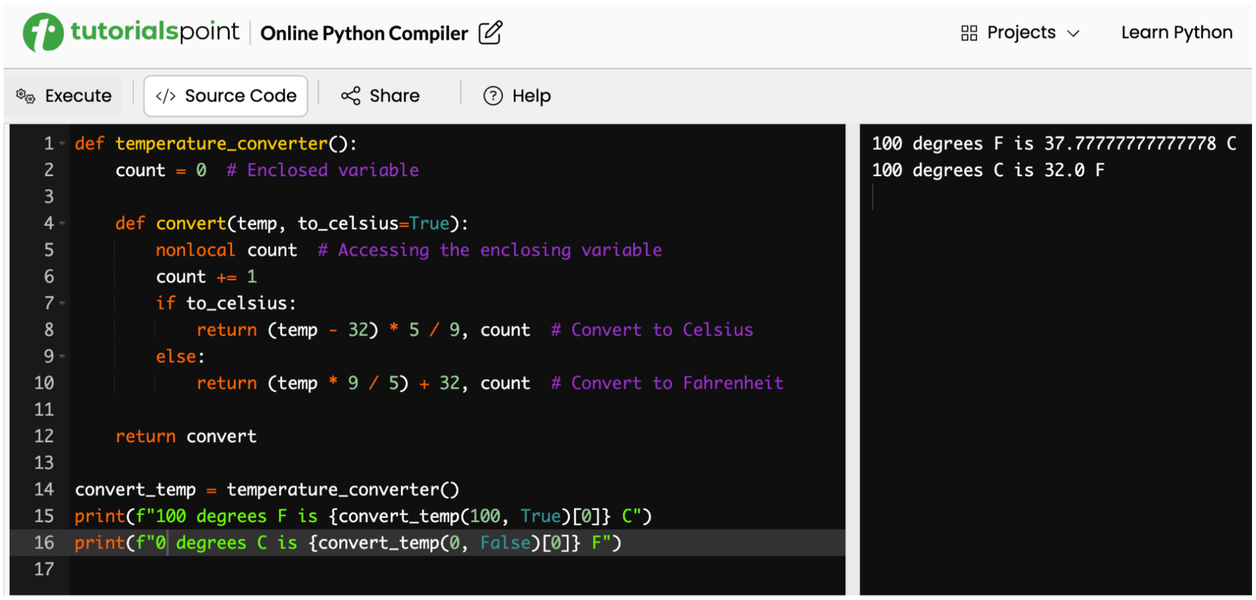
In Python and JS, the errors come up during runtime and the program stops executing after the first error. Meanwhile in C++, all the errors are caught during compilation.

As you can see in the screenshots, the errors in Python are descriptive with detailed traceback with line numbers. In JavaScript, they are also descriptive with console logs that contain error types and messages. Whereas in C++, there's only generic compiler error messages with line numbers

Section 2:

I wrote a temperature converter program in Python, Java and C++ to show how type systems, scopes and closures are handled in each language

Python:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_2/temperature_converter.py



```
1  def temperature_converter():
2      count = 0  # Enclosed variable
3  
4      def convert(temp, to_celsius=True):
5          nonlocal count  # Accessing the enclosing variable
6          count += 1
7          if to_celsius:
8              return (temp - 32) * 5 / 9, count  # Convert to Celsius
9          else:
10             return (temp * 9 / 5) + 32, count  # Convert to Fahrenheit
11 
12     return convert
13 
14 convert_temp = temperature_converter()
15 print(f"100 degrees F is {convert_temp(100, True)[0]} C")
16 print(f"0 degrees C is {convert_temp(0, False)[0]} F")
17 
```

```
100 degrees F is 37.77777777777778 C
100 degrees C is 32.0 F
```

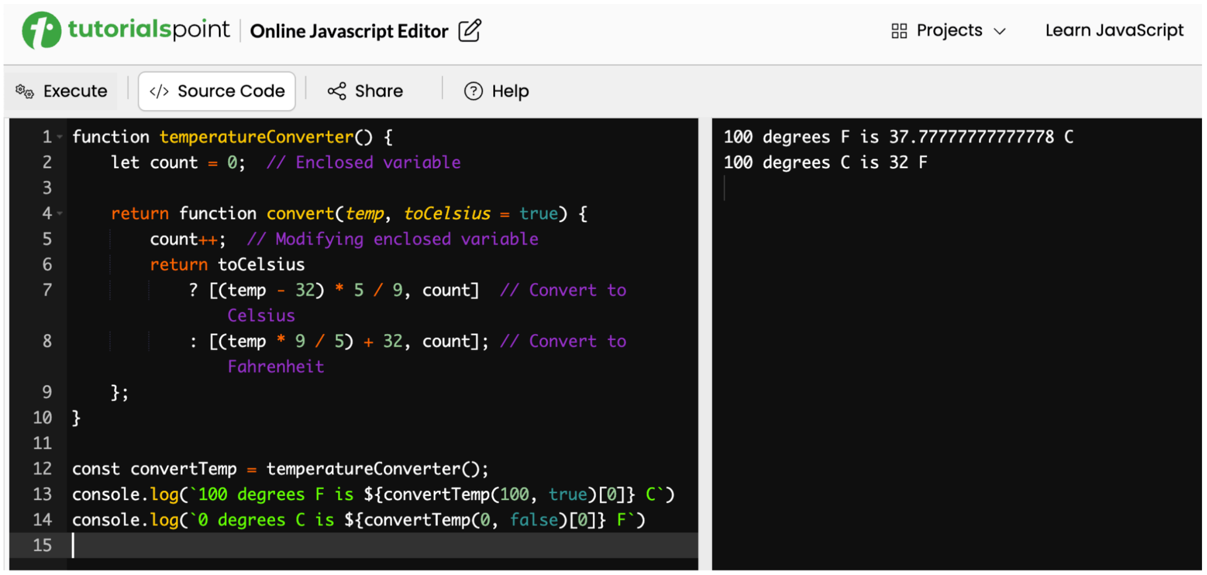
Type systems: Python is dynamically typed, and hence the types of variables are assigned at & can change at runtime. This offers flexibility for the developers but can produce runtime errors if types are used incorrectly.

This can be seen in line #2, where we didn't have to define the type of the data in the variable, and this is determined at runtime.

<u>Scopes & Closures:</u> Python supports lexical scoping and closures, allowing inner functions to capture and remember variables from their enclosing scope. (Ruqayyah Sara, 2020)

This can be seen in the methods, **convert()** and **temperature_converter()**, where the variable count is captured and remember within the **convert()** in line #6

JavaScript:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_2/temperature_converter.js



```
 1  function temperatureConverter() {
 2      let count = 0;  // Enclosed variable
 3
 4      return function convert(temp, toCelsius = true) {
 5          count++;  // Modifying enclosed variable
 6          return toCelsius
 7              ? [(temp - 32) * 5 / 9, count]  // Convert to Celsius
 8              : [(temp * 9 / 5) + 32, count]; // Convert to Fahrenheit
 9      };
10  }
11
12  const convertTemp = temperatureConverter();
13  console.log(`100 degrees F is ${convertTemp(100, true)[0]} C`)
14  console.log(`0 degrees C is ${convertTemp(0, false)[0]} F`)
15
```

```
100 degrees F is 37.77777777777778 C
100 degrees C is 32 F
```

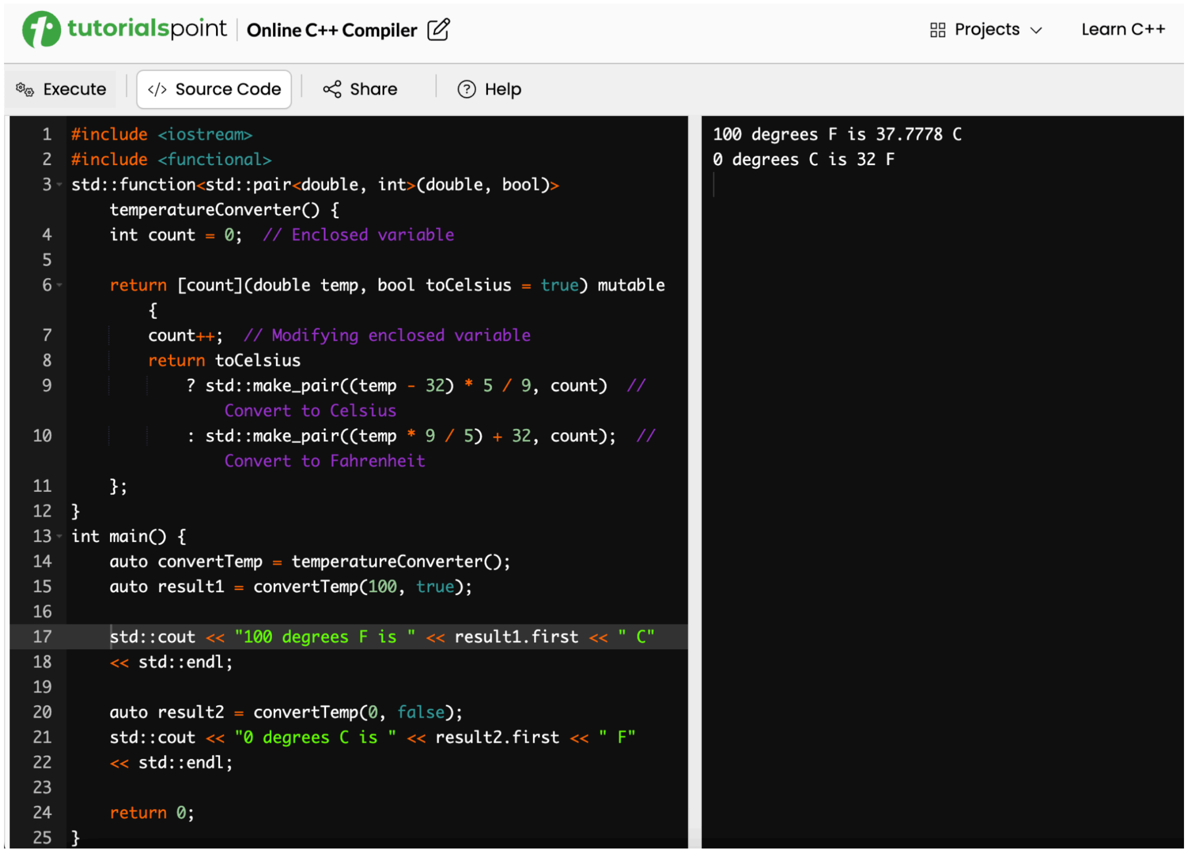
<u>Type systems:</u> JS is also dynamically typed, and hence the types of variables are assigned at & can change at runtime

This can be seen in line #2, where we didn't have to define the type of the data in the variable, and this is determined at runtime.

<u>Scopes & Closures:</u> JS also supports lexical scoping and closures, allowing inner functions to capture and remember variables from their enclosing scope. (Ruqayyah Sara, 2020)

This can be seen in the methods, **convert()** and **temperatureConverter()**, where the variable count is captured and remember within the **convert()** in line #5

C++:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_2/temperature_converter.cpp



```cpp
 1  #include <iostream>
 2  #include <functional>
 3  std::function<std::pair<double, int>(double, bool)>
        temperatureConverter() {
 4      int count = 0;  // Enclosed variable
 5
 6      return [count](double temp, bool toCelsius = true) mutable
            {
 7          count++;  // Modifying enclosed variable
 8          return toCelsius
 9              ? std::make_pair((temp - 32) * 5 / 9, count)  //
                    Convert to Celsius
10              : std::make_pair((temp * 9 / 5) + 32, count);  //
                    Convert to Fahrenheit
11      };
12  }
13  int main() {
14      auto convertTemp = temperatureConverter();
15      auto result1 = convertTemp(100, true);
16
17      std::cout << "100 degrees F is " << result1.first << " C"
18      << std::endl;
19
20      auto result2 = convertTemp(0, false);
21      std::cout << "0 degrees C is " << result2.first << " F"
22      << std::endl;
23
24      return 0;
25  }
```

```
100 degrees F is 37.7778 C
0 degrees C is 32 F
```

<u>Type systems:</u> C++ is statically typed, and hence the types of variables are checked at compile type. This helps in detecting the error during compile type, and hence performs better, as we can save on memory usage

This can be seen in line #4, where we have to define the type of the data in the variable (to **int** in this case)

<u>Scopes & Closures:</u> C++ uses lambda expressions with explicit capture syntax to implement closures

Key semantic differences:

<u>Memory management:</u>

Python and JavaScript utilize automatic garbage collection to manage memory, which makes it easy for the developer but can introduce unpredictable pauses during execution. Whereas **C++** requires manual memory management, giving developers manual control over resource allocation and deallocation, which can lead to more efficient programs but increases the risk of memory leaks and undefined behavior if not handled correctly by the developer. This also increases developer complexity (Stroustrup, 2013).

<u>Inheritance:</u>

Python supports multiple inheritance, allowing a class to inherit from multiple base classes. This provides flexibility, but can also lead to complexity and ambiguity. **C++** also supports multiple inheritance like Python but provides virtual inheritance to resolve ambiguities. **JavaScript** uses prototypal inheritance, where objects can directly inherit from other objects. This model is more flexible but less intuitive.

<u>Concurrency:</u>

Python uses Global Interpreter Lock (GIL) restricts multi-threading but supports concurrency through multi-processing and async programming. **JavaScript** uses a single-threaded, event-driven model for efficient concurrency without thread management. **C++** offers low-level threading for high efficiency but requires careful handling to prevent race conditions and deadlocks.

How these affect the program's behavior and performance:

Python's dynamic typing and garbage collection simplify development complexity but can result in slower execution & performance, and runtime errors if types are mismanaged.

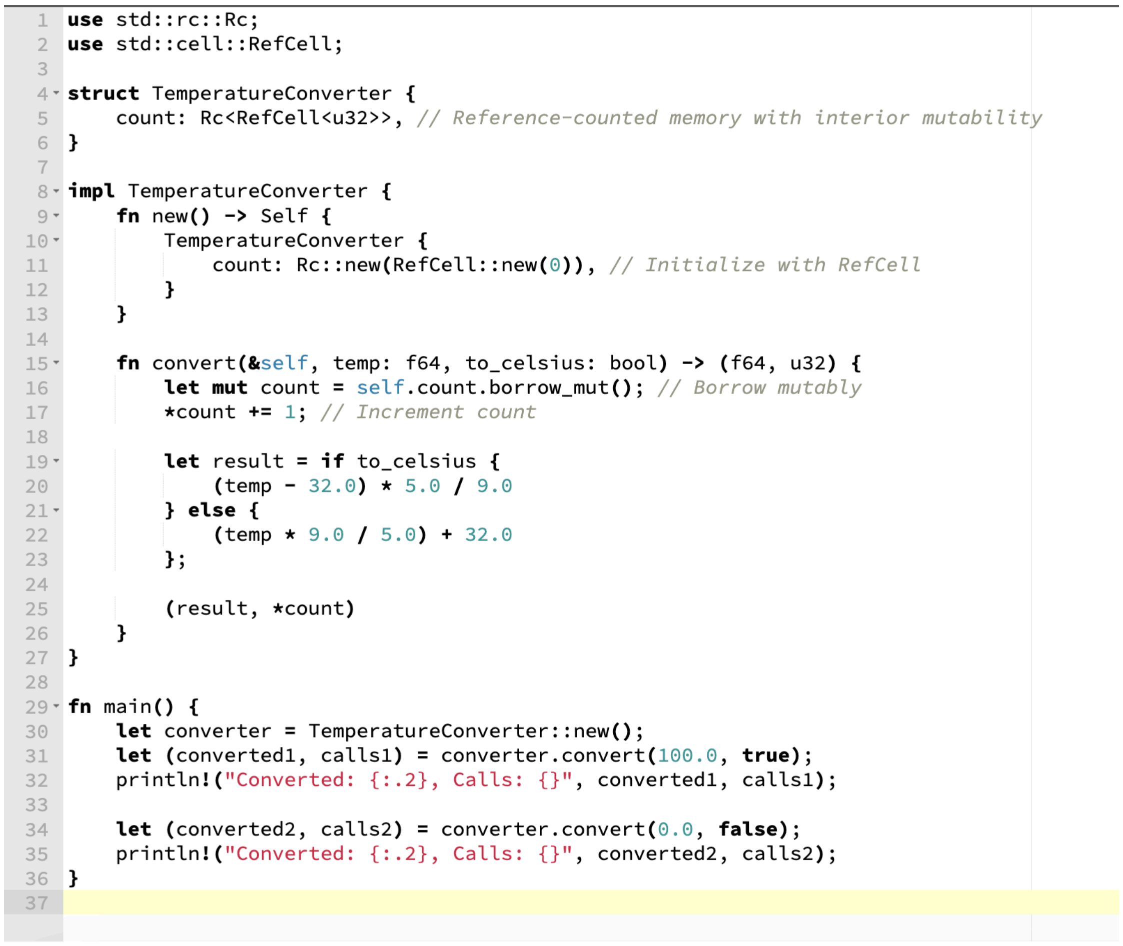
JavaScript's dynamic typing and event-driven concurrency model make it well-suited for asynchronous tasks, such as web development, but require developers to be mindful of type coercion and callback management.

C++'s static typing and manual memory management enable high-performance applications, particularly in system-level programming, but takes a lot of effort from the developers to manage resources safely and efficiently, increasing developer complexity (Stroustrup, 2013).

Part 2: Memory Management

Section 3:

Rust:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_3/temperature_converter.rs



```
 1 use std::rc::Rc;
 2 use std::cell::RefCell;
 3 
 4 struct TemperatureConverter {
 5     count: Rc<RefCell<u32>>, // Reference-counted memory with interior mutability
 6 }
 7 
 8 impl TemperatureConverter {
 9     fn new() -> Self {
10         TemperatureConverter {
11             count: Rc::new(RefCell::new(0)), // Initialize with RefCell
12         }
13     }
14 
15     fn convert(&self, temp: f64, to_celsius: bool) -> (f64, u32) {
16         let mut count = self.count.borrow_mut(); // Borrow mutably
17         *count += 1; // Increment count
18 
19         let result = if to_celsius {
20             (temp - 32.0) * 5.0 / 9.0
21         } else {
22             (temp * 9.0 / 5.0) + 32.0
23         };
24 
25         (result, *count)
26     }
27 }
28 
29 fn main() {
30     let converter = TemperatureConverter::new();
31     let (converted1, calls1) = converter.convert(100.0, true);
32     println!("Converted: {:.2}, Calls: {}", converted1, calls1);
33 
34     let (converted2, calls2) = converter.convert(0.0, false);
35     println!("Converted: {:.2}, Calls: {}", converted2, calls2);
36 }
37 
```

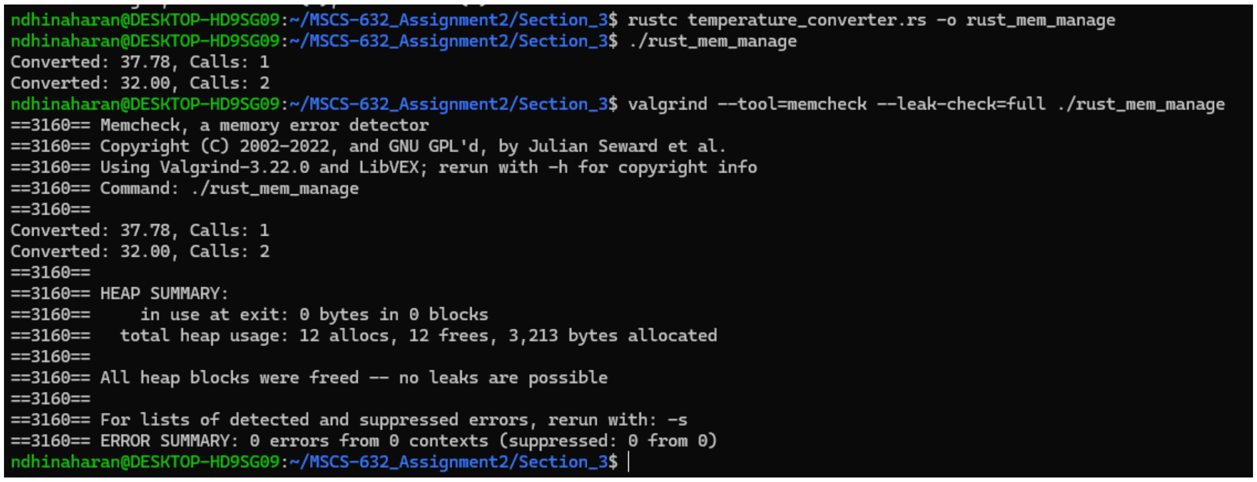
The variable **count** is wrapped in an **Rc<RefCell<u32>>** (Reference Counter) in line #4, meaning multiple parts of the program can own and share access to the same value

Ownership rules here:

- There is only one owner of TemperatureConverter at a time.
- When the owner goes out of scope, Rust automatically frees memory unless it's shared

The **&self** in line #15 and **borrow_mut()** in line #16 denotes mutable borrowing

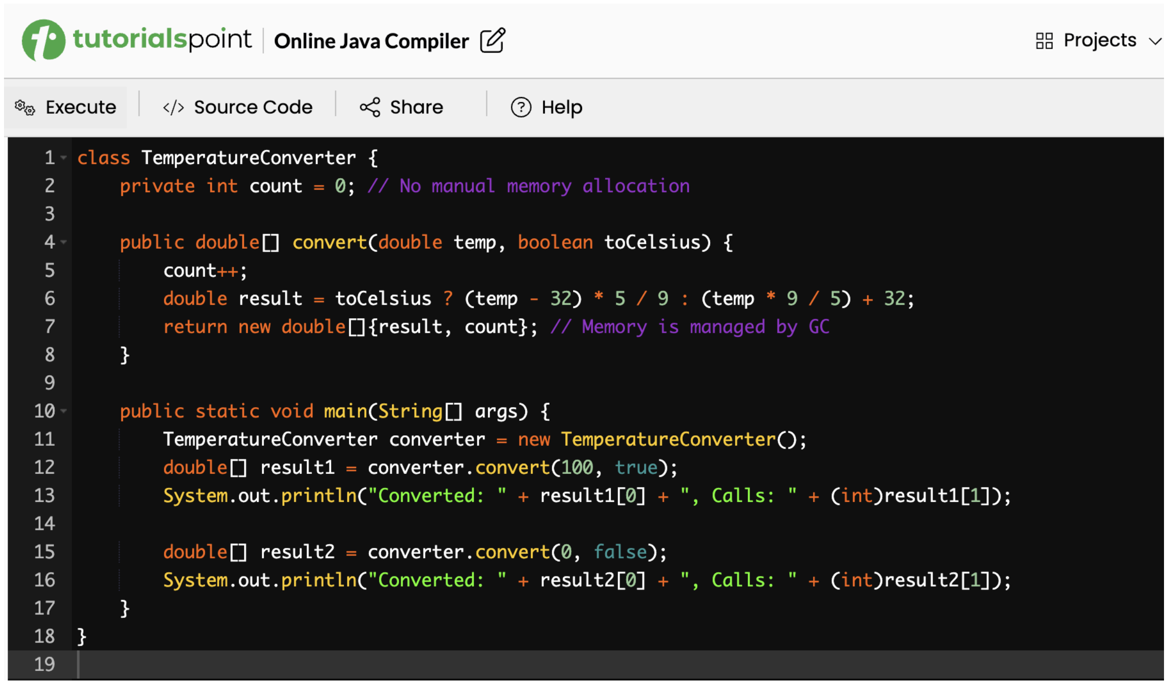
Ran my Rust program on a memory profiling tool, **Valgrind**, to analyze the memory usage and performance.



```
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ rustc temperature_converter.rs -o rust_mem_manage
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ ./rust_mem_manage
Converted: 37.78, Calls: 1
Converted: 32.00, Calls: 2
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ valgrind --tool=memcheck --leak-check=full ./rust_mem_manage
==3160== Memcheck, a memory error detector
==3160== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3160== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3160== Command: ./rust_mem_manage
==3160==
Converted: 37.78, Calls: 1
Converted: 32.00, Calls: 2
==3160==
==3160== HEAP SUMMARY:
==3160==     in use at exit: 0 bytes in 0 blocks
==3160==   total heap usage: 12 allocs, 12 frees, 3,213 bytes allocated
==3160==
==3160== All heap blocks were freed -- no leaks are possible
==3160==
==3160== For lists of detected and suppressed errors, rerun with: -s
==3160== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$
```

As you can see in the screenshot above, where I analyzed memory allocation using Valgrind, it can be seen that in my program there were 12 heap allocations, and the program also freed all of these 12 resources. Hence, this Valgrind analysis shows that the program was able to safely allocate and deallocate resources.

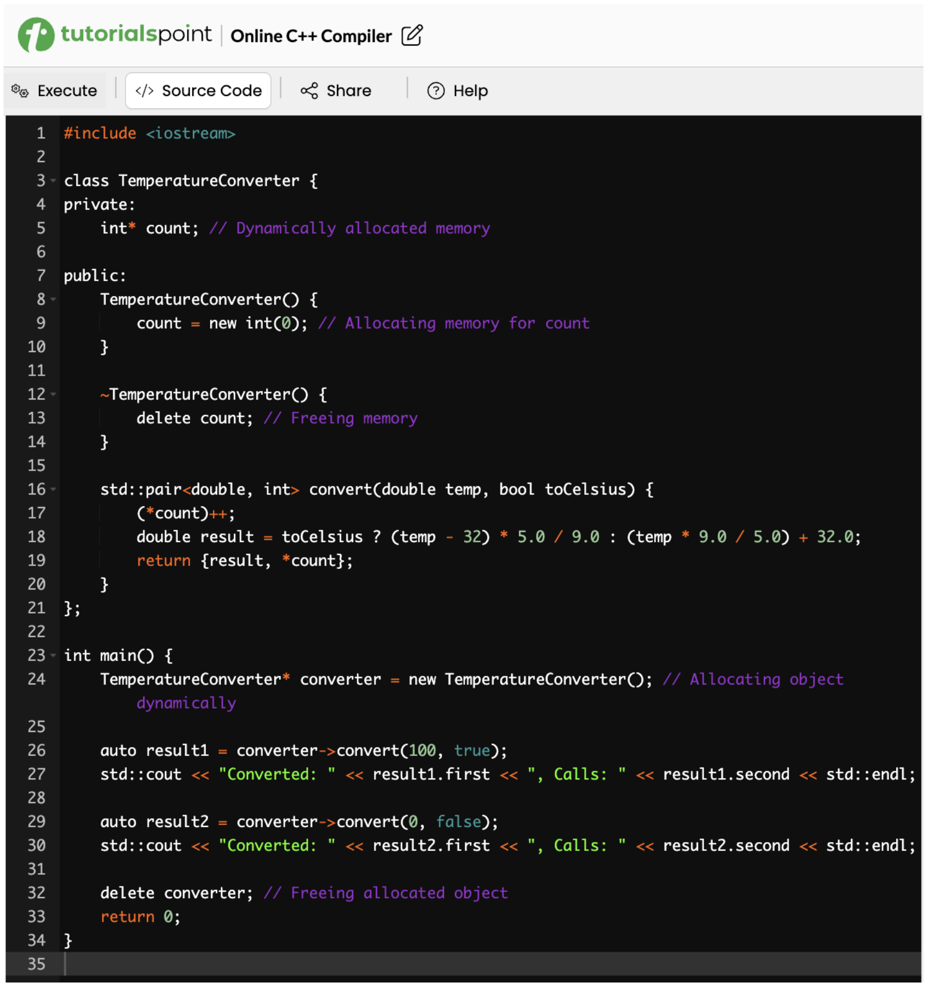
Java:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_3/TemperatureConverter.java



```java
class TemperatureConverter {
    private int count = 0; // No manual memory allocation

    public double[] convert(double temp, boolean toCelsius) {
        count++;
        double result = toCelsius ? (temp - 32) * 5 / 9 : (temp * 9 / 5) + 32;
        return new double[]{result, count}; // Memory is managed by GC
    }

    public static void main(String[] args) {
        TemperatureConverter converter = new TemperatureConverter();
        double[] result1 = converter.convert(100, true);
        System.out.println("Converted: " + result1[0] + ", Calls: " + (int)result1[1]);

        double[] result2 = converter.convert(0, false);
        System.out.println("Converted: " + result2[0] + ", Calls: " + (int)result2[1]);
    }
}
```

Java has automatic garbage collection feature, and memory is automatically when the data is invoked in line #7. As you can see in the code snippet above, there's no explicit memory allocation/deallocation.

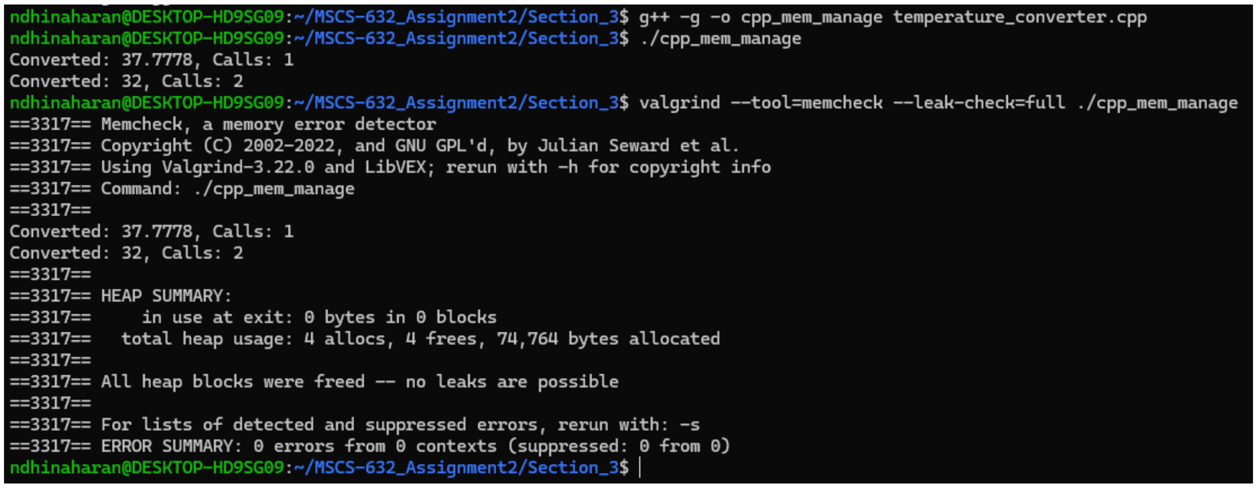
C++:
https://github.com/ndhinaharan36295/MSCS-632_Assignment2/blob/main/Section_3/temperature_converter.cpp



```cpp
1  #include <iostream>
2
3  class TemperatureConverter {
4  private:
5      int* count; // Dynamically allocated memory
6
7  public:
8      TemperatureConverter() {
9          count = new int(0); // Allocating memory for count
10     }
11
12     ~TemperatureConverter() {
13         delete count; // Freeing memory
14     }
15
16     std::pair<double, int> convert(double temp, bool toCelsius) {
17         (*count)++;
18         double result = toCelsius ? (temp - 32) * 5.0 / 9.0 : (temp * 9.0 / 5.0) + 32.0;
19         return {result, *count};
20     }
21 };
22
23 int main() {
24     TemperatureConverter* converter = new TemperatureConverter(); // Allocating object
           dynamically
25
26     auto result1 = converter->convert(100, true);
27     std::cout << "Converted: " << result1.first << ", Calls: " << result1.second << std::endl;
28
29     auto result2 = converter->convert(0, false);
30     std::cout << "Converted: " << result2.first << ", Calls: " << result2.second << std::endl;
31
32     delete converter; // Freeing allocated object
33     return 0;
34 }
35
```

As you can see in the above code snippet, memory is manually allocated (using **new** keyword) in lines #9 and #24. And then memory is also manually deallocated (using **delete** keyword) in lines #13 and #32. If this delete command is missed or forgotten, it leads to memory leaks.

Ran my C++ program on a memory profiling tool, **Valgrind**, to analyze the memory usage and performance.



```
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ g++ -g -o cpp_mem_manage temperature_converter.cpp
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ ./cpp_mem_manage
Converted: 37.7778, Calls: 1
Converted: 32, Calls: 2
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ valgrind --tool=memcheck --leak-check=full ./cpp_mem_manage
==3317== Memcheck, a memory error detector
==3317== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3317== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3317== Command: ./cpp_mem_manage
==3317==
Converted: 37.7778, Calls: 1
Converted: 32, Calls: 2
==3317==
==3317== HEAP SUMMARY:
==3317==     in use at exit: 0 bytes in 0 blocks
==3317==   total heap usage: 4 allocs, 4 frees, 74,764 bytes allocated
==3317==
==3317== All heap blocks were freed -- no leaks are possible
==3317==
==3317== For lists of detected and suppressed errors, rerun with: -s
==3317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ndhinaharan@DESKTOP-HD9SG09:~/MSCS-632_Assignment2/Section_3$ |
```

As you can see in the screenshot above, where I analyzed memory allocation using Valgrind, it can be seen that in my program there were 4 heap allocations, and the program also freed all of these 4 resources. Hence, this Valgrind analysis shows that the program was able to safely allocate and deallocate resources.

References

- Reed, M. (2024). *Python, JavaScript, Java, SQL, Linux: The complete coding and programming guide*.

- Mejia Alvarez, P., Gonzalez Torres, R. E., & Ortega Cisneros, S. (2024). *Exception handling: Fundamentals and programming*. Springer.

- Ruqayyah Sara (2020). *Semantic differences between JavaScript and Python*. Medium. https://medium.com/better-programming/semantic-differences-between-javascript-and-python-ed21b1f3ce50

- Stroustrup, B. (2013). *The C++ programming language (4th ed.)*. Addison-Wesley.