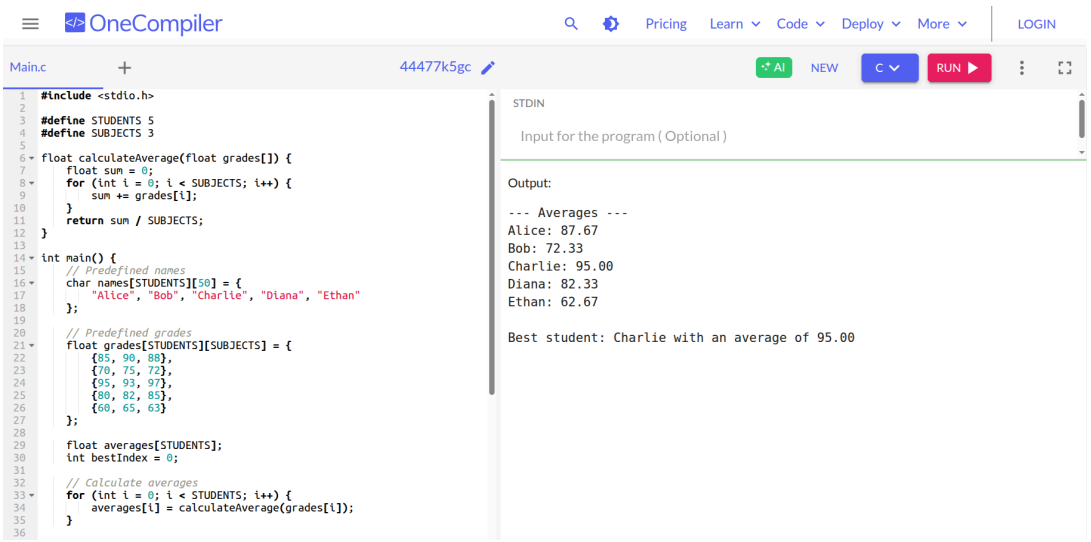
Activity #1 - Comparison and Design with Programming Languages

Structured language ©




```c
#include <stdio.h>

#define STUDENTS 5
#define SUBJECTS 3

float calculateAverage(float grades[]) {
    float sum = 0;
    for (int i = 0; i < SUBJECTS; i++) {
        sum += grades[i];
    }
    return sum / SUBJECTS;
}

int main() {
    // Predefined names
    char names[STUDENTS][50] = {
        "Alice", "Bob", "Charlie", "Diana", "Ethan"
    };

    // Predefined grades
    float grades[STUDENTS][SUBJECTS] = {
        {85, 90, 88},
        {70, 75, 72},
        {95, 93, 97},
        {80, 82, 85},
        {60, 65, 63}
    };

    float averages[STUDENTS];
    int bestIndex = 0;

    // Calculate averages
    for (int i = 0; i < STUDENTS; i++) {
        averages[i] = calculateAverage(grades[i]);
    }
```

Code explanation

#define is used to set the number of students (5) and subjects (3), making it easy to change the values without modifying the main code.

calculateAverage: receives an array of grades, adds up their values, and returns the average.

The user enters the names of the students and their three grades. Each student is reviewed, and the calculateAverage function is called to save their average in the averages array.

Each average is compared with the current best, and the index of the highest is saved in bestIndex. At the end, all averages are displayed, and finally, the student with the best average is displayed along with their value.

Object-oriented language (C#)

HelloWorld.cs + 44478g22n

```
using System;
using System.Collections.Generic;

class Student
{
    public string Name;
    public double Grade1;
    public double Grade2;
    public double Grade3;

    public Student(string name, double grade1, double grade2, double gr
    {
        Name = name;
        Grade1 = grade1;
        Grade2 = grade2;
        Grade3 = grade3;
    }

    public double CalculateAverage()
    {
        return (Grade1 + Grade2 + Grade3) / 3;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Create a list of students
        List<Student> students = new List<Student>()
```

STDIN

Input for the prograı

Output:

```
Students Averages
Alice: 87.67
Bob: 77.67
Charlie: 91.67
Diana: 69.00
Ethan: 88.00

Best Student
Name: Charlie
Average: 91.67
```

Code explanation

Student class: Represents a student with three attributes: Name, Grade1, Grade2, and Grade3. Its method is CalculateAverage() to calculate the average.

In Main, a list (List<Student>) is created with 5 students. The list is traversed with a foreach loop, the average for each student is calculated by calling the CalculateAverage() method, and then they are printed.

Functional language (Javascript)

index.js 44478w8wy AI

```javascript
// immutable list of students with their grades
const students = Object.freeze([
  { name: "Alice", grades: [85, 90, 78] },
  { name: "Bob", grades: [92, 88, 95] },
  { name: "Charlie", grades: [70, 75, 80] },
  { name: "Diana", grades: [88, 84, 89] },
  { name: "Ethan", grades: [95, 97, 99] }
]);

// Pure function
const calculateAverage = grades =>
  grades.reduce((sum, grade) => sum + grade, 0) / grades.length;

// Map each student to a new object with their average
const studentsWithAverages = students.map(student => ({
  name: student.name,
  average: calculateAverage(student.grades)
}));

// Find the BEST student
const bestStudent = studentsWithAverages.reduce((best, current) =>
  current.average > best.average ? current : best
);

console.log("Students and averages");
studentsWithAverages.forEach(s =>
  console.log(`${s.name}: ${s.average.toFixed(2)}`)
);

console.log("\nBest Student");
console.log(`${bestStudent.name} with an average of ${bestStudent.average
```

STDIN

Input for the program (Optional)

Output:

```
Students and averages
Alice: 84.33
Bob: 91.67
Charlie: 75.00
Diana: 87.00
Ethan: 97.00

Best Student
Ethan with an average of 97.00
```

Code explanation

Object.freeze() is used to ensure that the list of students cannot be modified. Each student has a name and an array of grades. calculateAverage receives a list of numbers and returns the average.

map is applied to generate a new list containing the students' names along with their averages, reduce is used to find the student with the highest average, and all averages are printed, followed by the student with the best average.

Comparación

Aspect	Object-Oriented (C#)	Structured (C)	Functional (JavaScript)

Syntax	Clear and modern syntax using classes and methods; easy to understand once you know OOP.	More complex and verbose; requires manual management of arrays and loops.	Clean because uses concise syntax with functions and array operations.
Data Handling	Uses classes and objects to encapsulate data and behavior.	Uses arrays and functions, no encapsulation.	Uses immutable objects and pure functions.
Lines of Code	63	55	33
Readability and Scalability	Very readable and scalable thanks to OOP principles.	Less readable; harder to scale and maintain.	Highly readable for small programs, scalable if functional principles are applied.

Reflection

For this task, the object-oriented programming (OOP) paradigm felt the most natural and intuitive. Using classes and objects in C# made it easier to represent each student as an independent entity with its own properties and methods. The logic for calculating averages was clearly encapsulated, improving readability and organization. OOP allowed the code to mirror real-world concepts, making it simpler to understand, extend, and maintain. Unlike the structured or functional versions, the OOP approach provided a clear separation of data and behavior, which made the solution both elegant and easy to reason about for this specific problem.

https://gitlab.com/jala-university1/cohort-1/ES.CSPR-366.GA.T2.25.M2/SA/paola-paredes/-/commit/8db6bfd144d24f92afbf158a4169e127eaa23523

Activity #2 - Functional Programming Workshop

Functional

OneCompiler

Pricing Learn

index.js 4447amfb3 AI

```
// Immutable list of transactions
const transactions = Object.freeze([
  { amount: 120.0, tag: "food" },
  { amount: -50.0, tag: "refund" },
  { amount: 200.0, tag: "electronics" },
  { amount: 40.0, tag: "test" },
  { amount: 15.0, tag: "food" },
  { amount: 300.0, tag: "travel" },
]);

const EXCLUDED_TAGS = Object.freeze(["test", "demo"]);

const TAX_RATE = 0.13;

// Keep only positive transactions
const isPositive = (tx) => tx.amount > 0;

// Keep only transactions whose tag is not excluded
const isAllowedTag = (tx) => !EXCLUDED_TAGS.includes(tx.tag);

// Apply tax to the amount and return a NEW object
const applyTax = (tx) => ({
  ...tx,
  amountWithTax: tx.amount * (1 + TAX_RATE).
```

STDIN

Input for the program (Optional)

Output:

```
Functional
Processed transactions with tax:
food: 135.60
electronics: 226.00
food: 16.95
travel: 339.00

Summary:
Total:   717.55
Average: 179.39
Max:     339.00
```

Imperative

OneCompiler

index.js 4447amfb3

```
// Immutable list of transactions
const transactions2 = [
  { amount: 120.0, tag: "food" },
  { amount: -50.0, tag: "refund" },
  { amount: 200.0, tag: "electronics" },
  { amount: 40.0, tag: "test" },
  { amount: 15.0, tag: "food" },
  { amount: 300.0, tag: "travel" },
];

const EXCLUDED_TAGS2 = ["test", "demo"];
const TAX_RATE2 = 0.13;

let total = 0;
let count = 0;
let max = 0;

const processedWithTax = [];

// Manual loop to process all steps
for (let i = 0; i < transactions2.length; i++) {
  const tx = transactions2[i];

  // Filter positive
  if (tx.amount <= 0) {
```

STDIN

Input for the program (Optional)

Output:

```
Imperative Version
Processed transactions with tax:
food: 135.60
electronics: 226.00
food: 16.95
travel: 339.00

Summary:
Total:   717.55
Average: 179.39
Max:     339.00
```

Reflection

In the functional version we start with an immutable list of transactions. Each transaction has two fields: amount and tag. Then we use the pure helper functions

- isPositive(tx) returns true only if the amount is greater than zero.
- isAllowedTag(tx) returns true if the tag is not in the excluded list.
- applyTax(tx) returns a new object that includes a new field amountWithTax. It does not change the original transaction.

First, we call .filter(isPositive) to keep only positive amounts, then we call .filter(isAllowedTag) to remove transactions with excluded tags, after that we call .map(applyTax) to apply the tax to each remaining transaction, and the result is a new list called processedTransactions. I use .reduce(summaryReducer, initialSummary) to compute:

- total: sum of all amountWithTax.
- count: number of valid transactions.
- max: maximum amountWithTax.

But for the imperative version we use the same list of transactions, the same excluded tags and tax rate, we declare total, count and max with let, and we start them at zero, also create an empty array processedWithTax to store the processed transactions.

We use a for loop to go through each transaction:

- If the amount is not positive, we continue to the next one.
- If the tag is in the excluded list, we also continue.
- We calculate amountWithTax and update total, count and max step by step.

After the loop, we compute the average and print all the results. Functional programming brings several important advantages for this kind of problem: the pipeline filter → filter → map → reduce describes what we want to do, not how to do it step by step. This makes the logic easier to read and understand.

The imperative version is also valid, but it mixes the logic of filtering, transforming and aggregating inside one loop. The functional style separates these concerns more clearly, which helps maintenance and makes the code closer to a mathematical description of the problem.