Basic C++ Questions

① Sum of digits

↳ given n positive int, find sum of them.

Code :-

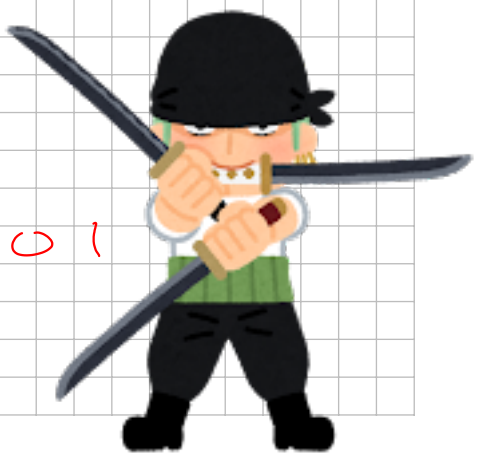
```cpp
#include <iostream>
using namespace std;

int main() {
    int n, number, sum = 0;

    cout << "Enter the number of integers: ";
    cin >> n; // reading n

    cout << "Enter " << n << " integers: ";
    for (int i = 0; i < n; i++) {
        cin >> number;
        sum += number;
    }

    cout << "The sum of the integers is: " << sum << endl;

    return 0;
}
```

n → no of digits to be added
number → current number
sum → sum of numbers
cin >> number; → reading current number
sum += number; ↳ Adding to find summation

② Reverse a number

Given an int number (positive) N, design an algo to reverse the number.

eg:- 123 ⟶ 321
4589 ⟶ 9854
10112 ⟶ 21101



Working:-

123 ⟶ ?

0 × 10 + 1
= 1 × 10 + 2
= 12 × 10 + 3 = 123

0 × 10 + 3 = 3 × 10 + 2
= 32 × 10 + 1
= 321

↳ (1) need to get last digit
(2) multiply the current reverse value by 10 and add extracted value.
(3) remove the last digit from original number.
(4) Repeat until no. is zero

Code:-

```cpp
#include <iostream>
using namespace std;

int main() {
    int n, reversed = 0, digit;

    cout << "Enter a number to reverse: ";
    cin >> n;

    while (n != 0) {
        digit = n % 10;              // Extract the last digit
        reversed = reversed * 10 + digit; // Append the digit to the reversed number
        n = n / 10;                  // Remove the last digit from the number
    }

    cout << "Reversed number: " << reversed << endl;

    return 0;
}
```

(4) (1) (2) (3)

↳ get the reversed number

↳ Important should know how to reverse a number.

Variable scope in Cpp

↳ defines the visibility of a variable.

* The following are types of scope in Cpp

① Local Scope

↳ Variable declared inside a function or block is local to that func or block.

↳ Not accessible outside the block and destroyed when block is exited.

Example of Local Scope.

```cpp
#include <iostream>
using namespace std;

void exampleLocalScope() {
    int x = 10;  // Local variable
    cout << "Inside function, x = " << x << endl;
}

// int x = 20;  // Uncommenting this would lead to a name conflict

int main() {
    exampleLocalScope();
    // cout << x;  // Error: x is not accessible here
    return 0;
}
```

② Global Scope

↳ declared outside of all function.

↳ can be accessed from any block or function in the prog

↳ should be used carefully

Eg :-

```cpp
#include <iostream>
using namespace std;

int globalVar = 100;  // Global variable

void exampleGlobalScope() {
    cout << "Accessing global variable: " << globalVar << endl;
}

int main() {
    cout << "Global variable in main: " << globalVar << endl;
    exampleGlobalScope();
    return 0;
}
```

③ Block Scope

- declared inside {} are scoped block.
- cannot be accessed outside block.

```cpp
#include <iostream>
using namespace std;

int main() {
    {
        int blockVar = 42;  // Block scope
        cout << "Inside block, blockVar = " << blockVar << endl;
    }
    // cout << blockVar;  // Error: blockVar is not accessible here
    return 0;
}
```

a block scope

④ Function Scope

- parameters of function have function scope.
- They are only accessible within the function body.

```cpp
#include <iostream>
using namespace std;

void addNumbers(int a, int b) {  // a and b have function scope
    int sum = a + b;  // Local variable with function scope
    cout << "Sum = " << sum << endl;
}

int main() {
    addNumbers(5, 10);
    // cout << sum;  // Error: sum is not accessible here
    return 0;
}
```

here sum, a, b have func scope.

(5) Static scope

↳ variable declared with the static keyword retains it value b/w multiple func calls.

↳ It is initialized only once and its values persists across multiple invocations of the functions.

```cpp
#include <iostream>
using namespace std;

void staticExample() {
    static int count = 0;  // Static variable
    count++;  <-- ++
    cout << "Count = " << count << endl;
}

int main() {
    staticExample();  // Count = 1
    staticExample();  // Count = 2
    staticExample();  // Count = 3
    return 0;
}
```

(x) Static value persist until the program ends.

⑥ File Scope

- Static variable declared outside the any function.
- accessible only with in the file.

```cpp
#include <iostream>
using namespace std;

static int fileScopedVar = 50;  // File scope

void accessFileScope() {
    cout << "File scoped variable: " << fileScopedVar << endl;
}

int main() {
    accessFileScope();
    return 0;
}
```

⑦ Namespace Scope

- declared inside namespace
- can be used by accessing the namespace name or through the 'using' directive.

```cpp
#include <iostream>
namespace myNamespace {
    int var = 10;  // Namespace scope
}

int main() {
    std::cout << "Accessing with namespace: " << myNamespace::var << std::endl;

    using namespace myNamespace;
    std::cout << "Accessing directly after 'using': " << var << std::endl;

    return 0;
}
```

} namespace created with var = 10

✓ var used.

(8) Class Scope

↳ Members of the class have class scope.

↳ They can be accessed through obj of the class or directly if static.

```cpp
#include <iostream>
using namespace std;

class MyClass {
private:
    int privateVar;  // Class scope

public:
    MyClass(int val) : privateVar(val) {}
    void display() {
        cout << "Private variable: " << privateVar << endl;
    }
};

int main() {
    MyClass obj(42);
    obj.display();
    // cout << obj.privateVar; // Error: privateVar is not accessible outside
                               //the class
    return 0;
}
```

→ class created, with int private var, and method and constructor

Summary

Scope Type	Where Declared	Accessed In	Lifetime
Local	Inside a function or block	Within the same block	Limited to block execution
Global	Outside all functions	Anywhere in the program	Entire program execution
Block	Inside `{}`	Within the block	Limited to block execution
Function	As function parameters	Within the same function	During function execution
Static	With `static` keyword	Within block or file	Entire program execution
File	With `static` keyword at file level	Only within the file	Entire program execution
Namespace	Inside a `namespace`	Using namespace or qualifier	Entire program execution
Class	Inside a class	Through objects or static access	Depends on class/object usage

Q→ WAP to print n fibonacci numbers

0, 1, 1, 2, 3, 5, 8 ; n = 7

Theory :-

Fibonacci numbers are sequence of numbers where each number is sum of the two preceding ones.

for $n \geq 2$,

$$f(n) = f(n-1) + f(n-2)$$

where $f(0) = 0$

$f(1) = 1$

Working :-

① a = 0, b = 1, initialize
(a = first term, b = second term)

② check if n at least 0 or 1.

③ Loop :-

Calculate the next number

next = a + b (next Term)

a = b (n-1)

b = next (n)

Loop will run till 'n' times.

Code :-

```cpp
#include <iostream>
using namespace std;

void generateFibonacciTerms(int n) {
    if (n <= 0) {
        cout << "Please enter a positive number of terms." << endl;
        return;
    }

    int a = 0, b = 1;

    cout << "First " << n << " Fibonacci terms:" << endl;

    for (int i = 1; i <= n; i++) {
        cout << a << " ";
        int next = a + b;  // Calculate the next Fibonacci term
        a = b;             // Update a to the current b
        b = next;          // Update b to the new term
    }

    cout << endl;
}

int main() {
    int n;

    cout << "Enter the number of Fibonacci terms to generate: ";
    cin >> n;

    generateFibonacciTerms(n);

    return 0;
}
```

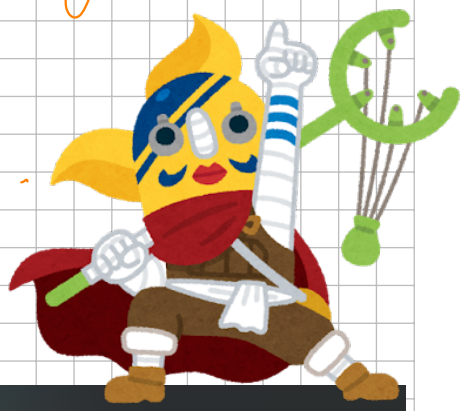
Q → Largest of N numbers

↳ given N int values find largest among them.

Working :-

1. Read value of N
2. Initialize 'largest' as first int

(3) take first number as largest.
(4) iterate through remaining n-1 terms, and check if any number is 'largest'.
(5) if true:-
largest = num.
(6) print the largest.



```cpp
#include <iostream>
using namespace std;

int main() {
    int n;

    cout << "Enter the number of integers: ";
    cin >> n;

    if (n <= 0) {
        cout << "Invalid input. The number of integers must be positive." << endl;
        return 1;
    }

    int largest, num;

    cout << "Enter the numbers:" << endl;

    // Initialize `largest` with the first number
    cin >> largest;

    // Iterate through the remaining n-1 numbers
    for (int i = 1; i < n; i++) {
        cin >> num;
        if (num > largest) {
            largest = num; // Update `largest` if a larger number is found
        }
    }

    cout << "The largest number is: " << largest << endl;

    return 0;
}
```

(1) (2) & (3) (4) (5) (6)

* Control Mechanism in loop

① Break Statement

↳ Used to exit loop prematurely.

eg:-

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    std::cout << i << " ";
}
```

→ output
0 1 2 3 4

② Continue Statement

↳ Skips the current iteration and continues to the next iteration.

eg:-

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) continue;
    std::cout << i << " ";
}
```

output
→ 0 1 3 4 5 6 7 8 9

③ Goto Statement

↳ Transfers control to labeled statement.

eg:-

```
int i = 0;
loop:
if (i < 5) {
    std::cout << i << " ";
    i++;
    goto loop;
}
```

to
go

output
0 1 2 3 4

④ Return Statement

↳ Exits loop the function it resides in.

eg:-

```
for (int i = 0; i < 10; i++) {
    if (i == 5) return;
    std::cout << i << " ";
}
```

→ exits function

output
0 1 2 3 4

Ternary operator in Cpp

Shorthand way of writing conditional statements.

Also called conditional operator because it evaluates conditional operator.

{ condition ? expression1 : expression2; }

condition :- evalues to true or false.

expression1 :- executed and returned, if the condition is true.

expression2 :- executed and returned if the condition is false.

basically :-

if condition true :- expression 1

if condition false :- Expression 2.

Examples :-

Basic example :-

10 > 20 → false

```
1   int a = 10, b = 20;
2   int max = (a > b) ? a : b;
3   std::cout << "Maximum: " << max << std::endl;
```

output: 20

Limitations of ternary operator

↳ Readability issues

↳ Limited functionality

↳ Has side effects.

Q → Check Prime

↳ Given positive number N (>1)

↳ design algo to check given number is prime.

Working :-

(1) Initial check

↳ N ≤ 1 (X) Not prime

(2) Iterate from 2 to $\sqrt{N}$

(We learned this)

(3) check if any number is perfectly divisible.

↳ If yes not prime

[N % i == 0]

(4) print result - (return)

Time Complexity

↳ Best Case → O(1) if N ≤ 3.

↳ Worst Case (if prime)

↳ $O(\sqrt{N})$

Code :-

```cpp
#include <iostream>
#include <cmath>

// Function to check if a number is prime
bool isPrime(int N) {
    if (N <= 1) return false;       // Numbers <= 1 are not prime
    if (N <= 3) return true;        // 2 and 3 are prime
    if (N % 2 == 0 || N % 3 == 0)   // Eliminate multiples of 2 and 3
        return false;

    // Check divisors from 5 to √N
    for (int i = 5; i * i <= N; i += 6) {
        if (N % i == 0 || N % (i + 2) == 0)
            return false;
    }
    return true;
}

int main() {
    int N;
    std::cout << "Enter a positive number: ";
    std::cin >> N;

    if (isPrime(N))
        std::cout << N << " is a prime number." << std::endl;
    else
        std::cout << N << " is not a prime number." << std::endl;

    return 0;
}
```

(1)
(2)
(3)
(4)

skips number with factors of 2, 3

eg :- 5, 11, 17, 19 etc

Q → Square root of a number.

eg :- $\sqrt{8} = 64$

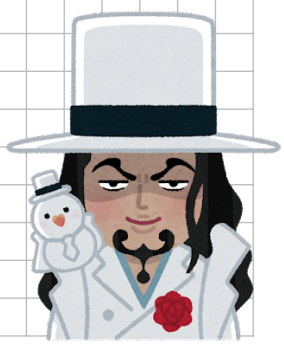
$\sqrt{30} = 5.47706$

Working :- (Newton method)

formula :- $\frac{x + N/x}{2}$

eg :- $N = 64$

let initial guess be → 64 (x)



Interations

(1) $x_1 = \frac{64 + 64/64}{2} = 32.5$

(2) $x_2 = \frac{32.5 + 64/64}{2} = 17.234375$

(3) $x_3 = \frac{17.234375 + 64/x_2}{2} = 10.4740$

(4) $x_4 = \frac{x_3 + 64/x_3}{2} = 8.292191$

(5) $x_5 = \frac{x_4 + 64/x_4}{2} = 8.005147$

(6) $x_6 = \frac{x_5 + 64/x_5}{2} = 8.000000$

output = $\sqrt{64} \approx 8.000000$

Code :-

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
double newtonSquareRoot(double N, double precision = 0.0001) {
    if (N == 0 || N == 1) return N;

    double x = N;  // Initial guess (same)
    double nextX;

    while (true) {   loop
        nextX = (x + N / x) / 2;  // Update using Newton's method
        if (abs(nextX - x) < precision) break;  // Check for convergence
        x = nextX;
    }
    return nextX;
}

int main() {
    double N;
    cout << "Enter a non-negative number: ";
    cin >> N;

    if (N < 0) {
        cout << "Square root is not defined for negative numbers." << endl;
        return 1;
    }

    double result = newtonSquareRoot(N, 0.0001);
    cout << "Square root of " << N << " is approximately: "
         << fixed << setprecision(5) << result << endl;

    return 0;
}
```

by checking if we reached result

Time Complexity

- Convergence rate is quadratic.
  - number of correct digits doubles roughly with each iter.
- Per-iteration cost:- $O(1)$
- Time Complexity:-
  - $O(\log(p))$