Problem: Write a program that checks if a given number is both:

o Greater than 10

. o Even.

```
        Console.Write("Enter a number: ");

        if (int.TryParse(Console.ReadLine(), out int number))
        {
            if (number > 10 && number % 2 == 0)
            {
                Console.WriteLine("The number is greater than 10 and even.")
            }
            else
            {
                Console.WriteLine("The number does not meet both conditions.
            }
        }
        else
        {
            Console.WriteLine("Invalid input. Please enter a valid integer."
        }
    }
}
```

input

```
Enter a number: 12
The number is greater than 10 and even.

..Program finished with exit code 0
```

Question: How does the && (logical AND) operator differ from the & (bitwise AND) operator?

`&&` — Logical AND (used with Booleans)

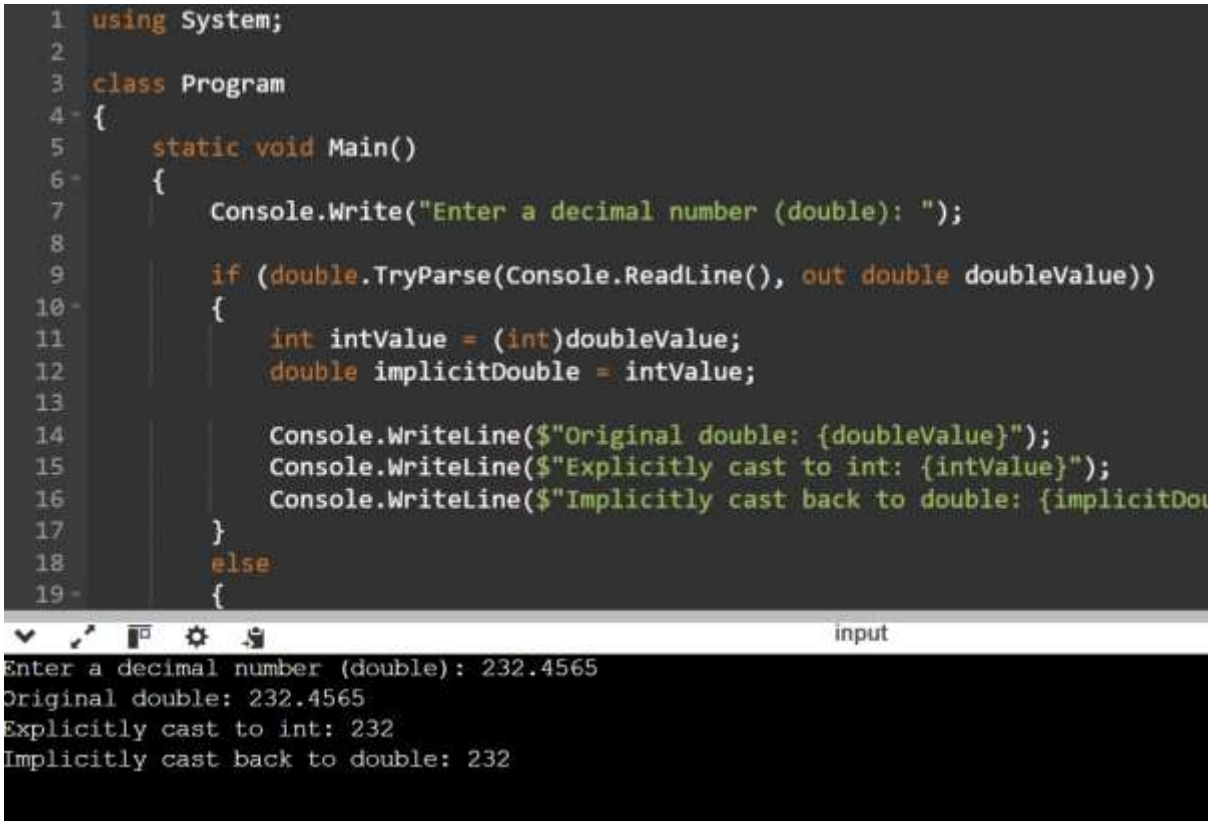
- Type: *Logical operator*
- Usage: Combines Boolean expressions.
- Short-circuit behaviour: Yes. If the first operand is `false`, the second one is not evaluated.

`&` — Bitwise AND (used with integers) or non-short-circuit Logical AND

1. Bitwise AND (when used with integers)

- Type: *Bitwise operator*
- Usage: Compares bits of two numbers
- .

Problem: Implement a program that takes a double input from the user and casts it to an int. Use both implicit and explicit casting, then print the results. Question: Why is explicit casting required when converting a double to an int?



Question: Why is explicit casting required when converting a `double` to an `int`?

Answer:
Because a `double` has a larger and more precise range than an `int`. When converting:

- You might lose data, such as the fractional part (e.g., `5.9` becomes `5`).
- There's also a risk of overflow if the double's value exceeds the range of `int`.

C# requires explicit casting (e.g., `(int)myDouble`) to force you to acknowledge this potential data loss and ensure you're doing it intentionally.

Q3)

```
using System


class Program

{

  static void Main()

  {

    Console.Write("Enter your age: ");
```

```
        string input = Console.ReadLine();

        try
        {
            int age = int.Parse(input);

            if (age > 0)
            {
                Console.WriteLine($"Your age is {age}.");
            }
            else
            {
                Console.WriteLine("Age must be greater than 0.");
            }
        }
        catch (FormatException)
        {
            Console.WriteLine("Invalid input: Please enter a valid number.");
        }
        catch (OverflowException)
        {
            Console.WriteLine("Invalid input: The number is too large or too small.");
        }
    }
}
```

❓ Question: What exception might occur if the input is invalid, and how can you handle it?

✳️ Exceptions that can occur:

FormatException

Occurs if the input cannot be parsed as an integer (e.g., letters, symbols, empty string).

Example: "abc" or ""

OverflowException

Occurs if the input is a number outside the range of int (e.g., 999999999999).

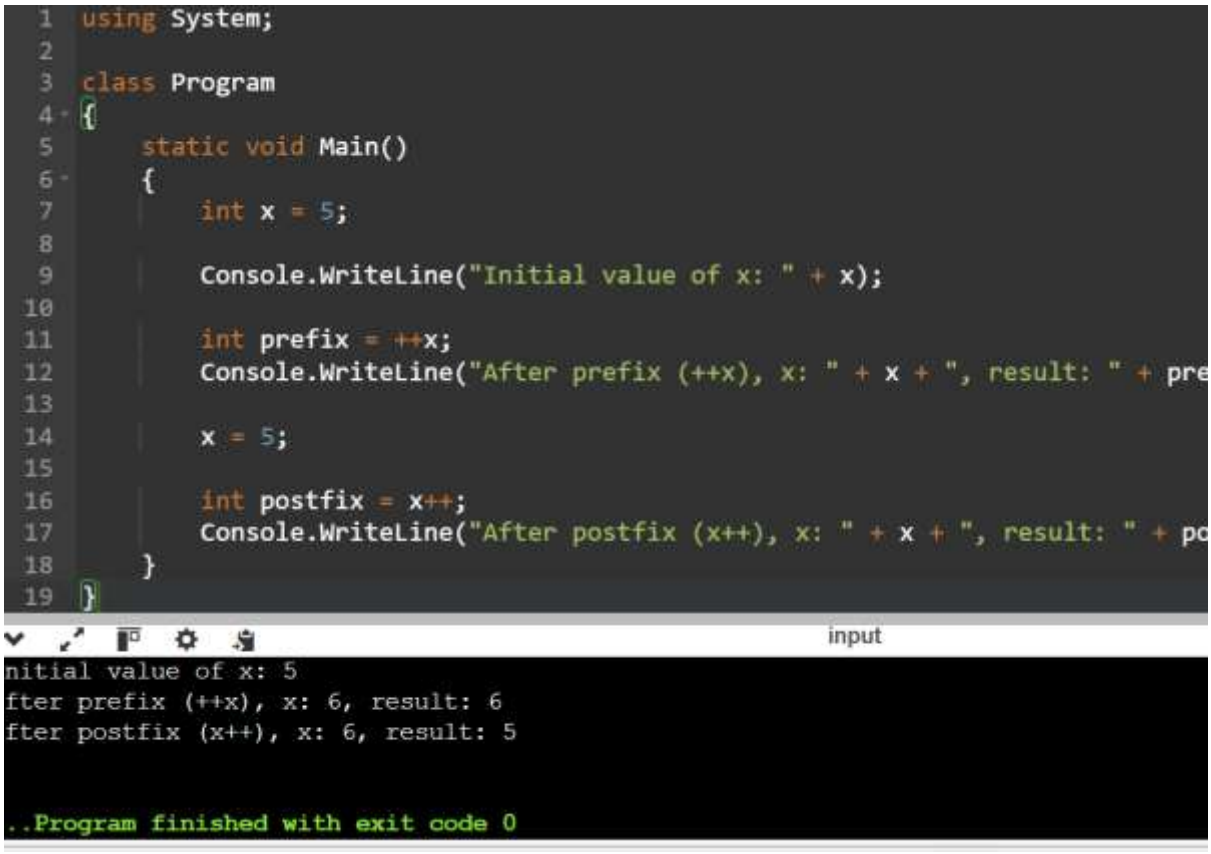
✅ How to handle it:

Use a try-catch block as shown above.

Catch specific exceptions (FormatException, OverflowException) to give clear error messages.

You could also use int.TryParse() if you want to avoid exceptions entirely.

Q4)



Q5) Final Answer:

- **`x = 7`**
- **Because:**
  - `++x` makes x = 6 and returns 6
  - `x++` uses 6, then x becomes 7

what's the difference between compiled and interpreted languages and in this way what about Csharp?

Compiled vs Interpreted Languages

🔧 Compiled Languages

- Code is **translated** from source (e.g. `.cs`, `.cpp`) to machine code **before** it runs.
- The compiler creates a **standalone executable** (like `.exe`).
- **Faster execution**, because translation happens only once.
- Examples: C, C++, Go, Rust

Pros: High performance
Cons: Compilation time, less flexible for on-the-fly changes

🎭 Interpreted Languages

- Code is **executed line-by-line** by an interpreter **at runtime**.
- No standalone executable — you need the interpreter to run the code.
- **Slower** than compiled code, but very flexible for quick changes and scripting.
- Examples: Python, JavaScript, Ruby

Pros: Easy debugging, fast iteration
Cons: Slower performance, depends on interpreter

🤖 So... What about C#?

C# is a **hybrid** — it's **compiled AND interpreted**, and here's how:

1. ✅ First, your C# code is **compiled** into **Intermediate Language (IL)** by the C# compiler (`csc`).
2. 🧠 Then, when the program runs, the **.NET runtime (CLR)** uses a **JIT (Just-In-Time) compiler** to convert the IL into machine code **at runtime**.

This means:

- C# benefits from **faster execution** (thanks to compilation),

- While also offering some **runtime flexibility** like interpreted languages.

So, C# sits **in the middle**:

🔄 Compiled to IL → Interpreted (JIT-compiled) at runtime by the CLR

Implicit Casting (Widening Conversion)

- ✅ **Automatic** conversion.
- 🔄 Happens when **no data loss** is possible.
- Used **between compatible types** (e.g. `int` to `double`).

✔ Example:

```
csharp
CopyEdit
int x = 10;
double y = x; // implicit
```

☑ Pros:

- Safe, no data loss.
- Clean syntax.

❌ Cons:

- Limited to widening conversions.

🧠 2. Explicit Casting (Narrowing Conversion)

- ❗ Requires **manual cast** using parentheses.
- 🔻 Used when **data loss is possible**, like from `double` to `int`.

✔ Example:

```
csharp
CopyEdit
double d = 9.8;
int i = (int)d; // explicit, value becomes 9
```

☑ Pros:

- Gives control to the programmer.

❌ Cons:

- Risk of precision loss or overflow.
- May throw exceptions at runtime (e.g. `InvalidCastException` in reference types).

🔁 3. Convert Class (System.Convert)

- ✅ Provides **flexible conversion** between many types (strings, numbers, bools...).
- 🔥 Can handle `null` safely (returns default value).
- Better for converting between **non-compatible types** (like string to int).

✔️ Example:

```
csharp
CopyEdit
string s = "123";
int x = Convert.ToInt32(s);
```

☑️ Pros:

- Handles nulls.
- Works across data types.
- Easy to use.

❌ Cons:

- May throw exceptions (`FormatException`, `OverflowException`).

🔍 4. Parse Methods (e.g., int.Parse)

- 🎯 Used for **string to value type conversion**.
- Works only with **strings** that are in valid format.
- **Stricter** than `Convert`.

✔️ Example:

```
csharp
CopyEdit
string s = "456";
int x = int.Parse(s);
```

☑️ Pros:

- Simple and direct.
- Good for guaranteed valid input.

❌ Cons:

- Throws `FormatException` or `ArgumentNullException`.
- No null or invalid string handling

BONUS

What is meant by "C# is managed code"?

When we say **C# is managed code**, we mean:

C# code runs under the control of the **.NET runtime environment**, specifically the **CLR (Common Language Runtime)**.

🔐 What "managed" really means:

- Memory is **automatically managed** (no need for manual `malloc` or `free` like in C/C++).
- The **Garbage Collector (GC)** handles cleanup of unused objects.
- Built-in **type safety**, **exception handling**, and **security checks**.
- The CLR also provides **Just-In-Time (JIT) compilation**, converting IL (Intermediate Language) into machine code at runtime.

🔍 Summary:

Managed code = code that runs inside the CLR with memory safety, garbage collection, and runtime services.

🧠 **Unmanaged code** (like native C++) runs directly on the OS without CLR supervision and requires manual memory management.

☑️ 2. What is meant by "struct is considered like class before"?

This statement refers to the **similarities** and **differences** between `struct` and `class` in C# — especially from a beginner's or C++ background perspective.

◇ Similarities (why people say "like class before"):

- Both can have **fields**, **methods**, **constructors**, **properties**, etc.
- Both support **encapsulation** and can implement **interfaces**.
- From the outside, you might use them the same way.

```
csharp
```

```
CopyEdit
struct Point { public int X; public int Y; }
class Car { public string Model; }
```

▽ But here's the key difference:

- `**struct**` is a **value type** (stored in the **stack** by default).
- `**class**` is a **reference type** (stored in the **heap**).

📌 Important struct characteristics:

- Cannot inherit from another class or struct.
- Lightweight and more efficient for small data models.
- Often used for performance-critical scenarios (like game engines or graphics code).