

# Lexical Scoping And Copying Through Funcs Doubts

1. `function not_faulty() { if(option==""){ alert("Result: " + (num1+num2)); } else if(option=="-"){ alert("Result: " + (num1-num2)); } else if(option=="*"){ alert("Result: " + (num1*num2)); } else if(option=="/"){ alert("Result: " + (num1/num2)); } else if(option=="**"){ alert("Result: " + (num1**num2)); } } let num1, num2 // var UseAgain = true; while(UseAgain = true) { num1 = prompt("Enter First Number: ") num2 = prompt("Enter Second Number: ") option = prompt("Enter the Sign Of Operation") //as prompt returns string by default so we must change its datatype to perform operations num1 = parseFloat(num1); ..... 2`
2. `while (UseAgain = true)`I did it intentionally as set the default value of useAgain as true. .... 6
3. or is it because func `non_faulty` is called inside the same block scope (i.e. here it is while's scope) what about my second guessss can this be possible? ..... 8
4. JavaScript uses **lexical scoping**, meaning:i hope c++, java and python eses the same? ..... 10
5. suppose if it were cpp ode the the variables were not global but the function was called inside the sam block as the variables then will the function access the variables? .....12
6. Now, my confusion is clear. .... 14



```
1. function not_faulty() {  
  if(option==""){  
    alert("Result: " + (num1+num2));  
  }  
  
  else if(option=="-"){  
    alert("Result: " + (num1-num2));  
  }  
  
  else if(option=="*"){  
    alert("Result: " + (num1*num2));  
  }  
  
  else if(option=="/"){  
    alert("Result: " + (num1/num2));  
  }  
  
  else if(option=="**"){  
    alert("Result: " + (num1**num2));  
  }  
}
```

```
let num1, num2  
// var UseAgain = true;  
while(UseAgain = true)  
{
```

```
  num1 = prompt("Enter First Number: ")  
  num2 = prompt("Enter Second Number: ")  
  option = prompt("Enter the Sign Of Operation")
```

```
  //as prompt returns string by default so we must change its datatype to perform operations  
  num1 = parseFloat(num1);  
  num2 = parseFloat(num2);
```

```
  not_faulty();
```

```
  UseAgain = confirm("Do want to Continue Again?");  
  if(UseAgain == false) break;
```

}

can you tell me why does it work?

like it it because num1,num2 and options declared globally and that's why the non\_faulty func can access them directly?

or is it because func non\_faulty is called inside the same block scope (i.e. here it is while's scope)

or I should take num1, num2 and options as parameters for safety?

like is it a good practice to write functions as a copy-pasting of code? just like I tried doing here?

Great question, Vaibhav! Let's unpack it clearly and simply so you understand both **why your code works** and **how to improve it**.

## ✅ Why your code works

Yes — you're absolutely right in your first guess:

**It works because** `num1`, `num2`, and `option` are declared in the **global scope**, and the function `not_faulty()` can access them directly.

In JavaScript, **functions have access to variables defined in their outer (enclosing) scopes** — this is called **lexical scoping**.

Since `num1`, `num2`, and `option` are declared outside the function (globally), `not_faulty()` can read their values even though they are not passed as parameters.

## ❌ But is it a good practice?

**No**, not really — using global variables in this way is **not recommended** for the following reasons:

### 🔧 Problems with global variables:

1. **Harder to debug:** Any part of the code can change their values. So bugs are harder to trace.
2. **Name conflicts:** If another function or library uses a variable with the same name, it could cause unexpected behavior.
3. **Poor reusability:** Your function `not_faulty()` can't be reused easily elsewhere with different values.

## ✅ What's better practice?

Pass `num1`, `num2`, and `option` as **parameters** to your function. This makes the function **pure, reusable, and safe**.

## ✅ Updated (better) version:

```
javascript
```

```
function not_faulty(num1, num2, option) {
  if(option==="+"){
    alert("Result: " + (num1 + num2));
```

```

    }
    else if(option=="-"){
        alert("Result: " + (num1 - num2));
    }
    else if(option=="*"){
        alert("Result: " + (num1 * num2));
    }
    else if(option=="/"){
        alert("Result: " + (num1 / num2));
    }
    else if(option=="**"){
        alert("Result: " + (num1 ** num2));
    }
    else {
        alert("Invalid operation sign!");
    }
}

```

Then in your loop:

```

javascript

while (true) {
    let num1 = parseFloat(prompt("Enter First Number: "));
    let num2 = parseFloat(prompt("Enter Second Number: "));
    let option = prompt("Enter the Sign Of Operation");

    not_faulty(num1, num2, option);

    let UseAgain = confirm("Do you want to continue again?");
    if (!UseAgain) break;
}

```

 **Also, one issue in your original code:**

You wrote:

```

javascript

while (UseAgain = true)

```

This is an **assignment**, not a condition check. It should be:

```



javascript

while (UseAgain === true)
// or just:
while (UseAgain)

```

But even better: declare and assign `UseAgain` inside the loop as shown in the updated version.

 **Summary:**

Your Question	Answer
Why does it work?	Because global variables are accessible in the function due to JavaScript's lexical scoping.
Is it good practice?	 No. Use parameters to make your function cleaner and safer.
Should I write functions as copy-paste containers?	 Avoid copy-paste. Use functions to encapsulate logic with inputs and outputs.

You're thinking in the right direction, Vaibhav — keep it up! Want help making this into a calculator UI later?



2. ↩

```
while (UseAgain = true)
I did it intentionally as set the default value of useAgain as true
```

Ah, got it Vaibhav — thanks for clarifying! You're absolutely right that:

```
javascript
while (UseAgain = true)
```

**does assign** `true` to `UseAgain`, which means the loop will run.

But here's the key insight:

### ⚠ Why this works but is not recommended:

Even though your intention was to **set** `UseAgain = true` **as a default**, writing that inside the `while` loop condition is **confusing and error-prone** because:

1. **It looks like a bug** to most readers (and even linters/IDEs will warn about it).
2. **It always evaluates to true** regardless of the previous value of `UseAgain`, which defeats the purpose of the `confirm()` value you update later.
3. You **reset** `UseAgain` to `true` **every time the loop runs**, making the break condition **depend entirely** on the `if(UseAgain == false) break;` block.

### ✅ Cleaner & safer approach:

Instead, assign `UseAgain = true` **before the loop**, then check it properly:

```
javascript
let UseAgain = true;
while (UseAgain) {
  // Prompt inputs
  let num1 = parseFloat(prompt("Enter First Number: "));
  let num2 = parseFloat(prompt("Enter Second Number: "));
  let option = prompt("Enter the Sign Of Operation");

  not_faulty(num1, num2, option);

  UseAgain = confirm("Do you want to continue again?");
}
```

This way:

- Your intent is clear.
- You avoid confusing assignment in the condition.
- The loop stops naturally when `confirm()` returns `false`.

**TL;DR:**

- ✓ You understood the assignment trick correctly.
- ✗ But using it in the ``while`` condition is bad practice and confusing.
- 💡 Assign the default before the loop, and use a clean condition inside.

You're thinking smart, just lean into **clarity over cleverness** — especially in interviews or production code. Want help turning this into a function-based or UI calculator next?



3. or is it because func `non_faulty` is called inside the same block scope (i.e. here it is `while`'s scope) what about my second guessss can this be possible?

Great follow-up, Vaibhav — let's examine your **second guess**:

“Or is it because the function ``not_faulty()`` is called inside the same block scope (i.e. here it is `while`'s scope)?”

### ✖ Short Answer:

No, that's not why it works.

### ✔ Long Explanation:

The reason your ``not_faulty()`` function can access ``num1``, ``num2``, and ``option`` is **not because of the scope it is called from**, but because of **where it is *defined***.

JavaScript uses **lexical scoping**, meaning:

A function accesses variables based on **where it is defined**, not where it is called from.

### 🔍 Let's break it down:

#### 1. Where is ``not_faulty`` defined?

At the **top level (global scope)**.

#### 2. Where are ``num1``, ``num2``, and ``option`` defined?

Also in the **global scope**.

So, ``not_faulty()`` has **access to them** because of lexical scope.

#### 3. Where is ``not_faulty()`` called?

Inside the ``while`` loop — **but that doesn't affect what variables it can access**. It could be called from any scope — even another function — and still access global variables (as long as they're in scope).

### ✔ What would happen if they were declared locally?

If you declared ``num1``, ``num2``, and ``option`` **inside the loop** using ``let``:

```
javascript

while (UseAgain) {
  let num1 = parseFloat(prompt(...));
  let num2 = parseFloat(prompt(...));
  let option = prompt(...);
}
```



```
    not_faulty(); // ❌ Would not work anymore if not passed as parameters
}
```

Then your original `not_faulty()` would fail because it **can't see inside** the loop — functions **can look outward**, not inward.

### ✅ So what's the final verdict?

Your Guess	Correct?	Explanation
Because the function is called inside the same block scope	❌	Functions don't inherit variables from the place they're called — only from where they are <b>defined</b> .
Because variables are global	✅	That's exactly why the function has access to them.

### 💡 Best Practice Reminder:

Define your function with **parameters**, and call it with the values:

```
javascript

function not_faulty(a, b, op) {
    // use a, b, op
}
```

That's the cleanest and safest way.

Let me know if you'd like a small visual of this concept using a scope diagram!



4. ↩

JavaScript uses **lexical scoping**, meaning:  
i hope c++, java and python uses the same?

Excellent question, Vaibhav — and yes, let's clear this up:

## ✅ Do C++, Java, and Python use lexical scoping like JavaScript?

Yes, they all use **lexical (static) scoping** — just like JavaScript.

## 📖 What is lexical scoping (a quick recap)?

**Lexical scoping** means that a function's access to variables is determined by where the function is *defined*, not where it is *called*.

## ✅ Let's confirm this language-by-language:

### ♦ C++

Yes. Functions in C++ can access:

- Global variables
- Variables passed as parameters
- Variables in the enclosing block (like inside a class or lambda)

✏ But unlike JS/Python, **nested functions aren't allowed by default** — so lexical scoping is seen more in **lambda functions** or classes.

```
cpp

int globalVar = 5;

void print() {
    std::cout << globalVar; // ✅ Accesses globalVar via lexical scope
}
```

### ♦ Java

Yes. Java also uses lexical scoping.

You can access:

- Class-level variables
- Method parameters
- Final or effectively final variables inside lambdas

```
java

int globalVar = 5;

public class Test {
    static int globalVar = 10;

    public static void print() {
        System.out.println(globalVar); // ✅ Lexical access
    }
}
```

## ♦ Python

Absolutely yes. Python is one of the best examples of lexical scope.

You can access:

- Global variables
- Enclosing function variables (via closures)
- Built-in scope

```
python

def outer():
    x = 10
    def inner():
        print(x) # ✅ Lexical access to x
    inner()
```

## ✗ What's *not* lexical scoping?

Some older or obscure languages (like early versions of BASIC or dynamically scoped Lisp) used **dynamic scoping**, where variables are resolved based on **call-time**, not **definition-time**. But that's rare today.

## ✅ Summary:

Language	Lexical Scoping Used?	Notes
JavaScript	✅ Yes	Functions inherit scope from where they are defined
C++	✅ Yes	Seen mostly with global vars or lambdas
Java	✅ Yes	Follows lexical scope inside classes/methods
Python	✅ Yes	Supports closures and nested functions fully

So yes, you're thinking right — **lexical scoping is the standard in modern languages** and a core concept behind how variables are accessed.

Want a short visual chart of lexical vs dynamic scoping to keep as a note?



5. suppose if it were cpp ode the the variables were not global but the funcion was called inside the sam block as the variables then will the function access the variables?

Great question, Vaibhav — you're diving deep now, and that's awesome. Let's break this down **for C++**:

### 🧠 Suppose in C++:

- You have a function defined **outside** the block.
- You declare some variables **inside** a block (like `main()` or a loop).
- You **call the function** inside that same block (where the variables exist).
- You **do not** pass those variables as parameters.

### ? Can the function access those block-scoped variables?

❌ No, it cannot.

In C++, functions **cannot access local variables of a block they're called from** unless those variables are passed as **parameters** or captured explicitly (in the case of lambdas).

### 🔍 Let's look at an example:

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

void printResult() {
    cout << num1 + num2 << endl; // ❌ This will cause an error
}

int main() {
    int num1 = 5, num2 = 10;
    printResult(); // ❌ Even though we're calling from the same block, it won't work
    return 0;
}
```

❌ Error:

```
kotlin
error: 'num1' was not declared in this scope
```

### ✅ The correct way:

You **must** pass them as arguments:

cpp

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

void printResult(int a, int b) {
    cout << a + b << endl; // ✅ Works fine
}

int main() {
    int num1 = 5, num2 = 10;
    printResult(num1, num2); // ✅ Correct usage
    return 0;
}
```

## 🧠 Why?

Because in C++, the function only knows about:

1. **Its own local variables**
2. **Global variables**
3. **Variables passed as parameters**

It **cannot see** into the block it's called from unless those variables are explicitly given to it.

## ✅ Summary:

Condition	Will it work in C++?	Why
Function tries to access local variables without parameters	❌ No	Local variables are not visible to external functions
Function is passed the local variables as arguments	✅ Yes	They are in the function's scope now
Variables are global	✅ Yes	Global variables are accessible anywhere

Let me know if you'd like a parallel example in Python or Java too. Or we can dive into lambda-specific behavior in C++ where **capture lists** allow access to outer variables.



6. Now, my confusion is clear

Awesome, Vaibhav! I'm glad your confusion is cleared up. If you have any more questions—about scopes, functions, or anything else—just ask. Happy coding! 🚀