Volatile

In this lesson, we'll understand how volatile variables behave.

WE'LL COVER THE FOLLOWING

Definition

> The **volatile** variable is one whose value may change due to an external event.

Usually, we can only change the value of a variable within our code. Let's say there is an external I/O event that tries to change the value of the variable. This would not be allowed.

However, it would be possible if the variable was volatile. A volatile variable can be declared using the `volatile` keyword.

```
volatile int myInt{2011}
```

We can find the keyword in Java and C# as well.

`volatile` vs `std::atomic`

What do the `volatile` keywords in C# and Java have in common with the `volatile` keyword in C++? Nothing!

It's so easy in C++.

1. `volatile` is for special objects, on which optimized read or write operations are not allowed.

2. `std::atomic` defines atomic variables, which are meant for thread-safe reading and writing. It's so easy, but the confusion starts exactly here. The `volatile` keyword in Java and C# is equivalent to `std::atomic` in C++. In other words, `volatile` has no multithreading semantics in C++.

`volatile` is typically used in embedded programming to denote objects that can change independently of the regular program flow. These are, for example, objects that represent an external device (memory-mapped I/O). Because these objects can change independent of the regular program flow, their values will be written directly in the main memory. Hence, there is no optimized storing in caches.

Example

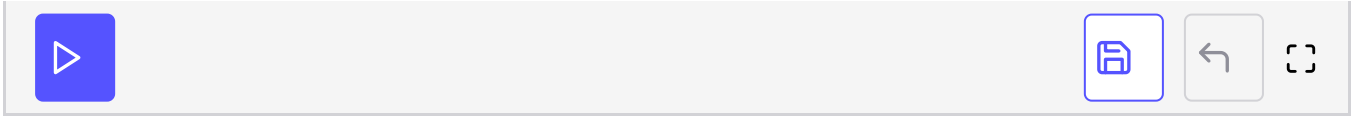
```
#include <iostream>
#include <thread>

volatile int x= 0;
volatile int y= 0;

void writing(){
  x= 2000;
  y= 11;
}

void reading(){
  std::cout << "y: " << y << " ";
  std::cout << "x: " << x << std::endl;
}

int main(){
  std::thread thread1(writing);
  std::thread thread2(reading);
  thread1.join();
  thread2.join();
};
```



So what happens when we declare the `int` variables as `volatile`? I guess we know. The program has a data race on the variables, `x` and `y`. So, the

program has undefined behavior and we cannot reason about `x` and `y`.

Further information

- data race

In the next chapter, we will discuss move semantics and perfect forwarding.