Hugo V. Teixeira

A little bit about coding, user interfaces, computer graphics and component-based software engineering.

Comparing the volatile keyword in Java, C and C++

DECEMBER 28, 2016DECEMBER 28, 2016 / HVIDAL

The **volatile** keyword exists in some programming languages, including Java, C and C++. I want to discuss in this blog post the differences, similarities and some details that every programmer should know about this keyword and help you understand how it affects your code. My focus will be only on Java, C and C++ because those are the languages I have most experience with. Feel free to share in the comments information about how other programming languages deal with the volatile modifier.

You probably know that Wikipedia has a good article about the volatile keyword (https://en.wikipedia.org/wiki/Volatile_(computer_programming)), but the article doesn't go very deep into the language details. The goal of this blog post is to compare the languages side by side and provide a bit more knowledge to anyone who is interested in this subject.

Declaring a volatile variable in Java

In Java, you can only add the volatile keyword to class members:

```
public class MyClass {
   private volatile int count = 0;
   public static volatile int sum = 0;
}
```

Arrays can be volatile, but the elements won't be volatile:

```
volatile int[] numbers = new int[10];
...
numbers[0] = 5;  // not volatile
```

Local variables cannot be volatile:

```
public void doStuff() {
   volatile int count = 0; // Error: illegal start of expression
}
```

Declaring a volatile variable in C and C++

In C and C++, the volatile keyword can be applied to local and global variables:

```
volatile int globalNumber = 0;
int* volatile pInt; // volatile pointer to nonvolatile int
volatile int* volatile pInt2; // volatile pointer to a volatile int

void doStuff() {
   volatile float localNumber = 5.0f;
}
```

structs can also have volatile fields:

```
struct A {
   volatile double n = 0.0;
};
```

In C++, class members can also be volatile:

```
class T {
   volatile long id = 0;
};
```

C++ also allows the volatile qualifier to be applied to return values, parameters and class methods. This is explained at the end of the post (see below).

In Java, volatile means memory visibility

When a variable is declared **volatile** in Java, the compiler is informed that such variable is **shared** and its value can be changed by another thread. Since the value of a volatile variable can change unexpectedly, the compiler has to make sure its value is propagated predictably to all other threads. Let's consider a simple piece of code:

```
volatile boolean isDone = false;
public void run() {
   while (!isDone)
      doWork();
}
```

The *isDone* variable is declared volatile. If that variable is set to *true* by another thread, this piece of code will read the new value and immediately leave the while-loop. If we forget to add the volatile modifier to that variable, there is a good chance the loop will never finish if the variable is set to true by another thread.

If a variable is declared volatile, all threads must read its most recent written value. The JVM does not cache volatile variables in registers or other caches that are hidden from other processors. This gives visibility to the variable and is considered a weaker form of synchronization between threads.

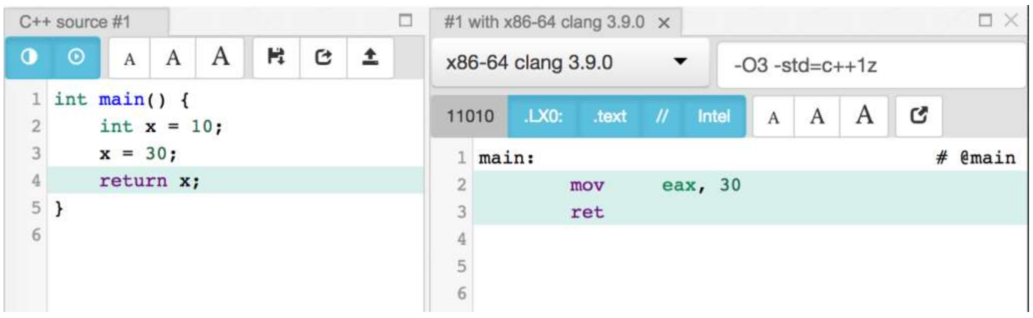
> **In Java, the volatile keyword guarantees memory visibility and it is meant to be used in concurrent programming.**

In C/C++, volatile means no optimizations

In C/C++, optimizations are done by the compiler. Most compilers have command line parameters that allow you to specify different levels of optimization. For example, take a look at some options from CLANG:

```
-O0 = no optimization
-O1 = somewhere between -O0 and -O2
-O2 = Moderate level of optimization, which enables most optimizations
-O3 = Like -O2 with extra optimizations to reduce code size
```

Some compiler optimizations include removing unused assignments and reordering of memory operations. This is easy to check with the gcc.goldbolt.com (http://gcc.godbolt.org/) website, let's take a look at a simple example:

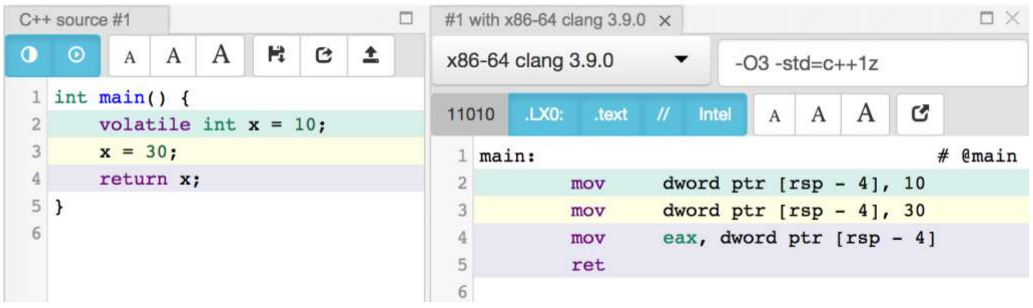


```
int main() {
    int x = 10;
    x = 30;
    return x;
}
```

```
main:                                   # @main
        mov     eax, 30
        ret
```

This code has an *int* variable x that receives a useless assignment on line 2. The compiler detects this issue, ignores the value 10 and decides to return 30 directly (see assembly code on the right).

The volatile keyword in C/C++ disables all optimizations on a variable. If we try the same code again, but with the volatile modifier applied to the x variable, the assembly code looks totally different:



```
int main() {
    volatile int x = 10;
    x = 30;
    return x;
}
```

```
main:                                   # @main
        mov     dword ptr [rsp - 4], 10
        mov     dword ptr [rsp - 4], 30
        mov     eax, dword ptr [rsp - 4]
        ret
```

Now the useless assignment isn't removed by the compiler and the assembly code on the right reflects exactly the source code on the left. You may ask why would someone use this kind of less efficient code. One answer is **memory-mapped I/O**, where Gadgets and electronic peripherals (e.g., sensors, displays, etc.) must communicate with the software in a very specific way. Optimizations could break the communication with those electronic parts.

Still in this topic, I have found an interesting difference between GCC and CLANG when it comes to optimizations and the volatile keyword. You can check it out in my previous article: a note about the volatile keyword in c++ (https://componenthouse.com/2016/10/21/a-note-about-the-volatile-keyword-in-cpp/)

> **In C/C++, the volatile keyword is NOT suitable for concurrent programming. It is meant to be used only with special memory (e.g., memory-mapped I/O).**

In Java, volatile doesn't guarantee atomicity and thread-safety

A common misunderstanding about the volatile keyword in Java is whether operations on such variables are atomic or not. The answer to this question can be YES and NO, so it depends on what the code is doing. Let me explain.

Operations are atomic if you look at individual reads and writes of the volatile variable. In other words, reading or writing a value works as if you are entering a synchronized block. So the integrity of the value is kept safe and there is no chance you will read a broken value because some other thread was still in the process of writing it.

Operations are NOT atomic if the code has to read the value before updating it. For example, consider this code:

```
volatile int i = 0;
i += 10;  // not atomic
```

The += operator isn't atomic because operations on volatile variables perform no locking. Because there is no locking, the executing thread cannot block other threads before the operation is completed. So *read-update-write* sequences on volatile variables are not thread-safe in Java. Another thread could just sneak into the sequence and create a race condition.

One way of implementing atomicity in Java is to use atomic classes from the *java.util.concurrent.atomic* package, like *AtomicInteger*, *AtomicBoolean*, *AtomicReference*, etc. The previous code would look like this:

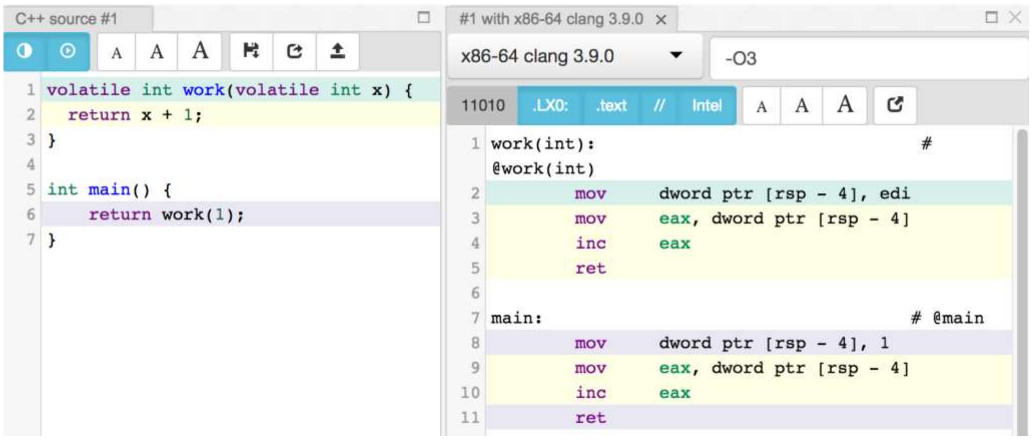
```
AtomicInteger i = new AtomicInteger(0);
i.addAndGet(10);  // atomic
```

In C/C++, volatile can be used in return types and parameters

C++ allows return types and parameters to be volatile. The following function takes a volatile int parameter and returns another volatile int:

```
volatile int work(volatile int x) {
  return x + 1;
}
```

As discussed before, the only difference the volatile qualifier makes in this case is that it won't be optimized by the compiler. The same function without the volatile keywords would be optimized. Here is the assembly code generated by the CLANG compiler:



x86-64 clang 3.9.0 -O3

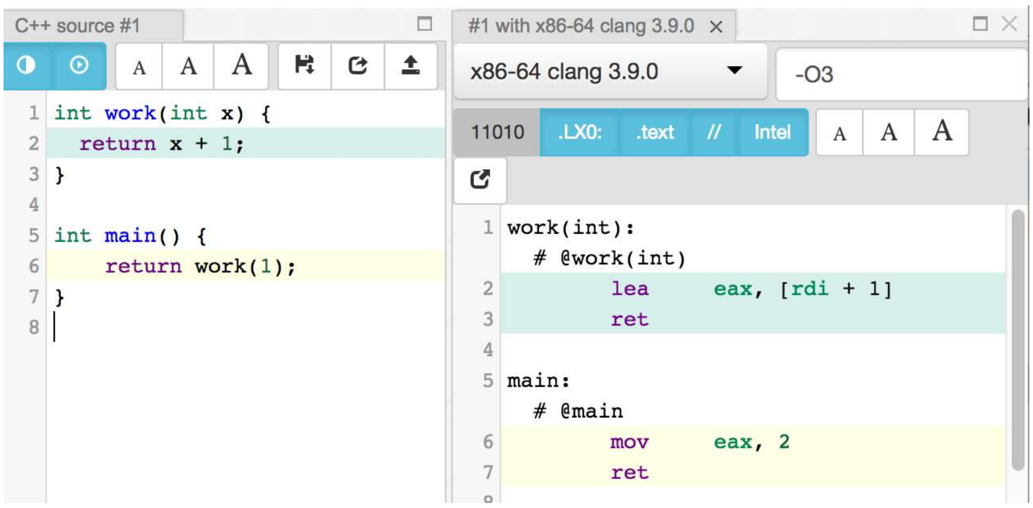
```
volatile int work(volatile int x) {
  return x + 1;
}

int main() {
    return work(1);
}
```

```
work(int):                              # @work(int)
        mov     dword ptr [rsp - 4], edi
        mov     eax, dword ptr [rsp - 4]
        inc     eax
        ret

main:                                   # @main
        mov     dword ptr [rsp - 4], 1
        mov     eax, dword ptr [rsp - 4]
        inc     eax
        ret
```

Now compare the same function without the volatile qualifiers. The assembly code is simplified:



x86-64 clang 3.9.0 -O3

```
int work(int x) {
  return x + 1;
}

int main() {
    return work(1);
}
```

```
work(int):
  # @work(int)
        lea     eax, [rdi + 1]
        ret

main:
  # @main
        mov     eax, 2
        ret
```

In C++, volatile can also be used in class methods, constructors and operators

C++ also allows you to use the volatile qualifier in class methods. Here is a simple example:

```
class MyClass {
  public:
   int normalMethod() {
      return 1;
    }
   int volatileMethod() volatile {
      return 2;
    }
};
```

If an instance of this class is declared volatile, then **the code can only call volatile methods on that object:**

```
int main() {
   volatile MyClass a;
   a.normalMethod();  // error: nonvolatile method cannot be called
   a.volatileMethod();  // OK
   return 0;
}
```

If the instance is not volatile, then the code can call both volatile and nonvolatile methods:

```
int main() {
   MyClass a;
   a.normalMethod();  // OK
   a.volatileMethod();  // OK
   return 0;
}
```

Conclusion

The meaning of the volatile keyword differs from language to language and this blog post briefly explains how it works in Java, C and C++.

In Java, volatile variables are meant to be used in multithreaded contexts. Threads that concurrently read a volatile variable will always read its latest value, which means the JVM will not cache the variable internally and its value will be shared across all processors.

Atomicity and thread-safety in Java are not guaranteed by volatile variables. In summary, you should use the volatile keyword if writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value. Besides that, there shouldn't be other variables participating in the concurrent state of the class, otherwise you will have race conditions. Finally, remember that locking is not required for any other reason while the variable is being accessed.

In C/C++, the volatile keyword is meant to be used with special memory: manipulating I/O registers or memory-mapped hardware. All optimizations on those variables will be disabled, which means that assignments and their order will be preserved. The volatile qualifier doesn't help us in multithreaded code.

We can finish with a table that summarizes the points discussed:

	Java	**C / C++**

	Java	**C / C++**
Purpose	Concurrent Programming	Special Memory (e.g., memory mapped I/O)
What difference does it make?	Adds visibility to the variable	Disables optimizations on the variable
Is the operation atomic?	**Yes** for individual reads and writes; **No** if the write depends on the current value;	**No**

Applies to	class fields	local and global variables;
		
		return types;
		
		function parameters;
		
		class fields and methods (C++ only);

Tags: C++, Java

5 thoughts on “Comparing the volatile keyword in Java, C and C++”

1. Pingback: Myths Programmers Believe about CPU Caches – Software the Hard way
2. Pingback: [Перевод] Мифы о кэше процессора, в которые верят программисты - Новини дня
3. **WangJinge** says:
   SEPTEMBER 11, 2019 AT 11:32 PM
   Why do you say “volatile in C++ is not used for concurrent programming”? Volatile variable always compiled to memory address instead of register, which also add visibility to the variable.

   REPLY
4. Pingback: Myths Programmers Imagine about CPU Caches – Tech 'n' Gadget News
5. **Lonna Baynes** says:
   JANUARY 31, 2020 AT 4:52 AM
   Are you seeking effective online promotion that actually gets good results? I apologize for sending you this message on your contact form but actually that was the whole point. We can send your ad message to sites through their contact pages just like you’re receiving this note right now. You can target by keyword or just execute bulk blasts to sites in any country you choose. So let’s say you’re looking to send a message to all the real estate agents in the

United States, we'll scrape websites for only those and post your ad text to them. Providing you're advertising some kind of offer that's relevant to that type of business then you'll be blessed with an amazing response!

Send an email to mark3545tho@gmail.com to get details about how we do this

REPLY

WORDPRESS.COM.