Free Atomic Functions

This lesson gives an overview of free atomic functions used from the perspective of concurrency in C++.

WE'LL COVER THE FOLLOWING

- `std::shared_ptr`
  - Assertions:

The functionality of the flag `std::atomic_flag` and the class template `std::atomic` can also be used with free functions. Because these functions use pointers instead of references, they are compatible with C. The atomic free functions are available for the types that you can use with the class template `std::atomic`.

There is one prominent exception to the rule: you can apply the atomic free functions to the smart pointer `std::shared_ptr`.

`std::shared_ptr`

`std::shared_ptr` is the only non-atomic data type on which you can apply atomic operations. First, let me write about the motivation for this exception.

The C++ committee saw the necessity that instances of smart pointers should provide a minimal atomicity guarantee if they are accessed with atomic operations. What is the meaning of the minimal atomicity guarantee for `std::shared_ptr`? Atomic operations on `std::shared_ptr` will increase and decrease the reference-counter in a thread-safe way because the control block of `std::shared_ptr` is thread-safe. You also have the guarantee that the resource will be destroyed exactly once.

Assertions:

The assertion that a `std::shared_ptr` provides are described by Boost.

1. A shared_ptr instance can be “read” (accessed using only const operations) simultaneously by multiple threads.
2. Different shared_ptr instances can be “written to” (accessed using mutable operations such as operator= or reset) simultaneously by multiple threads (even when these instances are copies, and share the same reference count underneath).

To make the two statements clear, let me show a simple example. When you copy a `std::shared_ptr` in a thread, all is fine.

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

int main(){
  std::shared_ptr<int> ptr = std::make_shared<int>(2011);

  for (auto i= 0; i<10; i++){
     std::thread([ptr]{
       std::shared_ptr<int> localPtr(ptr);
       localPtr = std::make_shared<int>(2014);
       std::cout<<"localPtr: "<<*(localPtr)<<std::endl;
      }).detach();
    }
}
```

Let’s first look at line 10. By using copy construction for the `std::shared_ptr` `localPtr`, only the control block is used. This is thread-safe. Line 11 is a little bit more interesting; the `localPtr` is set to a new `std::shared_ptr`. This is not a problem from the multithreading point of view, as the lambda-function (line 9) binds `ptr` by copy. Therefore, the modification of `localPtr` takes place on a copy. The story will change dramatically if I get the `std::shared_ptr` by reference.

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

int main(){
```

```
int main(){
  std::shared_ptr<int> ptr = std::make_shared<int>(2011);

  for (auto i= 0; i<10; i++){
     std::thread([&ptr]{
       ptr= std::make_shared<int>(2014);
       std::cout<<"ptr: "<<*(ptr)<<std::endl;
     }).detach();
  }
}
```

The lambda-function binds the `std::shared_ptr ptr` in line 9 by reference. Therefore, the assignment (line 10) is a race condition on the resource and the program has undefined behavior.

Admittedly that last example was not very easy to achieve. `std::shared_ptr` requires special attention in a multithreading environment. They are very special; they are the only non-atomic data types in C++ for which atomic operations exist.