Thread-Safe Linked List Using Atomic Pointers

This lesson describes the use of thread-safe singly linked list using atomic pointers.

> Let’s see the C++11 version of the thread-safe singly linked list first; then we’ll enhance it using atomic smart pointers from C++20.

```
#include <iostream>
#include <atomic>
#include <memory> //for shared_ptr
using namespace std;

template<typename T> class concurrent_stack {
  struct Node {
      T t;
      shared_ptr<Node> next;
    };
     shared_ptr<Node> head;
  concurrent_stack(concurrent_stack &) = delete;
  void operator=(concurrent_stack &) = delete;

 public:
    concurrent_stack() = default;
   ~concurrent_stack() = default;

  class reference{
      shared_ptr<Node> p;
    public:
      reference(shared_ptr<Node> p_) : p{p_} { }
      T& operator* () {return p->t;}
      T* operator->() {return &p->t;}
    };

  auto find( T t) const {
      auto p = atomic_load(&head);
      while(p && p->t != t)
          p = p->next;
      return reference(move(p));
    }

  auto front() const{
       return reference(atomic_load(&head));
    }

  void push_front(T t){
       auto p = make_shared<Node>();
       p->t = t;
       p->next = atomic_load(&head);
    while(p && !atomic_compare_exchange_weak(&head,&p->next,p)){ }
    }
    }

  void pop_front(){
      auto p = atomic_load(&head);
      while(p && !atomic_compare_exchange_weak(&head,&p,p->next)){ }
    }

};

int main(){
    concurrent_stack<int> cS;
    cS.push_front(3);
    cS.push_front(6);
    cS.find(6);
    cS.pop_front();
    cS.front();
}
```

Now let’s see how we can use atomic smart pointers to modify the above code:

```
template<typename T> class concurrent_stack {
  struct Node {
      T t;
      shared_ptr<Node> next;
    };
  atomic_shared_ptr<Node> head;
      // in C++11: remove "atomic_" and remember to use the special.
      // functions every time you touch the variable
  concurrent_stack(concurrent_stack &) = delete;
  void operator=(concurrent_stack &) = delete;

 public:
    concurrent_stack() = default;
   ~concurrent_stack() = default;

  class reference{
      shared_ptr<Node> p;
    public:
      reference(shared_ptr<Node> p_) : p{p_} { }
      T& operator* () {return p->t;}
      T* operator->() {return &p->t;}
    };

  auto find( T t) const {
      auto p = head.load(); // in C++11: atomic_load(&head)
      while(p && p->t != t)
          p = p->next;
      return reference(move(p));
    }

  auto front() const{
       return reference(head); // in C++11: atomic_load(&head)
    }

  void push_front(T t){
       auto p = make_shared<Node>();
       p->t = t;
       p->next = head;       // in C++11: atomic_load(&head)
      while(!head.compare_exchange_weak(p->next,p)){ }

     // in C++11: atomic_compare_exchange_weak(&head,&p->next,p)
    }

  void pop_front(){
      auto p = head.load(); // in C++11: atomic_load(&head)
      while(p && !head.compare_exchange_weak(p,p->next){ }
    // in C++11: atomic_compare_exchange_weak(&head,&p,p->next)
    }

};
```

C++20’s type system does not permit it to use a non-atomic operation on a `std::atomic_shared_ptr`.