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

using namespace std;
//long sum{ 0 };

void Increment(int i, atomic<int>& I) {
	for (int j = 0; j < i; ++j) I += 5;//load and store won't be able to express this.
}

int main() {
	//int I{ 0 };
	atomic<int> I(0);
	Increment(1000000, I);
	cout << I.load()<< endl;
	thread t1(Increment, 200000, ref(I));
	thread t2(Increment, 200000, ref(I));
	thread t3(Increment, 200000, ref(I));
	thread t4(Increment, 200000, ref(I));
	thread t5(Increment, 200000, ref(I));

	t1.join();
	t2.join();
	t3.join();
	t4.join();
	t5.join();

	cout << I.load() << endl;
	return 0;
}

//READ-MODIFY-WRITE ATOMIC

//What data type can be made stomic?
```

std::atomic
template <class T> struct atomic;
T: trivially copyable type.

trivially copyable type:

A trivially copyable type is a type whose storage is contiguous
(and thus its copy implies a trivial memory block copy,
as if performed with memcpy). This is true for scalar types,
trivially copyable classes and arrays of any such types.

A trivially copyable class is a class (defined with class, struct or union) that:
//extern, volatile, explicit, no-except, default, static, const, constexpr, mutable, preprocessor

uses the implicitly defined copy and move constructors, copy and move assignments (L-value operator= and R-value operator=), and destructor.
has no virtual members.
its base class and non-static data members (if any) are themselves also trivially copyable types.

```
// is_trivially_copiable example
#include <iostream>
#include <type_traits>

struct A { int i; };

struct B {
    int i,j;
    B (const B& x) : i(x.i), j(1) {};     // copy ctor
};

int main() {
    std::cout << std::boolalpha;
    std::cout << "is_trivially_copyable:" << std::endl;
    std::cout << "int: " << std::is_trivially_copyable<int>::value << std::endl;
    std::cout << "A: " << std::is_trivially_copyable<A>::value << std::endl;
    std::cout << "B: " << std::is_trivially_copyable<B>::value << std::endl;
    return 0;
}
```

Output:

```
is_trivially_copyable:
int: true
A: true
B: false
```

```
#include <iostream>
#include <type_traits>

struct A {
        int m;
};

struct B {
        B(B const&) {}
};

struct C {
        virtual void foo();
};

struct D {
        int m;

        D(D const&) = default; // -> trivially copyable

        D(int x): m(x+1) {}
};

int main()
{
        std::cout << std::boolalpha;
        std::cout << std::is_trivially_copyable<A>::value << '\n';
        std::cout << std::is_trivially_copyable<B>::value << '\n';
        std::cout << std::is_trivially_copyable<C>::value << '\n';
        std::cout << std::is_trivially_copyable<D>::value << '\n';
}
```

Output:

```
true
false
false
true
```

```
std::atomic<T> t;

//What operations can be performed on atomic?
/*
Assignment.
Some common operations.
Some data type dependent operations.
*/


#include <iostream>
#include <atomic>

using namespace std;
struct A {
            long x;
            long y;
            long z;
```

```
    A() {}
    A(long i, long j, long k): x(i), y(j), z(k){}
//~A() {}; vs ~A() = default;
};
int main() {
    atomic<int> x = 10;

    ++x;//atomic
    x++;//atomic
    x += 1;//atomic
    x |= 2;//atomic
    //*= 2;//compile time error
    int y = x * 2; //atomic
    x = y + 1;//atomic
    x = x + 1;//atomic; Race!!
    x = x * 2;//atomic; Race!!

    atomic<A> a({ 3, 4, 5});
    //AA.store({ 2,3 });
    cout << a.load().x << " " << a.load().y << " "<<a.load().z<<endl;
    //atomic<A> aa{ 5 };

    cout << boolalpha << std::is_trivially_copyable<A>::value << endl;
    return 0;
}
```

Some common operations on std::atomic<T> x;

```
//read and write
T y = x.load();     //y = x; OK
x.store(y); //x = y;     OK

//atomic exchange
T z = x.exchange(y); //z = x; x=y;     One atomic operation.
//x = x+5; //two atomic operations


//compare-and-swap
bool Bool = x.compare_exchange_strong(y,z);
//if x==y, make x=z and return true;
//Else, set y = x and return false;


//Example: atomic increment with CAS
std::atomic<int> x {10};
int x0 = x;
while (!x.compare_exchange_strong(x0, x0+1)) {}

//x becomes x+1, whichis an atomic operation
//atomic increment used to be only valid for int;
```

```
//Now, increment operations can be performed atomically even if it is not of type int.
//The concept can be used for increment more general type (such as doubles),
//multiply integers, and many more.

while (!x.compare_exchange_strong(x0, x0*2)){}
//x becomes x*2.     An atomic operation.
```