In addition to booleans, there are atomics for pointers, integrals and user defined types. The rules for user-defined types are special.

Both. The atomic wrapper on a pointer T\* std::atomic<T\*> or on an integral type integ std::atomic<integ> enables the CAS (compare-and-swap) operations.

std::atomic<T\*>

The atomic pointer std::atomic<T\*> behaves like a plain pointer T\*. So std::atomic<T\*> supports pointer arithmetic and pre- and post-increment or pre- and post-decrement operations. Have a look at the short example.

int intArray[5];

std::atomic<int\*> p(intArray);

p++;

assert(p.load() == &intArray[1]);

p+=1;

assert(p.load() == &intArray[2]);

--p;

assert(p.load() == &intArray[1]);

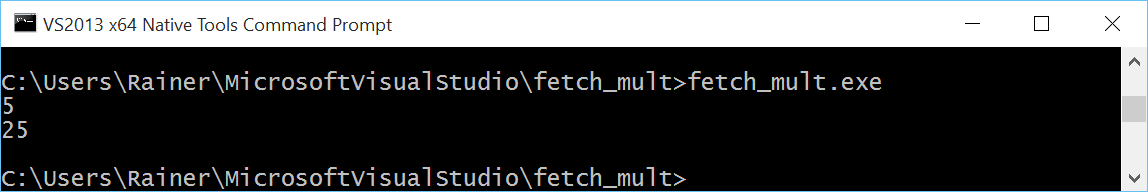
std::atomic<integral type>

In C++11 there are atomic types to the known integral data types. As ever you can read the whole stuff about atomic integral data types - including there operations - on the page [en.cppreference.com.](http://en.cppreference.com/w/cpp/atomic/atomic)A std::atomic<integral type> allows all, what a std::atomic\_flag or a std::atomic<bool> is capable of, but even more.

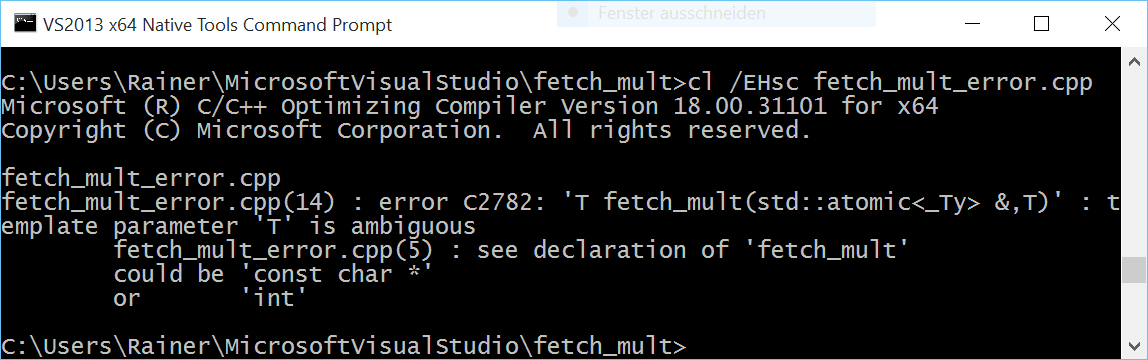
The composite assignment operators +=, -=, &=, |= and ^= and there pedants std::atomic<>::fetch\_add(), std::atomic<>::fetch\_sub(), std::atomic<>::fetch\_and(), std::atomic<>::fetch\_or() and std::atomic<>::fetch\_xor() are the most interesting ones. There is a little difference in the atomic read and write operations. The composite assignment operators return the new value, the fetch variations the old value. A deeper look gives more insight. There is no multiplication, division and shift operation in an atomic way. But that is not that big restriction. Because these operations are relatively seldom needed and can easily be implemented. How? Look at the example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // fetch\_mult.cpp  #include <atomic>  #include <iostream>  template <typename T>  T fetch\_mult(std::atomic<T>& shared, T mult){  T oldValue= shared.load();  while (!shared.compare\_exchange\_strong(oldValue, oldValue \* mult));  return oldValue;  }  int main(){  std::atomic<int> myInt{5};  std::cout << myInt << std::endl;  fetch\_mult(myInt,5);  std::cout << myInt << std::endl;  } |

I should mention one point. The addition in line 9 will only happen, if the relation oldValue == shared holds. So to be sure that the multitplication will always take place, I put the multitplication in a while loop. The result of the program is not so thrilling.



The implementations of the function template fetch\_mult is generic, too generic. So you can use it with an arbitrary type. In case I use instead of the number 5 the C-String 5, the Microsoft compilers complains that the call is ambiguous.



"5" can be interpreted as a const char\* or as an int. That was not my intention. The template argument should be an integral type. The right use case for concepts lite. With concepts lite, you can express constraints to the template parameter. Sad to say but they will not be part of C++17. We should hope for C++20 standard.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | template <typename T>  requires std::is\_integral<T>::value  T fetch\_mult(std::atomic<T>& shared, T mult){  T oldValue= shared.load();  while (!shared.compare\_exchange\_strong(oldValue, oldValue \* mult));  return oldValue;  } |

The predicate std::is\_integral<T>::value will be evaluated by the compiler. If T is not an integral type, the compiler will complain. std::is\_integral is a function of the new [type-traits library](http://en.cppreference.com/w/cpp/header/type_traits), which is part of C++11. The requires condition in line 2 defines the constraints on the template parameter. The compiler checks the contract at compile time.

You can define your own atomic types.

## std::atomic<user defined type>

There are a lot of serious restrictions on a user defined type to get an atomic type std::atomic<MyType>. These restrictions are on the type, but these restrictions are on the available operations that std::atomic<MyType> can perform.

For MyType there are the following restrictions:

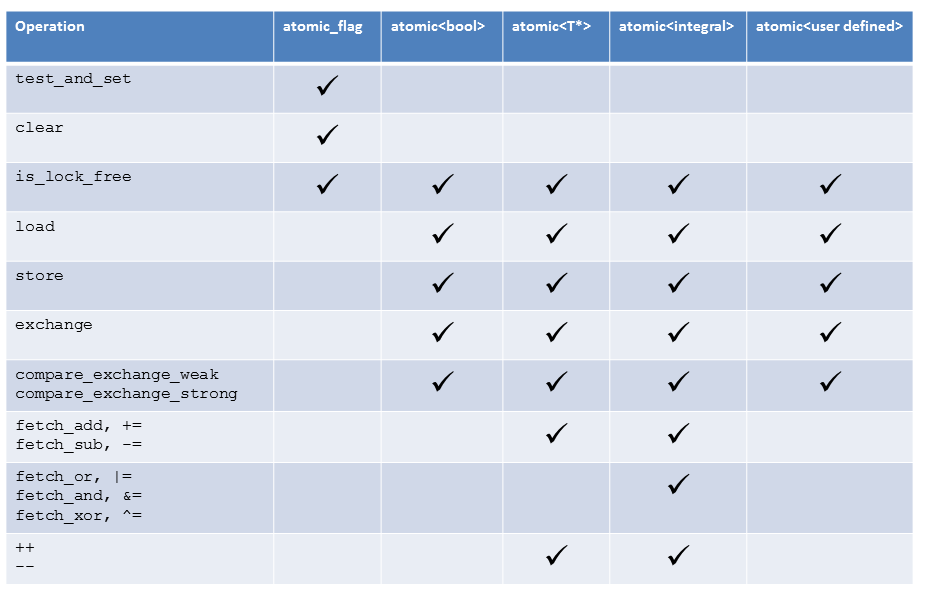
* The copy assignment operator for MyType, for all base classes of MyType and all non-static members of MyType must be trivial. Only an automatically by the compiler generated copy assignment operator is trivial. To say it the other way around. User defined copy assignment operators are not trivial.
* MyType must not have virtual methods or base classes.
* MyType must be bitwise comparable so that the C functions memcpy or memcmp can be applied.

You can check the constraints on MyType with the function std::is\_trivially\_copy\_constructible, std::is\_polymorphic and std::is\_trivial at compile time. All the functions are part of the [type-traits library.](http://en.cppreference.com/w/cpp/header/type_traits)

For the user defined type std::atomic<MyType> only a reduced set of operations is supported.

## Atomic operations

To get the great picture, I displayed in the following table the atomic operations dependent on the atomic type.



## Free atomic functions and smart pointers

The functionality of the class templates std::atomic and the Flag std::atomic\_flag can be used as a [free function](http://en.cppreference.com/w/cpp/atomic). Because the free functions use atomic pointers instead of references they are compatible with C. The atomic free functions support the same types as the class template std::atomic but in addition to that the smart pointer std::shared\_ptr. That is special because of std::shared\_ptr is not an atomic data type. The C++ committee recognised the necessity, that instances of smart pointers that maintain under their hood the reference counters and object must be modifiable in an atomic way.

std::shared\_ptr<MyData> p;

std::shared\_ptr<MyData> p2= std::atomic\_load(&p);

std::shared\_ptr<MyData> p3(new MyData);

std::atomic\_store(&p, p3);

To be clear.**The atomic characteristic will only hold for the reference counter, but not for the object.** That was the reason,  we get a std::atomic\_shared\_ptr in the future (I'm not sure if the future is called C++17 or C++20. I was often wrong in the past.), which is based on a std::shared\_ptr and guarantees the atomicity of the underlying object. That will also hold for std::weak\_ptr. std::weak\_ptr, which is a temporary owner of the resource, helps to break cyclic dependecies of std::shared\_ptr. The name of the new atomic std::weak\_ptr will be std::atomic\_weak\_ptr. To make the picture complete, the atomic version of std::unique\_ptr is called std::atomic\_unique\_ptr.