

Implementation of Critical Resource Concurrency Pattern in C++

The article is written by Vladimir Frolov, system developer of ApriorIT. The topic area of concurrency patterns is poorly covered in literature, so this text is called to fill a gap, at least partly.

Many of those who have ever used Critical Section object for synchronization between threads were applying it together with Auto Critical Section object. Indeed, Auto Critical Section is a powerful synchronization object which use has a set of advantages. The code containing Auto Critical Section for Critical Section management turns safe, more readable and structured when compared with the code directly working with Critical Section. Moreover, Auto Critical Section has trivial implementation. Regarding Auto Critical Section object one might say that it *wraps* in Critical Section the scope in which it is being created.

However, whether Auto Critical Section used or not, there is still a series of problems relating to Critical Section object:

- the Critical Section and resource which it protects are being created separately and can potentially have different lifetime.
- the code which uses the resource must know that access to it is synchronized by critical section.

Both these problems are of little importance if the critical section is used for creation of a thread-safe class. In that case the critical section is being created as data member and about it should be aware only function members of the given class while creating an Auto Critical Section object on call. However, a problem appears if it is necessary to provide access to a resource or class object which doesn't make any attempts at synchronization from within.

Log system can serve as an example here. For logging purpose it is reasonable to create an object of `std::ofstream` class and give global access to this object. Though, if single-threaded application using such tactics exists, then start of one more **thread** for logging purpose in this same application can lead to a number of problems:

- It will be necessary to create a critical section near object of `std::ofstream` class and give a global access to it.
- It will be necessary to wrap up all places already using the object of `std::ofstream` class in Auto Critical Section, and a problem here is that if any place appears not wrapped up such mistake will be hard to reveal.

The Critical Resource structural pattern serves for solving these problems.

The known synonyms are: Common Resource, Guardian Type.

Purpose: wraps not-synchronized object (unlike Auto Critical Section object which wraps scope) in critical section so that access to an object could be provided only through an entry in critical section, the leaving from which occurs right after end of access. So, Critical Resource pattern creates powerful alternative to using WinAPI Interlocked functions family.

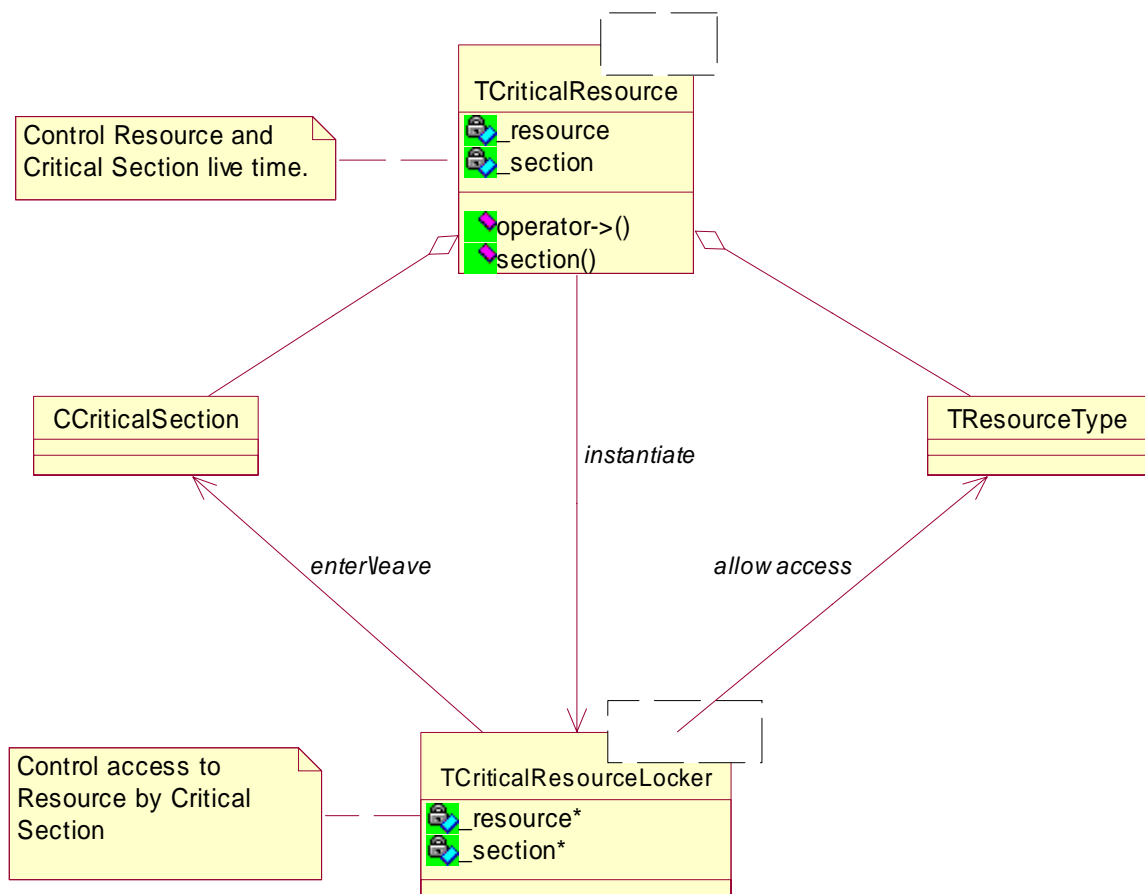
Critical Resource object can be implemented by means of temporary objects via which access to a resource provided (by analogy with Auto Critical Section pattern which is implemented by means of automatic object).

In C++ it's the most naturally implementing Critical Resource as a template which accepts as a parameter the resource type and has as data members the resource itself and critical section, controlling access to it. Access to a resource will be made via temporary object of Resource Locker class which will be storing pointer to a resource and corresponding Critical Section. Critical Resource should have a function member returning temporary object of Resource Locker class which, by analogy with Auto Critical Section, will be entering into critical section at constructor and leave it at destructor. Resource Locker class as well should have a function member giving access to the reference to a resource. It should be noted that in C++ along with Resource Locker class, a Const Resource Locker class must be implemented. This class will provide access to a constant resource and objects of this class will be returned by constant function member of Critical Resource object.

Also it is important to take into account that sometimes it is required to synchronize not only an access to a resource, but also a series of such accesses. For this purpose section () method in Critical Resource class should be implemented in order to return a reference to the critical section. Thus, if critical section is recursive (the critical section is recursive if one thread can enter it several times, at that in order some else thread could enter given critical section, the first one has to leave it as much times as it had entered. WinAPI CRITICAL_SECTION synchronization object is recursive, but if implementation of critical section is performed on the basis of Auto Reset Event synchronization object, then in order to make such critical section recursive, additional actions should be undertaken), then there is no necessity to implement in the Critical Resource a function member providing uncontrollable access to a resource. Resource Locker will be entering into critical section recursively.

Also should be noticed that at Critical Resource' template it is convenient to make specialization for references. So it will be possible to create Critical Resource supervising access to already existing objects (for example, std::cout).

At the diagram Critical Resource concurrency pattern can look as follows:



Implementation of Critical Resource concurrency pattern can look this way:

```
// It is supposed that CCriticalSection class is already implemented in
// the CriticalSection.h file
#include "CriticalSection.h"

// TCriticalResource class is template and receives as a parameter
// the type of a resource.
template<typename TResourceType>
class TCriticalResource
{
private:
// Resource Locker and Const Resource Locker classes are private members of
// TCriticalResource to avoid creation of their objects by somebody else.

class TResourceLocker;
class TConstResourceLocker;

public:
// The constructor creates by default a new instance of resource
// and critical section.
TCriticalResource();

// The constructor from a resource. Creates new instance of a resource
// by means of the copy constructor.
explicit TCriticalResource(const TResourceType&);

// The copy constructor. Creates resource copy and new critical section.
```

```
TCriticalResource(const TCriticalResource<TResourceType>&);

// The assignment operator, assigns one resource to another, performing
// access control at the same time by means of critical sections, but
// not entering simultaneously in both. For this purpose initially
// the copy of a first resource is being created, and then it is being
// assigned to the second one.
TCriticalResource<TResourceType>& operator=(const TCriticalResource<TResourceType>&);

// operators serve for receiving time object which
// will be performing access control to a resource.
TResourceLocker operator->();
TResourceLocker operator*();
TConstResourceLocker operator->() const;
TConstResourceLocker operator*() const;

// It is dead-lock unsafe to use many critical resources in one expression,
// but it is safe to use many copies of resources there if any of critical
// resources used.
TResourceType copy() const;

// To get access to critical section for synchronizing a series of accesses
// to a resource. It's recommended to use the result of a function as
// a parameter of the constructor of CAutoCriticalSection object
// to guarantee that thread leaves Critical Section.
// It's assumed that CCriticalSection is recursive.
sync::CCriticalSection& section();
private:
// The resource object access to which is being controlled.
TResourceType _resource;

// Critical Section object by means of which access to a resource
// is being controlled.
// The object is declared as mutable in order to enable control over
// a constant resource.
mutable sync::CCriticalSection _access;
}; // class TCriticalResource

// Implementation of Resource Lockers
template<typename TResourceType>
class TCriticalResource<TResourceType>:TResourceLocker
{
public:
// Constructor of a Locker accepts resource object which is being controlled and
// Critical Section object by means of which the control is being performed.
TResourceLocker(TResourceType&, sync::CCriticalSection&);

TResourceLocker& operator=(const TResourceType&);

// This constructor is needed to enable creation of
// time objects of TResourceLocker class, by means of which
// access will be provided. The constructor also assumes recursiveness of
// Critical Section because it should enter in Critical Section from which
// the copied object has not leave yet. There is no danger of entering in deadlock while
// object
// created in one thread and copied to another is being copied, because
// objects of TResourceLocker class will be created only by TCriticalResource,
// they will be temporary and be copied within one thread.
TResourceLocker(const TResourceLocker&);

// Destructor leaves Critical Section and by doing that it free the access to a resource
// from other threads
~TResourceLocker() throw();
```

```
// Ensuring access to a resource.
operator TResourceType&() const;
TResourceType& get() const;

// Operators are needed to make it possible to organize corresponding operators
// TCriticalResource
TResourceLocker* operator->();
TResourceLocker* operator*();

private:
// deadlock-unsafe operator of assignment one Resource Locker to another.
// Has no implementation.
TResourceLocker& operator=(const TResourceLocker&);

// Pointer to a resource and Critical Section of access to it, being kept in
TCriticalResource
TResourceType* _resource;
sync::CCriticalSection* _access;
}; // class TResourceLocker

// Provides controllable access to a constant resource.
template<typename TResourceType>
class TCriticalResource<TResourceType>::TConstResourceLocker
{
//..
// Implementation is similar to TResourceLocker except for instructions regarding
constantness of a resource.
}; // class TConstResourceLocker

// ...
// Specialization of reference for TCriticalResource is similar.

// Implementation of TResourceLocker class is almost obvious.

template<typename TResourceType>
TCriticalResource<TResourceType>::TResourceLocker::TResourceLocker(TResourceType&
resource,
sync::CCriticalSection& access
)
: _resource(&resource)
, _access (&access)
{
_access-> enter ();
}

template<typename TResourceType>
TCriticalResource<TResourceType>::TResourceLocker::TResourceLocker
(const TResourceLocker& resource)
: _resource(resource._ resource)
, _access(resource._ access)
{
_access->enter();
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::TResourceLocker&
TCriticalResource<TResourceType>::TResourceLocker::operator=(const TResourceType&
resource)
{
(*_resource) = resource;
return (*this);
}

template<typename TResourceType>
```

```
TCriticalResource<TResourceType>::~TResourceLocker::~~TResourceLocker() throw ()
{
    _access->leave();
}

template<typename TResourceType>
TCriticalResource<TResourceType>::~TResourceLocker::operator TResourceType&() const
{
    return (*_resource);
}

template<typename TResourceType>
TResourceType& TCriticalResource<TResourceType>::~TResourceLocker::get() const
{
    return (*_resource);
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::~TResourceLocker*
TCriticalResource<TResourceType>::~TResourceLocker::operator->()
{
    return this;
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::~TResourceLocker*
TCriticalResource<TResourceType>::~TResourceLocker::operator*()
{
    return this;
}

// ...
// TConstResourceLocker class is being implemented similar to TResourceLocker

// Implementation of TCriticalResource

template<typename TResourceType>
TCriticalResource<TResourceType>::~TCriticalResource()
: _resource ()
, _access ()
{
}

template<typename TResourceType>
TCriticalResource<TResourceType>::~TCriticalResource(const TResourceType& resource)
: _resource(resource)
, _access()
{
}

template<typename TResourceType>
TCriticalResource<TResourceType>::~TCriticalResource(const TCriticalResource
<TResourceType>& resource)
: _resource(resource._ resource)
, _access ()
{
}

template<typename TResourceType>
TCriticalResource<TResourceType>& TCriticalResource<TResourceType>::operator=
(const TCriticalResource<TResourceType>& resource)
{

// The copy of a resource is being created, therefore operator= does not enter in two
// Critical Section simultaneously.
```

```
    if (this != &resource)
    {
        TResourceType resourceCopy = resource.copy();
        (*this)->get() = resourceCopy;
    }
    return (*this);
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::TResourceLocker
TCriticalResource<TResourceType>::operator->()
{
    return TCriticalResource<TResourceType>::TResourceLocker(_resource, _access);
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::TConstResourceLocker
TCriticalResource<TResourceType>::operator->() const
{
    return TCriticalResource<TResourceType>::TConstResourceLocker(_resource, _access);
}

template <typename TResourceType>
typename TCriticalResource<TResourceType>::TResourceLocker
TCriticalResource<TResourceType>::operator*()
{
    return TCriticalResource<TResourceType>::TResourceLocker(_resource, _access);
}

template<typename TResourceType>
typename TCriticalResource<TResourceType>::TConstResourceLocker
TCriticalResource<TResourceType>::operator*() const
{
    return TCriticalResource<TResourceType>::TConstResourceLocker(_resource, _access);
}

template<typename TResourceType>
TResourceType TCriticalResource<TResourceType>::copy() const
{
    return (*this)->get();
}

template<typename TResourceType>
sync::CCriticalSection& TCriticalResource<TResourceType>::section()
{
    return _access;
}
```

Examples of using such TCriticalResource can look as follows:

```
TCriticalResource<int> intResource;

(*intResource) = 10;
// In this place other thread can access to intResource and change resource value.
intResource->get() += 10;

TCriticalResource<std::ostream&> sync_cout(std::cout);

sync_cout->get() << "It is thread-safety" << "to use many<< operators"
<< "in one expression." << std::endl;
// Other thread cannot get access to std::cout object while all
// operator<< not executed and temporary CResourceLocker object not destroyed, i.e. until
expression using TCriticalResource not executed.
```

The main disadvantage of using Critical Resource lies in the fact that using of more than one Critical Resource in one expression is deadlock-unsafe, and also can lead to serious problems which will probably be hard to reveal. In order to avoid this kind of problems we can break the mentioned expression to several ones, each of which will be using one Critical Resource and copies of other resources at that. To resolve this problem copy() method was implemented (copy() method can be used in one expression as many times as it is necessary only if get() method wasn't invoked in this expression).

Also it is necessary to pay attention to the fact that usage of TCriticalResource which limits access to STL containers (such as std::vector or std::list) for creation of a thread-safe container probably is not the best architectural decision. For this purpose it is better to use concurrency container patterns, such as: Buffer Monitor, Object Pool, Question Answer Pool, Blackboard.

Full implementation of TCriticalResource class can be taken from [CriticalResource.zip](#)