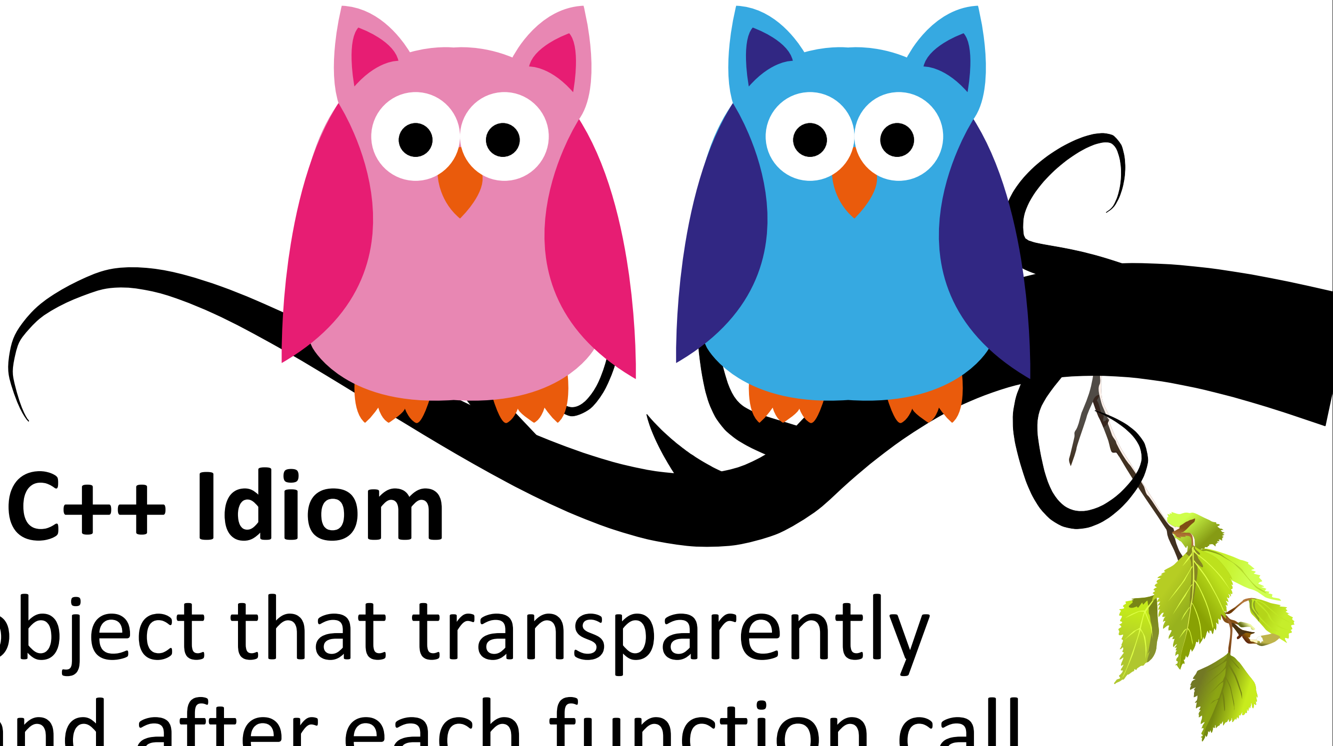
Execute-Around Pointer C++ Idiom

Introduction



- **Execute-Around Pointer C++ Idiom**
  - Provide a smart pointer object that transparently executes actions before and after each function call on an object
- **Condtion:**
  - The actions performed are the same for all functions.

Classic Scenario

- In multi-threaded application, it is necessary to lock before modifying the data structure/reference count/index and unlock it immediately afterwards

1. Lock and get exclusive access to shared variable
2. Modify the thread shared variable under exclusive access
3. Unlock and release exclusive lock

Classic Example

Lock

Unlock

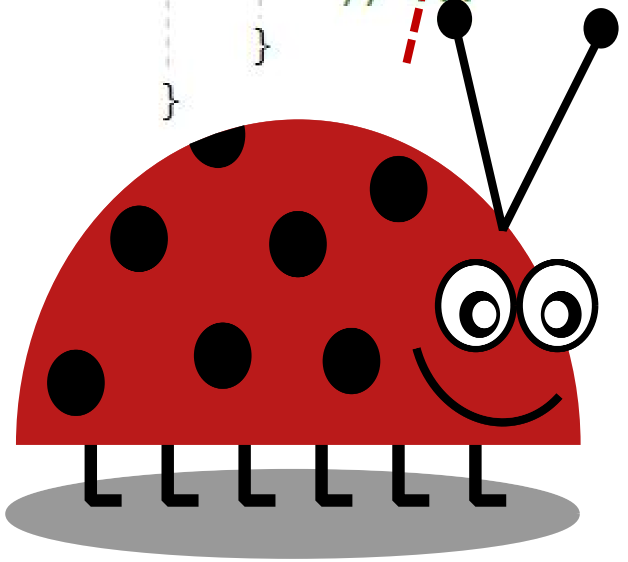
```
static std::vector<int> g_DataVector;
static std::mutex g_DataCounterLock;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...
        g_DataCounterLock.lock();              // 1. Lock and get exclusive access to shared varaible.
        g_DataVector.push_back( nIndex );      // 2. Modify the shared variable under exclusive access.
        g_DataCounterLock.unlock();            // 3. Unlock and release exclusive lock.
        // ...
        // Do some more thread specific code executions
        // ...
    }
}

int main() {
    std::thread Thread1( DoSomethingAsThread );
    std::thread Thread2( DoSomethingAsThread );
    Thread1.join();
    Thread2.join();
    return 0;
}
```

Issue Example

```
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...
        g_DataCounterLock.lock();          // 1. Lock and get exclusive access to shared varaible.
        g_DataVector.push_back( nIndex ); // 2. Modify the shared variable under exclusive access.

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

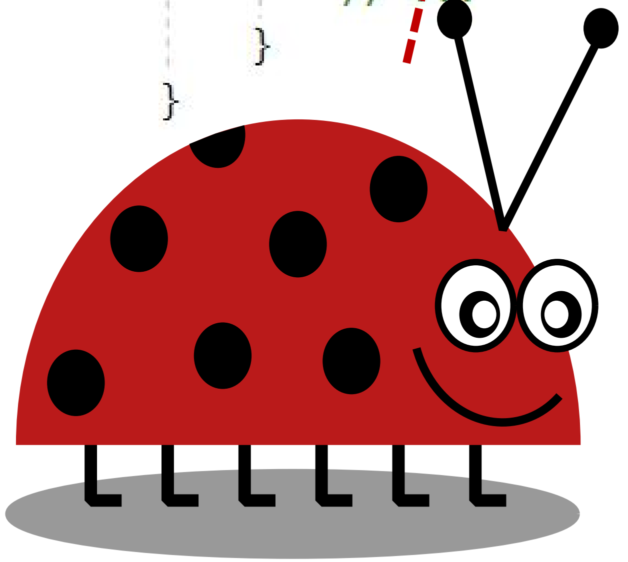
Issue Example

```
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...
        g_DataCounterLock.lock();          // 1. Lock and get exclusive access to shared varaible.
        g_DataVector.push_back( nIndex ); // 2. Modify the shared variable under exclusive access.

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```



Issue Example

```
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...
        g_DataCounterLock.lock();           // 1. Lock and get exclusive access to shared varaible.
        g_DataVector.push_back( nIndex ); // 2. Modify the shared variable under exclusive access.

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

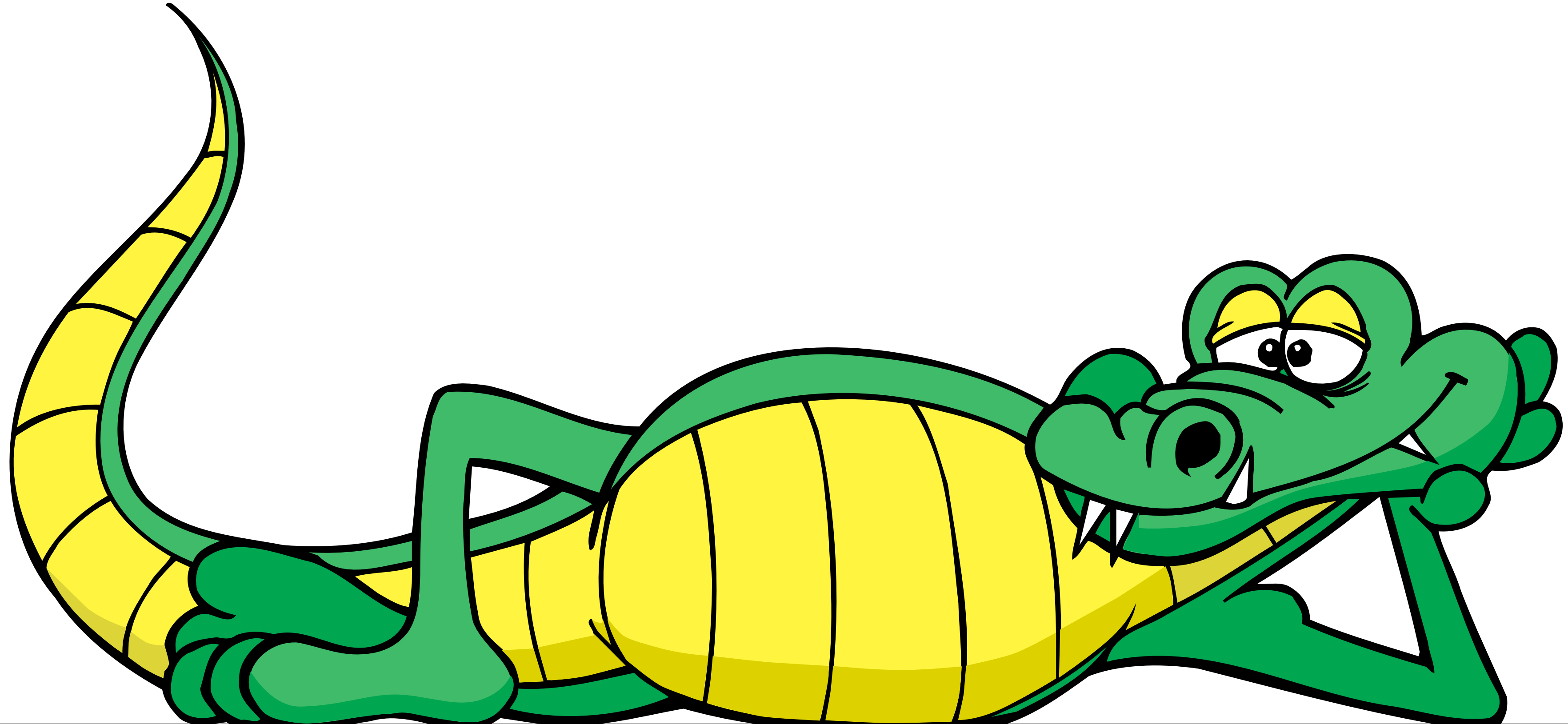


Forgot to write code for releaseing the lock.

- **Owning thread never releases its ownership.**
- **Waiting threads wait indefinitly.**
- **Waiting threads enters into HANG state.**

Lesson learned

- One should always remember to write code to unlock the lock if ever had locked.



After some time, say 1 year

- **Forgot to remember the promise.**

After some time, say 1 year

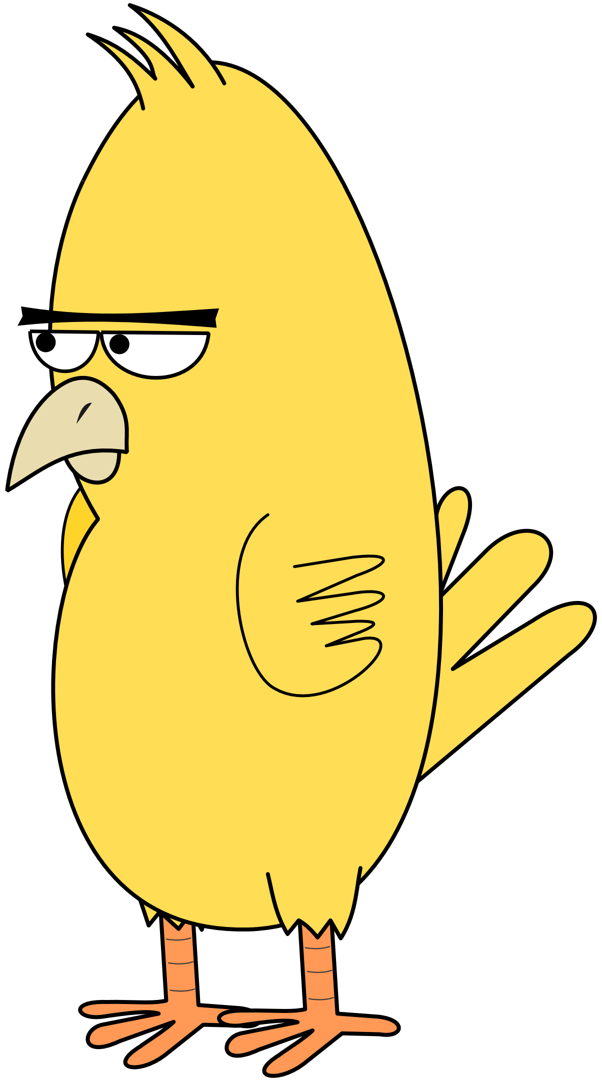
- Forgot to remember the promise.
- **Issue re-appeared again.**

After some time, say 1 year

- Forgot to remember the promise.
- Issue re-appeared again.
- **This time the issue was bit more sophisticated.**

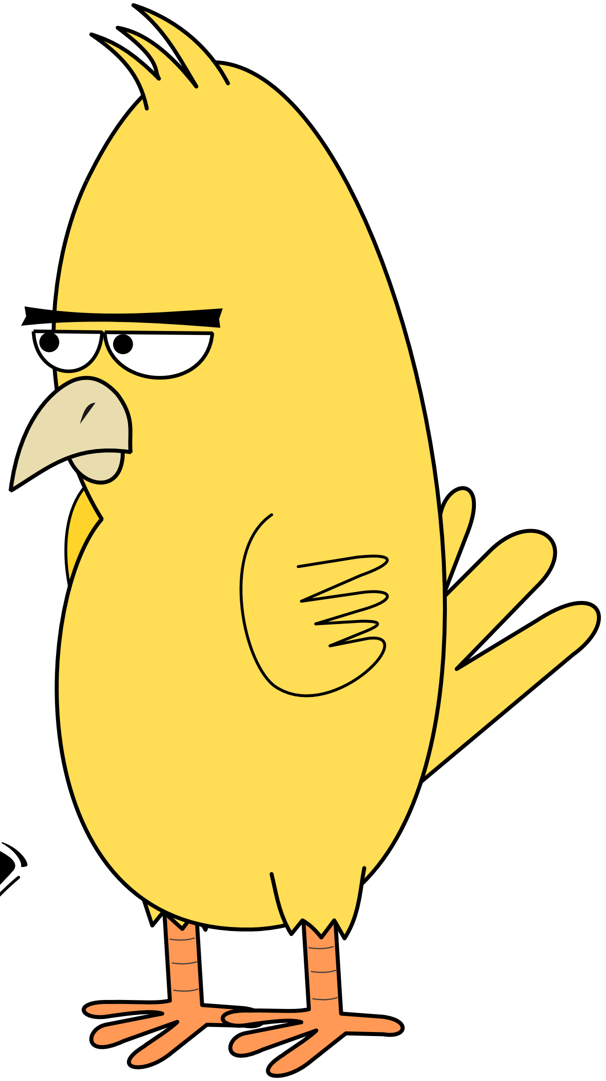
After some time, say 1 year

- Forgot to remember the promise.
- Issue re-appeared again.
- This time the issue was bit sophisticated.
- **The hang situation occurred at <u>production site</u>.**



After some time, say 1 year

- Forgot to remember the promise.
- Issue re-appeared again.
- This time the issue was bit sophisticated.
- The hang situation occurred at production site.
- **And the production site was a Nuclear reactor.**



Execute-Around Pointer Idiom

- The issue would not have occurred, if the shared data variable could ensure thread safety by itself
- I.e., with out requiring the developer to explicitly write code to lock and unlock the lock objects

Solution

By using Execution-Around Pointer Idiom,

Shared Data variable lock and unlock can be performed automatically with out writing any lock/unlock codes.

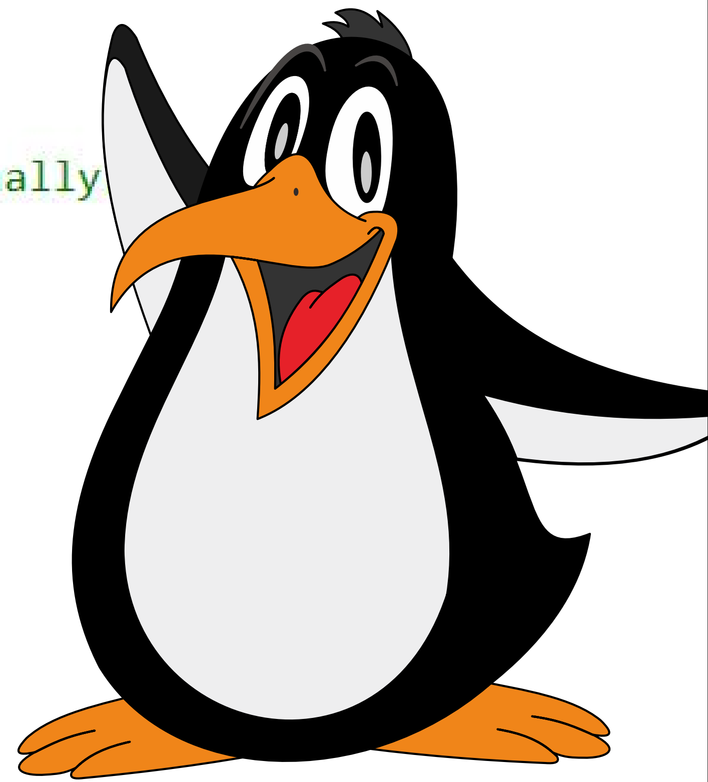
Scenario with Execution-Around Pointer Idiom

- In multi-threaded application, it is necessary to lock before modifying the data structure/reference count/index and unlock it immediately afterwards

1. ~~**Lock and get exclusive access to shared variable**~~
2. **Modify the thread shared variable under exclusive access**
3. ~~**Unlock and release exclusive lock**~~

Execute-Around Pointer Idiom

```
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally
        g_DataVector.push_back( nIndex );

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```



Implementation Details

1. Implement a proxy class.

```
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io,
                         std::vector<int>* DataVector_i )
                       : m_pMutexLock{ pMutexLock_io },
                         m_pDataVector{ DataVector_i } {
        m_pMutexLock->lock();
    }
    std::vector<int>* operator->() {
        return m_pDataVector;
    }
    ~ThreadSafeDataProxy() {
        m_pMutexLock->unlock();
    }

private:
    std::vector<int>* m_pDataVector= nullptr;
    std::mutex* m_pMutexLock       = nullptr;
};
```

Proxy class member variables are **pointers**

POINTER member variables representing the corresponding members in Container class.

Implementation Details

1. Implement a proxy class.

```
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io,
                         std::vector<int>* DataVector_i )
                       : m_pMutexLock{ pMutexLock_io },
                         m_pDataVector{ DataVector_i } {
        m_pMutexLock->lock();
    }
    std::vector<int>* operator->() {
        return m_pDataVector;
    }
    ~ThreadSafeDataProxy() {
        m_pMutexLock->unlock();
    }

private:
    std::vector<int>* m_pDataVector= nullptr;
    std::mutex* m_pMutexLock       = nullptr;
};
```

proxy initialization ctor() with paramters same as that of container class.
Initialize the member pointer varaibles with actual values from container class.

Implementation Details

1. Implement a proxy class.

```
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io,
                         std::vector<int>* DataVector_i )
                       : m_pMutexLock{ pMutexLock_io },
                         m_pDataVector{ DataVector_i } {
        m_pMutexLock->lock();
    }
    std::vector<int>* operator->() {
        return m_pDataVector;
    }
    ~ThreadSafeDataProxy() {
        m_pMutexLock->unlock();
    }

private:
    std::vector<int>* m_pDataVector= nullptr;
    std::mutex* m_pMutexLock       = nullptr;
};
```

Inside proxy ctor(), perform **lock** operation of std::mutex member pointer variable

Implementation Details

1. Implement a proxy class.

```
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io,
                         std::vector<int>* DataVector_i )
                       : m_pMutexLock{ pMutexLock_io },
                         m_pDataVector{ DataVector_i } {
        m_pMutexLock->lock();
    }
    std::vector<int>* operator->() {
        return m_pDataVector;
    }
    ~ThreadSafeDataProxy() {
        m_pMutexLock->unlock();
    }

private:
    std::vector<int>* m_pDataVector= nullptr;
    std::mutex* m_pMutexLock        = nullptr;
};
```

Inside proxy dtor(), perform **unlock** operation of std::mutex member pointer variable

Implementation Details

1. Implement a proxy class.

```
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io,
                         std::vector<int>* DataVector_i )
                       : m_pMutexLock{ pMutexLock_io },
                         m_pDataVector{ DataVector_i } {
        m_pMutexLock->lock();
    }
    std::vector<int>* operator->() {
        return m_pDataVector;
    }
    ~ThreadSafeDataProxy() {
        m_pMutexLock->unlock();
    }

private:
    std::vector<int>* m_pDataVector= nullptr;
    std::mutex* m_pMutexLock       = nullptr;
};
```

Overload operator->()
Return the Data vector pointer member varaible.
which was set via proxy ctor().

Implementation Details

2. Implement a container class.

```
class ThreadSafeData {
public:
    ThreadSafeData() {
    }
    ThreadSafeDataProxy operator->() {
        return ThreadSafeDataProxy( &m_mutexLock, &m_DataVector );
    }
    ~ThreadSafeData() {
    }

private:
    std::vector<int> m_DataVector;
    std::mutex m_mutexLock;
};
```

Member varaibles

1. Shared Data vector
2. std::mutex lock object for exclusive access

Implementation Details

2. Implement a container class.

```
class ThreadSafeData {
public:
    ThreadSafeData() {
    }
    ThreadSafeDataProxy operator->() {
        return ThreadSafeDataProxy( &m_mutexLock, &m_DataVector );
    }
    ~ThreadSafeData() {
    }

private:
    std::vector<int> m_DataVector;
    std::mutex m_mutexLock;
};
```

Overload operator->()

Return a temporary stack object of proxy class

Implementation Details

3. Use the container class object in the application program

```
static ThreadSafeData g_ThreadSafeDataVector;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally.
        g_ThreadSafeDataVector->push_back( nIndex );

        // ...
        // Do some more
        // ...
    }
}
```

Access & Modify the container class contained std::vector member variable via operator->()

Execute-Around Pointer Idiom

Thread Function

```
static ThreadSafeData g_ThreadSafeDataVector;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally.
        g_ThreadSafeDataVector->push_back( nIndex );

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

Before std::vector push_back API call

Execute-Around Pointer Idiom

Automatically Accquire an exclusive lock

without manually writing any code to Lock

Thread Function

```
static ThreadSafeData g_ThreadSafeDataVector;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally.
        g_ThreadSafeDataVector->push_back( nIndex );

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

Before std::vector push_back API call

Execute-Around Pointer Idiom

Thread Function

Automatically Accquire an exclusive lock

without manually writing any code to Lock

```
static ThreadSafeData g_ThreadSafeDataVector;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally.
        g_ThreadSafeDataVector->push_back( nIndex );

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

Before std::vector push_back API call

After std::vector push_back API call

Execute-Around Pointer Idiom

Thread Function

```
static ThreadSafeData g_ThreadSafeDataVector;
void DoSomethingAsThread() {
    for( int nIndex = 0; nIndex < 10; ++nIndex ) {
        // ...
        // Do some thread specific code executions
        // ...

        // Modify the shared variable under exclusive access.
        // The lock/unlock shall be automatically performed internally.
        g_ThreadSafeDataVector->push_back( nIndex );

        // ...
        // Do some more thread specific code executions
        // ...
    }
}
```

Automatically Accquire an exclusive lock

without manually writing any code to Lock

Before std::vector push_back API call

After std::vector push_back API call

Automatically Release exclusive lock

without manually writing any code to Lock

Template class can make the program generic to hold any class object types as ThreadSafeData.

```
struct IntType{
    int m_Count = 0;
};
template<typename T>
class ThreadSafeDataProxy {
public:
    ThreadSafeDataProxy( std::mutex* pMutexLock_io, T* pIntCounter_io ) : m_pMutexLock{ pMutexLock_io },
                                                                          m_pData{ pIntCounter_io } {

        m_pMutexLock->lock();
        std::cout << "Locked" << " TID:" << std::this_thread::get_id() << "\t Lock Object : " << m_pMutexLock;
    }
    T* operator->() {
        std::cout << "\tData Access" << " TID:" << std::this_thread::get_id() << "";
        return m_pData;
    }
    ~ThreadSafeDataProxy() {
        std::cout << "\tUnLocked" << " TID:" << std::this_thread::get_id() << "\t Lock Object : " << m_pMutexLock <<"\n";
        m_pMutexLock->unlock();
    }
private:
    T* m_pData                  = nullptr;
    std::mutex* m_pMutexLock = nullptr;
};

template<typename T>
class ThreadSafeData {
public:
    ThreadSafeData(){}
    ThreadSafeDataProxy<T> operator->() {
        return ThreadSafeDataProxy<T>( &m_mutexLock, &m_Data );
    }
    ~ThreadSafeData(){}
private:
    T m_Data;
    std::mutex m_mutexLock;
};
```

Template class made the program generic to hold any class object types as ThreadSafeData.

```
static ThreadSafeData<IntType> g_SharedIntData;
static ThreadSafeData<std::vector<int>> g_ThreadSafeVector;
static ThreadSafeData<std::map<int,int>> g_ThreadSafeMap;

void DoSomethingAsThread()
{
    for( auto nIndex = 0; nIndex < 10; ++nIndex ) {
        ++g_SharedIntData->m_Count; // integer variable
         g_ThreadSafeVector->push_back( 10 ); // std::vector
        g_ThreadSafeMap->emplace( nIndex, nIndex*10 ); // std::map
    }
}

int main() {
    std::thread Thread1( DoSomethingAsThread );
    std::thread Thread2( DoSomethingAsThread );
    Thread1.join();
    Thread2.join();
    return 0;
}
```

Different types of shared data types with out explicit lock/unlock. The lock/unlock is automatically occurs during accessing the underlying object via operator->()

Thank you

LIKE & SHARE & Subscribe to our YouTube Channel

https://youtu.be/oTpGyu2L8C8