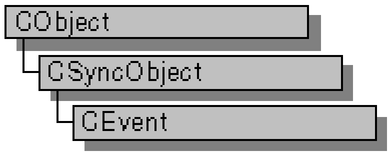
CEvent



An object of class **CEvent** represents an “event” — a synchronization object that allows one thread to notify another that an event has occurred. Events are useful when a thread needs to know when to perform its task. For example, a thread that copies data to a data archive would need to be notified when new data is available. By using a **CEvent** object to notify the copy thread when new data is available, the thread can perform its task as soon as possible.

CEvent objects have two types: manual and automatic. A manual **CEvent** object stays in the state set by SetEvent or ResetEvent until the other function is called. An automatic **CEvent** object automatically returns to a nonsignaled (unavailable) state after at least one thread is released.

To use a **CEvent** object, construct the **CEvent** object when it is needed. Specify the name of the event you wish to wait on, and that your application should initially own it. You can then access the event when the constructor returns. Call SetEvent to signal (make available) the event object and then call Unlock when you are done accessing the controlled resource.

An alternative method for using **CEvent** objects is to add a variable of type **CEvent** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CEvent** data member specifying if the event is initially signaled, the type of event object you want, the name of the event (if it will be used across process boundaries), and desired security attributes.

To access a resource controlled by a **CEvent** object in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource’s access member function. Then call the lock object’s **Lock** member function (for example, CMultiLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, call **SetEvent** to signal the event object, and then use the lock object’s **Unlock** member function (for example, CMultiLock::Unlock), or allow the lock object to fall out of scope.

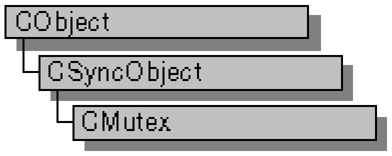
For more information on using **CEvent** objects, see the article Multithreading: How to Use the Synchronization Classes in *Visual C++ Programmer's Guide*.

#include <afxmt.h>

Class Members | Base Class | Hierarchy Chart

Sample MFC Sample MTGDI

CMutex



An object of class **CMutex** represents a “mutex” — a synchronization object that allows one thread mutually exclusive access to a resource. Mutexes are useful when only one thread at a time can be allowed to modify data or some other controlled resource. For example, adding nodes to a linked list is a process that should only be allowed by one thread at a time. By using a **CMutex** object to control the linked list, only one thread at a time can gain access to the list.

To use a **CMutex** object, construct the **CMutex** object when it is needed. Specify the name of the mutex you wish to wait on, and that your application should initially own it. You can then access the mutex when the constructor returns. Call CSyncObject::Unlock when you are done accessing the controlled resource.

An alternative method for using **CMutex** objects is to add a variable of type **CMutex** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CMutex** data member specifying if the mutex is initially owned, the name of the mutex (if it will be used across process boundaries), and desired security attributes.

To access resources controlled by **CMutex** objects in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource’s access member function. Then call the lock object’s **Lock** member function (for example, CSingleLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, use the lock object’s **Unlock** member function (for example, CSingleLock::Unlock), or allow the lock object to fall out of scope.

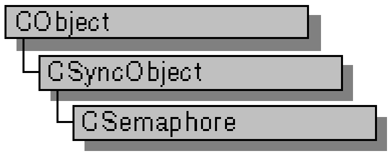
For more information on using **CMutex** objects, see the article Multithreading: How to Use the Synchronization Classes in *Visual C++ Programmer’s Guide*.

#include <afxmt.h>

Class Members | Base Class | Hierarchy Chart

Sample MFC Sample MUTEXES

CSemaphore



An object of class **CSemaphore** represents a “semaphore” — a synchronization object that allows a limited number of threads in one or more processes to access a resource. A **CSemaphore** object maintains a count of the number of threads currently accessing a specified resource.

Semaphores are useful in controlling access to a shared resource that can only support a limited number of users. The current count of the **CSemaphore** object is the number of additional users allowed. When the count reaches zero, all attempts to use the resource controlled by the **CSemaphore** object will be inserted into a system queue and wait until they either time out or the count rises above 0. The maximum number of users who can access the controlled resource at one time is specified during construction of the **CSemaphore** object.

To use a **CSemaphore** object, construct the **CSemaphore** object when it is needed. Specify the name of the semaphore you wish to wait on, and that your application should initially own it. You can then access the semaphore when the constructor returns. Call CSyncObject::Unlock when you are done accessing the controlled resource.

An alternative method for using **CSemaphore** objects is to add a variable of type **CSemaphore** as a data member to the class you wish to control. During construction of the controlled object, call the constructor of the **CSemaphore** data member specifying the initial access count, maximum access count, name of the semaphore (if it will be used across process boundaries), and desired security attributes.

To access resources contolled by **CSemaphore** objects in this manner, first create a variable of either type CSingleLock or type CMultiLock in your resource’s access member function. Then call the lock object’s **Lock** member function (for example, CSingleLock::Lock). At this point, your thread will either gain access to the resource, wait for the resource to be released and gain access, or wait for the resource to be released and time out, failing to gain access to the resource. In any case, your resource has been accessed in a thread-safe manner. To release the resource, use the lock object’s **Unlock** member function (for example, CSingleLock::Unlock), or allow the lock object to fall out of scope.

Alternatively, you can create a **CSemaphore** object stand-alone, and access it explicitly before attempting to access the controlled resource. This method, while clearer to someone reading your source code, is more prone to error.

For more information on how to use **CSemaphore** objects, see the article Multithreading: How to Use the Synchronization Classes in *Visual C++ Programmer’s Guide*.

#include <afxmt.h>

Class Members | Base Class | Hierarchy Chart

Sample MFC Sample MUTEXES

CMultiLock

CMultiLock does not have a base class.

An object of class **CMultiLock** represents the access-control mechanism used in controlling access to resources in a multithreaded program. To use the synchronization classes CSemaphore, CMutex, and CEvent, you can create either a **CMultiLock** or CSingleLock object to wait on and release the synchronization object. Use **CMultiLock** when there are multiple objects that you could use at a particular time. Use **CSingleLock** when you only need to wait on one object at a time.

To use a **CMultiLock** object, first create an array of the synchronization objects that you wish to wait on. Next, call the **CMultiLock** object's constructor inside a member function in the controlled resource's class. Then call the Lock member function to determine if a resource is available (signaled). If one is, continue with the remainder of the member function. If no resource is available, either wait for a specified amount of time for a resource to be released, or return failure. After use of a resource is complete, either call the Unlock function if the **CMultiLock** object is to be used again, or allow the **CMultiLock** object to be destroyed.

CMultiLock objects are most useful when a thread has a large number of **CEvent** objects it can respond to. Create an array containing all the **CEvent** pointers, and call **Lock**. This will cause the thread to wait until one of the events is signaled.

For more information on how to use **CMultiLock** objects, see the article Multithreading: How to Use the Synchronization Classes in *Visual C++ Programmer's Guide*.

#include <afxmt.h>

Class Members | Hierarchy Chart

CSingleLock

CSingleLock does not have a base class.

An object of class **CSingleLock** represents the access-control mechanism used in controlling access to a resource in a multithreaded program. In order to use the synchronization classes CSemaphore, CMutex, CCriticalSection, and CEvent, you must create either a **CSingleLock** or CMultiLock object to wait on and release the synchronization object. Use **CSingleLock** when you only need to wait on one object at a time. Use **CMultiLock** when there are multiple objects that you could use at a particular time.

To use a **CSingleLock** object, call its constructor inside a member function in the controlled resource's class. Then call the IsLocked member function to determine if the resource is available. If it is, continue with the remainder of the member function. If the resource is unavailable, either wait for a specified amount of time for the resource to be released, or return failure. After use of the resource is complete, either call the Unlock function if the **CSingleLock** object is to be used again, or allow the **CSingleLock** object to be destroyed.

CSingleLock objects require the presence of an object derived from CSyncObject. This is usually a data member of the controlled resource's class. For more information on how to use **CSingleLock** objects, see the article Multithreading: How to Use the Synchronization Classes in *Visual C++ Programmer's Guide*.

#include <afxmt.h>

Class Members | Hierarchy Chart

See Also CMultiLock