Notifications & Events

Notifying someone that something happened.

applied informatics

Overview

- Notifications and the NotificationCenter
- Sending notifications to other threads using a NotificationQueue
- Events

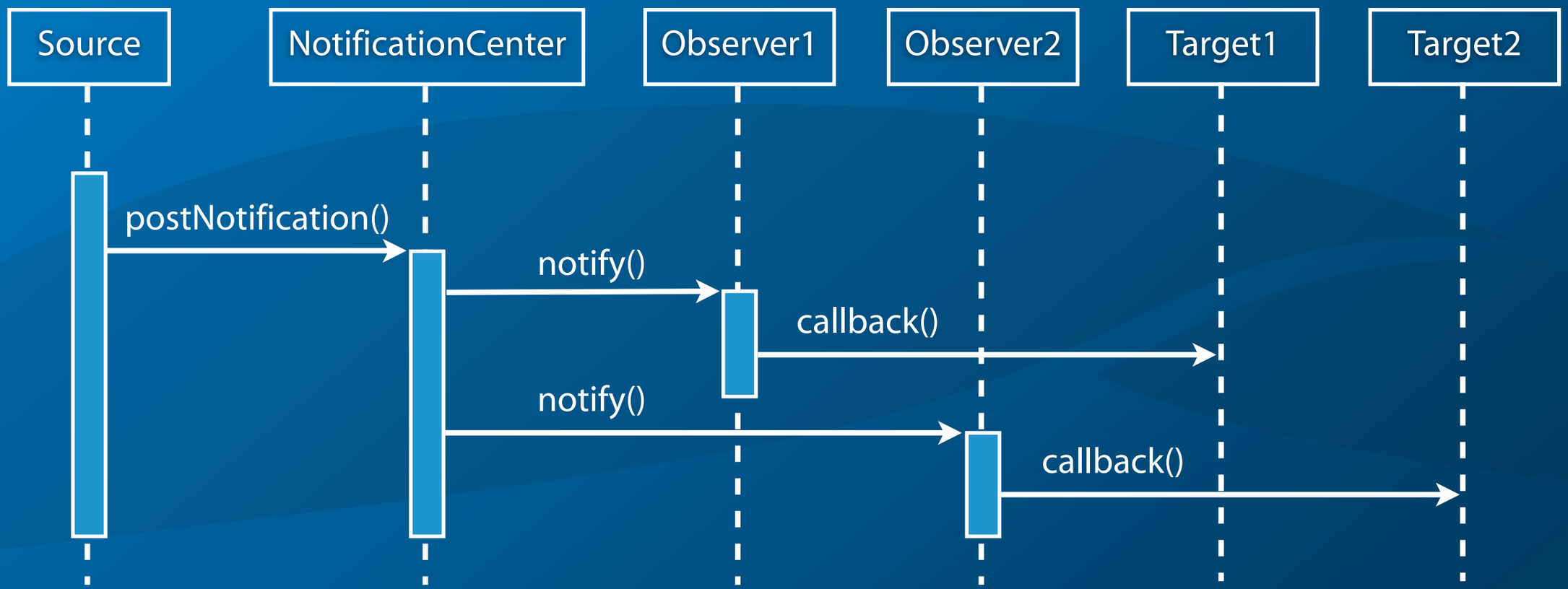
Notifications vs. Events

- Notifications and events are two mechanisms supported by POCO to tell a class (the target) that something happened in another class (the source).
- Notifications are used if an observer does not know or does not care about the source of an event. A Poco::NotificationCenter or Poco::NotificationQueue sits between, and decouples sources and targets. Notifications can be sent across thread boundaries.
- Events are used if an observer does care about the source of an event, or wants to receive events only from a particular source. Events also support asynchronous notification and other features, that notifications do not support.

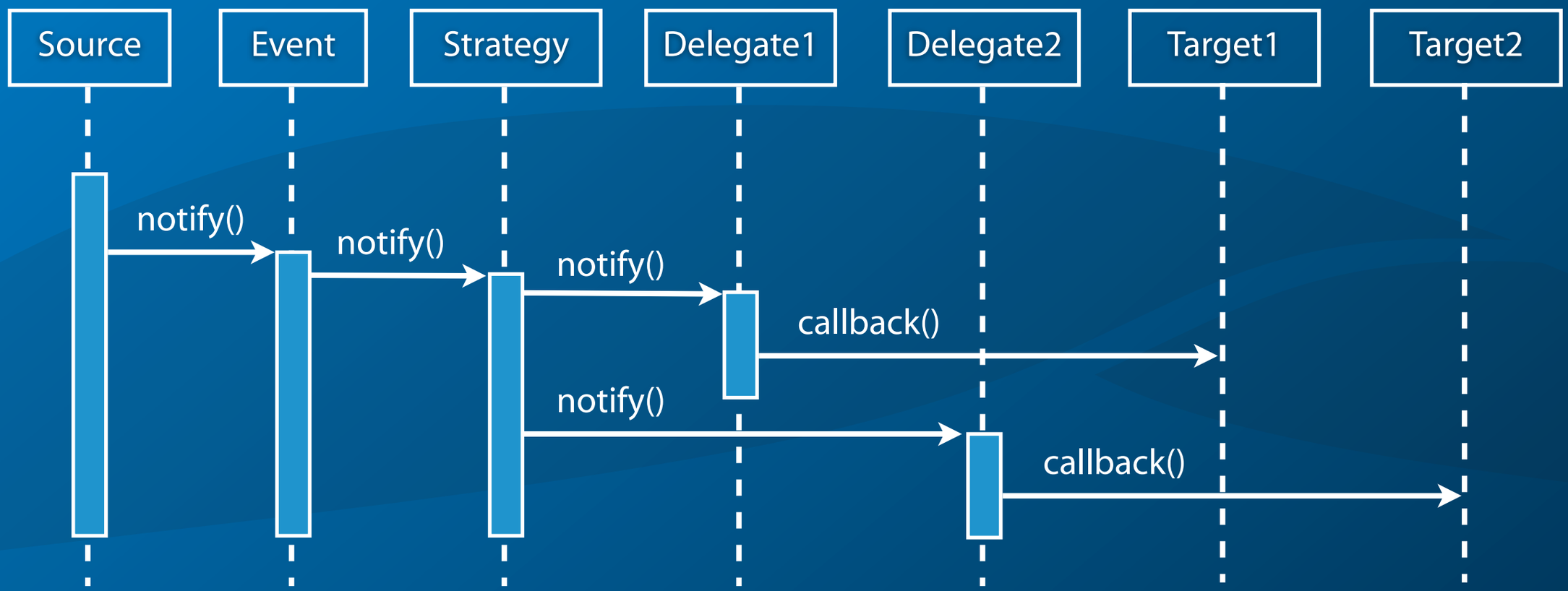
Notifications vs. Events (cont'd)

	Notifications	Events
targets know sources	–	✔
synchronous notification	✔	✔
asynchronous notification	–	✔
works across thread boundaries	✔	–
different notification strategies	–	✔
automatic expirations	–	✔
notification polymorphism	✔	–

Dispatching Notifications



Dispatching Events



Notification Classes

- Notification classes are derived from Poco::Notification and support reference counting (compatible with Poco::AutoPtr).
- A notification object can hold arbitrary data and can provide arbitrary operations.
- Notification classes do not support value semantics (no copy constructor, no assignment) and are always created on the heap.

The NotificationCenter Class

- Poco::NotificationCenter is a dispatcher for notification objects.
- #include "Poco/NotificationCenter.h"
- Poco::NotificationCenter uses observer objects (subclasses of Poco::AbstractObserver) to talk to its targets.
- An observer object stores a pointer to the target object, and a pointer to the target object's callback member function, and knows which notifications the target is interested in.

Subscribing to Notifications

- Targets can subscribe to notifications by registering themselves with a NotificationCenter using the addObserver() member function.
- void addObserver(const AbstractObserver& observer) registers a notification target with the NotificationCenter
- A subscription can be cancelled by calling removeObserver().
- void removeObserver(const AbstractObserver& observer) unregisters a notification target

Observers

- An Observer stores a pointer to the target object, and a pointer to the target object's callback member function, and knows which notifications the target is interested in.
- Observers are defined using either the Observer or the NObserver class template.
- Observer works with plain pointers to Notification objects.
- NObserver works with AutoPtr<Notification>.
- Observer and NObserver are instantiated for a Notification class and a target class.

Observers and Callback Functions

- For Observer, the target member function receiving the callback must be defined as:
  void someCallback(SomeNotification* pNf)
  where someCallback can be any name and SomeNotification is the notification to be registered for.
  The callback gets shared ownership of the notification object, and must release it when it's no longer needed.

- For NObserver, the target member function is:
  void someCallback(const AutoPtr<SomeNotification>& pNf)

Observers and Callback Functions (cont'd)

- During a callback, the callback function may unregister itself (or other callbacks) from the NotificationCenter, or register new callbacks with the NotificationCenter.
- Observers that have been added during a notification will be called the first time with the next notification.
- Observers that have been removed during a notification will not receive the current notification (unless they have already received it).

Posting Notifications

- Notifications are posted for dispatching by a NotificationCenter using the postNotification() method.
- void postNotification(Notification::Ptr pNotification) delivers the notification to all targets subscribed for the notification class (or a superclass of it)
- The notification is delivered to all registered targets. If a target throws an exception while handling the notification, dispatching stops and the exception is propagated to the caller.
- The NotificationCenter assumes ownership of the notification.

Notification Polymorphism

- Targets subscribed for a particular notification class also receive notifications that are subclasses of that class.
- If a target subscribes for Poco::Notification, it will thus receive all notifications posted to the NotificationCenter it has registered with.

```
#include "Poco/NotificationCenter.h"
#include "Poco/Notification.h"
#include "Poco/Observer.h"
#include "Poco/NObserver.h"
#include "Poco/AutoPtr.h"
#include <iostream>

using Poco::NotificationCenter;
using Poco::Notification;
using Poco::Observer;
using Poco::NObserver;
using Poco::AutoPtr;

class BaseNotification: public Notification
{
};

class SubNotification: public BaseNotification
{
};
```

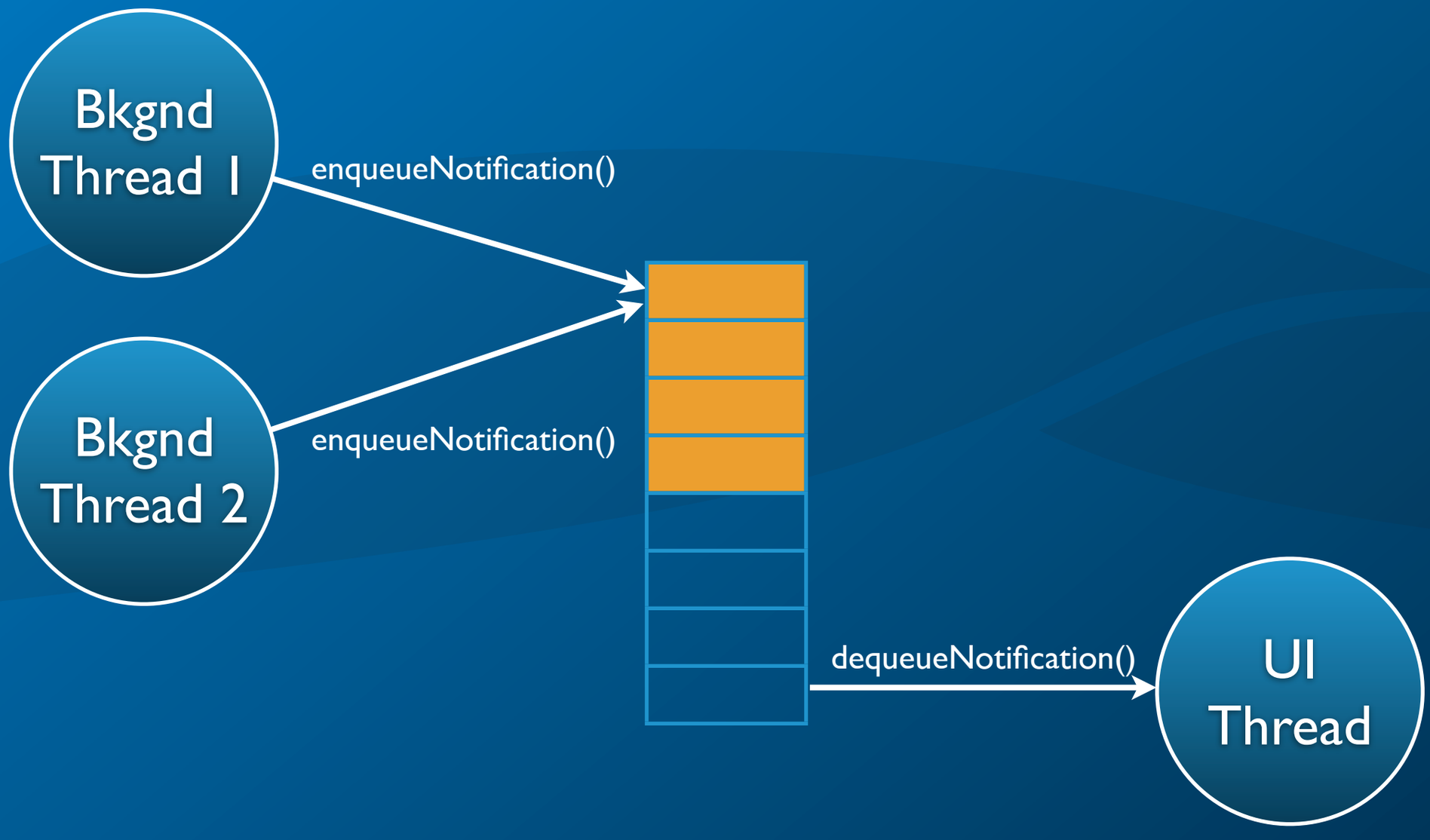
```
class Target
{
public:
    void handleBase(BaseNotification* pNf)
    {
        std::cout << "handleBase: " << pNf->name() << std::endl;
        pNf->release(); // we got ownership, so we must release
    }

    void handleSub(const AutoPtr<SubNotification>& pNf)
    {
        std::cout << "handleSub: " << pNf->name() << std::endl;
    }
};
```

```
int main(int argc, char** argv)
{
    NotificationCenter nc;
    Target target;

    nc.addObserver(
        Observer<Target, BaseNotification>(target, &Target::handleBase)
    );
    nc.addObserver(
        NObserver<Target, SubNotification>(target, &Target::handleSub)
    );

    nc.postNotification(new BaseNotification);
    nc.postNotification(new SubNotification);

    nc.removeObserver(
        Observer<Target, BaseNotification>(target, &Target::handleBase)
    );
    nc.removeObserver(
        NObserver<Target, SubNotification>(target, &Target::handleSub)
    );

    return 0;
}
```

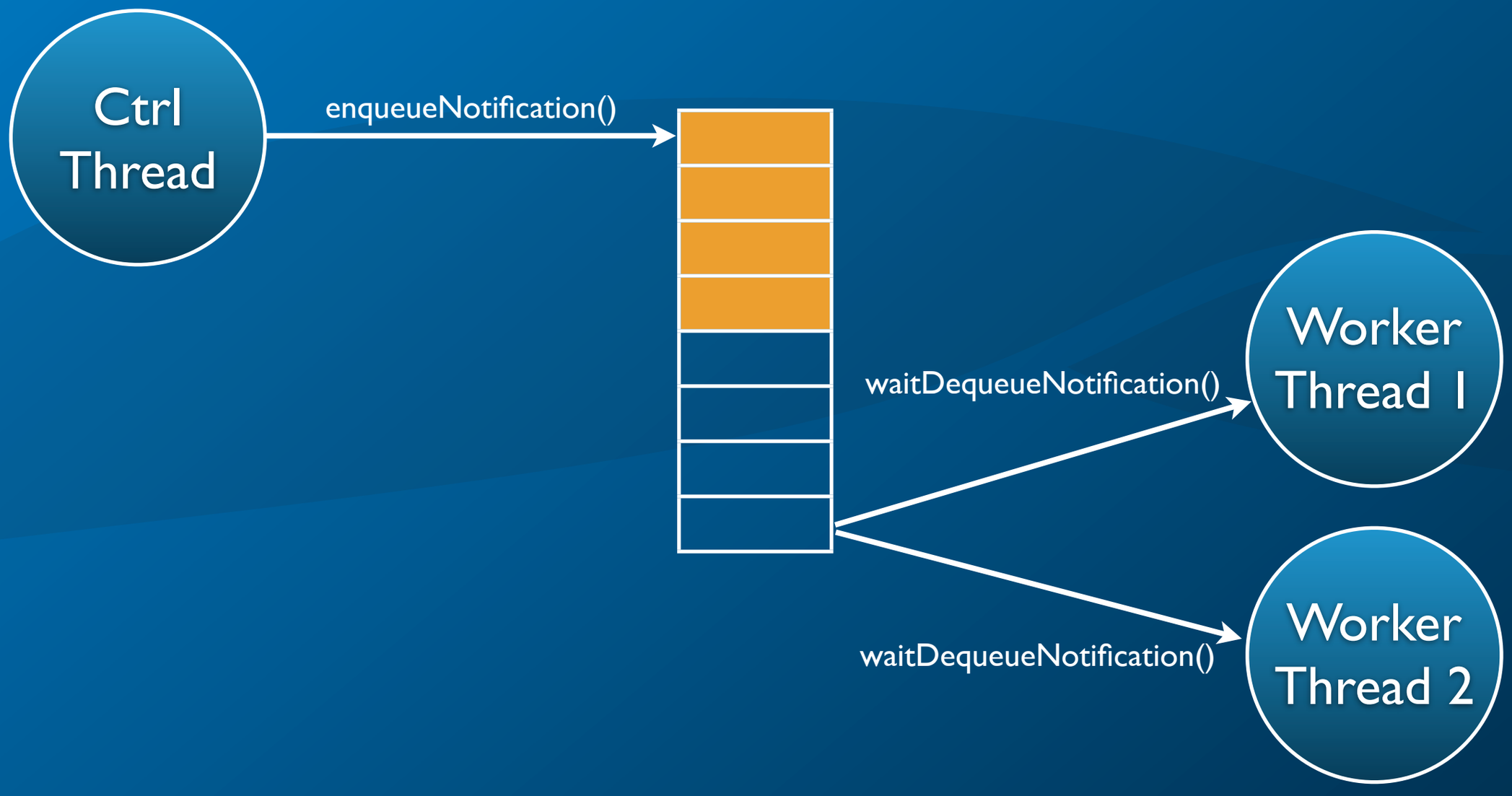
The NotificationQueue Class

- A Poco::NotificationQueue can be used to send notifications asynchronously from one thread to another.
- #include "Poco/NotificationQueue.h"
- More than one thread can read from a NotificationQueue.
- Use a NotificationQueue to
  - send notifications from background processing threads to the user interface thread, or
  - send notifications from a controlling thread to one or more worker threads.

Background Thread ➔ UI Thread



Controller Thread ➔ Worker Thread



Enqueueing Notifications

- void enqueueNotification(Notification::Ptr pNotification)
  enqueues the given notification by adding it to the end of the queue (FIFO principle). The queue takes ownership of the notification.

- void enqueueUrgentNotification(Notification::Ptr pNotification)
  enqueues the given notification by adding it to the beginning of the queue (LIFO principle). The queue takes ownership of the notification.

Dequeueing Notifications

- Notification* dequeueNotification()
  dequeues the next pending notification from the beginning of the queue, or null if no notification is available. The caller gains ownership of the notification.

- Notification* waitDequeueNotification()
  Notification* waitDequeueNotification(long timeout)
  dequeues the next pending notification. If no notification is available, waits (at most timeout milliseconds) for a notification to be posted. Returns the notification, or null if none is available.

Shutting Down a Queue

- How to tell worker threads they are done?
- Three strategies:

1. Post a special QuitNotification for every worker thread;
2. Set a (global) stop flag and use waitDequeueNotification() with a timeout;
3. Use wakeUpAll(): every call to waitDequeueNotification() will immediately return null.

Shutting Down a Queue: QuitNotification

- One QuitNotification must be posted for every worker thread.
- Worker threads must test for and handle QuitNotification (using dynamic_cast or something similar), and immediately stop dequeueing more notifications.
- Controller must know the exact number of worker threads.
- Worker threads can use waitDequeueNotification() without a timeout.

Shutting Down a Queue: Stop Flag

- A (global) stop flag is set to notify workers of pending shutdown.
- Worker threads must use waitDequeueNotification() with a timeout, and periodically check the stop flag.
- Worker threads can only react to shutdown after timeout expires.

Shutting Down a Queue: wakeUpAll()

- void wakeUpAll()
  wakes up all threads waiting for a notification using waitDequeueNotification(). Every call to waitDequeueNotification() will immediately return null.
- wakeUpAll() only works if all worker threads are idle and waiting for notifications.
- Additionally, a stop flag must be maintained if worker threads use waitDequeueNotification() with a timeout.

```
#include "Poco/Notification.h"
#include "Poco/NotificationQueue.h"
#include "Poco/ThreadPool.h"
#include "Poco/Runnable.h"
#include "Poco/AutoPtr.h"

using Poco::Notification;
using Poco::NotificationQueue;
using Poco::ThreadPool;
using Poco::Runnable;
using Poco::AutoPtr;

class WorkNotification: public Notification
{
public:
    WorkNotification(int data): _data(data) {}

    int data() const
    {
        return _data;
    }
private:
    int _data;
};
```

```
class Worker: public Runnable
{
public:
    Worker(NotificationQueue& queue): _queue(queue) {}

    void run()
    {
        AutoPtr<Notification> pNf(_queue.waitDequeueNotification());
        while (pNf)
        {
            WorkNotification* pWorkNf =
                dynamic_cast<WorkNotification*>(pNf.get());
            if (pWorkNf)
            {
                // do some work
            }
            pNf = _queue.waitDequeueNotification();
        }
    }

private:
    NotificationQueue& _queue;
};
```

```
int main(int argc, char** argv)
{
    NotificationQueue queue;

    Worker worker1(queue); // create worker threads
    Worker worker2(queue);

    ThreadPool::defaultPool().start(worker1); // start workers
    ThreadPool::defaultPool().start(worker2);

    // create some work
    for (int i = 0; i < 100; ++i)
    {
        queue.enqueueNotification(new WorkNotification(i));
    }

    while (!queue.empty()) // wait until all work is done
        Poco::Thread::sleep(100);

    queue.wakeUpAll(); // tell workers they're done
    ThreadPool::defaultPool().joinAll();

    return 0;
}
```

Special Queues

- PriorityNotificationQueue
  notifications are tagged with a priority and dequeued in order of their priority (lower numerical value means higher priority)

- TimedNotificationQueue
  notifications are tagged with a timestamp and dequeued in order of their timestamps

Events

- Events in POCO are modeled after C# events, but implemented in a true C++ way.
- In contrast to notifications, events are part of a class interface. Events are defined as public data members.
- Events support asynchronous notifications, different notification strategies and automatic expirations.
- Events are defined using the Poco::BasicEvent class template.
- #include "Poco/BasicEvent.h"

Events (cont'd)

- A target subscribes to an event by registering a delegate, using the Poco::Delegate class template.
- #include "Poco/Delegate.h"
- An event has exactly one argument, which can be a subclass of Poco::EventArgs.
- #include "Poco/EventArgs.h"

Defining an Event

- An event is defined using the Poco::BasicEvent class template.
- Poco::BasicEvent is instantiated with the type of the event argument.
- Usually, an event is added as a public data member to a class.

Delegates

- A target uses Poco::Delegate to register a callback member function with the event.
- Poco::Delegate is instantiated with the target class and the event argument type.
- A delegate is registered with an event using the += operator of the event.
- Similarly, a delegate is unregistered using the -= operator.

Delegates and Callback Functions

- The callback function used with a delegate must be a function with one of the following signatures:
  `void handler(const void* pSender, EventArg& arg)`
- The first argument points to the object that fired the event.
- The second is a reference to the argument passed to the event.
- The callback function may modify the event argument (unless it has been declared `const`) to pass data back to the sender.

Firing Events

- An event can be fired synchronously by invoking its notify() member function (or its function call operator).
- An event can be fired asynchronously by invoking its notifyAsync() member function.
- If any event handler throws an exception, event dispatching stops immediately and the exception is propagated to the caller.

```
#include "Poco/BasicEvent.h"
#include "Poco/Delegate.h"
#include <iostream>

using Poco::BasicEvent;
using Poco::Delegate;

class Source
{
public:
    BasicEvent<int> theEvent;

    void fireEvent(int n)
    {
        theEvent(this, n);
//      theEvent.notify(this, n); // alternative syntax
    }
};
```

```
class Target
{
public:
    void onEvent(const void* pSender, int& arg)
    {
        std::cout << "onEvent: " << arg << std::endl;
    }
};

int main(int argc, char** argv)
{
    Source source;
    Target target;
    source.theEvent += Poco::delegate(&target, &Target::onEvent);
    source.fireEvent(42);
    source.theEvent -= Poco::delegate(&target, &Target::onEvent);
    return 0;
}
```

Synchronous vs. Asynchronous Events

- Use notify when your handler code is small and the expected number of delegates is low.
- Use notify when you require synchronization.
- Be careful with notify() (and also notifyAsync()) when your handler code can trigger other events, i.e. other notifies. In combination with mutexes dead-locks are possible.

Event Considerations

- Never forget to unregister delegates! Otherwise dangling pointers will cause undefined behavior (crash) in a later notify.
- Each target can only register one single delegate at one event. If a target registers two callback functions with a single event, the latter will replace the first.
- Unregistering a delegate that was never registered or has already expired is okay.
- Events are thread safe, i.e. you can modify the delegate set while a notify is in progress. The new delegate set will not influence the current notify but will take effect with the next notify.

Advanced Events

- Poco::FIFOEvent can be used instead of Poco::BasicEvent to ensure delegates are called in the same order in which they have been added.
- Poco::PriorityEvent can be used instead of Poco::BasicEvent to add priorities to delegates. Delegates must be added using the Poco::PriorityDelegate class template. Delegates are called in order of their priority, with lower priorities coming first.

Advanced Events (cont'd)

- Automatically expiring delegates can be defined using the Poco::Expire class template as a wrapper around Poco::Delegate.
- Poco::Expire only works with Poco::Delegate. For Poco::PriorityDelegate, use Poco::PriorityExpire.

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096