

Multithreading

Doing things in parallel with POOCO.

Overview

- > The Thread class
- > Thread pools
- > Thread Local Storage
- > Thread Error Handling
- > Synchronization
(Mutex, FastMutex, Event, Condition, Semaphore, RWLock)
- > Higher-level abstractions
(Timers, Active Objects, Tasks and Task Management)

The Thread Class

- > `Poco::Thread` represents a thread in POCO.
- > `#include "Poco/Thread.h"`
- > In POCO, a thread has
 - > an optional name
 - > a priority (`Thread::Priority`)
 - > a unique ID (integer)
- > `Poco::Runnable` provides an entry point for a thread.

Thread Priorities

- > POCO supports five distinct priority values for a thread. These are mapped to the operating system's thread priority values.
 - > `PRIO_LOWEST`
 - > `PRIO_LOW`
 - > `PRIO_NORMAL` (default)
 - > `PRIO_HIGH`
 - > `PRIO_HIGHEST`
- > Some platforms require special privileges (root) to set or change a thread's priority.

Thread Priorities (cont'd)

- > The five log levels supported by POCO might not be fine grained enough for some applications.
- > You can set an operating system specific thread priority with `void setOSPriority(int prio)`
- > You can use `int getMinOSPriority()` and `int getMaxOSPriority()` to find out the range of valid priority values.

Thread Stack Size

- > The stack size of a thread can be set with `void setStackSize(int size)`
If size is 0, the default stack size of the operating system is used.
- > There is also `int getStackSize() const` to find out the stack size of the thread.

The Runnable Class

- > `Poco::Runnable` is an interface class for thread entry points.
- > `#include "Poco/Runnable.h"`
- > Subclasses must override the `run()` method.
- > The `Poco::RunnableAdapter` class template can be used to specify an arbitrary no-argument void member function as a thread entry point.
- > `#include "Poco/RunnableAdapter.h"`

Working With Threads

- > To start a thread, call its `start()` member function and pass it a reference to a `Runnable`.
- > To join a thread, call its `join()` member function. It will wait until the thread terminates, then return.
- > A thread's priority can be changed with `setPriority()` and queried with `getPriority()`.
- > Every thread gets a unique numerical ID which can be obtained with the `id()` member function.
- > A thread's name (optionally specified at construction) can be obtained with `name()`.

Working With Threads (cont'd)

- > Check whether a thread is running by calling `isRunning()`.
- > A pointer to the `Thread` object for the current thread can be obtained with the static `current()` member function. The main thread does not have a `Thread` object, therefore `current()` returns a null pointer for the main thread.
- > `void Thread::sleep(long milliseconds)`
suspends the current thread for the specified amount of time.
- > `void Thread::yield()`
gives the CPU to another thread.

```
#include "Poco/Thread.h"
#include "Poco/Runnable.h"
#include <iostream>

class HelloRunnable: public Poco::Runnable
{
    virtual void run()
    {
        std::cout << "Hello, world!" << std::endl;
    }
};

int main(int argc, char** argv)
{
    HelloRunnable runnable;

    Poco::Thread thread;
    thread.start(runnable);
    thread.join();

    return 0;
}
```

```
#include "Poco/Thread.h"
#include "Poco/RunnableAdapter.h"
#include <iostream>

class Greeter
{
public:
    void greet()
    {
        std::cout << "Hello, world!" << std::endl;
    }
};

int main(int argc, char** argv)
{
    Greeter greeter;
    Poco::RunnableAdapter<Greeter> runnable(greeter, &Greeter::greet);

    Poco::Thread thread;
    thread.start(runnable);
    thread.join();

    return 0;
}
```

Thread Pools

- > The creation of a new thread takes some time, and often threads can be reused.
- > Managing the lifetime of **Thread** objects can be hard (who deletes the thread when it's done, ...)
- > You may want to control the number of threads created in your application. A **ThreadPool** can do that for you.

The ThreadPool Class

- > `Poco::ThreadPool` implements thread pooling in POCO.
- > `#include "Poco/ThreadPool.h"`
- > A thread pool has a maximum capacity; if the capacity is exhausted, a `NoThreadAvailableException` is thrown if a new thread is requested.
- > The capacity of a thread pool can be increased dynamically:
`void addCapacity(int n).`
- > POCO provides a default `ThreadPool` instance with an initial capacity of 16 threads.

```
#include "Poco/ThreadPool.h"
#include "Poco/Runnable.h"
#include <iostream>

class HelloRunnable: public Poco::Runnable
{
    virtual void run()
    {
        std::cout << "Hello, world!" << std::endl;
    }
};

int main(int argc, char** argv)
{
    HelloRunnable runnable;

    Poco::ThreadPool::defaultPool().start(runnable);
    Poco::ThreadPool::defaultPool().joinAll();

    return 0;
}
```

The ThreadPool Class (cont'd)

- > Threads in a thread pool can be started with a specific priority (and an optional name):

```
void start(Runnable& target)
```

```
void start(Runnable& target, const std::string& name)
```

```
void startWithPriority(Thread::Priority prio, Runnable& target)
```

```
void startWithPriority(Thread::Priority prio, Runnable& target,  
const std::string& name)
```

- > To wait for the completion of all threads in the pool:

```
void joinAll()
```


ThreadPool Maintenance

- > A thread pool will stop all threads that have been idle for a certain amount of time.
- > However, the thread pool collects idle threads only at the following occasions:
 - > `start()` or `startWithPriority()`
 - > `addCapacity()`
 - > `joinAll()`
- > To force a thread collection, you can call `collect()`.

Thread Local Storage

- > Thread Local Storage is a special kind of variable that has a different value in every thread.
- > A thread has no way to access the value of another thread.
- > Type safe Thread Local Storage in POCO is implemented by the `Poco::ThreadLocal` class template.
- > `#include "Poco/ThreadLocal.h"`
- > `Poco::ThreadLocal` can be instantiated with any class that is default constructible.
- > To access the actual value of a thread local variable, use the arrow (`->`) or asterisk (`*`) operator of `ThreadLocal`.

```
#include "Poco/Thread.h"
#include "Poco/Runnable.h"
#include "Poco/ThreadLocal.h"
#include <iostream>

class Counter: public Poco::Runnable
{
    void run()
    {
        static Poco::ThreadLocal<int> tls;

        for (*tls = 0; *tls < 10; ++(*tls))
        {
            std::cout << *tls << std::endl;
        }
    }
};
```

```
int main(int argc, char** argv)
{
    Counter counter;

    Poco::Thread t1;
    Poco::Thread t2;

    t1.start();
    t2.start();

    t1.join();
    t2.join();

    return 0;
}
```

Thread Error Handling

- > An unhandled exception in a thread causes the thread to terminate. However, such an unhandled exception is normally not reported to the outside.
- > It is possible to register a global error handler that gets notified about unhandled exceptions in threads.
- > Such an error handler must be derived from the `Poco::ErrorHandler` (`#include "Poco/ErrorHandler.h"`) class.

The ErrorHandler Class

- > The following virtual member functions of `Poco::ErrorHandler` can be overridden:
- > `void exception(const Poco::Exception& exc)` is called for every unhandled exception derived from `Poco::Exception`.
- > `void exception(const std::exception& exc)` is called for every unhandled exception derived from `std::exception` (but not `Poco::Exception`)
- > `void exception()` is called for every other unhandled exception.

The ErrorHandler Class (cont'd)

- > To install a custom error handler:
`ErrorHandler* ErrorHandler::set(ErrorHandler* pHandler)`
installs the new error handler and returns a pointer to the old one.
- > Only one `ErrorHandler` can be installed in a single process.
- > The `ErrorHandler`'s `exception()` member functions are always called in the context of the offending thread.


```
#include "Poco/Thread.h"
#include "Poco/Runnable.h"
#include "Poco/ErrorHandler.h"
#include <iostream>

class Offender: public Poco::Runnable
{
    void run()
    {
        throw Poco::ApplicationException("got you");
    }
};
```

```
class MyErrorHandler: public Poco::ErrorHandler
{
public:
    void exception(const Poco::Exception& exc)
    {
        std::cerr << exc.displayText() << std::endl;
    }

    void exception(const std::exception& exc)
    {
        std::cerr << exc.what() << std::endl;
    }

    void exception()
    {
        std::cerr << "unknown exception" << std::endl;
    }
};
```

```
int main(int argc, char** argv)
{
    MyErrorHandler eh;
    ErrorHandler* pOldEH = Poco::ErrorHandler::set(&eh);

    Offender offender;

    Thread thread;
    thread.start(offender);
    thread.join();

    Poco::ErrorHandler::set(pOldEH);

    return 0;
}
```


Thread Synchronization

- > POCO provides the following synchronization primitives:
 - > `Poco::Mutex`
 - > `Poco::FastMutex`
 - > `Poco::Event`
 - > `Poco::Condition`
 - > `Poco::Semaphore`
 - > `Poco::RWLock`

Mutex

- > A mutex (mutual exclusion) is a synchronization primitive used to control access to a shared resource in a concurrent scenario.
- > Mutexes come in two flavors:
 - > recursive: the same mutex can be locked multiple times by the same thread (but not by other threads)
 - > non-recursive: an attempt to lock an already locked mutex will result in a deadlock

The Mutex and FastMutex Classes

- > `Poco::Mutex` is a recursive mutex;
`Poco::FastMutex` is (conceptually) a non-recursive mutex.
- > `#include "Poco/Mutex.h"`
- > **Warning:** On Windows, `Poco::FastMutex` is actually recursive, too. This is a potential source for portability issues if you make wrong assumptions based on that behavior. 

Mutex and FastMutex Operations

- > `void lock()`
acquires the mutex and waits if the mutex is held by another thread.
- > `void lock(long millisecs)`
acquires the mutex and blocks up to the given number of milliseconds if the mutex is held by another thread. Throws a `TimeoutException` if the mutex can not be locked.
- > `void unlock()`
releases the mutex so that it can be acquired by another thread.

Mutex and FastMutex Operations (cont.)

- > `bool tryLock()`
tries to acquire the mutex. Returns `false` immediately if the mutex is held by another thread, or `true` if the mutex has been acquired.
- > `bool tryLock(long millisecs)`
tries to acquire the mutex within the given time period. Returns `false` if it fails to acquire the lock, or `true` if the mutex has been acquired.

Scoped Lock

- > The Scoped Lock is a powerful idiom in C++.
- > A Scoped Lock acquires a mutex in its constructor, and releases the mutex in its destructor.
- > Thus, a scoped lock at the beginning of a block is the perfect way to implement a critical section.
- > At the beginning of the block, in the construction of the scoped lock, the mutex is acquired.
- > No matter how the block is exited (normally, by return, or by exception), the destructor of the scoped lock ensures the release of the mutex.

The ScopedLock Class Template

- > In POCO, the Scoped Lock is implemented by the `Poco::ScopedLock` class template.
- > `#include "Poco/ScopedLock.h"`
- > `Poco::ScopedLock` can be instantiated for every class that supports `lock()` and `unlock()` member functions.
- > Both `Mutex` and `FastMutex` provide typedefs for corresponding `ScopedLock` instantiations:
`Mutex::ScopedLock` and `FastMutex::ScopedLock`.

```
#include "Poco/Mutex.h"

using Poco::Mutex;

class Concurrent
{
public:
    void criticalSection()
    {
        Mutex::ScopedLock lock(_mutex);

        // ...
    }

private:
    Mutex _mutex;
};
```

Events

- > An Event is a synchronization object that allows one thread to signal one or more other threads (possibly waiting for it) the occurrence of a certain event.
- > Events come in two flavors:
 - > auto reset: after waking up at most one waiting thread, the event will lose its signalled state
 - > manual reset: the event will maintain the signalled state until it is manually reset

The Event Class

- > Events in POCO are implemented by `Poco::Event`.
- > `#include "Poco/Event.h"`
- > `Poco::Event` supports both automatic and manual reset.
 - > For automatic reset (default), pass `true` to the constructor.
 - > For manual reset, pass `false` to the constructor.

Event Operations

- > `void set()`

signals the **Event**. If the **Event** is auto-resetting, at most one thread waiting for the event is waken up and the signalling state is reset. Otherwise, all threads waiting for the **Event** are waken up.

- > `void wait()`

`void wait(long milliseconds)`

waits for the **Event** to become signalled. If a timeout is given, and the **Event** does not become signalled within the given interval, a **TimeoutException** is thrown.

Event Operations (cont'd)

- > `bool tryWait(long milliseconds)`
waits for the `Event` to become signalled. If the `Event` is signalled within the given interval, returns `true`. Otherwise, returns `false`.
- > `void reset()`
resets a (manual reset) `Event`.

Conditions

- > A Condition is a synchronization object used to block a thread until a particular condition is met. A Condition object is always used in conjunction with a Mutex (or FastMutex) object.
- > Condition objects are similar to POSIX condition variables, with the difference that Condition is not subject to spurious wakeups.
- > Threads waiting on a Condition are resumed in FIFO order.

The Condition Class

- > Conditions in POCO are implemented by `Poco::Condition`.
- > `#include "Poco/Condition.h"`
- > `Poco::Condition` is template based and works with any kind of mutex object.
- > The implementation is based on `Poco::Event` and a `std::deque` for waiting threads on all platforms.

Condition Operations

- > `template <class Mtx> void wait(Mtx& mutex)`
`template <class Mtx> void wait(Mtx& mutex, long milliseconds)`
Unlocks the mutex (which must be locked upon calling `wait()`) and waits (for the given time) until the Condition is signalled. The given mutex will be locked again upon successfully leaving the function, even in case of an exception. Throws a `TimeoutException` if the Condition is not signalled within the given time interval.
- > `template <class Mtx> bool tryWait(Mtx& mutex, long millisecs)`
Non-throwing variant of `wait()`.

Condition Operations (cont'd)

- > `void signal()`
Signals the Condition and allows one waiting thread to continue execution.
- > `void broadcast()`
Signals the Condition and allows all waiting threads to continue their execution.

Semaphores

- > Semaphores in POCO are implemented by the `Poco::Semaphore` class.
- > `#include "Poco/Semaphore.h"`
- > For the details, please refer to the reference documentation.

Reader Writer Locks

- > Reader Writer Locks in POCO are implemented by the `Poco::RWLock` class.
- > `#include "Poco/RWLock.h"`
- > A `RWLock` allows multiple concurrent readers, xor one exclusive writer. In other words, at any given time, a resource managed by a `RWLock` can be accessed by
 - > multiple concurrent readers, or
 - > at most one writer

The RWLock Class

- > `void readLock()`
`bool tryReadLock()`
acquires a read lock. If another thread holds a write lock, waits until the write lock is released (or returns false immediately).
- > `void writeLock()`
`bool tryWriteLock()`
acquires a write lock. If other threads currently hold locks, waits until all locks have been released (or returns false immediately).
- > `void unlock()`
release a read or write lock.

Timers

- > POCO provides a "sequential" timer implementation in the `Poco::Timer` class.
- > `#include "Poco/Timer.h"`
- > A timer starts a thread (taken from a thread pool) that first waits for a given start interval.
- > Once the start interval expires, the timer invokes a callback function.
- > After the callback function returns, and the periodic interval is not zero, the timer repeatedly waits for the periodic interval, and then invokes the callback function.

The Timer Class

- > The timer can be stopped by setting the periodic interval to zero.
- > The timer callback runs in the timer's thread, so synchronization may be required.
- > The precision of the timer depends on many factors, like operating system, system load etc. `Poco::Timer` is absolutely not realtime capable.
- > Please see the reference documentation for more information.

```
#include "Poco/Timer.h"
#include "Poco/Thread.h"

using Poco::Timer;
using Poco::TimerCallback;

class TimerExample
{
public:
    void onTimer(Poco::Timer& timer)
    {
        std::cout << "onTimer called." << std::endl;
    }
};

int main(int argc, char** argv)
{
    TimerExample te;
    Timer timer(250, 500); // fire after 250ms, repeat every 500ms
    timer.start(TimerCallback<TimerExample>(te, &TimerExample::onTimer));
    Thread::sleep(5000);
    timer.stop();
    return 0;
}
```

Task Management

- > If you need to track the progress of one or more background processing threads in a GUI (or server) application, you can use the `Poco::Task` class, along with `Poco::TaskManager`.
- >

```
#include "Poco/Task.h"  
#include "Poco/TaskManager.h"
```
- > `Poco::Task` is a `Poco::Runnable` that provides facilities for reporting the progress of an operation, and for supporting cancellation of a task.
- > `Poco::TaskManager` manages a collection of `Poco::Task` objects (`SharedPtr`), and runs them using a `Poco::ThreadPool`.

Task Management (cont'd)

- > For progress reporting and cancellation to work:
 - > You create a subclass of `Poco::Task` and override the `runTask()` member function.
 - > From your `runTask()`, you periodically call `setProgress()` to report the task's progress, and call `isCancelled()` or `sleep()` to check for a cancellation request.
 - > If `isCancelled()` or `sleep()` returns `true`, you return from `runTask()`.
- > The `Poco::TaskManager` uses a `Poco::NotificationCenter` to notify interested parties about the progress of its tasks.

Task Management (cont'd)

- > The following notifications (all derived from `Poco::TaskNotification`) are available:
 - > `Poco::TaskStartedNotification`
 - > `Poco::TaskCancelledNotification`
 - > `Poco::TaskFinishedNotification`
 - > `Poco::TaskFailedNotification`
 - > `Poco::TaskProgressNotification`
 - > `Poco::TaskCustomNotification`


```
#include "Poco/Task.h"
#include "Poco/TaskManager.h"
#include "Poco/TaskNotification.h"
#include "Poco/Observer.h"

using Poco::Observer;

class SampleTask: public Poco::Task
{
public:
    SampleTask(const std::string& name): Task(name)
    {
    }

    void runTask()
    {
        for (int i = 0; i < 100; ++i)
        {
            setProgress(float(i)/100); // report progress
            if (sleep(1000))
                break;
        }
    }
};
```

```
class ProgressHandler
{
public:
    void onProgress(Poco::TaskProgressNotification* pNf)
    {
        std::cout << pNf->task()->name()
                  << " progress: " << pNf->progress() << std::endl;

        pNf->release();
    }

    void onFinished(Poco::TaskFinishedNotification* pNf)
    {
        std::cout << pNf->task()->name() << " finished." << std::endl;

        pNf->release();
    }
};
```

```
int main(int argc, char** argv)
{
    Poco::TaskManager tm;
    ProgressHandler pm;

    tm.addObserver(
        Observer<ProgressHandler, Poco::TaskProgressNotification>
            (pm, &ProgressHandler::onProgress)
    );

    tm.addObserver(
        Observer<ProgressHandler, Poco::TaskFinishedNotification>
            (pm, &ProgressHandler::onFinished)
    );

    tm.start(new SampleTask("Task 1")); // tm takes ownership
    tm.start(new SampleTask("Task 2"));

    tm.joinAll();

    return 0;
}
```

Active Objects

- > An active object is an object that runs (some of) its member functions in its/their own thread(s).
- > In POCO, active objects support two kinds of active member functions:
 - > An **Activity** is a possibly long running void/no arguments member function running in its own thread.
 - > An **ActiveMethod** is a non-void one-argument member function that runs in its own thread.
 - > All active methods can share a single thread (in this case, invocation will be queued), or each have its own thread.

Activities

- > Activities can be started automatically upon object construction, or manually at a later time.
- > Activities can be stopped at any time. For this to work, the activity must call the `isStopped()` member function periodically.
- > A method implementing an activity cannot have arguments or return a value.
- > The thread for an activity is taken from the default thread pool.

```
#include "Poco/Activity.h"
#include "Poco/Thread.h"
#include <iostream>

using Poco::Thread;

class ActivityExample
{
public:
    ActivityExample(): _activity(this, &ActivityExample::runActivity)
    {
    }

    void start()
    {
        _activity.start();
    }

    void stop()
    {
        _activity.stop(); // request stop
        _activity.wait(); // wait until activity actually stops
    }
}
```

```
protected:
    void runActivity()
    {
        while (!_activity.isStopped())
        {
            std::cout << "Activity running." << std::endl;
            Thread::sleep(200);
        }
    }

private:
    Poco::Activity<ActivityExample> _activity;
};

int main(int argc, char** argv)
{
    ActivityExample example;
    example.start();
    Thread::sleep(2000);
    example.stop();

    return 0;
}
```


Active Methods

- > An active method is a member function that executes in its own thread (taken from the default thread pool).
- > Active methods can share a thread. In this case, only one active method can execute at a time while others are waiting in a queue for execution.
- > It takes exactly one argument and returns a value.
- > To pass more than one argument, use a struct, a `std::pair`, or a `Poco::Tuple`.
- > The return value of an active method will be delivered in an `Poco::ActiveResult` (also called a Future).

Active Results (Futures)

- > Since the return value of an active method is not available immediately after invoking the method (since the method runs in parallel), a **ActiveResult** is used to deliver the result.
- > **ActiveResult** is a class template, instantiated for the function's returned type.
- > Starting an active method returns an **ActiveResult** that will eventually contain the result (or an exception).
- > You usually wait for a result to arrive (using **wait()** or **tryWait()**), and then obtain the result by calling **data()**.

```
#include "Poco/ActiveMethod.h"
#include "Poco/ActiveResult.h"
#include <utility>

using Poco::ActiveMethod;
using Poco::ActiveResult;

class ActiveAdder
{
public:
    ActiveAdder():
        add(this, &ActiveAdder::addImpl)
    {
    }

    ActiveMethod<int, std::pair<int, int>, ActiveAdder> add;

private:
    int addImpl(const std::pair<int, int>& args)
    {
        return args.first + args.second;
    }
};
```

```
int main(int argc, char** argv)
{
    ActiveAdder adder;

    ActiveResult<int> sum = adder.add(std::make_pair(1, 2));
    sum.wait();
    std::cout << sum.data() << std::endl;

    return 0;
}
```

Queueing Method Execution

- > The default behavior of `ActiveMethod` does not fit the "classic" definition of an active object, where methods are queued for execution in a single thread.
- > To get the classic behavior, you use `ActiveDispatcher` as a base class for your active object.
- > You also need to tell the `ActiveMethod` to use the `ActiveDispatcher` to schedule the method execution.

```
#include "Poco/ActiveMethod.h"
#include "Poco/ActiveResult.h"
#include "Poco/ActiveDispatcher.h"
#include <utility>

using Poco::ActiveMethod;
using Poco::ActiveResult;

class ActiveAdder: public Poco::ActiveDispatcher
{
public:
    ActiveObject(): add(this, &ActiveAdder::addImpl)
    {
    }

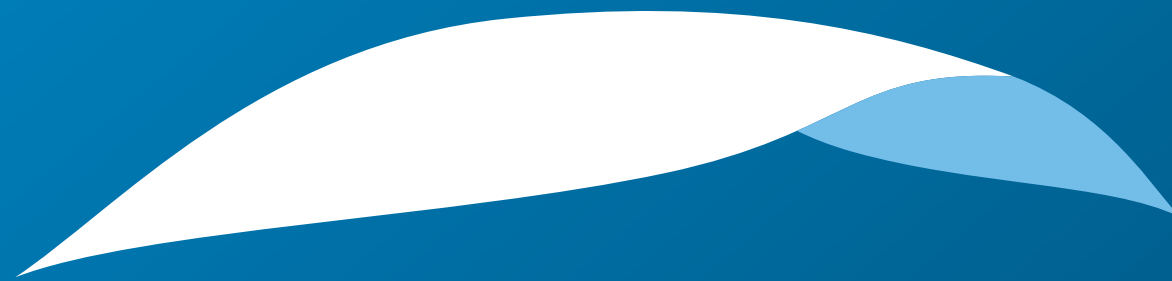
    ActiveMethod<int, std::pair<int, int>, ActiveAdder,
        Poco::ActiveStarter<Poco::ActiveDispatcher> > add;

private:
    int addImpl(const std::pair<int, int>& args)
    {
        return args.first + args.second;
    }
};
```

```
int main(int argc, char** argv)
{
    ActiveAdder adder;

    ActiveResult<int> sum = adder.add(std::make_pair(1, 2));
    sum.wait();
    std::cout << sum.data() << std::endl;

    return 0;
}
```

appliedinformatics

Copyright © 2006-2010 by Applied Informatics Software Engineering GmbH.
Some rights reserved.

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

