C++ Threading

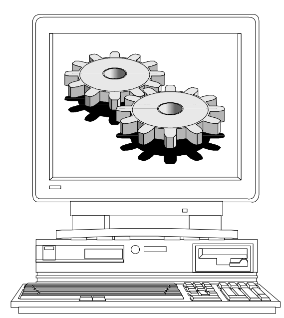
A Generic-Programming Approach

Kevlin Henney

kevlin@curbralan.com

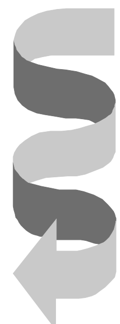
Agenda

- Intent
  - Building up from first principles, present a generic-programming model for multithreaded programming in C++
- Content
  - Programming with threads
  - Generic threading
  - Generic synchronisation



Programming with Threads

- Intent
  - Introduce primitive and some high-level thread programming concepts
- Content
  - C-style threading APIs
  - Active and passive objects
  - Inheritance versus delegation
  - Thread safety
  - Synchronisation primitives



Processes

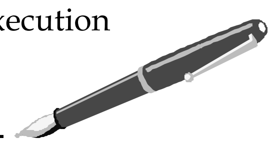
- A process can be considered to be a container of threads within a protected address space
  - Parts of the address space may be shareable
  - One or more threads execute through this space
- Multiple processes execute concurrently
  - Multiple processors, pre-emptive multitasking or, in the worst case, co-operative multitasking
- Execution is subject to priorities and policies

Threads

- A thread is an identifiable flow of control within a process, with its own stack
  - A sequential process without interrupts has a single thread
  - A sequential process with interrupts has a single main thread and other notional short-lived threads
  - Other types of process considered multithreaded
- Thread execution is also dependent on platform and program priorities and policies

C Programming Models

- Thread execution is normally treated as the asynchronous execution of a function
  - One with its own thread of control, a mini *main*
  - One thread can synchronise on the completion of another thread
- Ordinary functions can have arguments and a non-*void* return type
  - A thread can be passed data on execution and return a result on completion



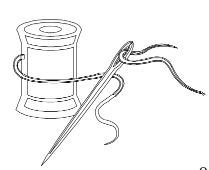
Typical C-Style Threading API

```
struct thread_t
{
    ... // platform-specific representation
};
struct thread_config_t
{
    ... // platform-specific representation
};
bool thread_create(
    thread_t *created_thread, const thread_config_t *config,
    void *main(void *), void *arg);
bool thread_join(thread_t, void **result);
bool thread_equal(thread_t, thread_t);
thread_t thread_current();
```

For brevity and simplicity, *true* and *false* are used to signal success and failure

Threads and Objects

- That the C model is function rather than object oriented is not a problem
  - Threading is inherently task oriented
- General awkwardness and lack of type safety and expressiveness is the real issue
  - Fiddly API details and *void* * for genericity
- Although a thread is an abstraction, it is not a program(mer) artefact



Active and Passive Objects

- A passive object is one that only speaks when spoken to
  - Only responds and calls other functions on other objects when one of its own functions is called
  - In essence, a traditional programming object
- An active object has a mind and life of its own
  - It owns its own thread of control, notionally associated with its own mini address space

Thread Wrapping

- Program design should be in terms of active and passive objects rather than free threads
  - Active objects should be freed from API detail
- At the object level there are essentially two approaches to encapsulating threading APIs...
  - The inheritance-based approach endows an active object with threadedness as part of its type
  - The delegation-based approach endows an active object with threadedness by association

Inheritance-based Approach

```
class threaded
{
public:
    void execute()
    {
        thread_create(&handle, 0, run, this);
    }
    void join()
    {
        thread_join(handle, 0);
    }
    ...
protected:
    virtual void main() = 0;
private:
    static void *run(void *that)
    {
        static_cast<threaded *>(that)->main();
        return 0;
    }
    thread_t handle;
};
```

Base class for active object types

In effect, an asynchronous command

Error handling omitted for brevity

In effect, an asynchronous template method

Inheritance Considered

- Tight coupling between the concept of a task object and the mechanism of its execution
  - What about event-driven or pool-based execution?
- Separate execution concerns: don't mix construction with execution
  - Initialising a threaded object is different to running it, just as starting a car is different to driving it
  - A thread in a constructor may start executing before the derived part of the object has initialised

Delegation-based Approach

```
class threadable
{
public:
    virtual void execute() = 0;
    ...
};
class threader
{
public:
    void execute(threadable *that)
    {
        thread_create(&handle, 0, run, that);
    }
    ...
private:
    static void *run(void *that)
    {
        static_cast<threaded *>(that)->execute();
        return 0;
    }
    thread_t handle;
};
```

Active objects are considered to be command objects (class threadable)

A separate object plays the role of command processor, allowing alternative executor implementations, such as pooling or time-based events (class threader)

Delegation Considered

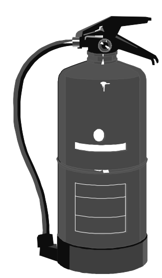
- Looser coupling than inheritance approach, trading a class relationship for an object one
  - Task independent from its execution mechanism, therefore easier to write, test and change
  - Still worth separating construction from execution
- Can be made looser by using template-based rather than *virtual* function polymorphism
  - What you can do is more important than who your parents are

Thread Safety

- Safety is not a bolt-on operational quality
  - Data integrity and liveness are common victims of incorrectly designed thread interactions
- The unit of thread safety is the function rather than the object
  - A function may be a true function or a primitive built-in operation, e.g. reading or writing to an *int*
- Safety may be achieved by immutability, atomicity or explicit locking

Safety Categories

- Safety, in terms of program data integrity, of a function or primitive can be classified as...
  - Safe only in a totally threadbare program
  - Safe only if accessed exclusively by a single thread
  - Safe only when access is explicitly requested and released by a thread
  - Safe regardless of thread access
- It is a mistake to think that all code should aspire to the last category



Critical Regions

- A region of code can be considered critical if concurrent access would be unsafe
- To be safe it must be embraced by a guard that permits no more than a thread at a time
  - A *lock* operation that blocks or lets a thread in
  - An *unlock* that releases the next waiting thread
- Synchronisation primitives are normally used to provide the basic lock and unlock features
  - Higher-level facilities are often built over an API

Synchronisation Primitives

- There are many common primitives...
  - The oldest and most basic mechanism is the binary semaphore
  - Mutexes are the most commonly used mechanism
  - Counting semaphores allow multiple threads to access a critical region
  - Reader–writer locks allow simultaneous read access but mutually exclusive write access
- Deadlock detection is often an optional quality-of-implementation feature

Simple Mutual Exclusion

- Mutexes provide mutual exclusion based on thread ownership
  - Can be strict or recursive: either deadlock or allow relocking by the same thread
- A common feature on many mutexes is a non-blocking lock operation
  - A *try_lock* allows the caller to acquire a mutex or move on and do something else without blocking
- Sometimes a timeout variant is supported

A Typical Mutex

```
struct mutex_t
{
    ... // platform-specific representation
};
struct mutex_config_t
{
    ... // platform-specific representation
};
bool mutex_create(mutex_t *, const mutex_config_t *config);
bool mutex_destroy(mutex_t *);
bool mutex_lock(mutex_t *);
bool mutex_try_lock(mutex_t *, bool *locked);
bool mutex_unlock(mutex_t *);
```

Note that *try_lock* returns *false* only in the event of an error: if the caller has acquired the mutex **locked* will be *true*

Conditional Mutual Exclusion

- Condition variables are used to notify threads of the occurrence of some condition
  - They are associated with a mutex which is reacquired on waking up
  - Actual associated condition predicate must be rechecked to ensure that it still holds true
- Mutex acquisition is subject to condition variable notification
  - Conceptually a condition is a parameter of a mutex lock, i.e. the mutex depends on it not vice-versa

A Typical Condition Variable

```
struct condition_t
{
    ... // platform-specific representation
};
struct condition_config_t
{
    ... // platform-specific representation
};
bool condition_create(condition_t *, const condition_config_t *);
bool condition_destroy(condition_t *);
bool condition_wait(condition_t *, mutex_t *);
bool condition_timed_wait(condition_t *, mutex_t *, time_spec);
bool condition_notify_one(condition_t *);
bool condition_notify_all(condition_t *);
```

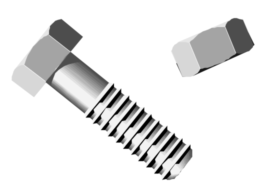
The *time_spec* type specifies the timeout as an absolute rather than relative time

Monitor Objects

- The integrity of mutable objects shared between threads can be ensured either by...
  - Locking and unlocking the object externally before and after each call or set of calls
  - Equating each function with a critical region, and locking and unlocking the object internally
- In either case, the synchronisation primitives are encapsulated within the monitor object
  - Just like free threads, avoid the kitchen synch

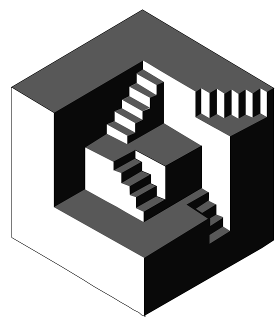
Generic Programming

- Generic programming is characterised by an open, orthogonal and expressive approach
  - Strong separation of concerns and loose coupling
  - More than just programming with templates
- Principal focus on conceptual design model rather than just on specific components
  - A stock set is typically provided for out-of-the-box use



Generic Threading

- Intent
  - Present an open and unified model for executing tasks in threads
- Content
  - The function metaphor
  - *Threadable* functional objects
  - *Threader* functional objects
  - *Joiner* functional objects
  - Uncaught exceptions



The Function is the Metaphor

- Recover the conceptual simplicity of the C model, but use generics for expressiveness
  - What happened to the result from the thread?
  - Loose coupling through templates and delegation
- Functional objects provide a unifying microarchitectural theme
  - Idiom relies on copyable objects with *operator()*, either functions or function objects

Mixed Metaphors

- A *Threadable* defines the task to be executed
  - A functional object that takes no arguments
- A *Threader* runs a *Threadable* in its own thread
  - A functional object that takes a *Threadable* object as its sole argument and returns a *Joiner*
- A *Joiner* is used to synchronise with and pick up the result from a *Threadable*
  - A functional object that takes no arguments and returns a suitable result type

Threadable Function Objects

- Ordinary nullary function objects
  - And should be callable as such

```
class threadable
{
public:
    threadable(const threadable &);
    ... // other suitable constructors, any other functions
    typedef result_type result_type;
    result_type operator()()
    {
        ... // lifecycle of the thread
    }
private:
    ... // representation accessible by call operator
};
```

Threadable Functions

- Ordinary functions can also be used
  - Additional trait support required to allow simple return-type deduction

```
template<typename nullary_function>
struct return_type
{
    typedef typename nullary_function::result_type type;
};
template<typename function_result_type>
struct return_type<function_result_type (*)()>
{
    typedef function_result_type type;
};
```


Threader Functional Objects

- Thread launch and execution policy details are separated from actual launch
  - Effectively results in an adapted function object that is executed in another thread

```
class threader
{
public:
    threader(const threader &);
    ... // other suitable constructors, any other functions
    template<typename threadable>
      joiner operator()(threadable);
private:
    ... // representation for configuring thread launch
};
```

Threader Variations

- Different kinds of *Threader* can provide for different thread configuration options
  - Concrete types encapsulate policy and mechanism
- Constructors offer the site for extension, not the function-call operator
  - Can address common per-thread requirements, such as stack sizing and priority
  - Can also handle other application-level configuration concepts, such as thread pooling

Joiner Functional Objects

- A *Joiner* is a...
  - Function proxy that stands in for the execution of the real *Threadable* object
  - Future variable for asynchronous evaluation

```
class joiner
{
public:
    joiner();
    joiner(const joiner &);
    joiner &operator=(const joiner &);
    typedef result_type result_type;
    result_type operator()();
    ...
};
```

threadof and Thread Identity

- Thread identity is treated as an opaque type
  - Supports only *operator*== and *operator!*=
- Thread identity is associated with the joiner, not the threader or the threadable
  - *threadof* applied to a joiner returns the thread identity currently associated with the joiner
  - *threadof(0)* returns the identity of the calling thread

```
joiner join;
...
if(threadof(join) == threadof(0))
    ...
```


A *threader* Class

```
class threader
{
public:
    template<typename threadable>
    joiner<return_type<threadable>::type> operator()(
        threadable function)
    {
        typedef threaded<threadable> threaded;
        thread_t handle;
        if(!thread_create(
           &handle, 0, threaded::needle, new threaded(function)))
            throw bad_thread();
        return joiner<return_type<threadable>::type>(handle);
    }
private:
    template<typename threadable>
      class threaded;
};
```

Nested *threaded* helper forward-declared for exposition only

The *threaded* Helper

```
template<typename threadable>
class threader::threaded
{
public:
    explicit threaded(threadable main)
      : main(main)
    {
    }
    static void *needle(void *eye)
    {
        std::auto_ptr<threaded> that(static_cast<threaded *>(eye));
        return new return_type<threadable>::type(that->main());
    }
private:
    threadable main;
};
```

Handling of *void* return types has been omitted for brevity

A *thread* Function

- A wrapper function can be provided for launching default configured threads
  - In this example, and in this sense, *thread* is a verb

```
template<typename threadable>
joiner<return_type<threadable>::type> thread(threadable function)
{
    return threader()(function);
}
```

A *joiner* Class Template

```
template<typename result_type>
class joiner
{
public:
    joiner();
    joiner(const joiner &);
    ~joiner();
    joiner &operator=(const joiner &);
    result_type operator()();
    ...
private:
    thread_t handle;
    bool joined;
    result_type *result;
};
template<>
class joiner<void>
{
    ...
};
```

Copy of threadable result owned by each *joiner* instance

Specialisation needed to handle *void* return case

The Act of Union

```
template<typename result_type>
class joiner
{
public:
    ...
    result_type operator()()
    {
        if(!joined)
        {
            void *thread_result;
            if(threadof(*this) == threadof(0) ||
               !thread_join(handle, &thread_result))
                throw bad_join();
            joined = true;
            result = static_cast<result_type *>(thread_result);
        }
        return *result;
    }
    ...
};
```

Uncaught Exceptions

- Ideally a thread should return normally rather than terminate with an exception
  - Just as, ideally, a program should not terminate with an exception
- However, an exception terminating a thread will, by default, also take down the program!
  - Therefore, trap the exception and map to a *std::bad_exception* on join

Rethreaded

```
template<typename threadable>
class threader::threaded
{
public:
    ...
    static void *needle(void *eye)
    {
        std::auto_ptr<threaded> that(static_cast<threaded *>(eye));
        try
        {
            return new return_type<threadable>::type(that->main());
        }
        catch(...) // one of the few times you'd ever want this...
        {
            return 0;
        }
    }
    ...
};
```

Rejoined

```
template<typename result_type>
class joiner
{
public:
    ...
    result_type operator()()
    {
        if(!joined)
        {
            void *thread_result;
            if(threadof(*this) == threadof(0) ||
               !thread_join(handle, &thread_result))
                throw bad_join();
            joined = true;
            result = static_cast<result_type *>(thread_result);
        }
        if(!result)
            throw std::bad_exception();
        return *result;
    }
    ...
};
```


Unexpected Handlers

- It is possible to further extend the design to allow an unexpected handler to be installed
  - This would become part of the threader's thread launching configuration
  - The handler would be called in the threadneedle's *catch* all clause

```
class threader
{
public:
    explicit threader(std::unexpected_handler handler = 0);
    ...
};
```

Generic Synchronisation

- Intent
  - Present an open and unified model for synchronisation between threads
- Content
  - Lockability
  - Locking semantics and substitutability
  - Timeouts
  - Lock traits
  - Lockers

Lockability

- Syntactic and semantic requirements can be used to express the range of lock alternatives
  - Core requirement of lockability must be satisfied by primitives and externally locked monitors
    - A *lock* member function acquires the lock
    - An *unlock* member function releases the lock
  - Lockability is separated from locking strategy
- Deadlock response is implementation defined
  - Either infinite blocking or *bad_lock* is thrown

A *mutex* Class

```
class mutex
{
public:
    mutex()
    {
        if(!mutex_create(&handle, 0))
            throw bad_lockable();
    }
    ~mutex()
    {
        mutex_destroy(&handle);
    }
    ...
private:
    mutex(const mutex &);
    mutex &operator=(const mutex &);
    mutex_t handle;
};
```

Throwing an exception on failure is not really an option in a destructor — a callback handler would be more appropriate

Implicit copyability does not make sense for resource objects

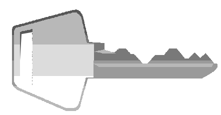
Locking a *mutex*

```
class mutex
{
public:
    ...
    void lock()
    {
        if(!mutex_lock(&handle))
            throw bad_lock();
    }
    void unlock()
    {
        if(!mutex_unlock(&handle))
            throw bad_lock();
    }
    bool try_lock()
    {
        bool locked;
        if(!mutex_try_lock(&handle, &locked))
            throw bad_lock();
        return locked;
    }
    ...
};
```

Exceptions simplify robust use of mutex

Lockable Categories

- A *Lockable* object supports the basic features required to delimit a critical region
  - Supports the basic *lock* and *unlock* functions
- A *TryLockable* object supports non-blocking
  - Additionally supports a *try_lock* function
- A *ConditionLockable* allows a condition variable to be associated with a lockable
  - Supports additional wait-related locking functions and type



ConditionLockable Interface

```
class condition_lockable
{
public:
    void lock();
    void unlock();
    bool try_lock();
    template<typename predicate>
      void lock_when(condition &, predicate);
    template<typename predicate>
      void relock_when(condition &, predicate);
    void lock_on(condition &);
    void relock_on(condition &);
    class condition;
    ...
};
```

Lockable (lock, unlock)

TryLockable (try_lock)

ConditionLockable (lock_when, relock_when, lock_on, relock_on, condition)

```
class condition
{
public:
    void notify_one();
    void notify_all();
    ...
};
```

Locking Semantics

- Locking behaviour can be further subdivided for each locking and unlocking
  - Ownership (thread affinity): owned or unowned
  - Re-entrancy: recursive or non-recursive

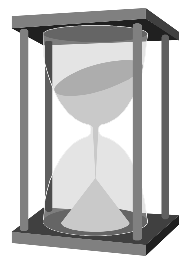
Example synchronisation primitive	Ownership	Re-entrancy
Strict mutex	Owned	Non-recursive
Recursive mutex	Owned	Recursive
Binary semaphore	Unowned	Non-recursive
Null semaphore	Unowned	Recursive

Lock Substitutability

- The categories form a subtyping hierarchy
  - *Lockable* ← *TryLockable* ← *ConditionLockable*
- Substitutability applies both to the degree of syntactic support and to the locking semantics
  - A recursive mutex and binary semaphore are substitutable in code written against a strict mutex
  - A null semaphore is substitutable for all others in a single-threaded environment

Timeouts

- Timeout variants may be optionally supported for each of the locking functions
  - A *lock* with a timeout throws a *timed_out* exception on expiry
  - A *try_lock* with a timeout simply returns *false* on expiry
  - Any of the conditional locks throw a *timed_out* exception on expiry
- An absolute time is passed to the locking function as an argument



Time Enough

```
class time
{
    ... // operator-only interface, effectively an opaque type
};
time nanosecond();
time millisecond();
time second();
time minute();
time now();
...
time operator+(const time &, const time &);
time operator-(const time &, const time &);
time operator*(int, const time &);
time operator*(const time &, int);
...
```

```
guard.lock(now() + 40 * second());
```

Lock Traits

```
template<typename lockable>
struct lock_traits
{
    typedef... lock_category;
    static const lock_ownership ownership = ...;
    static const lock_reentrancy reentrancy = ...;
    static const bool has_lock_with_timeout = ...;
    static const bool has_try_lock_with_timeout = ...;
    static const bool has_conditional_lock_with_timeout = ...;
};
```

```
template<
  typename        lock_category,
  lock_ownership  ownership                         = unowned,
  lock_reentrancy reentrancy                        = nonrecursive,
  bool            has_lock_with_timeout             = false,
  bool            has_try_lock_with_timeout         = false,
  bool            has_conditional_lock_with_timeout = false>
struct lockable;
```


Lock Inverse Traits

- Can specify characteristics to perform a reverse lookup to find a primitive lock type
  - Can find by exact match or by substitutable match

```
template<
    typename        lock_category,
    lock_ownership  ownership  = unowned,
    lock_reentrancy reentrancy = nonrecursive,
    ...>
struct find_best_lock
{
    typedef ... lock_type;
};
```

```
typedef find_best_lock<
    try_lockable_tag, owned, recursive>::lock_type recursive_mutex;
```

Lockable Objects

- The lockable model can be extended to include reader–writer and counting locks
  - *const* is taken to mean physically immutable, not just a conceptual protocol
  - Simply means that the *lock* and *unlock* count are allowed to rise higher than one
- But it would be wrong to think that lockable objects were synchronisation primitives only
  - Lockability is a generic capability and is not restricted to a handful a primitive type

Lockers

- A locker is any object or function responsible for coordinating the use of lockable objects
  - Lockers depend on lockable objects, not vice-versa, which avoids loops in the dependency graph
  - Lockers are applications of lockable objects and, as such, form a potentially unbounded family
- Most common role of lockers is for exception safety and programming convenience
  - Lockers execute-around the *lock*–*unlock* pairing

Scoped Locking

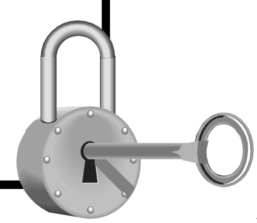
```
template<typename lockable>
class locker
{
public:
    explicit locker(lockable &lockee)
      : lockee(lockee)
    {
        lockee.lock();
    }
    ~locker()
    {
        lockee.unlock();
    }
private:
    locker(const locker &);
    locker &operator=(const locker &);
    lockable &lockee;
};
```

Substitutability between lockers and lockables does not make sense (points to `explicit locker(lockable &lockee)`)

Implicit copyability does not make sense for exclusive acquisition objects (points to `locker(const locker &);` and `locker &operator=(const locker &);`)

Scoped and Non-blocking

```
template<typename try_lockable>
class try_locker
{
public:
    explicit try_locker(try_lockable &lockee)
      : lockee(lockee), locked(lockee.try_lock())
    {
    }
    ~locker()
    {
        if(locked)
            lockee.unlock();
    }
    operator bool() const
    {
        return locked;
    }
    ...
    bool locked;
};
```



Temporary Work

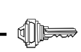
- Two mechanisms allow *try_locker*s to be used directly in a condition...
  - A variable can be declared in a condition if its type is convertible to *bool*
  - Temporaries are scope bound to references to *const*
- Can be further simplified with a common base

```
typedef try_locker<try_lockable> try_lock;
if(const try_lock &trial = try_lock(guard))
    ... // locked
else
    ... // unlocked
```

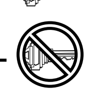

Smart Pointer Locking

```
template<typename lockable>
class locking_ptr
{
public:
    class pointer;
    explicit locking_ptr(lockable *target = 0)
      : target(target)
    {
    }
    pointer operator->() const
    {
        return pointer(target);
    }
private:
    lockable *target;
};
```

Forward declared for brevity

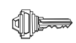


```
ptr->operation();
```



Smart Pointer Chaining

```
template<typename lockable>
class locking_ptr::pointer
{
public:
    explicit pointer(lockable *target)
      : target(target), locked(false)
    {
    }
    ~pointer()
    {
        if(locked)
            target->unlock();
    }
    lockable *operator->()
    {
        target->lock();
        locked = true;
        return target;
    }
private:
    lockable *target;
    bool locked;
};
```



Flag needed, and locking deferred from constructor until *operator()*, in case of no return-value optimisation

Conclusions

- Building from first principles it is easier to see the strengths and weaknesses in the C model
  - C threading is powerful but can be cumbersome, lacking type safety and transparent error handling
- The generic C++ model presented is a simple and unifying one
  - Moves away from C-like primitiveness
  - Loosely coupled and open
  - No more constraining than is strictly necessary