

More C++ Threading

From Procedural to Generic, by Example

Kevlin Henney

kevin@curbralan.com

Presented at the *ACCU Spring Conference*, Oxford, 16th April 2004.

Kevlin Henney

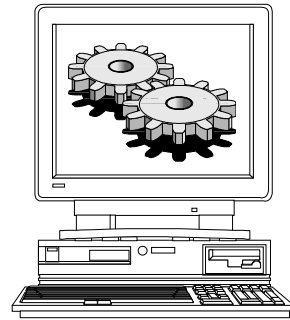
kevin@curbralan.com
kevin@acm.org

Curbralan Ltd

<http://www.curbralan.com>
Voice: +44 117 942 2990
Fax: +44 870 052 2289

Agenda

- Intent
 - ♦ Building up, by example, present a generic-programming model for multithreaded programming in C++
- Content
 - ♦ Concurrency Concepts
 - ♦ Procedural Threading
 - ♦ Object-Oriented Threading
 - ♦ Generic Threading



2

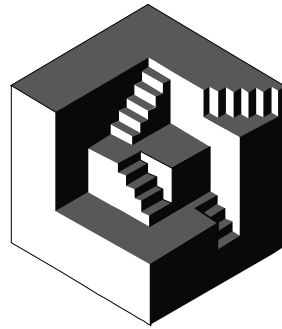
A lot has been written about multithreading, C++ and multithreading in C++. There have a number of different higher-level threading APIs written and proposed. Some are influenced by object models that are not necessarily appropriate to C++'s own idioms, and some of them suffer from looking too obviously like C API wrappers, in spite of their specific goal of API independence. In many cases the resulting object model is less expressive, although far simpler and safer to use, than the underlying API.

This talk revisits basic principles – concurrency, procedural threading, OO threading and generic programming techniques – to develop a different model for threading in C++. It is simple, idiomatic, and generic, and its thinking is more obviously unchained from the view of thread objects as C API wrappers. The generic-programming approach is more than simply using templates: it is orthogonal and open. Elements of the resulting thread programming model can also be realized in other programming languages.

This tutorial is targeted at C++ programmers with some experience of template usage, preferably based on the STL, and at least basic familiarity with concurrency concepts. It revisits and follows on from *C++ Threading*, presented at the ACCU Spring Conference in 2003.

Concurrency Concepts

- Intent
 - ◆ Introduce some primitive and high-level thread-programming concepts, plus an example
- Content
 - ◆ Processes and threads
 - ◆ Thread safety
 - ◆ Critical regions
 - ◆ Synchronisation primitives
 - ◆ Producer-consumer example



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

In most pre-emptive multitasking operating systems a process represents the coarsest unit of concurrency available. In the limit, where single-threaded processes are the norm, a process is also the smallest unit of concurrency. A process also represents a protected region of address space, a fully encapsulated unit that must make explicit and determined effort to share its data by some inter-process communication (IPC) mechanism, whether via named pipes or shared memory.

The motive for pre-emptive scheduling on a single processor is to create a virtual multi-processing model, an illusion of multiple processors. Even within multiple processors, the virtual multi-processing metaphor still holds because there is typically a many-to-one relationship between process and processor.

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

5

Another way of viewing a process is as a container of one or more threads of control, each with their own stack. In the limit, each process contains the thread associated with `main` and no others. Sequential programming has the benefit of safety and simplicity.

A virtual notion of concurrency is introduced with interrupts, such as Posix signals, which pre-empt and suspend the current thread of execution, and execute their handlers (typically) in a different stack context.

Fuller concurrency is realised in the shape of explicit multi-threading. The attraction of multiple threads in a single process is, depending on the operating system, a lower start-up and execution overhead for each thread task and simpler inter-thread communication and sharing. However, this last is also a liability: a shared address space is not something that undisciplined threads should be walking over.

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

6

Multiple threads accessing shared data raises questions of safety and determinism. A race condition can occur where two threads are trying to work on – reading from and writing to – a shared piece of data. Their access can clash, overlap and overwrite the common data, leading to data coherence problems and worse. Such defects are subtle, insidious and hard to replicate.

Although it is natural to think and talk in terms of thread safety with respect to data, it is not the data that defines correctness: it is the form of access. In other words, the concept of safety is defined with respect to function on data rather than data alone. An exclusive set of strictly immutable operations on a piece of data is implicitly safe. Similarly any operation that is defined to be atomic, i.e. uninterruptible.

Safety Categories

- Safety, in terms of program data integrity, of a function or primitive can be classified as...
 - ♦ *Threadbare*: safe only in a threadbare program
 - ♦ *Exclusive*: safe if accessed exclusively by one thread
 - ♦ *Requested*: safe only when access explicitly requested and released by a thread
 - ♦ *Transparent*: safe regardless of access
- It is a mistake to think that all code should aspire to the last category



7

The thread safety of a function should ideally be limited to the safety of its operands rather than with respect to implicit and hidden – as opposed to encapsulated – state.

For example, the C standard `strtok` function is stateful because state is held between calls. It is most simply implemented in terms of `static` data. This is a nuisance in a single-threaded program, but a potentially fatal liability in a multi-threaded program. In theory, because the vanilla C standard says nothing about threading, one should not assume any more than threadbare thread safety for `strtok`. In practice, exclusive safety can be assumed and, if all uses of `strtok` are under programmer control, requested safety is potentially possible, in conjunction with a programmer-provided and co-ordinated lock. In practice, an implementation of `strtok` may use thread-specific storage, which should be considered a facility for dealing with code reliant on global data rather than as a common design tool. There are also re-entrant versions of `strtok` that capture their context in a structure, i.e. `strtok` is effectively recognised as the object that it is.

By contrast, a pure function, such as `std::sort`, is considered to be stateless because its only dependency on mutable state is in terms of its own local, non-`static` variables, which includes arguments.

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 built on such primitives

8

A critical region of code — and associated data — is critical in the sense that it cannot support multi-threaded access, only single-threaded. In the presence of multiple threads such a critical region must have access through it negotiated to ensure that only one thread is allowed in at a time. A related concept is that of a crossroads whose flow is managed by traffic lights: the junction is the critical region, the lanes of traffic are the threads and the traffic lights represent the signalling primitives that ensure correct and exclusive access to the critical region.

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 (but useful) quality-of-implementation feature

9

Semaphores allows locking and unlocking (traditionally P and V) of a resource that has either single or counted ownership, blocking any threads that are awaiting ownership. However, one of the weaknesses of a binary semaphore is that there is no notion of thread affinity in ownership: any thread can unlock a locked semaphore, not just the one that acquired the lock. This more often a problem than a solution opportunity – property is theft.

Mutexes provide mutual exclusion based on thread ownership. They can be strict or recursive, deadlocking or allowing relock by the same thread, respectively. A common feature on many mutexes is a non-blocking lock operation. Such *try-lock* operations allow the caller to acquire a mutex or move on and do something else without blocking. A timeout variant is also normally supported.

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

10

A condition variable indicates a condition — as in a state rather than a Boolean value — that can be waited on and signalled, allowing a thread to wait until the predicate (the truth) associated with the condition becomes true and another thread to indicate that it has become true. The predicate itself has programmer-defined meaning.

Producer–Consumer Example

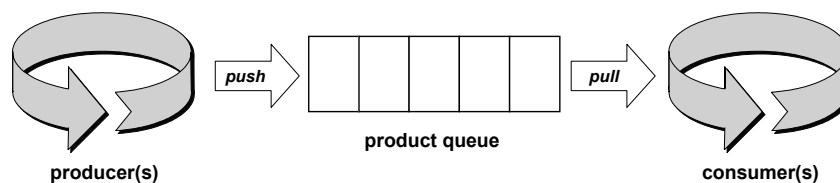
- A fairly classic demonstration of concurrency is a producer–consumer arrangement
 - ♦ A producer task creates products to be consumed by a consumer task
 - ♦ The rate of production is not necessarily constant, e.g. may be in response to external events
 - ♦ The producer and consumer should work as independently as possible, i.e. production should not depend on the rate of consumption
 - ♦ There may be multiple producers and consumers

11

Producer and consumer roles are, with the exception of the products they produce and consume, independent. To avoid lock-stepping the processing of one unnecessarily with respect to another, an intermediary, buffering role in the form of a queue can be introduced. This decouples producers and consumers not only from each others' identities, but also from each others' processing rates.

Threading and Queuing

- A product queue between the producer and consumer absorbs busyness and slack
 - ♦ The producer(s) and consumer(s) execute as separate threads associated with the queue



12

Perhaps the simplest and most common queuing arrangement is one in which both the producer and consumer roles govern control flow, but the event flow and data flow are consistently in the direction of the consumer from the producer. This leads to a push-pull architecture, where the producer pushes products onto the queue and the consumer pulls them from the queue. Producers and consumers are therefore active and the queue is passive.

Other configurations are possible: push-push, where the queue is active with respect to the consumer; pull-pull, where the producer is passive with respect to the queue; pull-push, where the queue is active with respect to both the producer and the consumer.

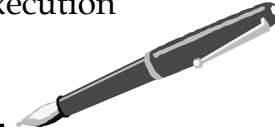
Procedural Threading

- Intent
 - ◆ Present a representative C threading API and typical usage
- Content
 - ◆ Typical C threading model
 - ◆ C-style producer and consumer
 - ◆ C-style product queue
 - ◆ Product queue condition notification and waiting



Typical C Threading Model

- 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



Most, but by no means all, multi-threading APIs in C handle threads as callbacks executed on their own stack, the task of the thread being given as a function pointer. The function pointers take an argument, used for passing in contextual data, and return a value, used for passing back results on thread completion.

C-Style Producer-Consumer

```
struct product_queue
{
    ... // representation and supporting details
};
void *produce(void *product_queue_ptr);
void *consume(void *product_queue_ptr);
```

```
product_queue products;
... // initialise products
thread_t consumer, producer;
thread_create(&consumer, 0, consume, &products);
thread_create(&producer, 0, produce, &products);
thread_join(producer, 0);
thread_join(consumer, 0);
```

Error handling, configuration and result collection omitted for brevity

15

The following is a simple C-like API, representative of a many C threading APIs. For brevity and simplicity, true and false are used to signal success and failure:

```
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();
```

C-Style Product Queue

- Queue type collocates data and mechanism
 - ◆ The underlying queue with its synchronisation

```
struct product;
struct product_queue
{
    std::queue<product *> queue;
    mutex_t guard;
    ...
};
```

```
product_queue products;
mutex_create(&products.guard, 0);
... // produce and consume
mutex_destroy(&products.guard);
```

16

The following is simple C-like API is representative of a many C mutex APIs:

```
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`.

C-Style Product Consumption

```
void *consume(void *product_queue_ptr)
{
    product_queue *products =
        static_cast<product_queue *>(product_queue_ptr);
    ...
    while(...)
    {
        product *to_consume = 0;
        mutex_lock(&products->guard);
        if(!products->queue.empty())
        {
            to_consume = products->queue.front();
            products->queue.pop();
        }
        mutex_unlock(&products->guard);
        ... // consume to_consume, if it is not null
    }
    ...
}
```

17

The product consumption is typified by housekeeping tasks: type laundering, locking up and tidying up. For brevity the code omits error handling which, when added, obscures the core logic even further. Likewise handling of thread cancellation or termination does nothing to make this any simpler. So, the C model is powerful, but mired in detail, with an impressive scope for error.

Product Queue Revisited

```
struct product;
struct product_queue
{
    std::queue<product *> queue;
    mutex_t guard;
    condition_t not_empty;
    ...
};
```

```
product_queue products;
mutex_create(&products.guard, 0);
condition_create(&products.not_empty, 0);
... // produce and consume
condition_destroy(&products.not_empty);
mutex_destroy(&products.guard);
```

18

In the following representative C API, the `time_spec` type specifies the timeout as an absolute rather than relative time:

```
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 *);
```

C-Style Product Production

```
void *produce(void *product_queue_ptr)
{
    product_queue *products =
        static_cast<product_queue *>(product_queue_ptr);
    ...
    while(...)
    {
        product *produced = ...;
        mutex_lock(&products->guard);
        products->queue.push(produced);
        condition_notify_one(&products->not_empty);
        mutex_unlock(&products->guard);
    }
    ...
}
```

19

The use of `condition_notify_one`, rather than `condition_notify_all`, is based on the idea that each waiting thread, which will be by definition a consumer, is effectively equivalent to any other thread, i.e. all consumers are equivalent. There is no reason to wake all waiting threads up because they all perform the same task: it is not as if a specific and critical task could be starved of access to the resource.

Product Consumption Again

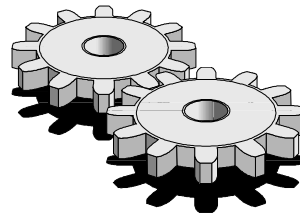
```
void *consume(void *product_queue_ptr)
{
    product_queue *products =
        static_cast<product_queue *>(product_queue_ptr);
    ...
    while(...)
    {
        mutex_lock(&products->guard);
        while(products->queue.empty())
            condition_wait(
                &products->not_empty, &products->guard);
        product *to_consume = products->queue.front();
        products->queue.pop();
        mutex_unlock(&products->guard);
        ... // consume to_consume
    }
    ...
}
```

20

Although, in principle, only one consumer will be woken at a time, in practice there are many scenarios where more than one will be signalled to wake in a way that a simple `if` statement would not be guarantee enough of correctness. The predicate associated with the condition — `!products->queue.empty()` — could be false after being woken because another consumer has already been woken and scheduled ahead of time, and therefore eaten another thread's breakfast. Hence the `while` loop.

Object-Oriented Threading

- Intent
 - ◆ Present a classic OO threading models and usage
- Content
 - ◆ Threads and objects
 - ◆ Active and passive objects
 - ◆ Inheritance-based threading
 - ◆ Delegation-based threading
 - ◆ Monitor objects
 - ◆ Internally locked queue



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



22

It is tempting to point to the common C model of threading and criticise it for not being object oriented. However, in a multi-paradigm world this is not really a reasonable criticism: not all that is OO is good, and not all that is not OO is bad. A more considered critique is based on safety and ease of use. Quite simply, the C solution is not well encapsulated, offering too many opportunities for error and duplicate coding.

In coming up with an object-based abstraction, it is tempting to equate a thread with an object. However, threads are not objects in a programmer accessible sense of the word: a thread is a path of execution. Having a class named `thread` that holds the task of a thread confuses the task of a thread with the thread itself. There are cases when such an identity can be useful, but a separation of concerns in this case offers a clearer vocabulary based on capabilities and roles.

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



23

Active objects are animated by an associated thread of control whereas passive objects are more classic objects: they speak only when spoken to. Reactive objects are sometimes identified as a third category, a variant of passive objects. Reactive objects express the idea of control-flow inversion, and are responsive and re-entrant.

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

24

Many threaded programs suffer from free-threaded design, where threads are visible to other threads, along with their synchronisation primitives. These architectures are weakly encapsulated, difficult to evolve, hard to debug (the requirement for which, in some cases, is almost an inevitability) and almost impossible to unit test.

Process-based design offers a much better model for thread design. Data should be considered unshared until proven otherwise, whereas many thread programmers assume that data is always shared because it is in the same address space. A process as a unit of encapsulated data equates nicely to an object, and this offers object-oriented approaches to threading a natural advantage over procedural models.

A further step is to adopt a virtual uniprocessor metaphor of design. Like a perceived stream of consciousness that emerges from the inherently parallel architecture of the human brain, a multi-threaded program should not appear to be overtly multi-threaded from any local perspective in the code. Synchronisation primitives and threading mechanisms should be hidden rather than exposed, leading to local regions of apparently sequential code that are driven externally by threads. Such architectures are easier to reason about and their parts can be unit tested more easily in isolation from one another.

Inheritance-based Approach

```
class threaded ← Base class for active
{ object types
public:
  void execute() ← In effect, an
  { asynchronous
    thread_create(&handle, 0, run, this); ← command
  }
  void join()
  {
    thread_join(handle, 0); ← Error handling
  } omitted for brevity
  ...
protected:
  virtual void main() = 0; ← In effect, an
private: asynchronous
  static void *run(void *that) template method
  {
    static_cast<threaded *>(that)->main();
    return 0;
  }
  thread_t handle;
};
```

25

One natural way to think of threading with respect to active objects is that of an inherited property, so active object types inherit from a base class that endows them with threaded capabilities.

Inheritance-based Consumer

```
class threaded_consumer : public threaded
{
public:
    explicit threaded_consumer(product_queue *);
    ...
protected:
    virtual void main();
private:
    product_queue *products;
};
```

```
product_queue products;
... // initialise products
threaded_consumer consumer(&products);
consumer.execute();
... // initialise, run and join a producer thread
consumer.join();
```

Active object types override a Hook Method that is used by a hidden Method Skeleton responsible for initiating threading and wrapping an underlying thread API.

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

27

Inheritance is the strongest form of coupling possible in an OO system, and in this case the coupling is perhaps too strong: the task of an active object can only be executed in a separate thread. It cannot be tested or used separately. Such tight coupling is inappropriate, and these independent ideas need to be expressed independently.

It is also possible to use an upside-down, templated form of inheritance, a Parameterized Derivation, that adapts a non-threaded task as a threaded one, introducing threading as a derived rather than a base capability.

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<threadable *>(that)->execute();  
        return 0;  
    }  
    thread_t handle;  
};
```

Active objects are considered to be command objects

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

28

A delegation- or forwarding-based approach demonstrates a clearer separation of concerns and conforms to a cleaner class hierarchy design: `threader` is concrete and `threadable`, which is the root for active objects, is purely abstract, an Interface Class. The inheritance-based design was not rooted in a fully abstract class.

Delegation-based Consumer

```
class threadable_consumer : public threadable
{
public:
    explicit threadable_consumer(product_queue *);
    virtual void execute();
    ...
private:
    product_queue *products;
};
```

```
product_queue products;
... // initialise products
threadable_consumer consumer(&products);
threader consumer_threader;
consumer_threader.execute(&consumer);
... // initialise, run and join a producer thread
consumer_threader.join();
```

In the delegation-based approach it is the collaboration of a `threadable` descendent and a `threader` that forms an active object, rather than an active object having a single classifying type.

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

30

Although the delegation-based approach is more loosely coupled than the inheritance-based approach, for simple cases it does involve more house-keeping work on the part of the user.

Mutex Wrapping

- Expressing mutexes as objects is a natural step
 - ♦ Exceptions simplify common usage of interface

```
class mutex
{
public:
    mutex();
    ~mutex();
    void lock();
    void unlock();
    bool try_lock();
    ...
private:
    mutex(const mutex &); ← Implicit copyability does
    mutex &operator=(const mutex &); ← not make sense for
    mutex_t handle;           resource objects
};
```

31

```
mutex::mutex()
{
    if(!mutex_create(&handle, 0))
        throw bad_lockable();
}
mutex::~~mutex()
{
    mutex_destroy(&handle); // simplistic quashing of failure
}
void mutex::lock()
{
    if(!mutex_lock(&handle))
        throw bad_lock();
}
void mutex::unlock()
{
    if(!mutex_unlock(&handle))
        throw bad_lock();
}
bool mutex::try_lock()
{
    bool locked;
    if(!mutex_try_lock(&handle, &locked))
        throw bad_lock();
    return locked;
}
```

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 synchronisation

32

Rather than associating objects that need explicit management with a separate synchronisation primitive, the use of primitives should be hidden from view. This encapsulation ensures that what are actually dependent concepts are expressed dependently – unification rather than separation.

Requiring a caller to manually lock an object still suffers some problems, e.g. too easy to forget, but does offer the opportunity to provide a critical region over an arbitrary number and set of calls. In some cases these can be better encapsulated using Execute-Around Methods and Combined Methods. Perhaps the biggest risk with an exclusively externally locked approach is that of denial of service: a rogue client can call a lock without the corresponding unlock.

Internally Locked Queue

```
class product_queue
{
public:
    product_queue();
    ~product_queue();
    void push(product *);
    product *pull();
    product *try_pull();
    ...
private:
    std::queue<product *> queue;
    mutex guard;
    ...
};
```

The constructor and destructor automate initialisation and finalisation of synchronisation mechanisms, the *push* function simplifies the producer code and the *pull* function simplifies the consumer code

33

Internally locked structures are easier to test and reason about because their interfaces look like ordinary passive object interfaces without temporal coupling governing use and ordering of function calls in the interface.

Internal Critical Regions

```
void product_queue::push(product *pushed)
{
    guard.lock();
    queue.push(pushes);
    ... // notify non-empty condition
    guard.unlock();
}
```

```
product *product_queue::pull()
{
    guard.lock();
    while(queue.empty())
        ... // wait on non-empty condition
    product *pulled = queue.front();
    queue.pop();
    guard.unlock();
    return pulled;
}
```

34

Internally locked objects are fully encapsulated with respect to their concerns and their synchronisation. It is here that synchronisation primitives can be used. Note that the code shown has not been made exception safe – that concern is dealt with later.

Optional Critical Region

```
product *product_queue::try_pull()
{
    product *pulled = 0;
    if(guard.try_lock())
    {
        if(!queue.empty())
        {
            pulled = queue.front();
            queue.pop();
        }
        guard.unlock();
    }
    return pulled;
}
```

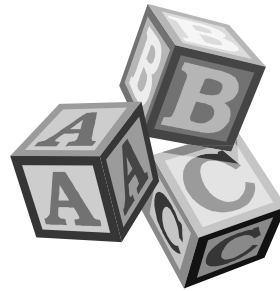
try_pull represents a combined operation, combining a pure query on emptiness with an already combined operation, *pull*, that yields a value and modifies the state of the queue

35

A non-blocking variation of the `pull` operation allows consumer, potentially, to engage in other tasks in the absence of products, should that be a relevant and reasonable requirement.

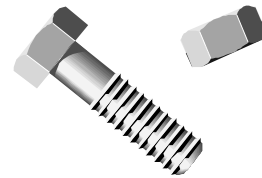
Generic Threading

- Intent
 - ◆ Present an open and unified model for threaded task execution and synchronisation
- Content
 - ◆ The function metaphor
 - ◆ Uncaught exceptions
 - ◆ Lockability and substitutability
 - ◆ Lock traits
 - ◆ Lockers



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



37

Generic programming is an approach to program composition that emphasises algorithmic abstraction. It is built on compile-time polymorphism and value-based programming: templates, overloading and conversions; copying and no explicit memory management.

Perhaps one of the most important distinctions to make is that generic programming is more than simply programming with generics. A lot of code that uses templates does not qualify. Generic programming is more broadly about a separated and loosely coupled design model that allows different features to vary independently.

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



38

Software and its development are inherently metaphor driven: real-world concepts and terms are used to describe and define virtual concepts. Metaphors provide vocabulary and consistency to a design, where used coherently.

The asynchronous function model of threading is a deep metaphor, not just a superficial convenience or dressing of an API. Everything really is usable as a function. The underlying idea is that a function represents a task in a control flow, and this leads to the roles present in the model.

Mixed Metaphors

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

39

There are three function object types associated with the asynchronous function model: *Threadable*, *Threader* and *Joiner*. *Threadable* objects or functions hold the task or lifecycle that is executed within a thread. *Threader* objects or functions are Executors that launch *Threadable* objects to be executed within a thread. A *Joiner* is the result of calling a *Threader* and is used to rendezvous with a thread by picking up the result of the *Threadable*. There is an element of higher-order functional programming: a *Threader* is effectively a function that takes a function and yields a function.

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
};
```

40

In addition to the syntactic model shown above, the use of a *Threadable* object must not result in a thrown exception. It is the job of a *Threadable*'s designer to ensure that exceptions do not escape as a matter of course, just as one would not expect to allow or encourage exceptions to escape from `main`.

Ordinary functions can also satisfy *Threadable* requirements. However, additional trait support is required to allow simple return-type deduction for both function objects and functions:

```
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
};
```

41

Different kinds of *Threader* can provide for different thread configuration options: each concrete types encapsulate policy and mechanism.

Constructors on the specific *Threader* type offer the site for extension, not the function-call operator, i.e. you do not add arguments to the launching function-call operator to affect the operational behaviour of the thread to be run. Such constructor-based parameterization can address common per-thread requirements, such as stack sizing and priority, without interfering with the essential generic threading concept. The *Threader* concept is sufficiently flexible that it can also handle other application-level configuration concepts, such as thread pooling: a thread pool can be expressed to satisfy *Threader* requirements.

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()();
    ...
};
```

42

Notionally, where a *Threader* forks the flow of control into a new thread, a *Joiner* joins with it. There is an element of symmetry to this design: a function is used to launch a thread and a function is used to meet it.

A *Joiner* may optionally support a conversion to `bool` for joinability, which indicates whether a thread has been explicitly detached or not, i.e. whether it is actually joinable or not.

An ideal implementation of this threading model allows *Joiner* objects to be copied freely so that multiple threads can appear to join a given thread, when in fact only one joins it and the rest pick up the same result. This requires a thread-safe, reference-counted implementation to handle lifetime and validity issues.

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))  
...  
...
```

43

The identity of a thread is treated as an opaque type, the result of a *threadof* operation. Given platforms can extend this by associating non-member functions that work with the thread identity to provide additional operations, e.g. thread priority modification or termination. Arguably many of these could be seen as operations on a joiner.

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

44

Handling of void return types has been omitted for brevity:

```
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;
};
```

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
- Parallels with procedural thread model
 - ♦ But simplified for common use

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

45

It is a common enough task to simply launch a thread with default settings, and without worrying specifically about *Threader* objects, that a helper with this role can be defined to simplify syntax and use. Such deducing helper function templates are common in generic programming, e.g. `std::make_pair`. Depending on common use in a particular system, many such helpers could be defined.

In many ways the introduction of such helpers recalls the C-style `thread_create`, but capturing common use rather than all parameters of variation.

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; ← Copy of threadable result
                           owned by each joiner
                           instance
};
template<>
class joiner<void> ← Specialisation needed to
                   handle void return case
{
    ...
};
```

46

The code outlines a simple *joiner*, one that does not support joining to an already joined thread or detection of detachment:

```
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;
    }
    ...
};
```

Producer-Consumer Revisited

- The threading model leads to simplified thread usage code
 - ◆ Thread launching and rendezvous are expressed directly through the function metaphor

```
product_queue products;  
joiner<void> join_consumer = thread(consumer(&products));  
joiner<void> join_producer = thread(producer(&products));  
join_producer();  
join_consumer();
```

47

The resulting producer-consumer launch and join code looks significantly simpler than the other approaches. It recalls the C model, in many ways, but is more self-contained and less error prone.

Producer Revisited

```
class producer
{
public:
    typedef void result_type;
    explicit producer(product_queue *);
    void operator()()
    {
        while(...)
        {
            product *produced = ...;
            products->push(produced);
        }
    }
private:
    product_queue *products;
};
```

48

The producer is rendered as a conventional, nullary function object type. Its main loop is simple and also easy to test in isolation, assuming the ability to substitute a Mock Object for a product_queue.

Consumer Revisited

```
class consumer
{
public:
    typedef void result_type;
    explicit consumer(product_queue *);
    void operator()()
    {
        while(...)
        {
            product *to_consume = products->pull();
            ... // consume to_consume
        }
    }
private:
    product_queue *products;
};
```

49

Similarly consumers are expressed through a conventional and independently testable function object type.

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



50

```
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);
    ...
};
```

51

Assuming handler is remembered by the threaded helper, and an empty function is used if null (i.e. a Null Object), *needle* becomes:

```
template<typename threadable>
void *threader::threaded::needle(void *eye)
{
    std::auto_ptr<threaded> that(
        static_cast<threaded *>(eye));
    try
    {
        return new return_type<threadable>::type(
            that->main());
    }
    catch(...)
    {
        try
        {
            that->handler();
        }
        catch(...)
        {
        }
        return 0;
    }
}
```

Lockability

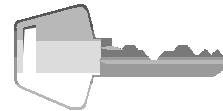
- Syntactic and semantic requirements can be used to express the range of lock alternatives
 - ♦ Core requirement of lockability must be satisfied by primitives and externally locked monitors
 - A *lock* member function acquires the lock
 - An *unlock* member function releases the lock
 - ♦ Lockability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

52

It is tempting to define locking in terms of synchronisation primitives. However, lockability is better treated as a general semantic concept that any object, primitive or high-level, could in principle satisfy.

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



53

ConditionLockable – which adds to *TryLockable*, which in turn adds to *Lockable* – can be sketched as follows:

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

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



54

Locking behaviour can be further subdivided for each locking and unlocking:

- *Ownership* (thread affinity): owned or unowned
- *Re-entrancy*: recursive or non-recursive

We can see these two axes of variation expressed against some primitive synchronisation mechanisms:

- *Strict Mutex*: owned and non-recursive
- *Recursive Mutex*: owned and recursive
- *Binary Semaphore*: unowned and non-recursive
- *Null Semaphore*: unowned and recursive

Importantly, substitutability is very context dependent: one cannot arbitrarily substitute a null semaphore (a Null Object) for any use of any other locks!

Waiting on a Condition

```
class product_queue
{
public:
    ...
    product *pull()
    {
        guard.lock();
        while(queue.empty())
            guard.relock_on(not_empty);
        product *pulled = queue.front();
        queue.pop();
        guard.unlock();
        return pulled;
    }
    ...
};
```

55

Treating condition locking as a property of mutexes rather than vice-versa has the benefit of making clear how something is locked and accessed, as it were emphasising it in the first person.

Waiting with a Predicate

```
class product_queue
{
public:
    ...
    product *pull()
    {
        guard.lock_when(not_empty, has_products(queue));
        product *pulled = queue.front();
        queue.pop();
        guard.unlock();
        return pulled;
    }
    ...
};

class has_products
{
public:
    explicit has_products(std::queue<product *> &);
    bool operator()() const
    {
        return !queue.empty();
    }
private:
    std::queue<product *> &queue;
};
```

56

Requiring the user of a condition variable to implement a `while` loop to verify a condition's predicate is potentially error prone. It can be better encapsulated by passing the predicate as a function object to the locking function.

Timeouts

- Timeout variants may be optionally supported for 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
- The timeout is passed to locking functions as an extra argument



57

Use of timeouts can create more robust programs, by not blocking forever, but at the same time one needs to avoid annoyingly arbitrary limits. The following is an opaque-type approach to handling time:

```
class time
{
    ... // operator-only interface
};
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);
...
```

This leads to usage such as the following:

```
guard.lock(10 * second());
```

Lock Traits and Inverse Traits

- Traits can characterise lockable types
 - ♦ Inverse traits can match a lockable type based on specific traits, for a given family of lock types

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

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

58

Trait definition can be assisted with a helper template, used as a base:

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

It is also possible to specify characteristics to perform a reverse lookup to find a primitive lock type, either by exact match or by substitutable match:

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

Lockers

- A locker is any object or function responsible for coordinating the use of lockable objects
 - ♦ Lockers depend on lockable objects – which need not be locking primitives – and not vice-versa
 - This avoids cycles 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

59

The lockable model can be extended to include reader-writer and counting locks. For reader-writer locks `const` is taken to mean physically immutable, and is not just a conceptual protocol. For counting locks, `lock` and `unlock` count are allowed to rise higher than one.

Note that it is easy to think that the only kinds of lockable objects are the basic synchronisation primitives. Lockability is a generic capability and is not restricted to a handful of primitive types. This becomes apparent with the concept of lockers, which realise the Execute-Around Object pattern (also known by the common misnomer of Resource Acquisition is Initialization).

A locker defines an execution strategy for locking and unlocking that is automated by construction and destruction. It simplifies common use of locking, and does so in an exception-safe fashion. As such, lockers depend on the interface of lockables – e.g. `lock` and `unlock` – but lockables do not depend on lockers. The relationship is strictly layered, open and extensible: lockable types may be whole, externally locked objects against which existing lockers can be used; new lockers can be defined that work against existing lockable types.

Scoped Locking

```
class product_queue
{
public:
    product_queue();
    ~product_queue();
    void push(product *pushed)
    {
        Locker<mutex> scoped(guard);
        queue.push(pushes);
        not_empty.notify_one();
    }
    ...
private:
    std::queue<product *> queue;
    mutex guard;
    mutex::condition not_empty;
    ...
};
```

60

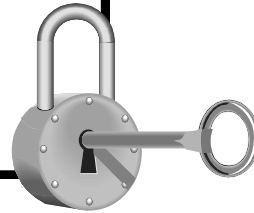
The following locker type depends only on Lockable requirements:

```
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 lockables and lockers does not make sense, so the constructor is explicit. Implicit copyability is also disabled.

Scoped and Non-blocking

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



61

The following code shows one way to use the try_locker code above:

```
product *product_queue::try_pull()
{
    product *pulled = 0;
    try_locker<mutex> scoped(guard);
    if(scoped && !queue.empty())
    {
        pulled = queue.front();
        queue.pop();
    }
    return pulled;
}
```

A safer strategy for a Boolean conversion is to use a member pointer rather than a bool, which is typically too permissive:

```
typedef bool try_locker::*is_locked;
operator is_locked() const
{
    return locked ? &try_locker::locked : 0;
}
```

Temporary Work

```
typedef const try_locker<mutex> &locked;
```

```
product *product_queue::try_pull()
{
    product *pulled = 0;
    if(locked scoped = guard)
    {
        if(!queue.empty())
        {
            pulled = queue.front();
            queue.pop();
        }
    }
    return pulled;
}
```

62

Two mechanisms can allow a `try_locker` to be used directly in a condition: a variable can be declared in a condition if its type is convertible to `bool` and temporaries are scope bound to references to `const`. Common usage can be captured and the mechanism further generalised for convenience, but the model and code above demonstrates the essential concepts.

Externally Locked Queue

- An alternative or complementary approach is to support external locking for an object
 - ♦ Multiple calls may be grouped within the same externally defined critical region

```
class product_queue
{
public:
    void lock(); ← Lockable
    void unlock(); ← Lockable
    ...
};

Locker<product_queue> guard(*products);
...
products->push(produced);
```

63

External locking has some associated risks for high-level objects. Incorrect usage can be too easy: a forgotten call to `lock` or `unlock` is more likely than with synchronisation primitives because the focus of using the object is on the rest of its non-*Lockable* interface, so it becomes easy to forget that to use the interface correctly also requires participation in a locking scheme.

To some extent lockers can help, but such a co-operative scheme should only be employed when internal locking is too restricted for a given use, e.g. multiple operations must be performed together. Ideally, if such operations are common they should be defined internally locked and defined in the interface of the object as Combined Methods.

Assuming that locks are re-entrant, external locking can be provided to complement the more encapsulated internal locking, i.e. by default if you want to call a single function you just call it and it automatically locks, but if you want to call multiple functions together you first apply an external lock.

Lockers as Smart Pointers

```
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;
};
```

```
products->push(produced);
```



64

Where only external locking is used, a safe approach is needed for calling single functions easily. The following code completes the detail of the Execute-Around Pointer shown above:

```
class 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;
};
```

Conclusions

- Building from first principles it is easier to see the strengths and weaknesses in the C model
 - ♦ Powerful, but can be tedious and error prone
- The generic C++ model presented is a simple and unifying one
 - ♦ Moves away from C-like primitiveness, but is more idiomatic than the classic OO model
 - ♦ Loosely coupled, open and no more constraining than is strictly necessary