C++ Advanced Design Issues

Asynchronous C++

Kevlin Henney

Presented at Visual Tools, Keble College, Oxford University, Monday 9th September 1996.

Overview

- **Asynchronous behaviour**
  - Timers
  - Threads
  - Synchronisation
  - Thread safety
  - Objects in shared memory
- **Techniques**
  - Application of idioms and patterns to create design solutions
  - Use of templates and inheritance for generalisation
  - Use of operator overloading to simplify class use
- **New language features**

This talk addresses the issues presented by asynchronicity in system architecture, with an emphasis on the methods and techniques used to model system features as objects and represent them in a robust C++ framework. Such frameworks are seen as the appropriate medium for presenting, and extending, system functionality to the programmer.

Comprehension and development of these classes is separate task to the development of typical applications, and often requires a deeper understanding of both detailed design and language. A number of advanced techniques, design patterns, idioms, and new language features are employed “under the hood” to construct these classes.

The aim is on the construction of a framework rather than on any specific run-time or operating system. Where appropriate, references will be made to Win32, Solaris, POSIX, etc. The emphasis on portability is not simply a reference to OS portability, but version to version portability. Developers who do not regard portability as an issue tend to get bitten more by the latter than those who take a more abstract, layered and mature approach.

A pattern is a solution to a problem in a context, and for idioms the context is closer to the language. Some commonly known C++ techniques are now enhanced by better support from the language. The draft C++ standard is nearing finalisation, and some new features will be presented in context as a preview of techniques that may soon available. Not all the features are supported by current compilers.

Callback idioms

- **The role of callbacks**
  - Decouple selection of functionality from its execution
  - Late binding of functionality to data
  - Provide some sort of generic interface to plug-in functionality
- **The C callback function model is too primitive**
  - Functions are stateless, so functions requiring a context need data passed in separately
  - Functions are defined at compile time, and cannot be customised or composed dynamically
  - Error prone and unsafe
  - Inextensible

Event driven programming, whether on a GUI messaged based system or down at the level of interrupts, is characterised in C by the use of function pointers. These present many apparent roots of control within a system.

A review of the techniques used in C reveals that the representation of callback concepts is done through clumsy interfaces: for data to be associated with an event it is often passed in as a parameter of generic form, i.e. `void *` or worse; for context, data must be associated with the callback at registration, only to be passed back to the function with the event; the definition of functions is fixed at compile time — although selection may be performed dynamically, definition cannot.

Multiple roots of control, association of data with function, grouping of related functions, generic interfaces with alternative implementations, etc. are mapped more easily into an object model than into a procedural one.

Callbacks via interfaces

- **Association of function(s) with data via common interfaces can be expressed with interface classes**

```
class updateable
{
public:
    virtual void update() = 0;
protected:
    ~updateable() {}
};
```

- **Classes can support multiple interfaces**

```
class cached_view
   : public virtual updateable, public virtual displayable { ... };
```

- **Interface support can be queried at runtime**

```
displayable *component = 0;
...
if(updateable *view = dynamic_cast<updateable *>(component))
    view->update();
```

Asynchronous C++ 4

Abstract base classes can be used to represent pure interfaces to functionality. They represent a simplified protocol through which an object can access the service of another object independently of its class implementation. One or more pure virtual functions represent the entry points to the behaviour, allowing the relationship of function and data to be inverted: a pointer is held to an object rather than a function.

This approach can be used in implementing notification architectures such as Observer pattern and Model-View-Controller configuration, of which the Document-View architecture is a degenerate variant.

In the case illustrated above, the destructor is declared `protected` because public deletion is not a property offered by this interface.

Multiple interfaces implies multiple inheritance, but it is not meaningful to have repeated interface inheritance so interfaces are used as `virtual` base classes. This is supported by an explicitly named construct in Java, but has always been available as a technique in C++ and other languages.

Run-time type information (RTTI) and `dynamic_cast` now offer better support for this technique. A pointer (or reference) can be cast down or across safely to a more derived or sibling class. Unlike hand crafted RTTI mechanisms, this works with virtual base classes and multiple inheritance. The result of a successful `dynamic_cast` is a correctly typed pointer, with returned null on failure. Declaration and test can be combined in a single condition, with the scope restricted to that of the control structure.

Interfacing with C callbacks

Header with prototypes

```
int timer(int, void (*)(void *), void *);
```

Wrapper class

```
class scheduled
{
public:
    void run_in(interval when)
    {
        timer(when, handler, this);
    }
    ...
protected:
    ...
    virtual void on_expiry() = 0;
private:
    static void handler(void *me)
    {
        reinterpret_cast<scheduled *>(me)->
            on_expiry();
    }
};
```

Virtual call

Simple derived class

```
class echo : public scheduled
{
    ...
    virtual void on_expiry()
    {
        cout << message << endl;
    }
    ...
};
```

```
echo delayed_welcome("Hello");
delayed_welcome.run_in(5);
```

Callback

Set up timer callback

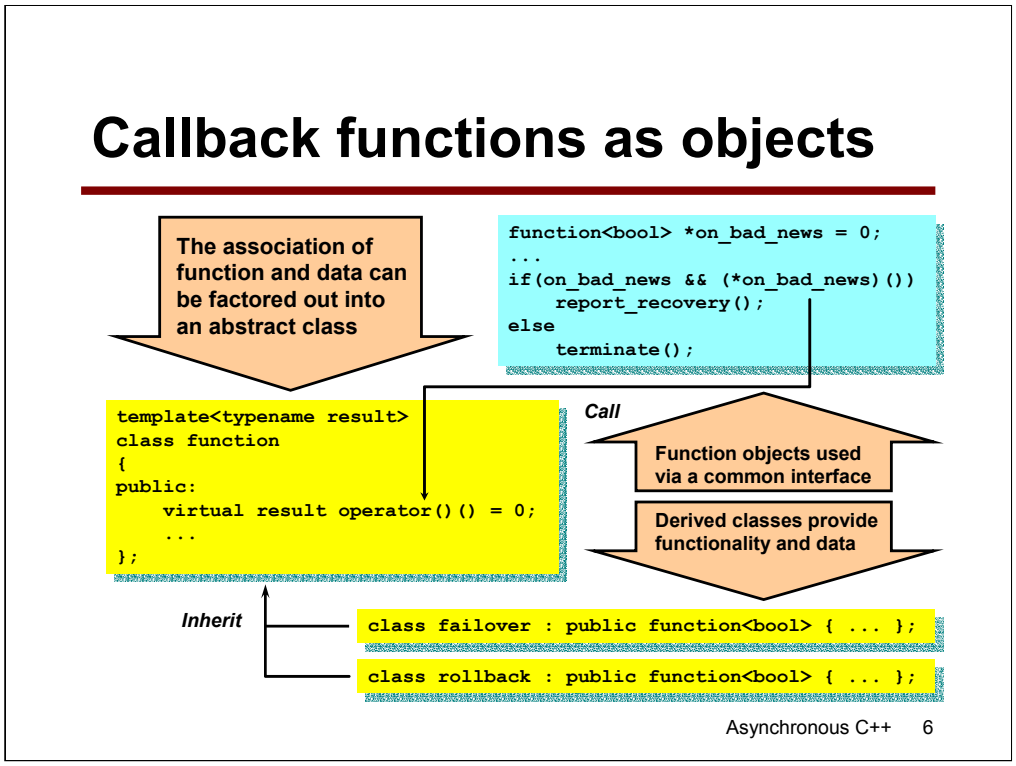
Asynchronous C++ 5

A common requirement is to take a callback system implemented in C, and wrap it up as C++. In the example shown here a generic interface to a timer registration is shown. The C function is fairly typical in its set of arguments: one or more arguments control the execution, one argument is the function, and one or more arguments represent user data to be passed back in.

To wrap this up as C++, the function to be called back is a class `static`. Its user data is the `this` pointer of the registering object. This is passed in as a `void *`, but is converted to the base class type and the place holder virtual function is called. This kind of algorithmic inversion, where a skeleton of a function is defined to rely on deferred functionality, is termed the Template Method pattern (not to be confused with C++ templates), and is described in *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides.

The `reinterpret_cast` is one of the new cast operators (the other two not mentioned so far are `static_cast` and `const_cast`). These operators explicitly name the kind of cast they perform, rather than the old style notation that mixes many of these roles (intentionally or otherwise). As such they make the code's intent clearer and are easier to *grep* for. A `reinterpret_cast` is one that reinterprets the underlying representation of its operand as another type.

Note that class body inline functions are used here and throughout the talk for brevity rather than as recommended practice — inlines tend to be inappropriate for functions that are `virtual` or whose address is used.



The idea that user data and functionality can be associated, can be modelled explicitly. A functor (or functional, functionoid, or simply function object) is an object that masquerades as a function. The class described here models a function hierarchy in which the function part, represented by `operator()`, takes no arguments. Derived classes then redefine this place holder, and add any support functions and data as necessary.

A further level of generalisation is possible in describing the return value of the functor. Templates are used here to represent *horizontal generalisation*, whereas inheritance represents *vertical generalisation*. The template parameter is used to name the return type of the functor. Note that a recent keyword, `typename`, is used rather than `class` to qualify the parameter.

In this context it is identical to `class`. However, it is felt to be clearer and more accurate than `class` when any type is being referred to. As a matter of convention, throughout the rest of this talk `typename` will be used to name any type, and `class` will refer to actual `class` types. `typename` has another more significant use within the language, but the same can be said of `class`!

The `bool` type is a built-in type associated with keyword constants `false` and `true`. It was introduced to resolve the difficulties (and indeed, impossibility) of defining a library type that has consistent Boolean behaviour appropriate to the language. It is also the final word on the many such mismatched types offered by library vendors. The `bool` type supports all the implicit conversions to make it compatible with its `int` predecessor.

Wrapping up function pointers

An adaptor class can be defined for function pointers

```
template<typename result>
class function_ptr
  : public function<result>
{
public:
    function_ptr(result callback())
      : ptr(callback) {}
    virtual result operator()()
        { return ptr(); }
private:
    result (*ptr)();
};
```

Transparent and uniform use of complex objects and function pointers

```
bool prompt_user_to_continue();
function_ptr<bool> prompt =
    prompt_user_to_continue;
...
on_bad_news = &prompt;
```

Asynchronous C++ 7

To make the function object model comprehensive in a hybrid language like C++, the low level mechanism such as function pointers must be brought into the fold. The code above illustrates such a class, wrapping a function pointer up in an object. The adaptor above is is an example of the Adapter pattern, whose intent is described in *Design Patterns* as

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

With function pointer declarations, the soundest policy is often to introduce a `typedef` to reduce the hieroglyphic index of the content. This was not felt to be necessary in the code above given the size of the example. Instead, advantage is taken of the C syntax sugar that allows function pointer arguments to be declared using function syntax.

Member function pointers

- **Applying member function pointers to objects**
  - Chosen behaviour can vary on a per-object basis
  - Different functionality can be chosen over time
  - Can be used as a form of customisable polymorphism

```
template<typename id, class mapped> class function_map
{
public:
    ...
    void operator()(mapped &object, id request) { (object.*lookup[request])(); }
private:
    map<id, void (mapped::*)()> lookup;
};
```

```
const pair<service, void (server::*)()> service_spec[size] =
    { make_pair(start, &server::on_start), make_pair(flush, &server::null), ... };
function_map<service, server> deliver(service_spec, service_spec + size);
...
deliver(handler_object, service_request);
```


Member function pointers allow separate selection of a member function and target object. Unlike the case of ordinary, unbound function pointers, there is no syntax sugaring on offer — the syntax is logical, but gruesome.

Being able to independently refer to a member function as almost an object in its own right allows programmers to program alternatives to normal, class defined polymorphism. Whereas conventional polymorphism is defined at compile time on a per-class basis, member function pointers can be used to implement per-object variations in behaviour, they may be changed at runtime, or they may be used to implement a pseudo-*vtable*, as above.

The `function_map` class is a container that maps some specified identifier type, such as an `enum` or string identifying a message or event, to a member function for a specified type. `function_map` objects are functors that are substitutes for the normal, unnamed function lookup mechanism, where statically bound identifiers are applied via the dot or arrow operators. This meta-level of programming is normally associated with more dynamic languages, such as CLOS, Dylan, Smalltalk and Self.

The `map` class is an associative array or dictionary class that is a part of the draft library standard, and is originally from the Standard Template Library (STL). Also from the STL is the `pair` template, which is an aggregate holding two public members. It is used for an *ad hoc* association of data couplets, a task simplified with the `make_pair` function. The `deliver` object is initialised from an array of associations in the STL iterator style of a range bound by an initial pointer and a one-past-the-end pointer.

Dynamic data+function binding

- **It is possible to reify the association between an object and the dynamically chosen function**
- **The bound pair can then be treated as another function object**

```
template<class target>
class remember_function
  : public function<void>
{
public:
    remember_function(
        target *on,
        void (target::*call)())
      : ptr(on), member(call) {}
    virtual void operator()()
        { (ptr->*member)(); }
private:
    target *ptr;
    void   (target::*member)();
};
```

```
multimap<event_id, function<void> *> event_table;
remember_function<connection> flusher(&db, &connection::flush);
event_table.insert(make_pair(bad_news, &flusher));
```


As with many strong associations, the delayed binding between an object and a member pointer can be wrapped up. An adaptor is shown above that implements fully bound, free standing function objects from that data+function pairing.

A `remember_function` public data member can be used as a pseudo-member function, providing the target object is initialised with `this` so that callback occurs back into the current object.

The `multimap` standard container class allows values to be stored against non-unique keys; here, an event could trigger multiple callbacks.

We can rewrite the `function_map` call mechanism, illustrated previously, in terms of `remember_function` and operator overloading — any of the left to right associating binary operators — to choose a more visually appealing call method. The comma operator would be of use as this gives the feeling of treating a call as a tuple/association between the receiver, the mechanism, and the selector:

```
(object, mechanism, selector)();
```

The code behind this is somewhat less appealing than its usage suggests:

```
template<typename id, class mapped>
  pair<mapped *, function_map<id, mapped> *>
    operator,(mapped &, function_map<id, mapped> &);
template<typename id, class mapped>
  remember_function<mapped> operator,(
    pair<mapped *, function_map<id, mapped> *> &, id);
```

Multi-threading

- **A thread is a path of control within a process**
  - A process is a container for one or more threads
  - Threads have their own execution context, but share their enclosing process address space
- **Many modern operating systems offer threading**
  - Scheduling is managed by the OS or a runtime library
  - Threading functionality is typically accessed via a C API
- **C++ can be used to build threaded frameworks**
  - Easier to use as shields user from API issues
  - Uniform and extensible interface model
  - Portability

Threads present a number of design and implementation challenges, as well shifts in perspective. The idea that an object encapsulates state and presents a safe, defined operational interface to users prevails even more strongly in multi-threaded OO. Many of the concepts used for process design are more applicable than those used for sequential, single process development.

Threads offer a means of sharing data more readily across concurrent tasks, but with this simplification comes the certainty that assuming "because it can be shared, it should" leads to more problems than approaching it from the other side — i.e. "assume no sharing, and design it in", rather than "everything is shared except that which is designed out".

Issues such as the integrity of global and static objects, and any that are implicitly operated on by unbound functions, are no longer assured.

Different operating systems handle the concept of threads differently: some implement them as user threads, which are scheduled within a process by a runtime support system; some as kernel threads, which are scheduled separately by the operating system; some offer both, such as Solaris threads and lightweight processes (LWPs). There are benefits and drawbacks to both approaches, although in terms of true pre-emptive, non-blocking, non-invasive architecture, kernel threads are preferred. In multi-processor systems implementing a shared memory architecture kernel threads can be scheduled on different CPUs.

The role of a thread class

- **A thread class is**
  - Abstract
  - A wrapper for the C API
  - A function with asynchronous execution semantics
- **Active objects have their own thread of control**
- **Embedded thread objects can be used to emulate *asynchronous functions* on passive objects**

Class for active connection server objects

```
class connection_server
  : public thread
{
    ...
private:
    virtual void main()
    {
        ... // lifecycle
    }
    ...
};
```

```
connection_server server(...);
...
server(); // run server
cout << "started" << endl;
```

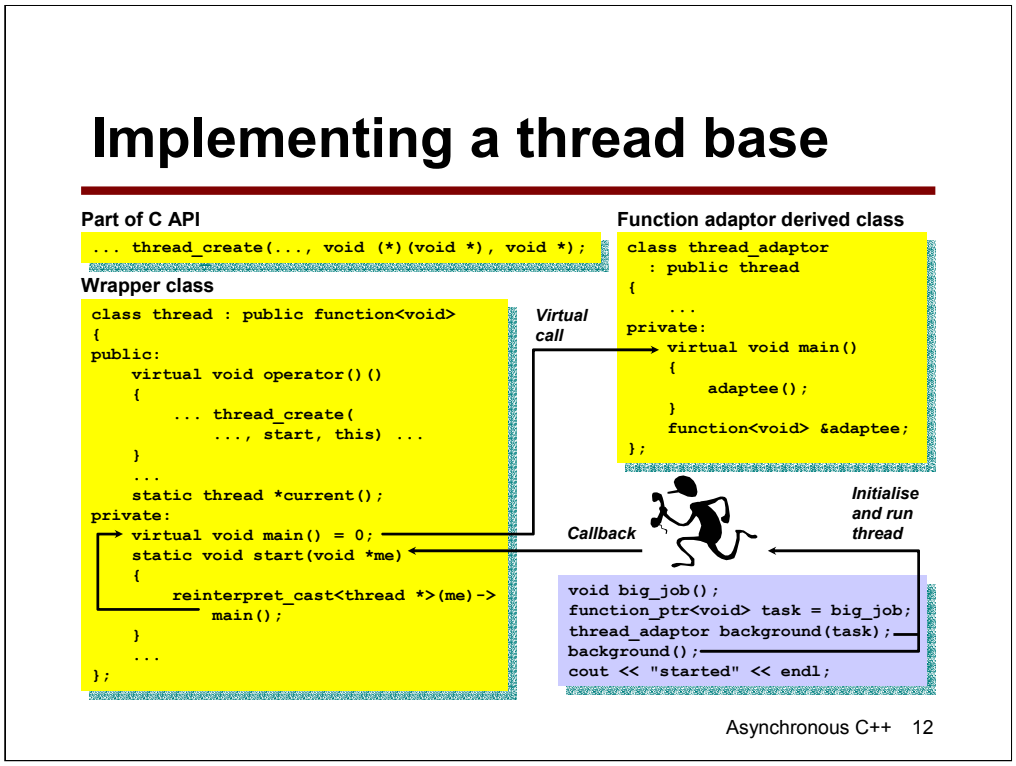
Asynchronous C++ 11

Active objects are those which own their own thread of control, either directly or indirectly. They are the *actors* of a system, initiating activity on other objects to carry out certain tasks and achieve specific aims. These collaborating objects are passive objects that are active only as the result of being called by another thread of control, hence they are reactive. They are conceptually servers where active objects are clients. Objects that have both characteristics are termed *agents*.

A thread base class should wrap up all the mechanism associated with thread handling and management, leaving the derived class free simply to define any data, its construction, provide a lifecycle, and any auxiliary functions as necessary. Additional synchronisation mechanisms, such as message queues, can be attached for building active agent objects that handle requests from other threads, as well as initiating them.

Active objects own a root of control and describe their own lifecycle. Execution and state are closely associated, so it is natural to model this both as a class and as a function.

Implicit threading behaviour is possible on some systems where features such as asynchronous I/O are implemented in the system API. Functions such as these can be associated with passive objects, such as a file handler, as asynchronous member functions, ie. they do not block the caller until they complete in the way that ordinary synchronous functions do. Explicit threading can also be used to achieve this. A similar concept is expressed by *oneway* functions in OMG's CORBA distributed object model.



C thread creation functions vary in detail, but tend to have the following features in common: resulting thread ID and success, callback function to start from, user data, and a host of options to control execution.

The thread wrapper class follows a similar form to the C callback wrapping method described earlier. In addition, most systems support thread specific data: the thread can hold data under a specific key. This can be used to associate the thread object pointer with the raw thread ID, ie. given an ID it is possible to retrieve the associated object, if any. A dummy class and singleton object (see the Singleton in *Design Patterns*) can be used to represent the original, main thread.

One variation on the typical thread creation process is the concept of a *thread fork*. This literally clones the current thread — stack, instruction pointer, etc. — at the point of the fork, but beware issues related to bytewise copying. Linux supports this feature with the `clone` call. If this is available on a system it makes the implementation of an asynchronous function — leaving aside synchronisation issues for a moment — trivial:

```
perform thread fork
if(in cloned thread)
    perform asynch task
```

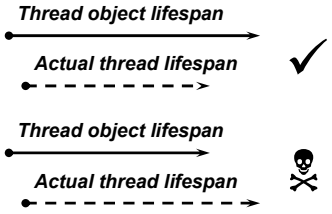
The `thread_adaptor` class illustrated above allows function objects, and by implication function pointers, to be used as the main task of a thread.

A `private virtual` function is used as an idiom to express that a function should only be provided by a derived class, but not called by it.

Traps for the unwary

- **There is a temptation to make threads auto-runnable**
  - However, execution may occur on a partially constructed object
  - Initialisation and execution are separate concepts
- **Lifetime of actual thread must be contained in lifetime of thread object**
  - Careful design of daemon threads
  - Co-ordinate destruction

```
class thread
  : public function<void>
{
    ...
protected:
    thread() : ...
    {
        ...
        (*this)();
    }
    ...
};
```



Asynchronous C++ 13

Attempting to run a thread from within a `thread` base class has unfortunate consequences. The thread can be scheduled to run before the derived constructors have completed, leaving the object incompletely initialised. An incorrect *vptr* can have disastrous effects: the `start` routine may attempt to execute a pure virtual function call on the object. It is always dangerous to call virtual functions from within a constructor and destructor; doubly so with multi-threading.

One way of looking at the solution is that it two phase initialisation. However, the idea of deferred execution — separating initialisation and main lifecycle — is closer to the truth. Decoupling like this makes `thread` objects easily rerunnable.

When a thread object reaches its destructor, it must either wait for the raw thread to complete or kill it. Whichever strategy is adopted, the point is that the underlying thread cannot be allowed to continue: it would be operating on deallocated memory.

Synchronisation primitives

- **Different types**
  - Mutexes
    - Recursive
    - Non-recursive
  - Semaphores
    - Binary
    - Counting
  - Condition variables
  - Events
- **Scope**
  - Intra-process
  - Inter-process

```
class mutex
{
public:
    ...
    void lock();
    bool try_lock();
    bool try_lock_for(interval);
    void unlock();
    ...
private:
    ...
    mutex(const mutex &);
    mutex &operator=(const mutex &);
};
```

```
mutex guard;
guard.lock();   // block to acquire
...             // critical section
guard.unlock(); // release
```

Asynchronous C++ 14

In the presence of multiple threads, synchronisation of access to data is vital. A number of primitives are typically available on a threading system.

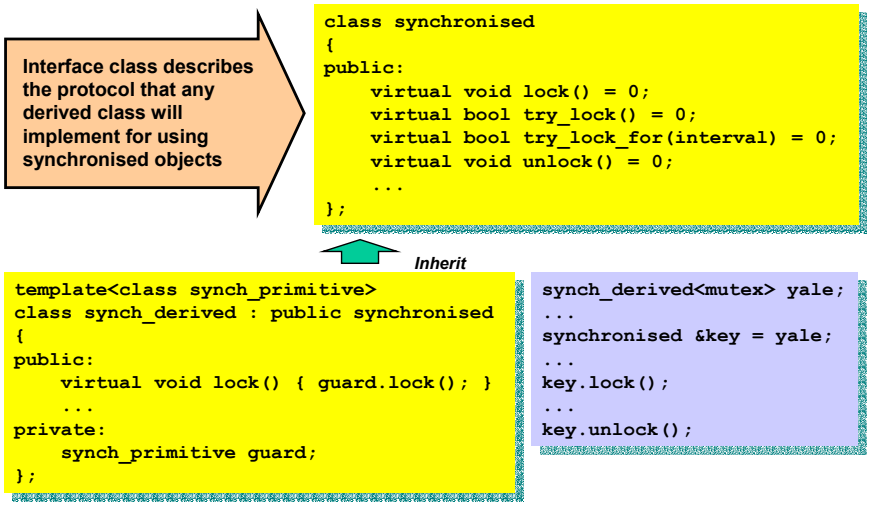
Mutexes and semaphores may be locked and unlocked, bracing a section of code termed the *critical section*. The execution of this code must appear to be executed atomically with respect to other threads, ie. although it may technically be pre-empted, it is not re-entered until completed by a given thread. This term is not to be confused with the Win32 critical section, which is simply a degenerate form of mutex.

Mutexes offer mutual exclusion in terms of a single thread, ie. the thread that locks it must be the thread that unlocks it. Binary semaphores are not quite so structured. It is system dependent as to whether or not a mutex is re-entrant, ie. whether or not a locked mutex may be relocked by the locker. Some systems, such as Solaris, offer locks that allow multiple reader / single writer access. These are more structured than the other primitives. They may be faked up using condition variables or events and mutexes, or pairs of counted semaphores. Win32 events are effectively "smart blocking flags". Condition variables are like events, but they auto-lock a mutex.

The scope or reach of these primitives may be within a single process address space, or may be system wide for synchronisation across processes.

Casual copying of mutexes, ie. copying objects that represent a unique lock, is prevented in the example above by declaring the copy operators `private` and providing them with no implementation.

Generalised locking



Asynchronous C++ 15

The mutex class introduced previously is a non-polymorphic class that encapsulates a system primitive and portion of C API. Clearly, many of the synchronisation primitives support common operations, and hence interfaces. In some cases a more general interface is useful.

The `synchronised` interface class may be used explicitly as a base class for a class supporting synchronisation. More usefully for primitives, which are best left as non-polymorphic, an adaptor class can be used to provide the interface — run-time polymorphism — on behalf of anything supporting the correctly named functions — sometimes known as compile time polymorphism. It easier to take a non-polymorphic class and adapt it to be polymorphic, than it is do it the other way around: the overhead and semantics of polymorphism can only introduced to a class, not removed.

It is also possible to create `read_write_synchronised` and `waitable` interfaces for hierarchies supporting reader/writer locking and simple event synchronisation. For instance, `thread` could support the `waitable` interface for synchronising on its termination.

Resource locking

- **Explicit lock/unlock is**
  - Tedious
  - Error prone
  - Open to abuse
  - Not exception safe
- **Natural pairing can be encapsulated as a control object**

```
template<class synch = synchronised>
class lock
{
public:
    lock(synch &key) : lockee(key)
        { lockee.lock(); }
    ~lock() { lockee.unlock(); }
private:
    synch &lockee;
};
```

```
mutex plain;
synch_derived<mutex> general;
```

```
{
    lock<mutex> region(plain);
    ...
}
```

Critical region

```
{
    lock<> region(general);
    ...
}
```

Asynchronous C++ 16

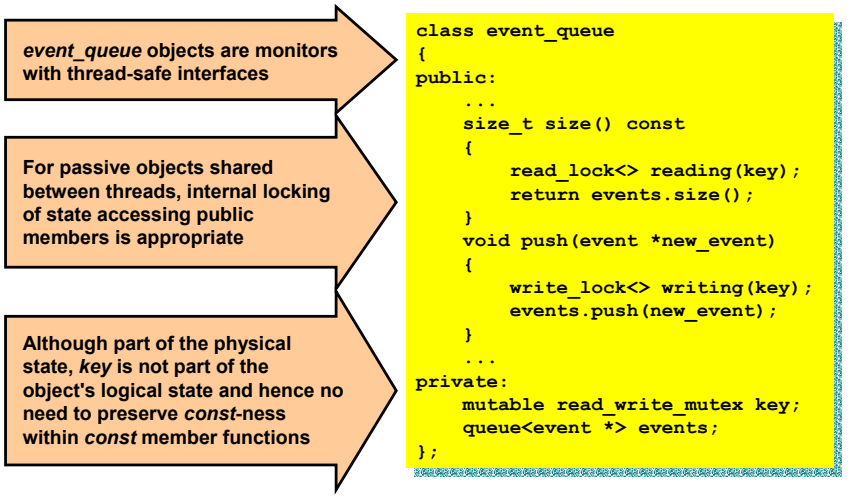
Although encapsulated, there are a number of problems waiting to happen with explicit `lock`/`unlock` code. Programmer error, such as premature return from a function, can leave a synchronised object in a locked state. Exceptions thrown in the critical section will bypass the call to `unlock` as the thrown exception unwinds the stack. It is also quite tedious and low level.

The control structure pairing can be encapsulated in a class that executes `lock` in the constructor and `unlock` in the destructor. Should a function be left prematurely, for whatever reason, the destructors are called as the stack is unwound, and hence the lock is freed even in exceptional circumstances.

This is known as the *resource-acquisition-is-initialisation* idiom. It simplifies the code and makes it more robust. The encapsulation of control in this case also ties the programmer's critical section in with the linguistic concepts of scope and lifetime. The critical region is thus explicitly delimited by the object's lifetime.

The solution presented also allows lock objects to be applied to an object either based on run-time or compile time polymorphism. A template parameter is used to specify what type is to be locked, and it has been provided with a default of `synchronised` for the polymorphic case. When the default is used, empty <> are still required with the type name.

Thread-safe interfaces



Passive objects that are shared between threads require locking when their interface is accessed. This can be done by negotiation through an additional separate synchronisation object, or closer to the class itself.

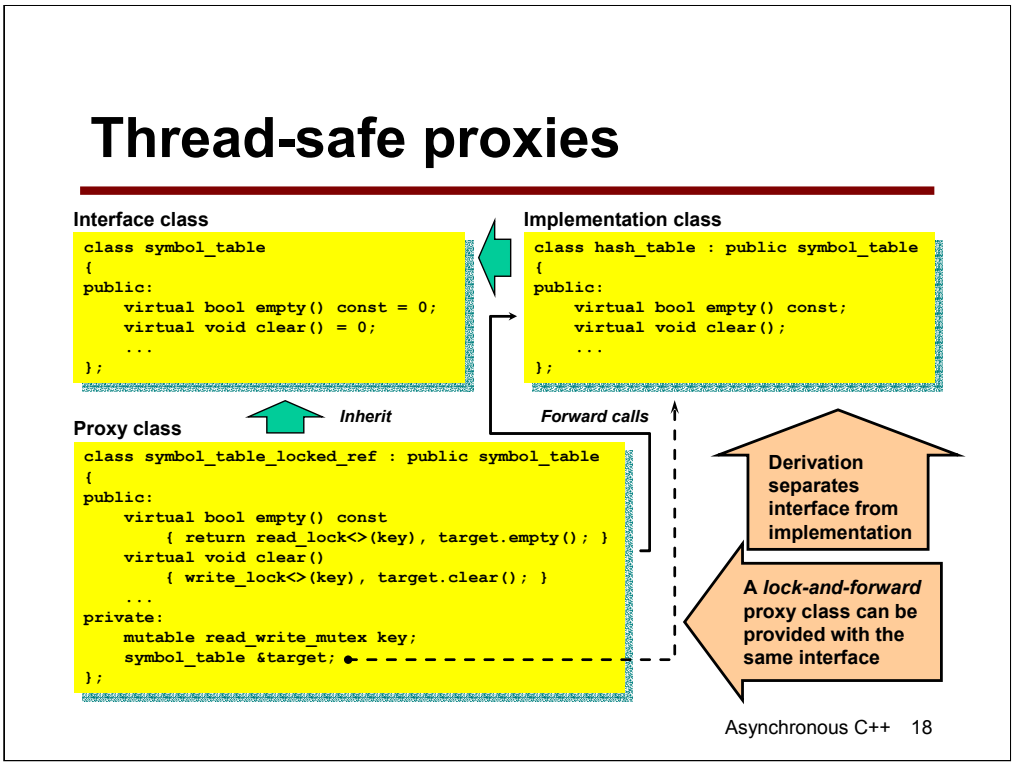
For objects that are known to be shared, it makes sense to provide the locking as part of the automatic behaviour, ie. when a public function is called it locks an internal synchronisation object. Objects of such a class are said to be *monitors* or, in Ada 95 parlance, they are *protected*. These simplify programming from the class users point of view.

The class author must be careful not to call other public member functions from within the class if the synchronisation object is not re-entrant.

The class above assumes read/write synchronisation classes and control objects, using the multiple reader capability to implement `const` member functions and single writer for modifier functions. `event_queue` forwards functions on to a `queue`, another class introduced from the STL.

The locking mechanism involves a change of state on the lock object, but this is part of the administration of `event_queue` rather than a logical property implied by the public interface, which should be `const`-correct. The `mutable` specifier informs both the compiler and reader that it is safe and expected that key may be modified within a `const` member function.

It is possible to template `event_queue` on a synchronisation primitive, and provide a `null_synchronisation` pass-through class for use in single threaded systems.



An alternative approach separates out the class of interest into an interface and an implementation. This is suitable where a *usage type* might support *creation types*. One or more of these implementation classes provide the actual class state and logic. A proxy class supporting the same usage interface provides the locking, and forwards the requests onto the actual object. The intent of the Proxy pattern is described in *Design Patterns* as

Provide a surrogate or place holder for another object to control access to it.

This technique can be used with existing unsynchronised classes, but also easily allows for the possibility of alternative implementations, or even chains of proxies that enhance the functionality (similar to the Decorator pattern). By holding a pointer to the interface class, users of this class need not be concerned with either the details of synchronisation or any impact in migrating the code to a single threaded system — the interface remains the same, and the object would be accessed directly.

Another way of looking at this form of proxy is as a "smart *lvalue*".

Note use of temporary unnamed variables and the normal comma operator in the synchronisation expressions. Such temporary variables are guaranteed to have their destructor called at the end of the full expression, ie. statement in this case, and hence automatically unlock.

Externally locked interfaces

- **Class users negotiate mutual exclusion explicitly**
- **The target class must be synchronisable**
  - By inheritance, requiring that the class is designed this way
  - Or by aggregation, with target class access using *operator->*

synchronised as mix-in

```
class symbol_table
  : public synchronised
{
public:
    ...
    virtual void lock();
    virtual void unlock();
    ...
};
```

Any class may be wrapped

```
template<class adaptee, class synch>
class synch_adaptor
  : public synch_derived<synch>
{
public:
    ...
    adaptee *operator->() const
        { return target; }
private:
    adaptee *target;
};
```


Alternatively classes may chose to require that users negotiate the use of their interface explicitly. One technique is to derive from the `synchronised` interface and implement a synchronisation strategy of choice. Alternative derivations include inheriting from a concrete synchronisation class that is either explicitly named or provided via a template parameter.

However, not all classes will have been, or even should be, designed as synchronisable in advance on the off chance that they might be used in a multi-threaded environment, shared between threads, and thus require synchronisation as an intrinsic property. It would be an unnecessary overhead in a number of cases, and introduce virtual functions into otherwise non-polymorphic classes. This would have a quantifiable impact on both their semantics and their footprint.

The `synch_adaptor` class provides a proxy approach that acts as a simpleton smart pointer: locking is available via member functions acquired through inheritance, but the actual target is accessed via the overloaded class member access operator, `operator->`. The `synch_adaptor` is an *object adaptor* as opposed to a *class adaptor* like `synch_derived`.

Smart pointers for locking

- **Main usage and synchronisation interfaces may be part of the same interface, but they are disjoint**
- **Safe access can be automated using a smart pointer to automatically lock the target on access**

```
symbol_table symbols;
synch_ptr<symbol_table *> symbols_ptr = &symbols;
...
symbols_ptr->insert("Thread-safety");
```

Lock required

Lock no longer required

```
stack<string> raw_stack;
typedef synch_adaptor<stack<string>, mutex> synch_stack;
synch_stack safe_stack(&raw_stack);
synch_ptr<synch_stack &> stack_ptr(safe_stack);
...
stack_ptr->put("Thread-safety first");
```

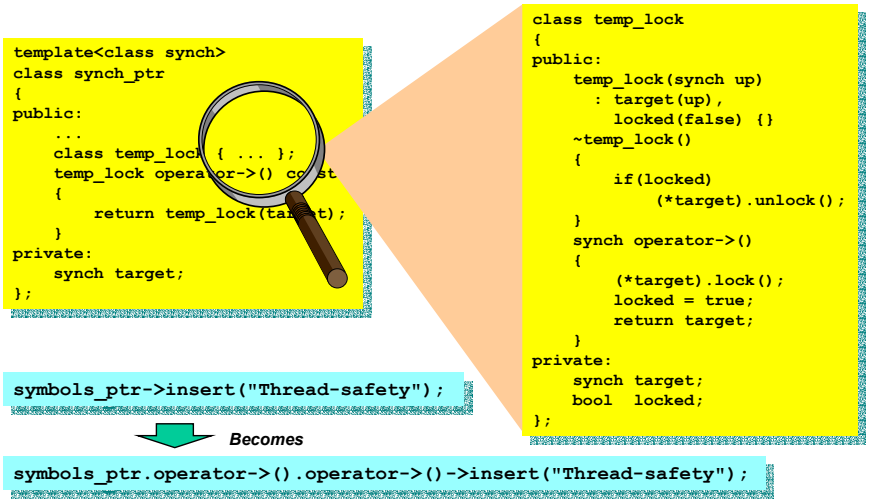
Lock required

Lock no longer required

Externally synchronisability may loosen the coupling between synchronisation and usage interface, but the cohesion and simplicity of the internally locking class approach is also missing. Externally locked classes may be synchronised with the locking classes discussed previously. For long transactions that are to be treated atomically this is ideal. However, this can make simple function calls tedious, prefixing them all with temporary statement duration locks.

An alternative is to provide a smart pointer that provides function call locking somehow via `operator->`. Smart pointers are another variant of Proxy.

Locking pointer implementation



The `synch_ptr` overloads `operator->` to return a temporary object that will perform the locking. This too provides an `operator->`. Calls to `operator->` are automatically chained by the compiler until a raw pointer type is returned. In `temp_lock`'s `operator->` the lock is applied and in its destructor, called at the end of a full expression, it is released.

The locking behaviour can easily be disabled for single threaded applications, requiring only modifications to `temp_lock` to make it a pass-through object, a recompile, and a relink.

Programmers should be careful about attempting to access the same object twice in a statement using `synch_ptrs`: this will cause deadlock if the synchronisation strategy is not re-entrant.

The class parameter should either be a pointer, a smart pointer or a reference for a type supporting `operator*` that yields a reference to a synchronised interface. For the above code to work with `synch_adaptor`, `synch_adaptor` must provide a dereference operator that implements the identity operation:

```
synch_adaptor &operator*()
{
    return *this;
}
```

An alternative solution, not discussed here, is to use partial specialisation for `synch_ptr` on references.

Reference counted pointers

- **Reference counting techniques can be used to automatically manage deletion**
- **A counting smart pointer holds pointers to**
  - The managed object
  - A shared counter

```
template<typename type> class counted_ptr
{
public:
    explicit counted_ptr(type *ptr_to_own = 0);
    template<typename other>
      counted_ptr(const counted_ptr<other> &);
    template<typename other>
      counted_ptr &operator=(const counted_ptr<other> &);
    ~counted_ptr() { release(); }
    type *operator->() const { return owned; }
    type &operator*() const { return *owned; }
private:
    void acquire(type *new_owned);
    void release();
    type *owned;
    ... // shared counter
};
```


Reference counting puts the responsibility for management of object ownership into the association, ie. smart pointers rather than the objects that use them are responsible for the deletion of the target object. Reference counted pointers keep a shared usage count of the object pointed to, updating its value when they are copied or destroyed, hence counted pointers are exception safe. On falling to zero the target object is effectively unused, and is deleted.

Reference counting can be overt or hidden. A common hidden idiom is to implement string copying in terms of a shared representation. The actual copy is only performed if and when any modifications are applied to the string. This is a *copy-on-write* strategy.

The example above is overt, and requires the users to use `counted_ptr` in their code rather than raw pointers:

```
counted_ptr<data> data_ptr(new data);
```

The `explicit` qualifier is used to prevent implicit copies and temporaries from being generated by the compiler.

Member function templates allow a new form of generalisation for smart pointers: a smart pointer to a derived class can be assigned to a smart pointer to a base class. The compiler performs the necessary type deduction:

```
counted_ptr<derived> leaf(new derived);
counted_ptr<base> root = leaf;
```

Thread-safe counting

No guarantee that decrement, dereference and test is atomic

Provide a class that performs increment and decrement atomically

Typical implementation

```
template<typename type>
class counted_ptr
{
    ...
    void release()
    {
        if(count && !--*count)
        {
            delete count;
            delete owned;
        }
    }
    type   *owned;
    size_t *count;
};
```

Thread-safe version

```
template<typename type>
class counted_ptr
{
    ...
    void release()
    {
        if(count && !count->down())
        {
            delete count;
            delete owned;
        }
    }
    type               *owned;
    threadsafe_counter *count;
};
```

Asynchronous C++ 23

Here be dragons! A typical reference counting implementation does not take atomicity into account. It is possible that a race condition could occur resulting in either a memory leak or memory trampling. This tends to be particularly insidious with hidden reference counting.

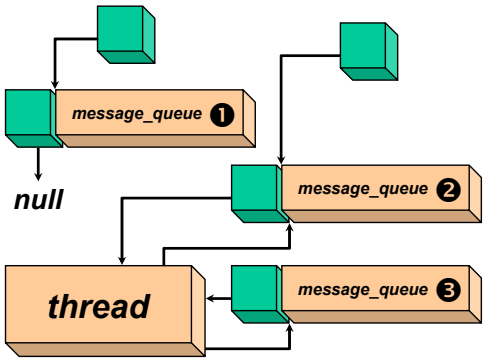
It is tempting to attempt to use one of the system synchronisation primitives for this count, but this has a significant overhead for such a lightweight class. It is also likely that if counted pointers are used extensively within an application, the system will run out of resources! It would also be inappropriate to reuse target object's synchronisation interface, assuming it supports one, as the synchronisation is on the count and not on the target object. Again, assuming it worked there would be a performance penalty to pay.

The most appropriate technique is known as *lock-free* synchronisation. All that is required is an atomic or interlocked increment and decrement operation, no locking is required. This can be wrapped up in a class, `threadsafe_counter`. If no such primitive operations are offered on a platform, it might be possible to use Dekker's or Peterson's algorithm, which only assume that reads and writes are atomic, to implement one.

Thread owned heap objects

- **The lifetime of an object can be tied to the lifetime of a given thread**
  - Non-intrusive base class technique
    - No virtual functions
    - No data members
  - Need to guarantee correct destruction

```
class message_queue
  : public thread::owned<message_queue>
{
    ...
};
```



Asynchronous C++ 24

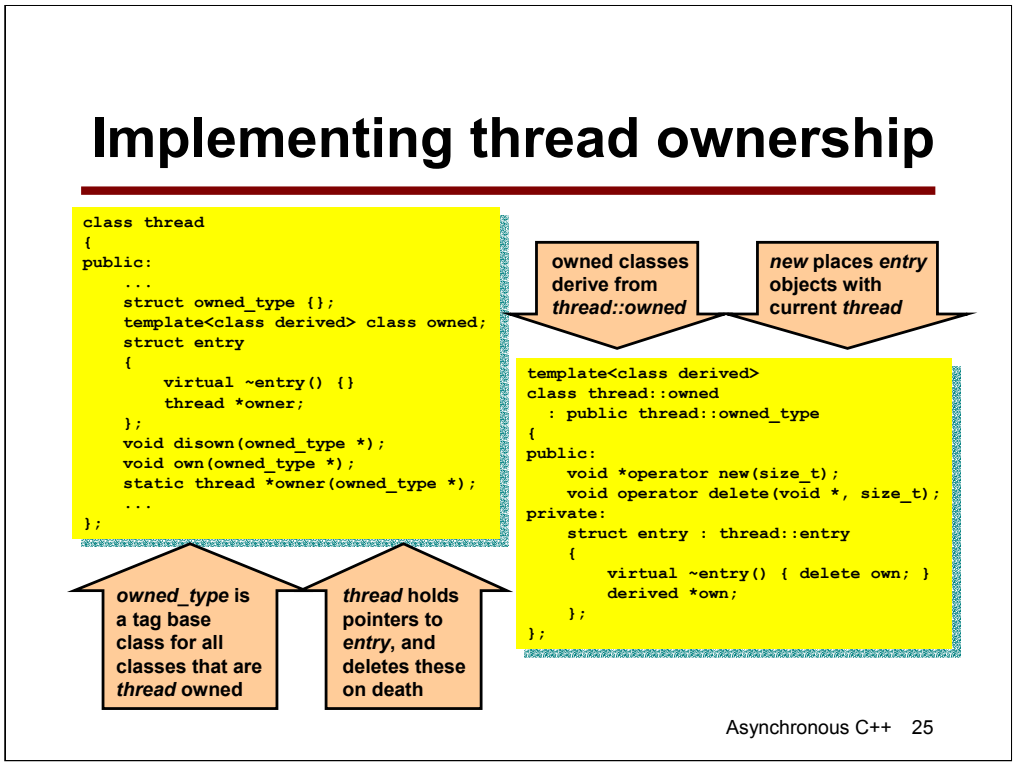
In a multi-threaded system, the lifetimes of many heap objects are bound to the thread that created them. It is possible to implement a non-intrusive generational garbage collector where the 'generation' is defined as ending at the termination of the thread's lifecycle, i.e. after `thread::main`.

Any such mechanism must ensure that objects can be deleted in advance of thread termination, without any ill effects, but must also ensure that all remaining allocated objects are deallocated with destructors called correctly.

This can be achieved with a base class, but should be done without changing the footprint of the class or introducing any polymorphism. Such classes derive from a place holder class that provides customised `new` and `delete` operators. The `thread::owned` class is templated on the derived class so that it is aware of the exact pointers types that must be maintained.

When asked for allocation these operators allocate extra memory that precedes the actual object pointer returned. This extra memory is used to indirectly point to the owning thread, and the owning thread is also aware of all the objects that it owns, such as object ❷. If an object is dissociated from a thread this effectively points to null, as in the case of object ❶. This is also how objects may be disowned and reowned by other threads. Remaining and forgotten objects are collected at thread end, such as ❸.

The effect of this approach is to implement bidirectional pointers. The `thread` class needs extension to offer all of this. The support is relatively light weight and, if implemented correctly, has no overhead if unused.



The `thread::entry` class represents the link that the `thread` object manages: it holds pointers to objects of this type, and on deleting the object — either at thread death or explicitly — the `thread` object deletes this link object. In the `thread::owned` class the forward pointer of correct type is added to a nested derived `entry` class. The extra allocated space for objects derived from `thread::owned` also points to the `entry` object. This double despatch mechanism, based on a variant of the Visitor pattern, is how correct type safe deallocation is achieved without relying on virtual destructors.

Both the `thread::owned` and the `thread::owned_type` classes are empty, ie. devoid of any data members or `virtual` functions. Although they have non-zero size when free standing, they can be implemented to have an apparently zero size when used as base classes. They can either be given addresses in any padding used to align other members, or offset halfway through a primitive data member, ie. a valid address that is not the address of another valid object — it is not possible to have half an object! As empty objects have no contents there is no fear of overwriting another genuine data member.

Note that the `thread::owned` template class is forward declared within the `thread` class, but is actually defined outside it.

Many of the types presented have been presented as unencapsulated `structs` for brevity. In truth, a careful balance of classes, privacy, friendship and safe constructors is the best approach.

Mapping objects into memory

- **Objects can be placed at specified addresses**

```
const volatile void *base = ...;
const volatile input_struct &in =
    *reinterpret_cast<const volatile input_struct *>(base);
```

- **Where more complex objects are used**
  - A placement form of *new* ensures correct construction
  - Destruction must be handled explicitly

```
void *operator new(size_t, void *place_at) throw()
{
    return place_at;
}
```

```
volatile results_cache *cache = new(base) results_cache;
...
cache->~results_cache();
```


Simple objects mapped at a given location in memory, such as memory mapped I/O, screen mapping, OS or run-time support structures, etc. can simply be taken and recast from a generic address to an appropriate type. It is important that the expected layout is matched exactly.

In the first example above, a reference is used in preference to a pointer. This binds `in` permanently to that location, permitting it to be used as a normal variable. `in`'s full *cv*-qualification specifies that it is a read-only variable whose value may change asynchronously — thus it is not `const` in the traditional sense of the word. It is important that the compiler does not aggressively optimise access to its value based on a strictly sequential execution model, hence `volatile`.

More complex objects may have important initialisation and finalisation semantics, such as the setting and clearing of control words on construction and destruction. The `new` operator provides allocation at some address and then calls the constructor. To provide a specific address or strategy, `operator new` may be overloaded and provided with extra arguments. In the case above, available in the standard library, the argument is simply the address for allocation — a remarkably trivial allocation operator...

Although placement forms are now possible for operator `delete`, these are used for recovering from failed constructors and cannot be called explicitly. To correctly destroy a placed object, its destructor must be called explicitly, as shown above.

Shared memory issues

- **Binary compatibility**
  - Mapped structures must have the same size and layout
- **Process compatibility**
  - If shared memory is mapped to same address in all processes, any pointers should be within shared region
  - Otherwise, offsets from a base address should be used
  - Any other descriptors used must be unique across system
- **Unless processes executed from the same image**
  - No virtual functions or virtual base classes
  - No use of run-time type information
- **Access must be synchronised**

Shared memory and memory mapped files offer forms of inter-process communication and persistence. Memory may be shared between processes and, where distinct from shared memory, memory mapped files also allow files to be treated as in-core data structures. If the issues above are respected and addressed, it is possible to use this as a medium for session persistence.

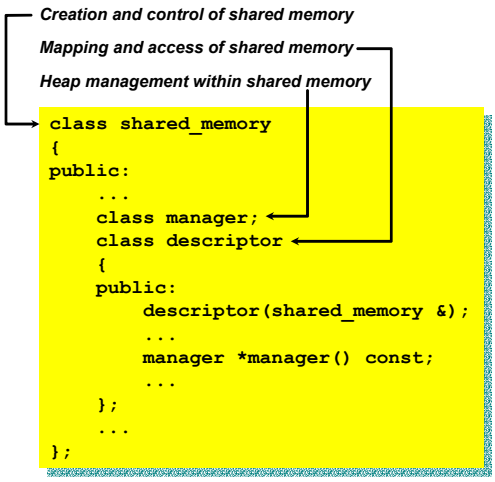
Flat data structures without pointers seem to be the safest bet. However, if certain constraints can be met and conditions guaranteed, pointers to both data and functions can be introduced. The simplest case is if shared memory is mapped to the same base address in all sharing processes, then internal pointers become possible. If executed from the same image, function pointers and hence polymorphism become a possibility.

If access to certain data members is guaranteed to be from a particular process, then in it may contain pointers back into that process' address space. This promise can be checked using a simple security mechanism: access to the data is by member function only, and this checks the PID of the caller against a PID stored within the object.

These constraints are part of the reason for preferring non-polymorphic system primitives over polymorphic ones, relying on adaptors to build the latter from the former. Synchronisation primitives that are simply wrappers for OS resource descriptors which are unique across the system can be placed in shared memory and used by all attached processes.

Managing shared allocation

- **Use a class to encapsulate mapping of memory**
- **Provide a simple heap manager that sits at the base of the shared region**
- **Overload *operator new* to simplify allocation**



There are many components to encapsulating shared memory. The physical concept of the shared segment is separate to actually attaching it to a process, in the same way that an ordinary file on a file system is different to holding an open descriptor to it for content access.

These roles are separated in the classes above: the outer `shared_memory` class is the one that describes the concept, the creation, and the removal of shared memory from a system. The nested `descriptor` class is responsible for its mapping. In practice it would have many constructors to support the many options available.

For convenience, a simple 'heap' manager can be used for allocation of objects into the shared memory region. The nested `manager` class covers this responsibility, and would normally reside near the base or end of the shared region. A call to `descriptor::manager` function returns the address of the `manager` object, or null if one is not being used.

The `new` operator can be overloaded to allocate from the shared memory heap manager rather than from the ordinary heap.

Shared heap management

```
class shared_memory::manager
{
public:
    ...
    template<typename type> void destroy(type *);
    template<typename type> type *find(const char *) const;
    ...
};

typedef shared_memory::manager shared;
void *operator new(size_t, shared *);
void *operator new(size_t, shared *, const char *name);
void *operator new(size_t, shared *, size_t offset);
void *operator new(size_t, shared *, const char *, size_t);
```

Calls destructor and deallocates

Finds a named object

Abbreviation

Create named object

Create object at a given offset

```
shared_memory::descriptor region(...);
results_cache *cache =
    new(region.manager(), "cache") results_cache(initial);
mutex *key = region.manager()->find<mutex>("lock");
...
region.manager()->destroy(cache);
```


Objects can be allocated freely and anonymously within the heap space. More useful is the concept of object naming to allow objects to be created by one process, and looked up by another without having to explicitly know the offset. The template member function `find` is used for this. Alternatively objects may be allocated at fixed offsets, and a corresponding `find` function can be provided for this.

Because placement `delete` is not a possibility, objects are destroyed explicitly with respect to the shared memory manager using `destroy`. This template member function ensures, through type deduction, that the object is correctly destructed.

The issues for overloading `operator new[]` in shared memory are a little more involved and are not covered here, suffice to say that it cannot be implemented without significant system dependent assumptions. Alternative methods must be employed to allocate shared arrays.

Summary

- **Functions as objects**
  - Callback idioms
  - Threads as asynchronous functions
- **Adaptors**
  - Adding polymorphic or thread-safe behaviour to classes
- **Proxies and smart pointers**
  - Control objects for synchronisation
  - Thread-safe object access
  - Thread-safe reference counting
- **Application of new and advanced language features**

A number of techniques — old and new — have been applied to the construction of asynchronous frameworks in C++. Patterns, idioms, and newer language features have been used to tackle many — although clearly not all — features of such systems. This talk has given an introduction and a significant foundation for further detailed design and implementation in this and related areas.