Using a *guarded<M>* metaclass

Today, by hand

```
struct MyData {
  vector<int>& v()  { assert( m_.is_held() );  return v_; }
  Widget*&     w()  { assert( m_.is_held() );  return w_; }
  void         lock()        { m_.lock(); }
  bool         try_lock()    { return m_.try_lock(); }
  void         unlock()      { m_.unlock(); }
private:
  vector<int>  v_;
  Widget*      w_;
  mutex_type   m_;
};
```

Today, with

```
struct MyData {
  GUARDED_WITH(
  GUARDED_MEMBE
  GUARDED_MEMBE
};
```

P0707R4

```
class(guarded<mutex_type>) MyData {
  vector<int> v;
  Widget* w;
};
```

macros? **metaclasses!**

If we are serious about getting rid of the remaining reasonable uses of macros, we will love:

modules
constexpr
reflection+generation (metaclasses)

Yes, this is about letting us write more **patterns** as library code

49

Example

Concurrency: `active`

But first, how we teach it today…

50

Learn to love threads — on your own terms

- Raw threads are what we have, but are too low-level:
  - Thread mainline has no guard rails: Can be any old undisciplined spaghetti code.
  - Communication is via shared state by default: Fun with mutexes, lock orders, etc.
- What we teach to do by hand (covers common cases, not all uses):
  - **Make the thread mainline a message pump: sequential ⇒ no races among "callee" msgs.**
    - Options: "Thread" can be a coroutine, a series of tasks on a pool, etc. — as-if sequential.
  - **Communicate by sending messages: queued requests, well-formatted (hint: well-typed!), typically with copies of state ⇒ no races between caller and asynchronous callee.**
    - Options: "Queue" can be a priority queue, use multiple channels, etc.
- But "CSP Pattern" is manual: How can we automate these best practices?
  - **Reality: ISO C++ will never add *active* as a language extension (unlike actor languages).**
  - Not directly expressing what we mean makes code harder to write, debug, and maintain.

51

Active objects: Overview

- An active object encapsulates a thread + message_queue.
  - Each object is an asynchronous worker whose mainline is a message pump.
  - Member function calls become async messages (⇒ strongly and statically typed).
- Example of our goal:

```
class worker {              // (coded specially)
public:
   future<int> func() { … }
};

// in calling code, using an active object
worker w;
auto result = w.func();     // nonblocking
… more work …               // … concurrent…
use(result.get());          // may block
```

Goal: Direct expression of intent

Covers many classes of concurrency:

Long-running workers
(physics thread, GUI thread, …)

Decoupled independent work
(background save, pipeline stages, …)

Encapsulated resources
(async I/O streams, …)

52

Attach thread lifetime to object lifetime…

- … By attaching the constructor and destructor: *← Yes, RAII for threads*
  - Constructor: Starts thread and message pump.
  - Destructor: Sends “done” signal, then **blocks** and waits for queue to drain.
- Lets us exploit existing rich language semantics to **control thread lifetimes**:

```
class active1 {
   active2 inner;        // by-value member, by-value lifetime:
   …                     // nested ~~object~~thread bound to enclosing ~~object~~thread
};

void some_function() {
   active1 a;            // stack-based object, stack-based lifetime:
   …                     // ~~object~~thread bound to local scope
} // waits for a and a.inner to complete
```

53

An “Active” helper

This sample shows the basic case: thread + FIFO queue

Variations (pools, channels, …) can be done similarly

- Encapsulates the core mechanics.

```
class Active {
public:
   using Message = function<void()>;
   Active() : thd([=]{ while (!done) mq.receive()(); }) { } // mainline: dispatch loop
   ~Active() { Send([=]{ done = true; }); thd.join(); }     // wait for queue to drain
   void Send( Message m ) { mq.send(m); }                   // enqueue a message
private:
   bool done = false;
   message_queue< Message > mq;
   thread thd;
};
```

54

Pop Quiz: What's the difference?

Mutex locks

```
class log {
  fstream f;
  mutex m;
public:
  void println( /*…*/ ) {
    lock_guard<mutex> hold(m);
    f << /*…*/ << endl;
  }
};
```

Active objects

classic "space vs. time" (concurrency)

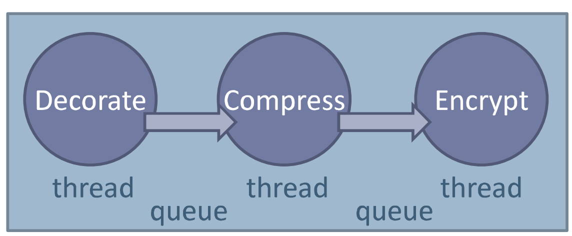
```
class log {
  fstream f;
  Active a;
public:
  void println( /*…*/ ) { a.Send( [=] {
    f << /*…*/ << endl;
  } ); }
};
```

Same calling code either way: `mylog.println( "Hello %1%", name );`

Group exercise (1 of 2)

- Implement a concurrent version of the following sequential code:

```
void SendPackets( Buffers& bufs ) {
  for( auto& b : bufs ) {
    Decorate( b );
    Compress( b );
    Encrypt( b );
  }
}
```



- Assume that:
  - Decorate(x) must end before Decorate(x+1) begins, same for Compress and Encrypt.
  - Decorate(x), Compress(x), and Encrypt(x) must execute in that order.
  - Decorate(x), Compress(y), and Encrypt(z) have no side effects w.r.t. each other and can run concurrently.

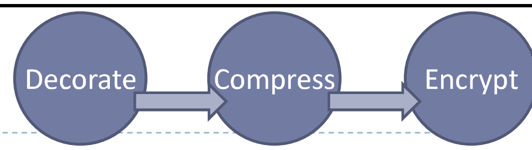
Pipeline stage

- Each stage does just one part.

```
class Stage {
public:
   Stage( function<void(Buffer*)> w ) : work{w} { }
   void Process( Buffer* b ) { a.Send( [=] {
      work( b );
   } ); }
private:
   function<void(Buffer*)> work;
   Active a;                          // NB: remember to put this member last!
};
```

57

Setting up the pipeline



- Three concurrent stages ("communicating sequential processes," anyone?):

```
void SendPackets( Buffers& bufs ) {
   Stage encryptor   ( []   (Buffer* b) { Encrypt(b); } );
   Stage compressor( [&](Buffer* b) { Compress(b);
                                                  encryptor.Process(b); } );
   Stage decorator   ( [&](Buffer* b) { Decorate(b);
                                                  compressor.Process(b); } );
   for( auto& b : bufs ) {
      decorator.Process( &b );
   }
} // automatically blocks waiting for pipeline to finish
```

Q: How is the pipeline destroyed? Be specific.

58

Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: **Do you see the problem?**

```
public void SendPackets( Buffers bufs ) {
  Stage encryptor = null;
  Stage compressor = null;
  Stage decorator = null;
  try {
    encryptor = new Stage( new EncryptRunnable() );
    compressor = new Stage( new CompressRunnable( encryptor ) );
    decorator = new Stage( new DecorateRunnable( compressor ) );
    for( b : bufs ) {
      decorator.Process( b );
    }
  } finally {
    if( encryptor != null )    encryptor.dispose();     // automatically block
    if( compressor != null )   compressor.dispose();    // waiting for the
    if( decorator != null )    decorator.dispose();     // pipeline to finish
  }
}
```

59

Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: **Do you see the problem?**

```
public void SendPackets( Buffers bufs ) {
  Stage encryptor = null;
  Stage compressor = null;
  Stage decorator = null;
  try {
    encryptor = new Stage( new EncryptRunnable() );
    compressor = new Stage( new CompressRunnable( encryptor ) );
    decorator = new Stage( new DecorateRunnable( compressor ) );
    for( b : bufs ) {
      decorator.Process( b );
    }
  } finally {
    if( encryptor != null )    encryptor.dispose();     // automatically block
    if( compressor != null )   compressor.dispose();    // waiting for the
    if( decorator != null )    decorator.dispose();     // pipeline to finish
  }
}
```

60

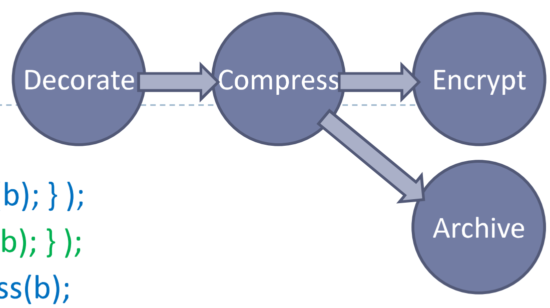
Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: Corrected (but still manual)

```
public void SendPackets( Buffers bufs ) {
   Stage encryptor = null;
   Stage compressor = null;
   Stage decorator = null;
   try {
      encryptor = new Stage( new EncryptRunnable() );
      compressor = new Stage( new CompressRunnable( encryptor ) );
      decorator = new Stage( new DecorateRunnable( compressor ) );
      for( b : bufs ) {
         decorator.Process( b );
      }
   } finally {
      if( decorator != null )     decorator.dispose();        // automatically block
      if( compressor != null )  compressor.dispose();       // waiting for the
      if( encryptor != null )     encryptor.dispose();        // pipeline to finish
   }
}
```

61

And more flexibility…




```
void SendPackets( Buffers& bufs ) {
   Stage encryptor    ( []   (Buffer* b) { Encrypt(b); } );
   Stage archiver     ( []   (Buffer* b) { Archive(b); } );
   Stage compressor ( [&](Buffer* b) { Compress(b);
                                         if (b->something()) encryptor.Process(b);
                                         else archiver.Process(b);
                                       } );
   Stage decorator    ( [&](Buffer* b) { Decorate(b);
                                         compressor.Process(b); );
                                       }
   for( auto b : bufs ) {
      decorator.Process( &b );
   }
} // automatically blocks waiting for pipeline to finish
```

62

Using an *active* metaclass

Today, by hand

```
class A {
public:
   Stage( function<void(Buffer*)> w )
      : work{w} { }
   void Process( Buffer* b ) { a.Send( [=] {
       work( b );
   } ); }
private:
   function<void(Buffer*)> work;
   Active a;        // remember to put this last!
};
```

P0707R4

```
class(active) Stage {
public:
   Stage( function<void(Buffer*)> w )
      : work{w} { }
   void Process( Buffer* b ) {
       work( b );
   }
private:
   function<void(Buffer*)> work;
};
```

63

Using an *active* metaclass (2)

Today, by hand

```
class log {
   fstream f;
   Active a;        // remember to put this last!
public:
   void println( /*…*/ ) { a.Send( [=] {
      f << /*…*/ << endl;
   } ); }
};
```

P0707R4

```
class(active) log {
   fstream f;
public:
   void println( /*…*/ )
      f << /*…*/ << endl;
   }
};
```

64

```
template<typename T>
constexpr void async(T source) {
    for... (auto o : source.member_variables()) {
        __generate o;
    }

    __generate class { Active __a; };

    for... (auto f : source.member_functions()) {
        auto ret = f.return_type();
        if (!f.is_constructor() && !f.is_destructor())
            f.make_private();

            __generate class {
                void idexpr(f,"_")(__inject(f.paramete
                    auto val = this->idexpr(f)(args...
                    __p->set_value( val );
                    delete __p;
                }
            };

            __generate struct {
                auto idexpr(f,"_")(__inject(f.paramete
                    auto p = new std::promise<typename
                    auto fut = p->get_future();
```

```
class Test {
    Active __a;
public:
    Test()     {
    }
    void h_(int i, std::promise<i
    auto h_(int i);
private:
    int h(int i)     {
        return i + 1;
    }
}
Compiler returned: 0
```

65

Using an *active* metaclass (3)

Today, by hand

```
class some_service {
  data stuff;
  Active a;          // remember to put this last!
public:
  future<double> GetResult() {
    promise<double> p;
    future<double> ret = p.get_future();
    a.Send( [p = move(p)]{
      this->DoGetResult( move(p) ); } );
    return ret;
  }
private:
  void DoGetResult( promise<double>&& p ) {
    p.set_value( result );
  }
};
```

P0707R4

```
class(active) some_service {
  data stuff;
public:
  double GetResult() {
    return result;
  }
};
```

Pop quiz:
What's this?

A: "Thread local storage"
But by construction &
with ordinary allocation

66