This Tutorial is about ...

➢ Writing aspect-oriented code with pure C++
  - basic implementation techniques using C++ idioms
  - limitations of the pure C++ approach

➢ Programming with AspectC++
  - language concepts, implementation, tool support
  - this is an AspectC++ tutorial

➢ Programming languages and concepts
  - no coverage of other AOSD topics like analysis or design
Introduction

Aspect-Oriented Programming

➢ AOP is about modularizing crosscutting concerns

- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

---

Why AOP with C++?

➢ Widely accepted benefits from using AOP
  • Avoidance of code redundancy, better reusability, maintainability, configurability, the code better reflects the design, ...

➢ Enormous existing C++ code base
  • Maintainance: extensions are often crosscutting

➢ Millions of programmers use C++
  • For many domains C++ is the adequate language
  • They want to benefit from AOP (as Java programmers do)

➢ How can the AOP community help?
  • Part II: Describe how to apply AOP with built-in mechanisms
  • Part III-V: Provide special language mechanisms for AOP

---

Scenario: A Queue utility class

```
namespace util {
    class Item {
        friend class Queue;
        Item* next;
        public:
        Item() : next(0) {};
    }
    class Queue {
        Item* first;
        Item* last;
        public:
        Queue() : first(0), last(0) {};
        void enqueue(Item* item) {
            printf(" > Queue::enqueue()\n");
            if (last) {
                last->next = item;
                last = item;
            } else
                first = last = item;
            printf(" < Queue::enqueue()\n");
        }
    }
    // class Queue
    namespace util

    Item* dequeue() {
        printf(" > Queue::dequeue()\n");
        if (first == last) {
            first = last = 0;
            return 0;
        } else
            first = first->next;
        printf(" < Queue::dequeue()\n");
        return last;
    }
    // class Queue
    // namespace util
```

---

The Simple Queue Class

```
util::Queue
  - first : util::Item
  - last : util::Item
  + enqueue(item : util::Item)
  + dequeue() : util::Item

util::Item
  - next
```

---
Introduction

Scenario: The Problem
Please extend the Queue class by an element counter!
I want Queue to throw exceptions!
Queue should be thread-safe!
Various users of Queue demand extensions:

The Not So Simple Queue Class

```cpp
class Queue {
  Item *first, *last;
  int counter;
  os::Mutex lock;
public:
  Queue () : first(0), last(0) {
    counter = 0;
  }
  void enqueue(Item* item) {
    lock.enter();
    try {
      if (item == 0)
        throw QueueInvalidItemError();
      if (last) {
        last->next = item;
        last = item;
      } else {
        last = first = item;
      }
      ++counter;
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
  }
  Item* dequeue() {
    Item* res;
    lock.enter();
    try {
      res = first;
      if (first == last)
        first = last = 0;
      else first = first->next;
      if (counter > 0) --counter;
      if (res == 0)
        throw QueueEmptyError();
      lock.leave();
      return res;
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
    return res;
  }
  int count() { return counter; }
}; // class Queue
```

What Code Does What?

Problem Summary

The component code is “polluted” with code for several logically independent concerns, thus it is ...

- hard to write the code
  - many different things have to be considered simultaneously
- hard to read the code
  - many things are going on at the same time
- hard to maintain and evolve the code
  - the implementation of concerns such as locking is scattered over the entire source base (a “crosscutting concern”)
- hard to configure at compile time
  - the users get a “one fits all” queue class
Part II – AOP with C++

Configuring with the Preprocessor?

```cpp
class Queue {
  Item *first, *last;
  #ifdef COUNTING_ASPECT
  int counter;
  #endif
  #ifdef LOCKING_ASPECT
  os::Mutex lock;
  #endif
public:
  Queue() :
    first(0),
    last(0) {
    #ifdef COUNTING_ASPECT
      counter = 0;
    #endif
  }
  void enqueue(Item *item) {
    #ifdef LOCKING_ASPECT
      lock.enter();
      try {
    #endif
    #ifdef ERRORHANDLING_ASPECT
      if (item == 0)
        throw QueueInvalidItemError();
    #endif
    if (last)
      last->next = item;
    last = item;
    #ifdef COUNTING_ASPECT
      ++counter;
    #endif
    #ifdef LOCKING_ASPECT
      } catch(...) {
        lock.leave();
        throw;
      }
      lock.leave();
    #endif
  }
  Item* dequeue() {
    Item* res;
    #ifdef LOCKING_ASPECT
      lock.enter();
      try {
    #endif
    res = first;
    if (first == last) {
      first = last = 0;
    } else {
      first = first->next;
    }
    #ifdef COUNTING_ASPECT
      if (counter > 0) --counter;
    #endif
    #ifdef ERRORHANDLING_ASPECT
      if (res == 0)
        throw QueueEmptyError();
    #endif
    #ifdef LOCKING_ASPECT
      } catch(...) {
        lock.leave();
        throw;
      }
      lock.leave();
    #endif
    return res;
  }
  #ifdef COUNTING_ASPECT
  int count() {
    return counter;
  }
  #endif
}; // class Queue
```

Outline

- We go through the Queue example and...
  - decompose the "one-fits-all" code into modular units
  - apply simple AOP concepts
  - use only C/C++ language idioms and elements
- After we went through the example, we...
  - will try to understand the benefits and limitations of a pure C++ approach
  - motivate the need for an advanced language with built-in AOP concepts: AspectC++

Preprocessor

- While we are able to enable/disable features
  - the code is not expressed in a modular fashion
  - aspectual code is spread out over the entire code base
  - the code is almost unreadable
- Preprocessor is the "typical C way" to solve problems
- Which C++ mechanism could be used instead?
  **Templates!**
AOP with C++

Using Templates

Templates are typically used to implement generic abstract data types:

```cpp
// Generic Array class
// Elements are stored in a resizable buffer
template class T >
class Array {
  T* buf; // allocated memory
public:
  T operator[]( int i ) const {
    return buf[ i ];
  }
  ...
};
```

AOP with Templates

Templates can be used to construct generic code

- To actually use the code, it has to be instantiated

Just as preprocessor directives templates are evaluated at compile-time do not cause any direct runtime overhead (if applied properly)

```cpp
#define add1(T, a, b) 
( ((T)a) + ((T)b) )
template <class T>
T add2(T a, T b) { return a + b; }
printf("%d", add1(int, 1, 2));
printf("%d", add2<int>(1, 2));
```

Aspects as Wrapper Templates

The counting aspect is expressed as a wrapper template class, that derives from the component class:

```cpp
// generic wrapper (aspect), that adds counting to any queue class
// Q, as long it has the proper interface
template <class Q>
class Counting_Aspect : public Q { // aspect should be applied on int counter;
public:
  void enqueue(Item* item) { // execute advice code after join point
    Q::enqueue(item); counter++; }
  Item* dequeue() { // again, after advice
    Item* res = Q::dequeue(item);
    if (counter > 0) counter--;
    return res;
  }
  // this method is added to the component code (introduction)
  int count() const { return counter; }
};
```
Weaving

We can define a type alias (typedef) that combines both, component and aspect code (weaving):

```
// component code
class Queue { ... }

// The aspect (wrapper class)
template <class Q>
class Counting_Aspect : public Q { ... }

// template instantiation
typedef Counting_Aspect<Queue> CountingQueue;

int main() {
    CountingQueue q;
    q.enqueue(new Item);
    q.enqueue(new Item);
    printf("number of items in q: %u\n", q.count());
    return 0;
}
```

Our First Aspect – Lessons Learned

➢ Aspects can be implemented by template wrappers
   - Aspect inherits from component class, overrides relevant methods
   - Introduction of new members (e.g. counter variable) is easy
   - Weaving takes place by defining (and using) type aliases

➢ The aspect code is generic
   - It can be applied to "any" component that exposes the same interface (enqueue, dequeue)
   - Each application of the aspect has to be specified explicitly

➢ The aspect code is clearly separated
   - All code related to counting is gathered in one template class
   - Counting aspect and queue class can be evolved independently (as long as the interface does not change)

Adding an error handling aspect (exceptions) is straightforward. We just need a wrapper template:

```
// another aspect (as wrapper template)
template <class Q>
class Exceptions_Aspect : public Q {
    void enqueue(Item* item) { // this advice is executed before the component code (before advice)
        if (item == 0) throw QueueInvalidItemError();
        Q::enqueue(item);
    }

    Item* dequeue() { // after advice
        Item* res = Q::dequeue();
        if (res == 0) throw QueueEmptyError();
        return res;
    }
};
```

Combining Aspects

We already know how to weave with a single aspect. Weaving with multiple aspects is also straightforward:

```
// component code
<class Queue { ... }>
// wrappers (aspects)
template <class Q>
class Counting_Aspect : public Q { ... }
template <class Q>
class Exceptions_Aspect : public Q { ... }

// template instantiation (weaving)
typedef Exceptions_Aspect< Counting_Aspect< Queue > > ExceptionsCountingQueue;
```
Ordering

➢ In what order should we apply our aspects?
Aspect code is executed outermost-first:

```cpp
typedef Exceptions_Aspect< // first Exceptions, then Counting
  Counting_Aspect< Queue > > ExceptionsCountingQueue;

typedef Counting_Aspect< // first Counting, then Exceptions
  Exceptions_Aspect< Queue > > ExceptionsCountingQueue;
```

➢ Aspects should be independent of ordering
- For dequeue(), both Exceptions_Aspect and Counting_Aspect give after advice. Shall we count first or check first?
- Fortunately, our implementation can deal with both cases:

```cpp
Item* res = Q::dequeue(item);
// its ok if we run before Exceptions_Wrapper
if (counter > 0) counter--;
return res;
```

Locking Aspect

With what we learned so far, putting together the locking aspect should be simple:

```cpp
template <class Q>
class Locking_Aspect : public Q {
  public:
    Mutex lock;
    void enqueue(Item* item) {
      lock.enter();
      try {
        Q::enqueue(item);
      } catch (...) {
        lock.leave();
        throw;
      }
      lock.leave();
    }
  Item* dequeue() {
    lock.enter();
    try {
      res = Q::dequeue(item);
      } catch (...) {
        lock.leave();
        throw;
      }
      lock.leave();
      return res;
    }
};
```

Locking Advice (2)

Locking_Aspect uses an around advice, that proceeds with the component code in the middle of the aspect code:

```cpp
template <class Q>
class Locking_Aspect : public Q {
  public:
    Mutex lock;
    void enqueue(Item* item) {
      lock.enter();
      try {
        Q::enqueue(item);
      } catch (...) {
        lock.leave();
        throw;
      }
      lock.leave();
    }
  Item* dequeue() {
    lock.enter();
    try {
      res = Q::dequeue(item);
      } catch (...) {
        lock.leave();
        throw;
      }
      lock.leave();
      return res;
    }
};
```

Advice Code Duplication

Specifying the same advice for several joinpoints leads to code duplication:
Dealing with Joinpoint Sets

To specify advice for a set of joinpoints, the joinpoints must have a uniform interface:

```cpp
template <class Q>
class Locking_Aspect : public Q {
public:
    Mutex lock;
    // wrap joinpoint invocations into action classes
    struct EnqueueAction {
        Item* item;
        void proceed(Q* q) { q->enqueue(item); }
    };
    struct DequeueAction {
        Item* res;
        void proceed(Q* q) { res = q->dequeue(); }
    };
    ...
};
```

Reusable Advice Code

The advice code is expressed as template function, which is later instantiated with an action class:

```cpp
template <class Q>
class Locking_Aspect : public Q {
    ...
    template <class action> // template inside another template
    void advice(action* a) {
        lock.enter();
        try {
            a->proceed(this);
        } catch (...) {
            lock.leave();
            throw;
        }
        lock.leave();
    }
    ...
};
```

Binding Advice to Joinpoints

Using the action classes we have created, the advice code is now nicely encapsulated in a single function:

```cpp
template <class Q>
class Locking_Aspect2 : public Q {
    ...
    void enqueue(Item* item) {
        EnqueueAction tjp = {item};
        advice(&tjp);
    }
    Item* dequeue() {
        DequeueAction tjp;
        advice(&tjp);
        return tjp.res;
    }
    ...
};
```

Reusing Advice – Lessons Learned

➢ We avoided advice code duplication, by...
    ▶ delegating the invocation of the original code (proceed) to action classes
    ▶ making the aspect code itself a template function
    ▶ instantiating the aspect code with the action classes

➢ Compilers will probably generate less efficient code
    ▶ Additional overhead for storing argument/result values
Putting Everything Together

We can now instantiate the combined Queue class, which uses all aspects:
(For just 3 aspects, the typedef is already getting rather complex)

```cpp
typedef Locking_Aspect<Exceptions_Aspect<Counting_Aspect<Queue>>> CountingQueueWithExceptionsAndLocking;
// maybe a little bit more readable ...
typedef Counting_Aspect<Queue> CountingQueue;
typedef Exceptions_Aspect<CountingQueue> CountingQueueWithExceptions;
typedef Locking_Aspect<CountingQueueWithExceptions> CountingQueueWithExceptionsAndLocking;
```

“Obliviousness”

... is an essential property of AOP: the component code should not have to be aware of aspects, but ...

➢ the extended Queue cannot be named “Queue”
  - our aspects are selected through a naming scheme (e.g. CountingQueueWithExceptionsAndLocking).
➢ using wrapper class names violates the idea of obliviousness

Preferably, we want to hide the aspects from client code that uses affected components.

Hiding Aspects

➢ Aspects can be hidden using C++ namespaces
➢ Three separate namespaces are introduced
  - namespace components: component code for class Queue
  - namespace aspects: aspect code for class Queue
  - namespace configuration: selection of desired aspects for class Queue
➢ The complex naming schemes as seen on the previous slide is avoided

```cpp
namespace components {
    class Queue { ... };
}
namespace aspects {
    template <class Q>
    class Counting_Aspect : public Q { ... };
}
namespace configuration {
    typedef aspects::Counting_Aspect<components::Queue> Queue;
}
// client code can import configuration namespace and use // counting queue as “Queue” using namespace configuration;
void client_code () {
    Queue queue; // Queue with all configured aspects
    queue.enqueue (new MyItem);
}
```
Obliviousness – Lessons Learned

➢ Aspect configuration, aspect code, and client code can be separated using C++ namespaces
  - name conflicts are avoided
➢ Except for using the configuration namespace the client code does not have to be changed
  - obliviousness is (mostly) achieved on the client-side

What about obliviousness in the extended classes?

Limitations

For simple aspects the presented techniques work quite well, but a closer look reveals limitations:

➢ Joinpoint types
  - no distinction between function call and execution
  - no generic interface to joinpoint context
  - no advice for private member functions
➢ Quantification
  - no flexible way to describe the target components (like AspectJ/AspectC++ pointcuts)
  - applying the same aspect to classes with different interfaces is impossible or ends with excessive template metaprogramming

Conclusions

➢ C++ templates can be used for separation of concerns in C++ code without special tool support
➢ However, the lack of expressiveness and scalability restricts these techniques to projects with ...
  - only a small number of aspects
  - few or no aspect interactions
  - aspects with a non-generic nature
  - component code that is “aspect-aware”
➢ However, switching to tool support is easy!
  - aspects have already been extracted and modularized.
  - transforming template-based aspects to code expected by dedicated AOP tools is only mechanical labor
References/Other Approaches

- A comprehensive analysis of doing AOP with pure C++: what's possible and what not
  - http://www.heise.de/ix/artikel/2001/08/143/

A. Alexandrescu: "Modern C++ Design – Generic Programming and Design Patterns Applied", Addison-Wesley, C++ in depth series, 2001
- Introduces "policy-based design", a technique for advanced separation of concerns in C++
- Policy-based design tries to achieve somewhat similar goals as AOP does
  - http://www.moderncppdesign.com/

Other suggestions towards AOP with pure C++:
- C. Diggins: “Aspect Oriented Programming in C++”
- D. Vollmann: “Visibility of Join-Points in AOP and Implementation Languages”
Part III – Aspect C++

Queue: Demanded Extensions

I. Element counting

II. Errorhandling
   (signaling of errors by exceptions)

III. Thread safety
    (synchronization by mutex variables)

Element counting: The Idea

➢ Increment a counter variable after each execution of `util::Queue::enqueue()`

➢ Decrement it after each execution of `util::Queue::dequeue()`
We introduced a new aspect named `ElementCounter`. An aspect starts with the keyword `aspect` and is syntactically much like a class.

Like a class, an aspect can define data members, constructors and so on.
**ElementCounter1 - Elements**

The **pointcut expression** denotes where the advice should be given.
(After execution of methods that match the pattern)

```cpp
aspect ElementCounter {
    int counter;
    ElementCounter() { counter = 0; }
    advice execution("% util::Queue::enqueue(...)") : after()
        { ++counter;
          printf(" Aspect ElementCounter: # of elements = %d\n", counter);
        }
    advice execution("% util::Queue::dequeue(...)") : after()
        { if( counter > 0 ) --counter;
          printf(" Aspect ElementCounter: # of elements = %d\n", counter);
        }
};
ElementCounter1.ah
```

**Aspect member elements can be accessed from within the advice body**

```cpp
aspect ElementCounter {
    int counter;
    ElementCounter() { counter = 0; }
    advice execution("% util::Queue::enqueue(...)") : after()
        { ++counter;
          printf(" Aspect ElementCounter: # of elements = %d\n", counter);
        }
    advice execution("% util::Queue::dequeue(...)") : after()
        { if( counter > 0 ) --counter;
          printf(" Aspect ElementCounter: # of elements = %d\n", counter);
        }
};
ElementCounter1.ah
```

**ElementCounter1 - Result**

```cpp
int main() {
    util::Queue queue;
    printf("main(): enqueueing an item\n");
    queue.enqueue( new util::Item );
    printf("main(): dequeuing two items\n");
    util::Item* item = queue.dequeue();
    item = queue.dequeue();
}
main.cc
```

```cpp
int main() {
    util::Queue queue;
    printf("main(): enqueueing an item\n");
    queue.enqueue( new util::Item );
    printf("main(): dequeuing two items\n");
    util::Item* item = queue.dequeue();
    item = queue.dequeue();
}
```

**ElementCounter1 – What’s next?**

- The aspect is not the ideal place to store the counter, because it is shared between all Queue instances
- Ideally, counter becomes a member of Queue
- In the next step, we
  - move counter into Queue by introduction
  - expose context about the aspect invocation to access the current Queue instance
ElementCounter2

```cpp
aspect ElementCounter {
  private:
    advice "util::Queue": int counter;
  public:
    advice "util::Queue": int count { return counter; } const
    advice execution("% util::Queue::enqueue(...)"
      & that(queue) : after( util::Queue& queue ) {
      ++queue.counter;
      printf(" Aspect ElementCounter: # of elements = %d\n", queue.count());
    }
    advice execution("% util::Queue::dequeue(...)"
      & that(queue) : after( util::Queue& queue ) {
      if (queue.count() > 0) --queue.counter;
      printf(" Aspect ElementCounter: # of elements = %d\n", queue.count());
    }
    advice construction("util::Queue"
      & that(queue) : before( util::Queue& queue ) {
      queue.counter = 0;
    });
};
```

ElementCounter2 - Elements

We also introduce a public method to read the counter

```cpp
int ElementCounter2::count() { return counter; }
```

ElementCounter2 - Elements

A context variable queue is bound to that (the calling instance). The calling instance has to be an util::Queue

```cpp
... queue is bound to that (the calling instance). 
The calling instance has to be an util::Queue
```

ElementCounter2 - Elements

Introduces a new data member counter into all classes denoted by the pointcut "util::Queue"

```cpp
... Introduces a new data member counter into all classes denoted by the pointcut "util::Queue"
```
```cpp
aspect ElementCounter {
private:
  advice "util::Queue" : int counter;
public:
  advice "util::Queue" : int count { return counter; }
  advice execution("% util::Queue::enqueue(...)")
    & that(queue) : after( util::Queue& queue ) {
    queue.counter = 0;
    printf(" Aspect ElementCounter: # of elements = %d\n", queue.count());
  }
  advice execution("% util::Queue::dequeue(...)")
    & that(queue) : after( util::Queue& queue ) {
    if (queue.count() > 0) --queue.counter;
    printf(" Aspect ElementCounter: # of elements = %d\n", queue.count());
  }
  advice construction("util::Queue")
    & that(queue) : before( util::Queue& queue ) {
    queue.counter = 0;
 );
};

int main() {
  util::Queue queue;
  printf("main(): Queue contains %d items\n", queue.count());
  queue.enqueue(new util::Item);
  queue.enqueue(new util::Item);
  printf("main(): Queue contains %d items\n", queue.count());
  util::Item* item;
  item = queue.dequeue();
  printf("main(): dequeueing one items\n");
  printf("main(): Queue contains %d items\n", queue.count());
}
```
ElementCounter – Lessons Learned

You have seen...

➢ the most important concepts of AspectC++
  - Aspects are introduced with the keyword `aspect`
  - They are much like a class, may contain methods, data members, types, inner classes, etc.
  - Additionally, aspects can give `advice` to be woven in at certain positions (`joinpoints`). Advice can be given to
    • Functions/Methods/Constructors: code to execute (`code advice`)
    • Classes or structs: new elements (`introductions`)
  - Joinpoints are described by `pointcut expressions`

➢ We will now take a closer look at some of them

Joinpoints

➢ A `joinpoint` denotes a position to give advice
  - `Code` joinpoint
    a point in the **control flow** of a running program, e.g.
    • `execution` of a function
    • `call` of a function
  - `Name` joinpoint
    • a **named C++ program entity** (identifier)
      • class, function, method, type, namespace
  - Joinpoints are given by `pointcut expressions`
    • a pointcut expression describes a set of joinpoints

Pointcut Expressions

➢ Pointcut expressions are made from ...
  - `match expressions`, e.g. "% util::Queue::enqueue(…)"
    • are matched against C++ program entities → name joinpoints
    • support wildcards
  - `pointcut functions`, e.g `execution(…), call(…), that(…)`
    • `execution`: all points in the control flow, where a function is about to be executed → code joinpoints
    • `call`: all points in the control flow, where a function is about to be called → code joinpoints
  - Pointcut functions can be combined into expressions
    • using logical connectors: &&, ||, !
    • Example: `call("% util::Queue::enqueue(...)") && within("% main(...)")`

Syntactic Elements

```
aspect ElementCounter {
  advice execution("% util::Queue::enqueue(...)") : after()
  {
    printf( " Aspect ElementCounter: after Queue::enqueue\n" );
  }
  ...
};
```

```
Pointcut Expressions

➢ Pointcut expressions are made from ...
  - `match expressions`, e.g. "% util::Queue::enqueue(…)"
    • are matched against C++ program entities → name joinpoints
    • support wildcards
  - `pointcut functions`, e.g `execution(…), call(…), that(…)`
    • `execution`: all points in the control flow, where a function is about to be executed → code joinpoints
    • `call`: all points in the control flow, where a function is about to be called → code joinpoints
  ➢ Pointcut functions can be combined into expressions
    • using logical connectors: &&, ||, !
    • Example: `call("% util::Queue::enqueue(...)") && within("% main(...)")`
```
Advice to functions

- **before advice**
  - Advice code is executed before the original code
  - Advice may read/modify parameter values
- **after advice**
  - Advice code is executed after the original code
  - Advice may read/modify return value
- **around advice**
  - Advice code is executed instead of the original code
  - Original code may be called explicitly: tjp->proceed()

Introductions

- Additional methods, data members, etc. are added to the class
- Can be used to extend the interface of a class

Before / After Advice

```cpp
class ClassA {
  public:
    void foo()
      printf("ClassA::foo()\n");
    ...
};

int main()
  printf("main()\n");
  ClassA a;
  a.foo();
```

Around Advice

```cpp
class ClassA {
  public:
    void foo()
      printf("ClassA::foo()\n");
    ...
};

int main()
  printf("main()\n");
  ClassA a;
  a.foo();
```

Introductions

```cpp
class ClassA {
  public:
    void foo()
      printf("ClassA::foo()\n");
    ...
};
```

private:

```cpp
int main()
  printf("main()\n");
  ClassA a;
  a.foo();
```

public:

```cpp
int main()
  printf("main()\n");
  ClassA a;
  a.foo();
```
Queue: Demanded Extensions

I. Element counting

II. Errorhandling
   (signaling of errors by exceptions)

III. Thread safety
    (synchronization by mutex variables)

I want Queue to throw exceptions!

Errorhandling: The Idea

➢ We want to check the following constraints:
   – enqueue() is never called with a NULL item
   – dequeue() is never called on an empty queue
➢ In case of an error an exception should be thrown
➢ To implement this, we need access to ...
   – the parameter passed to enqueue()
   – the return value returned by dequeue()
   ... from within the advice

ErrorException

namespace util {
  struct QueueInvalidItemError {}
  struct QueueEmptyError {}
}

aspectErrorException {

  advice execution("% util::Queue::enqueue(...)") && args(item)
    : before(util::Item* item) {
    if( item == 0 )
      throw util::QueueInvalidItemError();
  }

  advice execution("% util::Queue::dequeue(...)") && result(item)
    : after(util::Item* item) {
    if( item == 0 )
      throw util::QueueEmptyError();
  }

};

ErrorException - Elements

namespace util {
  struct QueueInvalidItemError {}
  struct QueueEmptyError {}
}

aspectErrorException {

  advice execution("% util::Queue::enqueue(...)") && args(item)
    : before(util::Item* item) {
    if( item == 0 )
      throw util::QueueInvalidItemError();
  }

  advice execution("% util::Queue::dequeue(...)") && result(item)
    : after(util::Item* item) {
    if( item == 0 )
      throw util::QueueEmptyError();
  }

};
### ErrorException - Elements

```cpp
namespace util {
  struct QueueInvalidItemError {};
  struct QueueEmptyError {};
}

aspect ErrorException {
  advice execution("% util::Queue::enqueue(...)") && args(item) :
    before(util::Item* item) {
      if (item == 0)
        throw util::QueueInvalidItemError();
    }
  advice execution("% util::Queue::dequeue(...)") && result(item) :
    after(util::Item* item) {
      if (item == 0)
        throw util::QueueEmptyError();
    }
};
```

A context variable `item` is bound to the first argument of type `util::Item*` passed to the matching methods.

### ErrorException - Elements

```cpp
namespace util {
  struct QueueInvalidItemError {};
  struct QueueEmptyError {};
}

aspect ErrorException {
  advice execution("% util::Queue::enqueue(...)") && args(item) :
    before(util::Item* item) {
      if (item == 0)
        throw util::QueueInvalidItemError();
    }
  advice execution("% util::Queue::dequeue(...)") && result(item) :
    after(util::Item* item) {
      if (item == 0)
        throw util::QueueEmptyError();
    }
};
```

Here the context variable `item` is bound to the result of type `util::Item*` returned by the matching methods.

### ErrorException – Lessons Learned

You have seen how to ...

- use different types of advice
  - before advice
  - after advice
- expose context in the advice body
  - by using `args` to read/modify parameter values
  - by using `result` to read/modify the return value

### Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)
Thread Safety: The Idea

➢ Protect enqueue() and dequeue() by a mutex object

➢ To implement this, we need to
  - introduce a mutex variable into class Queue
  - lock the mutex before the execution of enqueue() / dequeue()
  - unlock the mutex after execution of enqueue() / dequeue()

➢ The aspect implementation should be exception safe!
  - in case of an exception, pending after advice is not called
  - solution: use around advice

---

LockingMutex

aspect LockingMutex {
  advice "util::Queue" : os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) && that(queue) :
    around( util::Queue& queue ) {
      queue.lock.enter();
      try {
        tjp->proceed();
      }
      catch(...) {
        queue.lock.leave();
        queue.lock.leave();
        throw;
      }
    };
};

---

LockingMutex - Elements

aspect LockingMutex {
  advice "util::Queue" : os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::%queue(...)";
  advice execution(sync_methods()) && that(queue) :
    around( util::Queue& queue ) {
      queue.lock.enter();
      try {
        tjp->proceed();
      }
      catch(...) {
        queue.lock.leave();
        throw;
      }
    };
}

We introduce a mutex member into class Queue

Pointcuts can be named. sync_methods describes all methods that have to be synchronized by the mutex
aspect LockingMutex {
    advice "util::Queue" : os::Mutex lock;
    pointcut sync_methods() = "% util::Queue::%queue(...)";
    advice execution(sync_methods()) && that(queue) : around( util::Queue& queue ) {
        queue.lock.enter();
        try {
            tjp->proceed();
        }
        catch(...) {
            queue.lock.leave();
            throw;
        }
        queue.lock.leave();
    };
}

sync_methods is used to give around advice to the execution of the methods.

By calling tjp->proceed() the original method is executed.

You have seen how to ...

➢ use named pointcuts
  - to increase readability of pointcut expressions
  - to reuse pointcut expressions
➢ use around advice
  - to deal with exception safety
  - to explicit invoke (or don’t invoke) the original code by calling tjp->proceed()
➢ use wildcards in match expressions
  - "% util::Queue::%queue(...)" matches both enqueue() and dequeue()
Interrupt Safety: The Idea

➢ Scenario
- Queue is used to transport objects between kernel code (interrupt handlers) and application code
  - If application code accesses the queue, interrupts must be disabled first
  - If kernel code accesses the queue, interrupts must not be disabled

➢ To implement this, we need to distinguish
  - if the call is made from kernel code, or
  - if the call is made from application code

We define two pointcuts. One for the methods to be synchronized and one for all kernel functions

This pointcut expression matches any call to a sync_method that is not done from kernel_code
LockingIRQ1 – Problem
util::Queue queue;
void do_something() {
    printf("do_something\n");
    queue.enqueue( new util::Item );
}

namespace kernel {
    void irq_handler() {
        printf("kernel::irq_handler\n");
        do_something();
    }
}

int main() {
    printf("main\n");
    kernel::irq_handler(); // irq
    printf("back in main\n");
    queue.dequeue();
}

main.cc

LockingIRQ1 – Result
util::Queue queue;
void do_something() {
    printf("do_something\n");
    queue.enqueue( new util::Item );
}

namespace kernel {
    void irq_handler() {
        printf("kernel::irq_handler\n");
        do_something();
    }
}

int main() {
    printf("main\n");
    kernel::irq_handler(); // irq
    printf("back in main\n");
    queue.dequeue();
}

main.cc

LockingIRQ2
aspect LockingIRQ {
    pointcut sync_methods() = "% util::Queue::queue(...)";
    pointcut kernel_code() = "% kernel::...\n";

    advice execution(kernel_code) & & cflow(kernel_code) : around() {
        os::disable_int();
        try {
            tjp->proceed();
            catch(...) {
                os::enable_int();
            }
        }
        os::enable_int();
    }
};

LockingIRQ2.ah

Solution
Using the cflow pointcut function

This pointcut expression matches the execution of sync_methods if no kernel_code is on the call stack. cflow checks the call stack (control flow) at runtime.
LockingIRQ2 – Result

util::Queue queue;
void do_something() {
    printf("do_something()
");
    queue.enqueue(new util::Item);
}

namespace kernel {
    void irq_handler() {
        printf("kernel::irq_handler()
");
        queue.enqueue(new util::Item);
        do_something();
    }
}

int main() {
    printf("main()
");
    queue.enqueue(new util::Item);
    kernel::irq_handler();
    // irq
    printf("back in main()\n");
    queue.dequeue();
}

LockingIRQ – Lessons Learned

You have seen how to ...

➢ restrict advice invocation to a specific calling context

➢ use the within(...) and cflow(...) pointcut functions
   - within is evaluated at compile time and returns all code
     joinpoints of a class' or namespaces lexical scope
   - cflow is evaluated at runtime and returns all joinpoints
     where the control flow is below a specific code joinpoint

AspectC++: A First Summary

➢ The Queue example has presented the most
   important features of the AspectC++ language
   • aspect, advice, joinpoint, pointcut expression,
     pointcut function, ...

➢ Additionally, AspectC++ provides some more
   advanced concepts and features
   • to increase the expressive power of aspectual code
   • to write broadly reusable aspects
   • to deal with aspect interdependence and ordering

➢ In the following, we give a short overview on
   these advanced language elements

AspectC++: Advanced Concepts

➢ Join Point API
   • provides a uniform interface to the aspect invocation context,
     both at runtime and compile-time

➢ Abstract Aspects and Aspect Inheritance
   • comparable to class inheritance, aspect inheritance allows to
     reuse parts of an aspect and overwrite other parts

➢ Generic Advice
   • exploits static type information in advice code

➢ Aspect Ordering
   • allows to specify the invocation order of multiple aspects

➢ Aspect Instantiation
   • allows to implement user-defined aspect instantiation models
The Joinpoint API

➢ Inside an advice body, the current joinpoint context is available via the implicitly passed `tjp` variable:

```cpp
advice ...
{
  struct JoinPoint {
    ...
  } *tjp; // implicitly available in advice code ...
}

➢ You have already seen how to use `tjp`, to ...
  • execute the original code in around advice with `tjp->proceed()`

➢ The joinpoint API provides a rich interface
  • to expose context independently of the aspect target
  • this is especially useful in writing reusable aspect code

Abstract Aspects and Inheritance

➢ Aspects can inherit from other aspects...
  • Reuse aspect definitions
  • Override methods and pointcuts
➢ Pointcuts can be pure virtual
  • Postpone the concrete definition to derived aspects
  • An aspect with a pure virtual pointcut is called abstract aspect
➢ Common usage: Reusable aspect implementations
  • Abstract aspect defines advice code, but pure virtual pointcuts
  • Aspect code uses the joinpoint API to expose context
  • Concrete aspect inherits the advice code and overrides pointcuts

The Join Point API (Excerpt)

Types (compile-time)

- object type (initiator)
  ```
  That
  ```
- object type (receiver)
  ```
  Target
  ```
- result type of the affected function
  ```
  Result
  ```
- type of the i'th argument of the affected function
  ```
  Arg<i>::Type
  ```
- type of the i'th argument value of a function call (compile-time index)
  ```
  Arg<i>::ReferredType
  ```
- type of the i'th argument value of a function call (runtime index)
  ```
  void* arg(int i)
  ```
- textual representation of the joinpoint
  ```
  static const char* signature()
  ```

Values (runtime)

- pointer to the object initiating a call
  ```
  That* that()
  ```
- pointer to the object that is target of a call
  ```
  Target* target()
  ```
- pointer to the result value of the function
  ```
  Result* result()
  ```
- typed pointer to the i'th argument value of a function call
  ```
  Arg<i>::ReferredType* arg()
  ```
- textual representation of the joinpoint
  ```
  static const char* signature()
  ```
- executes the original joinpoint code in an around advice
  ```
  void proceed()
  ```
- returns the runtime action object
  ```
  AC::Action& action()
  ```

Abstract Aspects and Inheritance

The abstract locking aspect declares two pure virtual pointcuts and uses the joinpoint API for an context-independent advice implementation.
Abstract Aspects and Inheritance

```
#include "mutex.h"
aspect LockingA {
  pointcut virtual sync_classes() = 0;
  pointcut virtual sync_methods() = 0;
  advice sync_classes() :
  os::Mutex lock;
  advice execution(sync_methods()) :
  tjp->that() -> lock.enter();
  try {
    tjp->proceed();
  } catch(...) {
    tjp->that() -> lock.leave();
    throw;
  } tjp->that() -> lock.leave();
};
```

The concrete locking aspect derives from the abstract aspect and overrides the pointcuts.

```
#include "LockingA.ah"
aspect LockingQueue : public LockingA {
  pointcut sync_classes() :
  "util::Queue";
  pointcut sync_methods() :
  "% util::Queue::%queue(...)";
};
```

The concrete locking aspect derives from the abstract aspect and overrides the pointcuts.

Generic Advice

Uses static JP-specific type information in advice code
- in combination with C++ overloading
- to instantiate C++ templates and template meta-programs

```
aspect TraceService {
  advice call(...) : after() {
    ... cout << tjp->result();
  }
};
```

Resolves to the statically typed return value
- no runtime type checks are needed
- unhandled types are detected at compile-time
- functions can be inlined

```
... operator <<(..., int)
... operator <<(..., long)
... operator <<(..., bool)
... operator <<(..., Foo)
```

Aspect Ordering

➢ Aspects should be independent of other aspects
  - However, sometimes inter-aspect dependencies are unavoidable
  - Example: Locking should be activated before any other aspects
➢ Order advice
  - The aspect order can be defined by order advice
    advice pointcut-expr : order(high, ..., low)
  - Different aspect orders can be defined for different pointcuts
➢ Example

```
advice "% util::Queue::%queue(...)"
  : order("LockingIRQ", "%" && !"LockingIRQ");
```
Aspect Instantiation

➢ Aspects are singletons by default
  • `aspectof()` returns pointer to the one-and-only aspect instance
➢ By overriding `aspectof()` this can be changed
  • e.g. one instance per client or one instance per thread

```cpp
aspect MyAspect {
    // ....
    static MyAspect* aspectof() {
        static __declspec(thread) MyAspect* theAspect;
        if (theAspect == 0)
            theAspect = new MyAspect;
        return theAspect;
    }
};
```

Example of an user-defined `aspectof()` implementation for per-thread aspect instantiation by using thread-local storage. (Visual C++)

Summary

➢ AspectC++ facilitates AOP with C++
  • AspectJ-like syntax and semantics
➢ Full obliviousness and quantification
  • aspect code is given by `advice`
  • joinpoints are given declaratively by `pointcuts`
  • implementation of crosscutting concerns is fully encapsulated in `aspects`
➢ Good support for reusable and generic aspect code
  • aspect inheritance and virtual pointcuts
  • rich `joinpoint` API

And what about tool support?
Part IV – Tool Support

Overview

- ac++ compiler
  - open source and base of the other presented tools
- ag++ wrapper
  - easy to use wrapper around g++ for make-based projects
- AspectC++ Add-In for Microsoft® Visual Studio®
  - commercial product by pure-systems GmbH
- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

➔ demonstration with the tutorial CD

About ac++

- Available from www.aspectc.org
  - Linux, Win32, Solaris, MacOS X binaries + source (GPL)
  - documentation: Compiler Manual, Language Reference, ...
- Transforms AspectC++ to C++ code
  - machine code is created by the back-end (cross-)compiler
  - supports g++ and Visual C++ specific language extensions
- Current version: 1.0
  - stable
  - (almost) feature-complete
  - no optimizations for compilation speed, yet

Aspect Transformation

```c++
aspect Transform { 
  advice call("% foo()") : before() {
    printf("before foo call\n";)
  }
  advice execution("% C::%()") : after() {
    printf(tjp->signature());
  }
};
```

```c++
class Transform { 
  static Transform __instance;
  // ...
  void __a0_before () { 
    printf ("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature());
    }
};
```
Aspect Transformation

```c++
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
    advice execution("% C::%()") : after()
    { printf(tjp->signature());
    }
  }
}
```

Advice becomes a member function

```c++
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature());
    }
};
```

One global aspect instance is created by default

```c++
aspect Transform {
  advice call("% foo()") : before() {
    printf("before foo call\n");
    advice execution("% C::%()") : after()
    { printf(tjp->signature());
    }
  }
}
```

“Generic Advice” becomes a template member function

```c++
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature());
    }
};
```

Aspects are transformed into ordinary classes

```c++
class Transform {
  static Transform __instance;
  // ...
  void __a0_before () {
    printf("before foo call\n");
  }
  template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
      printf (tjp->signature());
    }
};
```

One global aspect instance is created by default
Joinpoint Transformation

int main() {
  foo();
  return 0;
}

main.cc

Joinpoint Transformation

int main() {
  struct __call_main_0_0 {
    static inline void invoke (){
      AC::__a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}

main.cc'

Joinpoint Transformation

int main() {
  foo();
  return 0;
}

main.cc

Joinpoint Transformation

int main() {
  struct __call_main_0_0 {
    static inline void invoke (){
      AC::__a0_before ();
      ::foo();
    }
  };
  __call_main_0_0::invoke ();
  return 0;
}

main.cc'

Tool Support

➢ Whole Program Transformation-Mode
  - e.g. ac++ -p src -d gen -e cpp -Iinc -DDEBUG
  - transforms whole directory trees
  - generates manipulated headers, e.g. for libraries
  - can be chained with other whole program transformation tools

➢ Single Translation Unit-Mode
  - e.g. ac++ -c a.cc -o a-gen.cc -p .
  - easier integration into build processes
Tool Demos

- AspectC++ Add-In for Microsoft® Visual Studio®
  - by pure-systems GmbH (www.pure-systems.com)
- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

Summary

➢ Tool support for AspectC++ programming is based on the ac++ command line compiler
  - full "obliviousness and quantification"
  - delegates the binary code generation to your favorite compiler
➢ Commercial and a non-commercial IDE integration is available
  - Microsoft® Visual Studio®
  - Eclipse®
Part V – Examples

Aspect-Oriented Programming with C++ and AspectC++
AOSD 2004 Tutorial

Examples

AspectC++ in Practice - Examples

➢ Applying the observer protocol
  - Example: a typical scenario for the widely used observer pattern
  - Problem: implementing observer requires several design and code transformations

➢ Error handling in legacy code
  - Example: a typical Win32 application
  - Problem: error handling often “forgotten” as too much of a bother

➢ Product line development
  - Example: an embedded weather station software family
  - Problem: optional crosscutting concerns in optional components (a typical case for the #ifdef hell)

Observer Pattern: Scenario

Draw()
AnalogClock
Draw()
DigitalClock
GetHour() : int
SetTime (in h : int, in m : int, in s : int)
Tick()

update on change

Observer Pattern: Implementation

`IObserver` update (in s : ISubject)
`IObserver` updateAll()
`ISubject` observers
update (in s)
update (in s)
Observer Pattern: Problem

The 'Observer Protocol' Concern...

...crosscuts the module structure

Solution: Generic Observer Aspect

aspect ObserverPattern {
  ...
  public:
    struct ISubject {}
    struct IObserver {
      virtual void update (ISubject *) = 0;
    }
    pointcut virtual observers() = 0;
    pointcut virtual subjects() = 0;
    pointcut virtual subjectChange() = execution( "% ...::%(...)"
        && !"% ...::%(...) const" ) && within( subjects() );
    advice observers () : baseclass ( IObserver );
    advice subjects() : baseclass ( ISubject );
    advice subjectChange() : after () {
      ISubject* subject = tjp->that();
      updateObservers( subject );
    }
    void updateObservers( ISubject* subject ) { ... }
    void addObserver( ISubject* subject, IObserver* observer ) { ... }
    void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};

abstract pointcuts that define subjects/observers
(need to be overridden by a derived aspect)

Interfaces for the subject/observer roles

Examples
Solution: Generic Observer Aspect

aspect ObserverPattern {
  ... 
public:
  struct ISubject {}
  struct IObserver {
    virtual void update (ISubject *) = 0;
  }
  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;
  pointcut virtual subjectChange() = execution("% ...::%(...)
                          && !"% ...::%(...) const") && within(subjects());
  advice observers() : baseclass(IObserver);
  advice subjects() : baseclass(ISubject);
  advice subjectChange() : after()
    { ISubject* subject = tjp->that();
      updateObservers(subject); }
  void updateObservers(ISubject* subject) { ... }
  void addObserver(ISubject* subject, IObserver* observer) { ... }
  void remObserver(ISubject* subject, IObserver* observer) { ... }
};

Solution: Putting Everything Together

aspect ClockObserver : public ObserverPattern {
  // define the participants
  pointcut subjects() = "ClockTimer";
  pointcut observers() = "DigitalClock" || "AnalogClock";
  advice observers() : baseclass(IOserver);
  advice subjects() : baseclass(ISubject);
  advice subjectChange() : after()
    { ISubject* subject = tjp->that();
      updateObservers(subject); }
  void updateObservers(ISubject* subject) { ... }
  void addObserver(ISubject* subject, IObserver* observer) { ... }
  void remObserver(ISubject* subject, IObserver* observer) { ... }
};

Applying the Generic Observer Aspect to the clock example

aspect ClockObserver : public ObserverPattern {
  // define the participants
  pointcut subjects() = "ClockTimer";
  pointcut observers() = "DigitalClock" || "AnalogClock";
  public:
  // define what to do in case of a notification
  advice observers() : void update(ObserverPattern::ISubject* s) { s->Draw(); }
};
Observer Pattern: Conclusions

➢ Applying the observer protocol is now very easy!
  - all necessary transformations are performed by the generic aspect
  - programmer just needs to define participants and behaviour
  - multiple subject(observer relationships can be defined

➢ More reusable and less error-prone component code
  - observer no longer “hard coded” into the design and code
  - no more forgotten calls to update() in subject classes

➢ Full source code on Tutorial CD

Errorhandling in Legacy Code: Scenario

LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
  HDC dc = NULL; PAINTSTRUCT ps = {0};
  switch( nMsg ) {
    case WM_PAINT:
      dc = BeginPaint( hWnd, &ps );
      ...
      EndPaint(hWnd, &ps);
      break;
    ...
  }
  int WINAPI WinMain( ... ) {
    HANDLE hConfigFile = CreateFile( "example.config", GENERIC_READ, ... );
    WNDCLASS wc = {0, WndProc, 0, 0, ... , "Example_Class"};
    RegisterClass( &wc );
    HWND hwndMain = CreateWindowEx( 0, "Example_Class", "Example", ... );
    UpdateWindow( hwndMain );
    MSG msg;
    while( GetMessage( &msg, NULL, 0, 0 ) ) {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
    return 0;
  }
}

These Win32 API functions may fail!

Win32 Errorhandling: Goals

➢ Detect failed calls of Win32 API functions
  • by giving after advice for any call to a Win32 function
➢ Throw a helpful exception in case of a failure
  • describing the exact circumstances and reason of the failure

Problem: Win32 failures are indicated by a “magic” return value
• magic value to compare against depends on the return type of the function
• error reason (GetLastError()) only valid in case of a failure

<table>
<thead>
<tr>
<th>return type</th>
<th>magic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>FALSE</td>
</tr>
<tr>
<td>ATOM</td>
<td>(ATOM) 0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>INVALID_HANDLE_VALUE or NULL</td>
</tr>
<tr>
<td>HWND</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Detecting the failure: Generic Advice

```cpp
advice call(win32API ()) : after () {
  if (isError (*tjp->result()))
    // throw an exception
}
bool isError(ATOM);
bool isError(BOOL);
bool isError(HANDLE);
bool isError(HWND);
...```

Describing the failure: Generative Advice

```cpp
advice call(win32API ()) : after () {
  // throw an exception
  ostringstream s;
  DWORD code = GetLastError();
  s << "WIN32 ERROR " << code << ... << win32::GetErrorText( code ) << ...
  ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);
  throw win32::Exception( s.str() );
}
```

Reporting the Error

```cpp
LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
  HDC dc = NULL; PAINTSTRUCT ps = {0};
  switch ( nMsg ) {
  case WM_PAINT:
    dc = BeginPaint( hWnd, &ps );
    ... EndPaint(hWnd, &ps);
    break;
  ...}
int WINAPI WinMain( ... ) {
  HANDLE hConfigFile = CreateFile( "example.config", GENERIC_READ, ... );
  WNDCLASS wc = {0, WndProc, 0, 0, 0, "Example_Class"};
  RegisterClass( &wc );
  HWND hwndMain = CreateWindowEx( 0, "Example_Class", "Example", ... );
  UpdateWindow( hwndMain );
  MSG msg;
  while ( GetMessage( &msg, NULL, 0, 0 ) ) {
    TranslateMessage( &msg );
    DispatchMessage( &msg );
  }
  return 0;
}
```

Errorhandling in Legacy Code: Conclusions

- Easy to apply errorhandling for Win32 applications
  - previously undetected failures are reported by exceptions
  - rich context information is provided
- Uses advanced AspectC++ techniques
  - error detection by generic advice
  - context propagation by generative advice
- Full source code on tutorial CD
Part VI – Summary

---

**Summary – This Tutorial ...**

- showed basic techniques for AOP with pure C++
  - using templates to program generic wrapper code
  - using action classes to encapsulate the “proceed-code”
  - using namespaces to substitute types transparently
- introduced the AspectC++ language extension for C++
  - AspectJ-like language extension
  - ac++ transforms AspectC++ into C++
  - supports AOP even in resource constrained environments
- demonstrated the AspectC++ tools
- discussed the pros and cons of each approach

---

**Pros and Cons**

AOP with pure C++
- no special tool required
- requires in-depth understanding of C++ templates
- lack of “obliviousness”
  - the component code has to be aspect-aware
- lack of “quantification”
  - no pointcut concept, no match expressions

AspectC++
- the ac++ compiler transforms AspectC++ into C++
- various supported joinpoint types, e.g. execution and calls
- built-in support for advanced AOP concepts:
  - cflow, joinpoint-API
- longer compilation times

---

**Future Work – Roadmap**

- Parser improvements
  - full template support
  - speed optimization
  - full g++ 3.x and Visual C++ compatibility
- Language design/weaver
  - weaving in templates
  - advice for object access (set/get pointcut functions)
  - advice for object instantiations
- Tools
  - dependency handling
Thank you for your attention!