Konfigurierbare Systemsoftware (KSS)

VL 3 – Aspect-Oriented Programming (AOP)

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 2013-05-02

http://www4.informatik.uni-erlangen.de/Lehre/SS13/V_KSS

Implementation Techniques: Classification

- Decompositional Approaches
  - Text-based filtering (untyped)
  - Preprocessors

- Compositional Approaches
  - Language-based composition mechanisms (typed)
  - OOP, AOP, Templates

- Generative Approaches
  - Metamodel-based generation of components (typed)
  - MDD, C++ TMP, generators

About this Lecture

Solution Space

Domain Expert

Features and Dependencies

Problem Space

Architecture and Implementation

Specific Problem

Specific Solution

Implementation Techniques: Classification

Decompositional Approaches

Text-based filtering (untyped)

Preprocessors

Compositional Approaches

Language-based composition mechanisms (typed)

OOP, AOP, Templates

Generative Approaches

Metamodel-based generation of components (typed)

MDD, C++ TMP, generators

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 References

© dl KSS (VL 3 | SS 13) 3 AOP

3-2

3-3

3-4
3.1 Motivation: Separation of Concerns

Example: I4WeatherMon

Example: eCos

AOP

3.2 Tutorial: AspectC++

3.3 Summary and Outlook

3.4 References

### Agenda

The **embedded Configurable operating system**

- Operating system for embedded applications
- Open source, maintained by eCosCentric Inc.
- Many 16-bit and 32-bit platforms supported
- Broadly accepted real-world system

- More than **750** configuration options (kernel)
- Feature-based selection
- **Preprocessor-based** implementation
Aspect-Oriented Programming (AOP) [2]

> AOP is about modularizing crosscutting concerns

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

Static Configurability with the CPP?

Kernel policies:

- Tracing
- Instrumentation
- Synchronization

Mutex options:

- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

Cyg_Mutex::Cyg_Mutex() {
    locked = false;
    owner = NULL;
    protocol = DYNAMIC;
    ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
}

Instrumentation

Issue

Crosscutting Concerns

Mutex options:

- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

Kernel policies:

- Tracing
- Instrumentation
- Synchronization

Issue

Crosscutting Concerns

Synchronization
AOP: The Basic Idea

Separation of **what** from **where**:

- **Join Points** \(\mapsto\) **where**
  - positions in the static structure or dynamic control flow (event)
  - given declaratively by pointcut expressions

- **Advice** \(\mapsto\) **what**
  - additional elements (members, ...) to introduce at join points
  - additional behavior (code) to superimpose at join points

---

### Static Configurability with the CPP?

**Kernel policies:**
- Synchronization
- Instrumentation
- Tracing

**Result** after refactoring into aspects [4]

---

### Implementation of Crosscutting Concerns with AOP

**Aspect** `int sync {`  
**Pointcut** `sync() = execution(...) // kernel calls to sync`  
**Advice** `sync(): before() {`  
- `Cyg.Scheduler::lock();`  
- `Cyg.Scheduler::unlock();`  
`}`  
**Advice** `sync(): after() {`  
- `Cyg.Scheduler::lock();`  
- `Cyg.Scheduler::unlock();`  
`}`  

// In eCos, a new thread always starts with a lock value of 0  
**Advice** `execute(  
"Cyg.HardwareThread::thread::entry(...);" : before() {  
  Cyg.Scheduler::zero.sched.lock();  
}`  

---
AspectC++

AspectC++ is an AOP language extension for C++
- superset of ISO C++ 98 [1]
  ~ every C++ program is also an AspectC++ program
- additionally supports AOP concepts

Technical approach: source-to-source transformation
- ac++ weaver transforms AspectC++ code into C++ code
- resulting C++ code can be compiled with any standard-compliant compiler (especially gcc)
- ag++ weaver wrapper works as replacement for g++ in makefiles

Language and weaver are open source (GPL2)

http://www.aspectc.org

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
- Example Scenario
  - First Steps And Language Overview
  - Advanced Concepts
  - Weaver Transformations
  - Further Examples
3.3 Summary and Outlook
3.4 References

Scenario: A Simple Queue

The Simple Queue Class

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

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

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

Scenario: The Problem

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

Scenario: 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();
    } catch (...) {
      lock.leave(); throw;
    }
    lock.leave();
    return res;
  }
  int count() { return counter; }
}; // class Queue
```
Goal: A configurable Queue

Queue

Operations: enqueue(), dequeue()

Counting: count()

Thread safety: Thread safety

Error propagation: Error propagation

Return codes: Return codes

Exceptions: Exceptions

---

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
  - Example Scenario
  - First Steps And Language Overview
  - Advanced Concepts
  - Weaver Transformations
  - Further Examples
3.3 Summary and Outlook
3.4 References

---

Queue: Element Counting

Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

---

Configuring with the Preprocessor?

```cpp
class Queue {
    int first, last;
    int counter;
};

#define LOCKING_ASPECT
    {类 lock enter()
        #define COUNTING_ASPECT
            (counter > 0) --counter;
    #define ERRORHANDLING_ASPECT
        if (true) throw (exception);
;
    #define COUNTING_ASPECT
        if (counter > 0) --counter;
    #define ERRORHANDLING_ASPECT
        if (true) throw (exception);

    void enqueue(int time) {
        LOCKING_ASPECT
            lock enter()
        COUNTING_ASPECT
            counter += 1;
        ERRORHANDLING_ASPECT
            if (true) throw (exception);
    }
}
```

Please extend the Queue class by an element counter!
Queue: Element Counting

Element counting: The Idea

- Increment a counter variable after each execution of `util::Queue::enqueue()`
- Decrement it after each execution of `util::Queue::dequeue()`

ElementCounter1

```cpp
aspect ElementCounter1 {
    int counter;
    ElementCounter1() {
        counter = 0;
    }

    advice execution("% util::Queue::enqueue(...)") : after() {
        ++counter;
        printf( " AspectElementCounter: # of elements = %d\n", counter );
    }

    advice execution("% util::Queue::dequeue(...)") : after() {
        if( counter > 0 ) --counter;
        printf(" AspectElementCounter: # of elements = %d\n", counter );
    }
};
```

ElementCounter1 - Elements

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.

```cpp
aspect ElementCounter1 {
    int counter;
    ElementCounter1() {
        counter = 0;
    }

    advice execution("% util::Queue::enqueue(...)") : after() {
        ++counter;
        printf( " AspectElementCounter: # of elements = %d\n", counter );
    }

    advice execution("% util::Queue::dequeue(...)") : after() {
        if( counter > 0 ) --counter;
        printf(" AspectElementCounter: # of elements = %d\n", counter );
    }
};
```
Queue: Element Counting

ElementCounter1 - Elements

```cpp
aspect ElementCounter {
    int counter;
    ElementCounter() { counter = 0; }
}

advice execution("\% util::Queue::enqueuel(...)") : after() {
    ++counter;
    printf(" Aspect ElementCounter: # of elements = \%d\n", counter);
}
advice execution("\% util::Queue::dequeuel(...)") : after() {
    if( counter > 0 ) --counter;
    printf(" Aspect ElementCounter: # of elements = \%d\n", counter);
}
}
```

ElementCounter1.ah

---

Queue: Element Counting

ElementCounter1 - Elements

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

ElementCounter1.ah

---

Queue: Element Counting

ElementCounter1 - Result

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

main.cc
Queue: Element Counting

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

---

Queue: Element Counting

ElementCounter2

aspect ElementCounter
{
  advice "util::Queue" : slice {  
    int counter;
    public:
    int count() const { return counter; }
  };
  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.ah

---

Queue: Element Counting

ElementCounter2 - Elements

aspect ElementCounter
{
  advice "util::Queue" : slice {  
    int counter;
    public:
    int count() const { return counter; }
  };
  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.ah

---

Queue: Element Counting

ElementCounter2 - Elements

aspect ElementCounter
{
  advice "util::Queue" : slice {  
    int counter;
    public:
    int count() const { return counter; }
  };
  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.ah

---

Queue: Element Counting

ElementCounter2 - Elements

aspect ElementCounter
{
  advice "util::Queue" : slice {  
    int counter;
    public:
    int count() const { return counter; }
  };
  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.ah

---
Queue: Element Counting

### ElementCounter2 - Elements

**Aspect: ElementCounter**
```
aspect ElementCounter {
  advice "util::Queue": slice class {
    int counter;
    public:
    int count() const { return counter; }
  };
  advice execution("% util::Queue::enqueue(...)")
    if (queue.count() > 0) --queue.counter;
    printf(" AspectElementCounter: # of elements = %d\n", queue.count());

  advice execution("% util::Queue::dequeue(...)")
    if (queue.count() > 0) --queue.counter;
    printf(" AspectElementCounter: # of elements = %d\n", queue.count());

  advice construction("util::Queue")
    & that(queue): before (util::Queue& queue) {
      queue.counter = 0;
    }
};
```

---

Queue: Element Counting

### ElementCounter2 - Elements

**Aspect: ElementCounter**
```
aspect ElementCounter {
  advice "util::Queue": slice class {
    int counter;
    public:
    int count() const { return counter; }
  };
  advice execution("% util::Queue::enqueue(...)")
    & that(queue): after (util::Queue& queue) {
      ++queue.counter;
      printf(" AspectElementCounter: # of elements = %d\n", queue.count());
    }

  advice execution("% util::Queue::dequeue(...)")
    & that(queue): after (util::Queue& queue) {
      --queue.counter;
      printf(" AspectElementCounter: # of elements = %d\n", queue.count());
    }

  advice construction("util::Queue")
    & that(queue): before (util::Queue& queue) {
      queue.counter = 0;
    }
};
```

---

Queue: Element Counting

### ElementCounter2 - Result
```
main.cc

```
Queue: Element Counting

**ElementCounter2 - Result**

```c
int main() {
    util::Queue queue;
    printf("main(): Queue contains %d items\n", queue.count());
    queue.enqueue(new util::Item);
    printf("main(): Queue contains 1 item\n");
    item = queue.dequeue();
    printf("main(): Queue contains 0 items\n");
    printf("main(): dequeuing one item\n");
    queue.enqueue(new util::Item);
    item = queue.dequeue();
    printf("main(): Queue contains 2 items\n");
    item = queue.dequeue();
    printf("main(): Queue contains 1 items\n");
    item = queue.dequeue();
    printf("main(): Queue contains 0 items\n");
}
```

main.cc

Queue: Element Counting

**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

AspectC++ Language Elements

**Syntactic Elements**

- `aspect name`
- `pointcut expression`
- `advice type`

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

ElementCounter1.ah

```
```

AspectC++ Language Elements

**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(...)")`

Advice

- 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

- A slice of additional methods, types, etc. is added to the class
- Can be used to extend the interface of a class
Queue: Error Handling

Error Handling: 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

Error Exception

```c
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();
    });
};
```
Exception - Elements

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

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();
    }
};  // aspectErrorException
```

Exception - Elements

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

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();
    }
};  // aspectErrorException
```

Exception – 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: Thread Synchronization

Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

---

Queue: Thread Synchronization

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": slice class [ 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();
        };
}

LockingMutex.ah

---

LockingMutex - Elements

aspect LockingMutex {
    advice "util::Queue": slice class [ 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();
        };
}

LockingMutex.ah
LockingMutex - Elements

```cpp
aspect LockingMutex {
    advice "util::Queue" : slice class { os::Mutex lock; };

    pointcut sync_methods() = "% util::Queue::%queue(...)";

    advice execution(sync_methods()) & that(queque)
        : around( util::Queue & queue ) {
            queue.lock.enter();
            try {
                tsp->proceed();
            } catch(...) {
                queue.lock.leave();
                throw;
            }
            queue.lock.leave();
        }
};
```

Pointcuts can be named. `sync_methods` describes all methods that have to be synchronized by the mutex.

```
aspect LockingMutex {
    advice "util::Queue" : slice class { os::Mutex lock; };

    pointcut sync_methods() = "% util::Queue::%queue(...)";

    advice execution(sync_methods()) & that(queque)
        : around( util::Queue & queue ) {
            queue.lock.enter();
            try {
                tsp->proceed();
            } catch(...) {
                queue.lock.leave();
                throw;
            }
            queue.lock.leave();
        }
};
```

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

LockingMutex - Elements

```
aspect LockingMutex {
    advice "util::Queue" : slice class { os::Mutex lock; };

    pointcut sync_methods() = "% util::Queue::%queue(...)";

    advice execution(sync_methods()) & that(queque)
        : around( util::Queue & queue ) {
            queue.lock.enter();
            try {
                tsp->proceed();
            } catch(...) {
                queue.lock.leave();
                throw;
            }
            queue.lock.leave();
        }
};
```

LockingMutex - Lessons Learned

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 `tsp->proceed()`
- use wildcards in match expressions
  - "% util::Queue::%queue(...)" matches both `enqueue()` and `dequeue()`
Queue: IRQ Synchronization

Queue: A new Requirement

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

IV. Interrupt safety
   (synchronization on interrupt level)

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

aspect LockingIRQ1

```cpp
aspect LockingIRQ

pointcut sync_methods() = "% util::Queue::%queue(...)";
pointcut kernel_code() = "% kernel::%(...)";

advice call(sync_methods()) && !within(kernel_code()) : around() {
  os::disable_int();
  try {
    tjp->proceed();
  } catch (...) {
    os::enable_int();
    throw;
  }
  os::enable_int();
}
```

LockingIRQ1.ah

aspect LockingIRQ1 – Elements

```cpp
aspect LockingIRQ

pointcut sync_methods() = "% util::Queue::%queue(...)";
pointcut kernel_code() = "% kernel::%(...)";

advice call(sync_methods()) && !within(kernel_code()) : around() {
  os::disable_int();
  try {
    tjp->proceed();
  } catch (...) {
    os::enable_int();
    throw;
  }
  os::enable_int();
}
```

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

LockingIRQ1.ah
Queue: IRQ Synchronization

**LockingIRQ1 – Elements**

```java
aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::queue(...);"
  pointcut kernel_code() = "% kernel::%(...);";
  advice call(sync_methods()) & within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
    }
  }
};
```

This pointcut expression matches any call to a `sync_method` that is not done from `kernel_code`

---

**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");
    queue.enqueue( new util::Item );
    do_something();
  }
}
```

```
int main() {
  printf("main\n");
  queue.enqueue( new util::Item );
  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::%(...);";
  advice execution(sync_methods()) & within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
    }
    os::enable_int();
};
```

**Solution**

Using the `cflow` pointcut function

---

Queue: IRQ Synchronization

**LockingIRQ2 – 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");
    queue.enqueue( new util::Item );
    do_something();
  }
}
```

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

```
main.cc
```

---

Queue: IRQ Synchronization
Queue: IRQ Synchronization

LockingIRQ2 – Elements

```cpp
aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::enqueue(...)";
  pointcut kernel_code() = "% kernel::...(...)";
  advice execution(sync_methods()) && !flow(execution(kernel_code())) : around {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
      throw;
    }
    os::enable_int();
  };
}
```

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.

A First Summary

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
Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
   Example Scenario
   First Steps And Language Overview
   Advanced Concepts
   Weaver Transformations
   Further Examples
3.3 Summary and Outlook
3.4 References

Advanced Concepts

The Joinpoint API

- Inside an advice body, the current joinpoint context is available via the implicitly passed \texttt{tjp} variable:
  
  
  \begin{verbatim}
  advice ... { 
  struct JoinPoint { 
  ... 
  } tjp: // implicitly available in advice code 
  ... 
  }
  
  \end{verbatim}

- You have already seen how to use \texttt{tjp}, to ...
  - execute the original code in around advice with \texttt{tjp->proceed()}

- The joinpoint API provides a rich interface
  - to expose context independently of the aspect target
  - this is especially useful in writing \textit{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

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

```cpp
#include "lockingA.h"
aspect LockingQueue : public LockingA {
  pointcut sync_classes() = "util::Queue";
  pointcut sync_methods() = "% util::Queue::Queue(...)";
};
```

```cpp
#include "lockingA.h"
aspect LockingQueue : public LockingA {
  advice call(...) : after () {
    cout << *tjp->result();
  }
};
```

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

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

```cpp
... operator <<(..., int)
... operator <<(..., long)
... operator <<(..., bool)
... operator <<(..., Foo)
```
Advanced Concepts

Generic Advice

- Uses static JP-specific type information in advice code
- in combination with C++ overloading
- Resolves to the **statically typed** return value
- no runtime type checks are needed
- unhandled types are detected at compile-time
- functions can be inlined

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

Advanced Concepts

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;
    }
}
```

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

Agenda

- 3.1 Motivation: Separation of Concerns
- 3.2 Tutorial: AspectC++
  - Example Scenario
  - First Steps And Language Overview
  - Advanced Concepts
  - Weaver Transformations
  - Further Examples
- 3.3 Summary and Outlook
- 3.4 References
Weaver Transformations

Aspect Transformation

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

Class Transform {
    static Transform __instance;
    // ...
    void __a0 before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
        printf(tjp->signature());
    }
};

Advice becomes a member function

Aspects are transformed into ordinary classes

One global aspect instance is created by default
Weaver Transformations

Aspect Transformation

```cpp
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

Class Transform {
    static Transform __instance;
    // ...
    void __d0 before () {
        printf ("before foo call\n");
    }
    template<class JoinPoint>
    void __d1 after (JoinPoint *tjp) {
        printf (tjp->signature () );
    }
};

Joinpoint Transformation

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

```cpp
struct __call_main_0_0 {
    static inline void invoke (){
        AC::...a0 before ();
        ::foo();
    }
};
__call_main_0_0::invoke ()
    return 0;
```
Weaver Transformations

Translation Modes

- **Whole Program Transformation-Mode**
  - e.g. `act++ -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. `act++ -c a.cc -o a-gen.cc -p`
  - easier integration into build processes

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
- Example Scenario
- First Steps And Language Overview
- Advanced Concepts
- Weaver Transformations
- Further Examples
3.3 Summary and Outlook
3.4 References

Further Examples

Observer Pattern: Scenario

```
DigitalClock
Draw()

AnalogClock
Draw()

ClockTimer
GetHour() : int
SetTime(h : int, in m : int, in s : int)
Tick()
```

Observer Pattern: Implementation

```
IObserver
update(in s : ISubject)

iSubject
updateAll()

DigitalClock
Draw()
update(in s )

AnalogClock
Draw()
update(in s )

ClockTimer
GetHour() : int
SetTime(h : int, in m : int, in s : int)
Tick()
```
Further Examples

Observer Pattern: Problem

The 'Observer Protocol' Concern...

```
IObservable
update (i : ISubject)
```

```
ISubject
updateAll ()
```

```
DigitalClock
Draw() i
update (in s : ISubject)
```

```
AnalogClock
Draw() i
```

```
ClockTimer
GetHour() : int
SetTime(i : int, in m : int, in s : int)
Tick() i
```

...crosscuts the module structure

Further 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 () : slice class : public ObserverPattern::IObservable;
        advice subjects() : slice class : public ObserverPattern::ISubject;

        advice subjectChange (): after () {
            ISubject* subject = tsp->that();
            updateObservers( subject );
        }

        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void remObserver( ISubject* subject, IObserver* observer ) { ... }
}
```

Further 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 () : slice class : public ObserverPattern::IObservable;
        advice subjects() : slice class : public ObserverPattern::ISubject;

        advice subjectChange (): after () {
            ISubject* subject = tsp->that();
            updateObservers( subject );
        }

        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void remObserver( ISubject* subject, IObserver* observer ) { ... }
}
```

Further 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 () : slice class : public ObserverPattern::IObservable;
        advice subjects() : slice class : public ObserverPattern::ISubject;

        advice subjectChange (): after () {
            ISubject* subject = tsp->that();
            updateObservers( subject );
        }

        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void remObserver( ISubject* subject, IObserver* observer ) { ... }
}
```

Further 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 () : slice class : public ObserverPattern::IObservable;
        advice subjects() : slice class : public ObserverPattern::ISubject;

        advice subjectChange (): after () {
            ISubject* subject = tsp->that();
            updateObservers( subject );
        }

        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void remObserver( ISubject* subject, IObserver* observer ) { ... }
}
```
Solution: Generic Observer Aspect

```cpp
aspect ObserverPattern {
  ...
  public:
    struct ISubject {};
    struct IObserver {
      virtual void update(ISubject *) = 0;
    };
    pointcut virtual observers() = 0;
    pointcut virtual subjects() = 0;
    pointcut virtual subjectChange() = execution("% ...:%(....)") & & within(subjects());

    advice observers() : slice class : public ObserverPattern::IObserver;
    advice subjects() : slice class : public ObserverPattern::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: Generic Observer Aspect

```cpp
aspect ObserverPattern {
  ...
  public:
    struct ISubject {};
    struct IObserver {
      virtual void update(ISubject*) = 0;
    };
    pointcut virtual observers() = 0;
    pointcut virtual subjects() = 0;
    pointcut virtual subjectChange() = execution("% ...:%(....)") & & within(subjects());

    advice observers() : slice class : public ObserverPattern::IObserver;
    advice subjects() : slice class : public ObserverPattern::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: Generic Observer Aspect

```
After advice to update observers after execution of a state-changing method
```

Solution: Putting Everything Together

```cpp
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() : slice class {
      public:
        void update(ObserverPattern::ISubject* s) {
          Draw();
        }
    }
};
```

Further Examples

```
Introduction of the role interface as additional baseclass into subjects / observers
```

Further Examples

```
Further Examples
```
Further Examples

Errorhandling in Legacy Code: Scenario

```c
RESULT WINAPI WinProc( HANDLE 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, WinProc, 0, 0, 0, "Example", RegisterClass( &wc );
    HWND hWndMain = CreateWindowEx( 0, "Example", "Example", ... );
    UpdateWindow( hWndMain );
    MSG msg;
    while( GetMessage( &msg, NULL, 0, 0 ) ) {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
    }
    return 0;
  }
}
```

Further Examples

Win32 Errorhandling: Goals

- Detect failed calls of Win32 API functions
  - by giving 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>

Further Examples

Detecting the failure: Generic Advice

```
advice call(win32API()) {
  after () {
    if (isError( "*tjp->result()")
      // throw an exception
  }
}
```
Further Examples

Describing the failure: Generative Advice

```cpp
template <int I> struct ArgPrinter {
    template <class JP> static void work (JP &jp, ostream &s) {
        ArgPrinter<I-1>::work (jp, s);
    }
};

advice call(win32API) () : after () {
    if (throw an exception
        osringStream s;
        DWORD code = GetLastError();
        s << "WIN32 ERROR: " << code << ...
        << JP>signature () << "WITH: " << ...;
        ArgPrinter<JoinPoint::ARGS>::work (*jp, s);
    throw win32::Exception (s.str());
}
```

Further Examples

Reporting the Error

```cpp
LRESULT WINAPI WndProc (HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam) 
{
    switch (nMsg) {
        case WM_CREATE:
            s = "CreateWindow("; 
            h = CreateWindowEx (0, "Example", "Example", ...
            hWnd, hWnd); 
            break;
        ...
    }
}
```

Agenda

- 3.1 Motivation: Separation of Concerns
- 3.2 Tutorial: AspectC++
- 3.3 Summary and Outlook
- 3.4 References

Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
  - Avoidance of code tangling
    - implementation of optional features
  - Avoidance of code scattering
    - implementation of nonfunctional features

- Basic idea: separation of what from where
  - Join Points  where
    - positions in the static structure or dynamic control flow (event)
    - given declaratively by pointcut expressions
  - Advice     what
    - additional elements (members, ...) to introduce at join points
    - additional behavior (code) to superimpose at join points

- AspectC++ brings AOP concepts to the C++ world
  - Static source-to-source transformation approach
Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
  - Avoidance of code tangling
    → implementation of optional features
  - Avoidance of code scattering
    → implementation of nonfunctional features
- Basic idea: separation of what from where
  - **Join Points**  ↦→ **where**
    - positions in the static structure or dynamic control flow (event)
    - given declaratively by pointcut expressions
  - **Advice**  ↦→ **what**
    - additional elements (members, ...) to introduce at join points
    - additional behavior (code) to superimpose at join points
- AspectC++ brings AOP concepts to the C++ world
  - Static source-to-source transformation approach

Next Lecture:
How to use AOP to achieve loose coupling, granularity and variability for feature implementations in configurable system software

Aspect-aware design

References


AspectC++ Quick Reference

Concepts

aspect
Aspects in AspectC++ implement in a modular way cross-cutting concerns and are an extension to the class concept of C++. Additionally to attributes and methods, aspects may also contain advice declarations.

advice
An advice declaration is used either to specify code that should run when the join points specified by a pointcut expression are reached or to introduce a new method, attribute, or type to all join points specified by a pointcut expression.

slice
A slice is a fragment of a C++ element like a class. It may be used by introduction advice to implemented static extensions of the program.

join point
In AspectC++ join points are defined as points in the component code where aspects can interfere. A join point refers to a method, an attribute, a type (class, struct, or union), an object, or a point from which a join point is accessed.

pointcut
A pointcut is a set of join points described by a pointcut expression.

pointcut expression
Pointcut expressions are composed from match expressions used to find a set of join points, from pointcut functions used to filter or map specific join points from a pointcut, and from algebraic operators used to combine pointcuts.

match expression
Match expressions are strings containing a search pattern.

order declaration
If more than one aspect affects the same join point an order declaration can be used to define the order of advice code execution.

Aspects

Writing aspects works very similar to writing C++ class definitions. Aspects may define ordinary class members as well as advice.

aspect A {...};
defines the aspect A
aspect A: public B {...};
A inherits from class or aspect B

Advice Declarations

advice pointcut : before(...) {...}
the advice code is executed before the join points in the pointcut
advice pointcut : after(...) {...}
the advice code is executed after the join points in the pointcut
advice pointcut : around(...) {...}
the advice code is executed in place of the join points in the pointcut
advice pointcut : order(high, ...low);
high and low are pointcuts, which describe sets of aspects. Aspects on the left side of the argument list always have a higher precedence than aspects on the right hand side at the join points, where the order declaration is applied.

advice pointcut : slice class : public Base {...}
introduces a new base class Base and members into the target classes matched by pointcut.

advice pointcut : slice ASlice;
introduces the slice ASlice into the target classes matched by pointcut.

Pointcut Expressions

Type Matching

"int"
matches the C++ built-in scalar type int
"% *"
matches any pointer type

Namespace and Class Matching

"Chain"
matches the class, struct or union Chain
"Memory%"
matches any class, struct or union whose name starts with “Memory”

Function Matching

"void reset()"
matches the function reset having no parameters and returning void
"% printf(...)"
matches the function printf having any number of parameters and returning any type
"% . . . ::%(...)"
matches any function, operator function, or type conversion function (in any class or namespace)

Template Matching

"% std::set<...>"
matches all template instances of the class std::set
"% std::set<int>"
matches only the template instance std::set<int>
"% . . . ::<...>::%(...)"
matches any member function from any template class instance in any scope

Predefined Pointcut Functions

Functions

call(pointcut) \( \text{N} \rightarrow \text{C}_{\text{f}} \)
provides all join points where a named entity in the pointcut is called.

execution(pointcut) \( \text{N} \rightarrow \text{C}_{\text{f}} \)
provides all join points referring to the implementation of a named entity in the pointcut.

construction(pointcut) \( \text{N} \rightarrow \text{C}_{\text{con}} \)
all join points where an instance of the given class(es) is constructed.

destruction(pointcut) \( \text{N} \rightarrow \text{C}_{\text{con}} \)
all join points where an instance of the given class(es) is destructed.

pointcut may contain function names or class names. A class name is equivalent to the names of all functions defined within its scope combined with the :: operator (see below).

Control Flow

cflow(pointcut) \( \text{C} \rightarrow \text{C}_{\text{C}} \)
captures join points occurring in the dynamic execution context of join points in the pointcut. The argument pointcut is forbidden to contain context variables or join points with runtime conditions (currently cflow, that, or target).

Types

base(pointcut) \( \text{N} \rightarrow \text{C}_{\text{r}} \)
returns all base classes resp. redefined functions of classes in the pointcut

derived(pointcut) \( \text{N} \rightarrow \text{C}_{\text{r}} \)
returns all classes in the pointcut and all classes derived from them resp. all redefined functions of derived classes
Scope

within(pointcut) N→C
filters all join points that are within the functions or classes in the pointcut

Context

that(type pattern)
returns all join points where the current C++ this pointer refers to an object which is an instance of a type that is compatible to the type described by the type pattern

target(type pattern)
returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the type pattern

result(type pattern)
returns all join points where the result object of a call/execution is an instance of a type described by the type pattern

args(type pattern, ...)
a list of type patterns is used to provide all join points with matching argument signatures

Instead of the type pattern it is possible here to pass the name of a context variable to which the context information is bound. In this case the type of the variable is used for the type matching.

Algebraic Operators

pointcut &amp; pointcut
intersection of the join points in the pointcuts

pointcut || pointcut
union of the join points in the pointcuts

! pointcut
exclusion of the join points in the pointcut

JoinPoint-API

The JoinPoint-API is provided within every advice code body by the built-in object \texttt{JoinPoint} of class \texttt{JoinPoint}.

Compile-time Types and Constants

That
object type (object initiating a call)

Target
target object type (target object of a call)

Result
type of the object, which is used to store the result of the affected function

Res::Type, Res::ReferredType
result type of the affected function

Arg<i>::Type, Arg<i>::ReferredType [type]
type of the \textit{i}-th argument of the affected function (with \(0 \leq i < \text{ARGS}\))

ARGs [const]
number of arguments

JPID [const]
unique numeric identifier for this join point

JPTYPE [const]
umERIC identifier describing the type of this join point (\texttt{AC::CALL}, \texttt{AC::EXECUTION}, \texttt{AC::CONSTRUCTION}, or \texttt{AC::DESTRUCTION})

Runtime Functions and State

static const char *signature() gives a textual description of the join point (function name, class name, ...) static const char *filename() returns the name of the file in which the joinpoint shadow is located static int line() the source code line number that is associated with the joinpoint shadow

That *that() returns a pointer to the object initiating a call or 0 if it is a static method or a global function

Target *target() returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

Result *result() returns a typed pointer to the result value or 0 if the function has no result value

Arg<i>:Argtype *arg<i>() returns a typed pointer to the \textit{i}-th argument value (with \(0 \leq i < \text{ARGS}\))

void *arg(int i) returns a pointer to the memory position holding the argument value with index \textit{i}

void proceed() executes the original code in an around advice

AC::Action &action() returns the runtime action object containing the execution environment to execute (\texttt{trigger}) the original code encapsulated by an around advice

Runtime Type Information

static AC::Type resulttype() returns a C++ ABI V3 conforming string representation of the result type / argument type of the affected function

Example

A reusable tracing aspect.

aspect Trace {
  pointcut virtual functions() = 0;
  advice execution(functions()) : around() {
    cout << "before " << JoinPoint::signature() << "("
    for (unsigned i = 0; i < JoinPoint::ARGS; ++i)
      cout << (i ? ", " : ")" << JoinPoint::argtype(i);
    cout << ")" << endl;
    tjp->proceed();
    cout << "after " << endl;
  }
}

In a derived aspect the pointcut \textit{functions} may be redefined to apply the aspect to the desired set of functions.

aspect TraceMain : public Trace {
  pointcut functions() = "% main(...)";
};

This is a reference sheet corresponding to AspectC++ 1.0.

Version 1.12, April 18, 2011.

(c) Copyright 2011 pure-systems GmbH, Olaf Spinczyk and Daniel Lohmann. All rights reserved.

** support for template instance matching is an experimental feature

†† http://www.codesourcery.com/cxx-abi/abi.html#mangling

†‡‡ C, C++, C++11, C++14, C++17: Code (any, only \texttt{Class}, only \texttt{Function}, only \texttt{Type})

‡ Cons support for template instance matching is an experimental feature

alone support for template instance matching is an experimental feature

<table>
<thead>
<tr>
<th>Result</th>
<th>Exclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>(C,C)</td>
</tr>
<tr>
<td>N</td>
<td>C</td>
</tr>
</tbody>
</table>