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 15 – 2015-04-28

http://www4.informatik.uni-erlangen.de/Lehre/SS15/V_KSS

About this Lecture

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
Agenda

3.1 Motivation: Separation of Concerns
   Example: I4WeatherMon
   Example: eCos

3.2 Tutorial: AspectC++

3.3 Summary and Outlook

3.4 References

Case Study eCos [4]

- 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

Static Configurability with the CPP?

I4WeatherMon (CPP): Implementation (Excerpt)

Example: I4WeatherMon
example from last lecture
**Aspect-Oriented Programming (AOP)**

- AOP is about modularizing crosscutting concerns
  - well modularized concern
  - badly modularized

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

**Static Configurability with the CPP?**

```cpp
Cyg_Mutex::Cyg_Mutex() {
    CYG_REPORT_FUNCTION();
    locked = false;
owner = NULL;
}
```

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

**Issue**

Crosscutting Concerns

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

Implementation of Crosscutting Concerns with AOP

```java
aspect int sync {
  pointcut sync() = execution(...) // kernel calls to sync
     || construction(...) || destruction(...);
// advise kernel code to invoke lock() and unlock()
advice sync() : before() {
  Cfg.Scheduler::lock();
}
advice sync() : after() {
  Cfg.Scheduler::unlock();
}
}
```

Static Configurability with the CPP?

Kernel policies:
- Synchronization

Crosscutting Concerns
Can we do better with aspects?

Result
after refactoring into aspects [4]
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

Scenario: A Simple Queue

```cpp
namespace util {
  class Item {
    friend class Queue;
    Item* next;
    public:
      Item() : next(nullptr) {}
  };

  class Queue {
    Item* first;
    Item* last;
    public:
      Queue() : first(nullptr), last(nullptr) {}
      void enqueue(Item* item) {
        // implement enqueue
      };
    Item* dequeue() {
      // implement dequeue
    }
  };
}
```

The Simple Queue Class
Scenario: A Simple Queue

What Code Does What?

```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();
    }
}
```

```cpp
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;
}
```

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
Goal: A configurable Queue

Queue

Operations
enqueue() dequeue()

Counting
count()

Thread safety

Error propagation

Return codes

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

I. Element counting

II. Error handling
(signaling of errors by exceptions)

III. Thread safety
(synchronization by mutex variables)
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()`

```
aspect ElementCounter1 {
    int counter;
    ElementCounter1() {
        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

Queue: Element Counting

ElementCounter1 - Elements

We introduced a new aspect named `ElementCounter`. An aspect starts with the keyword `aspect` and is syntactically much like a class.

```
aspect ElementCounter1 {
    int counter;
    ElementCounter1() {
        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

Queue: Element Counting

ElementCounter1 - Elements

Like a class, an aspect can define data members, constructors and so on.

```
aspect ElementCounter1 {
    int counter;
    ElementCounter1() {
        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
### Queue: Element Counting

#### ElementCounter1 - Elements

```c
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

```c
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();
}
```

---

#### ElementCounter1.ah
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

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(" 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;
};

---

Queue: Element Counting

ElementCounter2

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(" 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;
};

---

Queue: Element Counting

ElementCounter2 - Elements

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(" 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;
};

---

Queue: Element Counting

ElementCounter2 - Elements

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(" 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;
};

---
Queue: Element Counting

**ElementCounter2 - Elements**

```cpp
aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
        public:
            int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)") {
        ++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") {
        queue.counter = 0;
    };
};
```

---

**ElementCounter - Result**

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

---

© dl KSS (VL 3 | SS 15) 3 AOP | 3.2 Tutorial: AspectC++ 3–38

© dl KSS (VL 3 | SS 15) 3 AOP | 3.2 Tutorial: AspectC++ 3–39

© dl KSS (VL 3 | SS 15) 3 AOP | 3.2 Tutorial: AspectC++ 3–40

© dl KSS (VL 3 | SS 15) 3 AOP | 3.2 Tutorial: AspectC++ 3–41
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 %d items\n", queue.count());
    queue.dequeue();
    printf("main(): Queue contains %d items\n", queue.count());
}
```

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

```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`
**AspectC++ Language Elements**

### 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

#### 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

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

---

### Before / After Advice

#### with execution joinpoints:

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

```
int main() {
    printf("main()\n");
    ClassA a;
    a.foo();
    tjp->proceed();
}
```

#### with call joinpoints:

```
int main() {
    printf("main()\n");
    a = new ClassA;
    a->foo();
    tjp->proceed();
}
```

---

### Around Advice

#### with execution joinpoints:

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

```
int main() {
    printf("main()\n");
    ClassA a;
    a = new ClassA;
    a->foo();
    tjp->proceed();
}
```

#### with call joinpoints:

```
int main() {
    printf("main()\n");
    a = new ClassA;
    a->foo();
    tjp->proceed();
}
```
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

```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();
            }
        }
};
```
Queue: Error Handling

**Error Exception - 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();
    };
}
```

---

Queue: Error Handling

**Error Exception - 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();
    };
}
```

---

Queue: Error Handling

**Error 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)

---

LockingMutex

```c++
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

---

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 - Elements

```c++
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();
};
```

We introduce a mutex member into class Queue

LockingMutex.ah
LockingMutex - Elements

```cpp
aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = "% util::Queue::xqueue(...)";
  advice execution(sync_methods()) & that(queue) : around( util::Queue & queue ) {
    queue.lock.enter();
    try {
      tjp->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::xqueue(...)";
  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.

LockingMutex - Elements

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

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

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 `tjp->proceed()`
- use wildcards in match expressions
  - "% util::Queue::xqueue(...)" 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

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 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
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();
            throw;
        }
        os::enable_int();
    }
}
```

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

LockingIRQ1.ah

Queue: IRQ Synchronization

LockingIRQ1 – Result

```c
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

<Output>

Queue: IRQ Synchronization

LockingIRQ1 – Problem

```c
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

<Output>

Queue: IRQ Synchronization

LockingIRQ2

```java
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();
            throw;
        }
        os::enable_int();
    }
}
```

Solution

Using the `cflow` pointcut function

LockingIRQ2.ah
LockingIRQ2 – Elements

```cpp
aspect LockingIRQ {
    pointcut sync_methods() = "% util::Queue::%queue(...)";
    pointcut kernel_code() = "% kernel::%(...)";
    advice execution(sync_methods()) && fastflow(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.

LockingIRQ2.ah

---

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 aspextual 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

---

Queue: IRQ Synchronization

LockingIRQ – Lessons Learned

- 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

---

Queue: IRQ Synchronization

LockingIRQ2 – Result

```cpp
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

<Output>
### 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 `tjp` variable:
  ```c
  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

---

### Advanced Concepts

#### The Join Point API (Excerpt)

<table>
<thead>
<tr>
<th>Types (compile-time)</th>
<th>Values (runtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>// object type (initiator)</td>
<td>// pointer to the object initiating a call</td>
</tr>
<tr>
<td>That</td>
<td>That* that()</td>
</tr>
<tr>
<td>// object type (receiver)</td>
<td>// pointer to the object that is target of a call</td>
</tr>
<tr>
<td>Target</td>
<td>Target* target()</td>
</tr>
<tr>
<td>// type of the 1st argument of the affected</td>
<td>// pointer to the result value</td>
</tr>
<tr>
<td>Result</td>
<td>Result* result()</td>
</tr>
<tr>
<td>// function with 0 &lt;= i &lt; ARGS</td>
<td>// typed pointer the 1st argument value of a</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::Type</td>
<td>Arg&lt;i&gt;::TypedRef arg()</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::RefereeType</td>
<td>// pointer the 7th argument value of a</td>
</tr>
<tr>
<td>// function call (compile-time index)</td>
<td>// function call (runtime index)</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::RefereeType</td>
<td>void* arg(int i)</td>
</tr>
<tr>
<td>// textual representation of the joinpoint</td>
<td>// static cast char* signature()</td>
</tr>
<tr>
<td>// function/class name, parameter types...</td>
<td>// executes the original joinpoint code</td>
</tr>
<tr>
<td>// in an around advice</td>
<td>// in an around advice</td>
</tr>
<tr>
<td>void proceed()</td>
<td>void proceed()</td>
</tr>
<tr>
<td>// returns the runtime action object</td>
<td>// returns the runtime action object</td>
</tr>
<tr>
<td>AC::joinpoint</td>
<td>AC::Action action()</td>
</tr>
</tbody>
</table>
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 an 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
Advanced Concepts

Generic Advice

Uses static JP-specific type information in advice code
  - in combination with C++ overloading
  - template meta-programs
  - no runtime type checks are needed
  - unhandled types are detected at compile-time
  - functions can be inlined

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

Advanced Concepts

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

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;
  }
}
```

Advanced Concepts

Agenda

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

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Weaver Transformations

Aspect Transformation

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

Aspect Transformation

```cpp
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 __a3_after (JoinPoint *tjp) {
        printf (tjp->signature ()�);
    }
};
```

Joinpoint Transformation

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

the function call is replaced by a call to a wrapper function

```cpp
int main() {
    struct __call_main_0_0 {
        static inline void invoke (){
            AC:......a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
```

a local class invokes the advice code for this joinpoint
Weaver Transformations

Translation Modes

- **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

Further Examples

Observer Pattern: Scenario

Observer Pattern: Implementation

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

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 = tgt->that();
        updateObservers(subject);
    }
    void updateObservers( ISubject* subject ) { ... }
    void addObserver( ISubject* subject, IObserver* observer ) { ... }
    void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};

Further Examples
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 subjectChangeEvent = execution("% ...:%(...)") & & within(subjects());
  advice observers() : slice class : public ObserverPattern::IObservable {
    advice subjects() : slice class : public ObserverPattern::ISubject {
      advice subjectChangeEvent : after () {
        ISubject* subject = tjp->that();
        updateObservers(subject);
      }
      void addObserver(ISubject* subject, IObserver* observer) {
      }
      void removeObserver(ISubject* subject, IObserver* observer) {
      }
    };
  }
}
```

Further Examples

Application of the role interface as additional baseclass into subjects / observers

```cpp
aspect ObserverPattern {
  ... public:
  struct ISubject {}
  struct IObserver {
    virtual void update(ISubject *) = 0;
    }
  }
  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;
  pointcut virtual subjectChangeEvent = execution("% ...:%(...)") & & within(subjects());
  advice observers() : slice class : public ObserverPattern::IObservable {
    advice subjects() : slice class : public ObserverPattern::ISubject {
      advice subjectChangeEvent : after () {
        ISubject* subject = tjp->that();
        updateObservers(subject);
      }
      void addObserver(ISubject* subject, IObserver* observer) {
      }
      void removeObserver(ISubject* subject, IObserver* observer) {
      }
    };
  }
}
```

Solution: Putting Everything Together

Applying the Generic Observer Aspect to the clock example

```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

Errorhandling in Legacy Code: Scenario

A typical Win32 application

```cpp
RESULT WINAPI WinProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};
    switch (nMsg) {
        case WM_PAINT:
            dc = BeginPaint(hWnd, &pp);
            // ... EndPaint(hWnd, &pp);
            break;
    }
    int WINAPI WinMain(...)
    HANDLE hConfigFile = CreateFile("example.config", GENERIC_READ, ...
    WNDCLASS wc = {0, WinProc, 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;
    }
}
```

Further Examples

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>

Further Examples

Detecting the failure: Generic Advice

```cpp
advice call(win32API()):
after() {
    bool isError(ATOM);
    bool isError(BOOL);
    bool isError(HANDLE);
    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 &tjp, ostream &s) {
        ArgPrinter<IP-1>::work (tjp.s); 
        s << " \t" << tjp.template arg<1-I>(); 
    }
};
```

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

### Agenda

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

---

Further Examples

### Reporting the Error

```cpp
RESULT WINAPI WndProc( HWND hWnd, U Long NM, WPARAM wParam, LPARAM lParam ) {
    switch (NM) {
        case WM_CREATE:
            hwnd = GetWindow(hWnd, GW_OWNER);
            break;
        ... 
    }
}
```

Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
  - Avoidance of code tangling
  - Avoidance of code scattering
- 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

Referenzen
**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<int>"matches all template instances of the class std::set
"std::set<...>"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)
N → C
provides all join points where a named entity in the pointcut is called.

execution(pointcut)
N → C
provides all join points referring to the implementation of a named entity in the pointcut.

construction(pointcut)
N → C
all join points where an instance of the given class(es) is constructed.

destruction(pointcut)
N → C
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 ll operator (see below).

**Control Flow**

cflow(pointcut)
C → 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)
N → N
returns all base classes resp. redefined functions of classes in the pointcut

derived(pointcut)
N → N
returns all classes in the pointcut and all classes derived from them resp. all redefined functions of derived classes
**Scope**

within(pointcut) \(N \rightarrow C\)

filters all join points that are within the functions or classes in the pointcut

**Context**

that(type pattern) \(N \rightarrow C\)

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) \(N \rightarrow C\)

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) \(N \rightarrow C\)

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, ...) \((N,...)\rightarrow C\)

returns a list of type patterns used to provide all joinpoints 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 & pointcut \((N,N)\rightarrow N, (C,C)\rightarrow C\)

intersection of the join points in the pointcuts

pointcut || pointcut \((N,N)\rightarrow N, (C,C)\rightarrow C\)

union of the join points in the pointcuts

! pointcut \(N\rightarrow N, C\rightarrow C\)

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 \(\text{tp}\) of class JoinPoint.

**Compile-time Types and Constants**

That [type]

object type (object initiating a call)

Target [type]

target object type (target object of a call)

Result [type]

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

Res::Type, Res::ReferredType [type]

result type of the affected function

Arg<i>::Type, Arg<i>::ReferredType [type]

type of the \(i^{th}\) 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]

numeric identifier describing the type of this join point (\(AC::\text{CALL}, AC::\text{EXECUTION}, AC::\text{CONSTRUCTION}\), or \(AC::\text{DESTRUCTION}\))

**Runtime Functions and State**

static const char *signature() [type]

gives a textual description of the join point (function name, class name, ...)

static const char *filename() [type]

returns the name of the file in which the joinpoint shadow is located

static int line() [type]

returns the source code line number that is associated with the joinpoint shadow

That *"that"* [type]

returns a pointer to the object initiating a call or 0 if it is a static method or a global function

Target *"target"* [type]

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"* [type]

returns a typed pointer to the result value or 0 if the function has no result value

Arg<i>::ReferredType *arg<i>[] [type]

returns a typed pointer to the \(i^{th}\) argument value (with \(0 \leq i < \text{ARGS}\))

void *arg(int i) [type]

returns a pointer to the memory position holding the argument value with index \(i\)

void proceed() [type]

executes the original code in an around advice (should be called at most once in around advice)

AC::Action &action() [type]

returns the runtime action object containing the execution environment to execute (trigger()) the original code encapsulated by an around advice

**Runtime Type Information**

static AC::Type retulttype() [type]

returns a C++ ABI V3 conforming string representation of the result type / argument type of the affected function

AC::Type argtype(int i) [type]

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


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