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 12 – 2012-05-23
http://www4.informatik.uni-erlangen.de/Lehre/SS12/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
I4WeatherMon (CPP): Implementation (Excerpt)

```cpp
# ifdef cfWM_PCCON_XML
XMLCon_process();
# endif

// application defined timer interrupt handler
int CiAO::AVRTimer1::tick () {
    ...) // continue with timer-related code
}
```

### 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

### Case Study eCos [4]

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

#### 3.2 Tutorial: AspectC++

#### 3.3 Summary and Outlook

#### 3.4 References
Synchronization

Kernel policies:
- Tracing
- Instrumentation
- Synchronization

Static Configurability with the CPP?

### Mutex options:
- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

#### Kernel policies:
- Tracing
- Instrumentation
- Synchronization

Aspect-Oriented Programming (AOP)

- AOP is about modularizing crosscutting concerns
  - Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

### Issue
Crosscutting Concerns
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

```cpp
aspect int sync {
  pointcut sync() = execution(...) // kernel calls to sync
  || construction(...) || destruction(...);

  // advise kernel code to invoke lock() and unlock()
  advice sync() : before() {
    Cgy.Scheduler::lock();
  }
  advice sync() : after() {
    Cgy.Scheduler::unlock();
  }

  // In eCos, a new thread always starts with a lock value of 0
  advice execution("%Gy.HardwareThread::thread.entry(...)") : before() {
    Cgy.Scheduler::zero.sched.lock();
  }
}
```

Crosscutting Concerns
Can we do better with aspects?
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

```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) {
                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 = nullptr;
                } else {
                    first = first->next;
                    printf("< Queue::dequeue()\n");
                    return res;
                }
            }
            // class Queue
    } // namespace util
```
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 QueueEmptyItemError();  
        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 QueueEmptyItem();  
        }  
      }  
      catch (...) {  
        lock.leave();  
        throw;  
      }  
      lock.leave();  
      return res;  
    }  
    int count() { return counter; }  
  };  
}; // class Queue
```

Scenario: The Problem

Various users of Queue demand extensions:

- Please extend the Queue class by an element counter!
- I want queue to throw exceptions!
- Queue should be thread-safe!
Goal: A configurable Queue

Queue: Element Counting

I. Element counting

II. Error handling (signaling of errors by exceptions)

III. Thread safety (synchronization by mutex variables)

Queue: Demanded Extensions

Please extend the Queue class by an element counter!
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 - Elements**

```cpp
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);
    }
}
```

Like a class, an aspect can define data members, constructors and so on.
Queue: Element Counting

ElementCounter1 - Elements

```cpp
aspect ElementCounter {
    int counter;
    ElementCounter() {
        counter = 0;
    }
    advice execution("% until::Queue::enqueue(...)") : after() {
        ++counter;
        printf(" Aspect ElementCounter: \# of elements = %d\n", counter);
    }
    advice execution("% until::Queue::dequeue(...)") : 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(): enqueueing an item\n");
    queue.enqueue(new util::Item);
    printf("main(): dequeuing two items\n");
    util::Item* item = queue.dequeue();
    item = queue.dequeue();
}
```

main.cc

```
main(): enqueueing an item
> Queue::enqueue(00320FD0)
< Queue::enqueue(00320FD0)
Aspect ElementCounter: \# of elements = 1
main(): dequeuing two items
> Queue::dequeue()
< Queue::dequeue(00320FD0)
Aspect ElementCounter: \# of elements = 0
main(): dequeue()
< Queue::dequeue() returning 00320FD0
Aspect ElementCounter: \# of elements = 0
< Output>
```
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

```cpp
aspect ElementCounter {

advice "util::Queue" : slice class {
    int counter;
    public:
    int count() const { return counter; }
};

advice execution("% util::Queue::enqueue(\ldots)")
    \& that(queue) : after( util::Queue& queue ) {
    ++queue.counter;
    printf( " Aspect ElementCounter: \# of elements = %d\n", queue.count() );
}

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

advice execution("% util::Queue::dequeue(\ldots)")
    \& 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
// We introduce a private counter element and a public method to read it
```
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:enqueu(...)")
        & that(queue) : after( util::Queue& queue ) {
            ++queue.counter;
            printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
        }
    advice execution("% util::Queue:dequeu(...)")
        & 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 - Result

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

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(): enqueuing some items\n");
    queue.enqueue(new util::Item);
    printf("main(): Queue contains 0 items\n");
    printf("main(): dequeuing one item\n");
    item = queue.dequeue();
    printf("main(): Queue contains 2 items\n");
    printf("main(): dequeuing one item\n");
    queue::dequeue();
}
```

```c
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

Syntax Elements

- Aspect name
- Pointcut expression
- Advice type
- Advice body

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

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

Advice to functions

- **before advice**
  - Advice code is executed before the original code
  - Advice code may read/modify parameter values
- **after advice**
  - Advice code is executed after the original code
  - Advice code 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

AspectC++ Language Elements

Before / After Advice

with execution joinpoints:

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

```cpp
advice execution("void ClassA::foo()" : before) {
  printf("%main()\n");
}
```

```cpp
advice execution("void ClassA::foo()" : after) {
  printf("%main()\n");
}
```

with call joinpoints:

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

```cpp
advice call("void ClassA::foo()" : before) {
  printf("%main()\n");
}
```

```cpp
advice call("void ClassA::foo()" : after) {
  printf("%main()\n");
}
```

Around Advice

with execution joinpoints:

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

```cpp
advice execution("void ClassA::foo()" : around) {
  before code
  tjp->proceed()
  after code
}
```

with call joinpoints:

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

```cpp
advice call("void ClassA::foo()" : around) {
  before code
  tjp->proceed()
  after code
}
```
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

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();
  });
}

ErrorException.ah
namespace util {
struct QueueEmptyError {};
}

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

## 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)

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

---

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

---

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

We introduce a mutex member into class Queue
Queue: Thread Synchronization

LockingMutex - Elements

```cpp
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

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

LockingMutex - Elements

```cpp
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

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

Queue: Thread Synchronization

LockingMutex - Elements

```cpp
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

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

Queue: Thread Synchronization

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::%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)

We need Queue to be synchronized 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::enable_int();
    try {
        tjp->proceed();
    } catch(...) {
        os::disable_int();
        throw;
    } os::enable_int();
};

LockingIRQ1.ah

aspect LockingIRQ1 – Elements

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

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

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

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();
  }
}
```

LockingIRQ1.ah

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

---

Queue: IRQ Synchronization

LockingIRQ1 – Problem

```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();
  }
}

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

main.cc

The pointcut within(kernel_code) does not match any indirect calls to sync_methods

---

Queue: IRQ Synchronization

LockingIRQ2

```cpp
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();
  }
}
```

LockingIRQ2.ah

Solution

Using the cflow pointcut function
Queue: IRQ Synchronization

LockingIRQ2 – Elements

```cpp
aspect LockingIRQ {
    pointcut sync_methods() = % util::Queue::enqueue(...);
    pointcut kernel_code() = % kernel::...;
    advice execution(sync_methods()) & if flow(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
```

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>
```

Queue: IRQ Synchronization

LockingIRQ2 – Lessons Learned

You have seen how to ...

- restrict advice invocation to a specific calling context
- use the `with(...)` 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>
```

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
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**

AspectC++: Advanced Concepts

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

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

- **Generic Advice**
  - exploits static type information in advice code

- **Aspect Ordering**
  - allows to specify the invocation order of multiple aspects

- **Aspect Instantiation**
  - allows to implement user-defined aspect instantiation models
Advanced Concepts

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 {
    osi::Mutex lock;
  }
  advice execution(sync_methods()) : around()
    try {
      tjp->that() -> lock.enter();
    } try {
      tjp->proceed();
    } catch (...) {
      tjp->that() -> lock.leave();
    }
}
```

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

Advanced Concepts

Abstract Aspects and Inheritance

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

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

Advanced Concepts

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
aspect TraceService {
  advice call(...) : after()
    ... operator<<(..., int) {
      ... cout << *tjp->result();
    }
    ... operator<<(..., long)
    ... operator<<(..., bool)
    ... operator<<(..., Foo)
}
```
Advanced Concepts

Generic Advice

Uses static JP-specific type information in advice code

- in combination with C++ overloading
- template meta-programs

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(); 
    }
}
```

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

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` pointcut
  - Different aspect orders can be defined for different pointcuts
- 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;
    }
}
```

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

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

```cpp
aspect Transform {

    advice call("%foo()") : before() {
        printf("before foo call\n");
    }

    advice execution("% C::%()") : after() {
        printf(tjp->signature());
    }
};
```

`Transform.ah`

Aspects are transformed into ordinary classes

```cpp
class Transform {

    static Transform __instance;

    // ...
    void __a0 before () {
        printf("before foo call\n");
    }

    template<class JoinPoint>
    void __a0_after (JoinPoint *tjp) {
        printf(tjp->signature());
    }
};
```

`Transform.ah`

Advice becomes a member function

```cpp
class Transform {

    static Transform __instance;

    // ...
    void __a0 before () {
        printf("before foo call\n");
    }

    template<class JoinPoint>
    void __a0_after (JoinPoint *tjp) {
        printf(tjp->signature());
    }
};
```

`Transform.ah`

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

```cpp
Class Transform {
    static Transform __instance;
    ...
    void __a0 before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1 after (JoinPoint *tjp) {
        printf (tjp->signature () );
    }
};
```

Weaver Transformations

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 () {
            ::foo();
            __call_main_0_0::invoke ();
            return 0;
        }
    };
}
```

a local class invokes the advice code for this joinpoint

```cpp
int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            ::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 -DEBUG`
  - 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: Problem

The 'Observer Protocol' Concern...

```
IObserver
update (in s : ISubject)
```

```
ISubject
updateAll()
```

```
DigitalClock
Draw() (in s)
```

```
AnalogClock
Draw() (in s)
```

```
ClockTimer
GetHour() (in int m : int, int n : int)
Tick()
```

...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 = (jโปร: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 = (jโปร:that());
updateObservers(subject);
}
```

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

Further Examples
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 subjectChange() = execution("% ...:=(...)"
        && "% ...:%(...)" && within( subjects() ));
    advice observers () : slice class : public ObserverPattern:IObserver;
    advice subjectObservers() : 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 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 subjectChange() = execution("% ...:=(...)"
        && "% ...:%(...)" && within( subjects() ));
    advice observers () : slice class : public ObserverPattern:IObserver;
    advice subjectObservers() : 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 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 subjectChange() = execution("% ...:=(...)"
        && "% ...:%(...)" && within( subjects() ));
    advice observers () : slice class : public ObserverPattern:IObserver;
    advice subjectObservers() : 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 removeObserver (ISubject* subject, IObserver* observer ) {... }
};
```

Further Examples

Solution: Putting Everything Together

Applying the Generic Observer Aspect to the clock example

```cpp
aspect ClockObserver : public ObserverPattern { 
    /* define the participants 
        pointcut subjects() = "ClockTime"; 
        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

Solution: Putting Everything Together

Applying the Generic Observer Aspect to the clock example

```cpp
aspect ClockObserver : public ObserverPattern { 
    /* define the participants 
        pointcut subjects() = "ClockTime"; 
        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();
        }
    ;
    }
};
```
Errorhandling in Legacy Code: Scenario

A typical Win32 application

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 (ATOM)</td>
<td>0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>INVALID_HANDLE_VALUE or NULL</td>
</tr>
<tr>
<td>HWND</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Detecting the failure: Generic Advice

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);
        s << "," << *jp. template arg<I-1>();
    }
};

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

Further Examples

Reporting the Error

```cpp
LRESULT WinProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam) {
    Hdc dc = NULL;
    switch(nMsg) {
        case WM_CREATE:
        dc = GetDC(hWnd);
        break;
        ...
    }
    used dc = BlendRect( dc, ...
    MessageBox(hWnd, "Sample message", "Sample title", MB_OK);
    return 0;
}
```

Further Examples

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
  - 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


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.

pointcut
affects the same join points as the advice pointcut

match expression
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 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.

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
"% ....::Service::%(...) const" matches any const member-function of the class Service defined in any scope
"% ....::operator %(...)" matches any type conversion function
"virtual % C::%(...)" matches any virtual member function of C

Template Matching
"std::set<int>" 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) provides all join points where a named entity in the pointcut is called.

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

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

destruction(pointcut) 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) 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) returns all base classes resp. redefined functions of classes in the pointcut

derived(pointcut) 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, ...) \((N,...) \rightarrow C\) a list of type patterns is 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 (\(N \rightarrow C\)) intersection of the join points in the pointcuts.

pointcut || pointcut (\(N \rightarrow C\)) 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 tjp of class JoinPoint.

**Compile-time Types and Constants**

That

object type (object initiating a call) [type]

Target

target object type (target object of a call) [type]

Result

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

Res::Type, Res::ReferencedType

result type of the affected function [type]

Arg<i>::Type, Arg<i>::ReferencedType

type of the \(i\)th argument of the affected function (with \(0 \leq i < \text{ARGS}\)) [type]

ARGS

number of arguments [const]

JPID

unique numeric identifier for this join point [const]

JPTYPE

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

**Runtime Functions and State**

static const char *signature() gives a textual description of the join point (function name, class name, ...) [type]

static const char *filename() returns the name of the file in which the joinpoint shadow is located [type]

static int line() the source code line number that is associated with the joinpoint shadow [type]

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

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 [type]

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

Arg<i>::ReferencedType *arg<i>(){returns a typed pointer to the \(i\)th argument value (with \(0 \leq i < \text{ARGS}\)) [type]

void *arg(int i) returns a pointer to the memory position holding the argument value with index \(i\) [type]

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

AC::Action&action() returns the runtime action object containing the execution environment to execute (trigger()) the original code encapsulated by an around advice [type]

**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 [type]

static AC::Type argtype(int i) returns a C++ ABI V3 conforming string representation of the result type / argument type of the affected function [type]

**Example**

A reusable tracing aspect.

```cpp
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 functions may be redefined to apply the aspect to the desired set of functions.

```cpp
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_type, C_ctor, C_dtor: Code (any, only Call, only Execution, only object Construction, only object Destruction), N, NC, NC, NR, NR: Names (any, only Namespace, only Class, only Function, only Type)