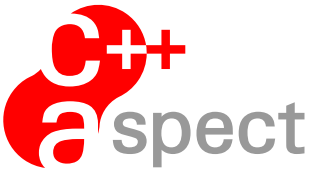

Documentation:

AspectC++ Language Reference

pure-systems GmbH

Matthias Urban

and Olaf Spinczyk

Version 1.1, 11th July 2003

[1]os@aspectc.org
www.aspectc.org

[2]aspectc@pure-systems.com
www.pure-systems.com
Agnetenstr. 14
39106 Magdeburg
Germany

Contents

1 About

This document is intended to be used as a reference book for the AspectC++ language elements. It describes in-depth the use and meaning of each element providing examples. For experienced users the contents of this document is summarized in the *AspectC++ Quick Reference*. A step-by-step introduction how to program with AspectC++ is given in the *AspectC++ Programming Guide*. Detailed information about the AspectC++ compiler **ac++** can be looked up in the *AC++ Compiler Manual*.

AspectC++ is an aspect-oriented extension to the C++ language[1]. It is similar to AspectJ[2] but, due to the nature of C++, in some points completely different. The first part of this document introduces the basic concepts of the AspectC++ language. The in-depth description of each language element is subject of the second part.

2 Basic Concepts

2.1 Pointcuts

Aspects in AspectC++ implement crosscutting concerns in a modular way. With this in mind the most important element of the AspectC++ language is the pointcut. Pointcuts describe a set of join points by determining on which condition an aspect shall take effect. Thereby each join point can either refer to a function, an attribute, a type, a variable, or a point from which a join point is accessed so that this condition can be for instance the event of reaching a designated code position. Depending on the kind of pointcuts, they are evaluated at compile time or at runtime.

2.1.1 Match Expressions

There are two types of pointcuts in AspectC++: code pointcuts and name pointcuts. Name pointcuts describe an intersection through the set of names known in a program. In detail these names can be the names of types, attributes, functions, variables, and namespaces. One way to describe name pointcuts are the so-called match expressions, i.e. string constants containing a search pattern. In such a search pattern the special character “%” is interpreted as a wildcard for names or parts of a signature. The special character sequence “...” matches any number of parameters in a function signature.

[1] defined in the ISO/IEC 14882:1998(E) standard

[2] www.aspectj.org

Example: match expressions (name pointcuts)

`"int"`
: matches the C++ built-in scalar type `int`

`"%List"`
: matches any class, struct or union whose name ends with “List”

`"% printf(...)"`
: matches the function `printf` having any number of parameters and returning any type

The signatures given in match expressions are internally separated into result type, entity name[3], and arguments[3]. This makes it possible to match even complex C++ signatures. The signature `"void (*fct())(int)"` describing a function taking no argument and returning a pointer to a function taking a single argument of type `int` can be for instance successfully matched by the match expression `"% fct()"`.

2.1.2 Pointcut Expressions

The other type of pointcuts, the code pointcuts, describe an intersection through the set of the points in the control flow of a program. A code pointcut can refer to a call or execution point of a function. They can only be created with the help of name pointcuts because all join points supported by AspectC++ require at least one name to be defined. This is done by calling predefined pointcut functions in a pointcut expression that expect a pointcut as argument. Such a pointcut function is for instance **within**(*pointcut*), which filters all join points that are within the functions or classes in the given pointcut.

Name and code pointcuts can be combined in pointcut expressions by using the algebraic operators “&&”, “||”, and “!”.

Example: pointcut expressions

`"%List" && !derived("Queue")`
: describes the set of classes with names that end with “List” and that are not derived from the class `Queue`

`call("void draw()") && within("Shape")`
: describes the set of calls to the function `draw` that are within methods of the class `Shape`

[3] if present

2.1.3 Types of Join Points

According to the two types of pointcuts supported by AspectC++ there are also two types of join points. Based on a short code fragment the differences and relations between these two types of join points shall be clarified.

```
class Shape;
void draw(Shape&);

namespace Circle {
  typedef int PRECISION;

  class S_Circle : public Shape {
    PRECISION m_radius;
  public:
    ...
    void radius(PRECISION r) { m_radius=r; }
  };

  void draw(PRECISION r) {
    S_Circle circle;
    circle.radius(r);
    draw(circle);
  }
}

int main() {
  Circle::draw(10);
  return 0;
}
```

Code join points are used to form code pointcuts and name join points (i.e. names) are used to form name pointcuts. Figure 1 on the following page shows some join points of the code fragment and how they correlate.

Every **execution** join point is associated with the name of an executable function. Pure virtual functions are not executable. Thus, advice code for execution join points would never be triggered for this kind of function. However, the call of such a function, i.e. a **call** join point with this function as target, is absolutely possible.

Every **call** join point is associated with two names: the name of the source and the target function of a function call. As there can be multiple function calls within the same function, each function name can be associated with a list of **call** join points.

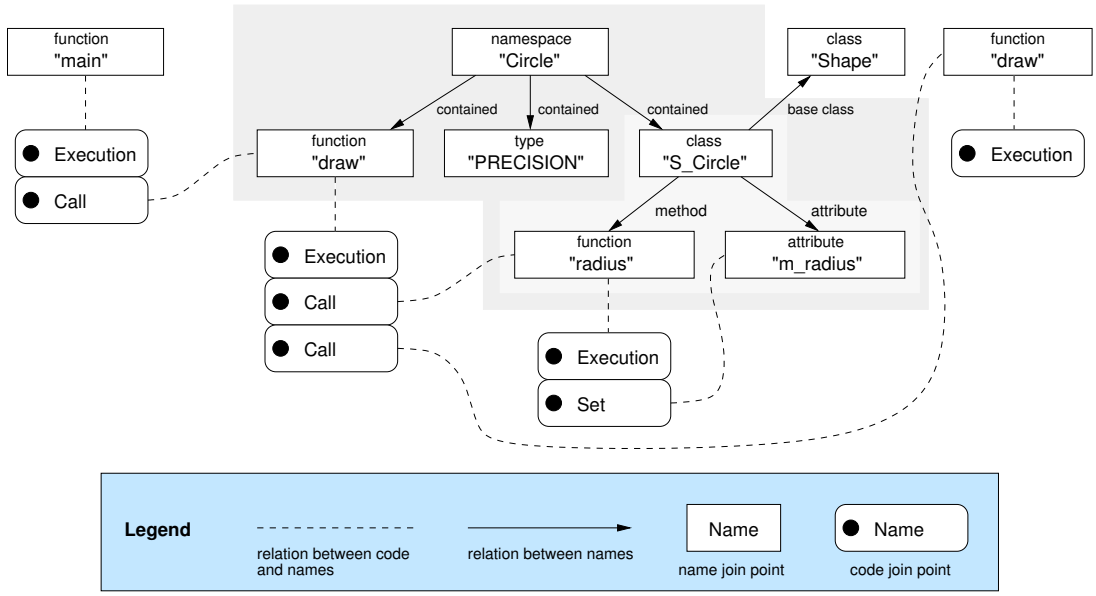

Figure 1: join points and their relations

2.1.4 Pointcut declarations

AspectC++ provides the possibility to name pointcut expressions with the help of pointcut declarations. This makes it possible to reuse pointcut expressions in different parts of a program. They are allowed where C++ declarations are allowed. Thereby the usual C++ name lookup and inheritance rules are also applicable for pointcut declarations.

A pointcut declaration is introduced by the keyword `pointcut`.

Example: pointcut declaration

```
pointcut lists() = derived("List");
```

`lists` can now be used everywhere in a program where a pointcut expression can be used to refer to `derived("List")`

Furthermore pointcut declarations can be used to define pure virtual pointcuts. This enables the possibility of having re-usable abstract aspects that are discussed in section 2.3. The syntax of pure virtual pointcut declarations is the same as for usual pointcut declarations except the keyword `virtual` following `pointcut` and that the pointcut expression is "0".

Example: pure virtual pointcut declaration

```
pointcut virtual methods() = 0;
```

`methods` is a pure virtual pointcut that has to be redefined in a derived aspect to

refer to the actual pointcut expression

2.2 Advice Code

To a code join point so-called advice code can be bound. Advice code can be understood as an action activated by an aspect when a corresponding code join point in a program is reached. The activation of the advice code can happen before, after, or before and after the code join point is reached. The AspectC++ language element to specify advice code is the advice declaration. It is introduced by the keyword `advice` followed by a pointcut expression defining where and under which conditions the advice code shall be activated.

Example: advice declaration

```
advice execution("void login(...)") : before() {
  cout << "Logging in." << endl;
}
```

The code fragment `:before()` following the pointcut expression determines that the advice code shall be activated directly **before** the code join point is reached. It is also possible here to use `:after()` which means **after** reaching the code join point respectively `:around()` which means that the advice code shall be executed instead of the code described by the code join point. In an **around** advice the advice code can explicitly trigger the execution of the program code at the join point so that advice code can be executed **before** and **after** the join point. There are no special access rights of advice code regarding to program code at a join point.

Beside the pure description of join points pointcuts can also bind variables to context information of a join point. Thus for instance the actual argument values of a function call can be made accessible to the advice code.

Example: advice declaration with access to context information

```
pointcut new_user(const char *name) =
  execution("void login(...)") && args(name);

advice new_user(name) : before(const char *name) {
  cout << "User " << name << " is logging in." << endl;
}
```

In the example above at first the pointcut `new_user` is defined including a context variable `name` that is bound to it. This means that a value of type `const char*` is supplied every time the join point described by the pointcut `new_user` is reached. The pointcut function

`args` used in the pointcut expression delivers all join points in the program where an argument of type `const char*` is used. Therefore `args(name)` in touch with the **execution** join point binds `name` to the first and only parameter of the function `login`.

The advice declaration in the example above following the pointcut declaration binds the execution of advice code to the event when a join point described in `new_user` is reached. The context variable that holds the actual value of the parameter of the reached join point has to be declared as a formal parameter of `before`, `after`, or `around`. This parameter can be used in the advice code like an oridinary function parameter.

Beside the pointcut function `args` the binding of context variables is performed by `that`, `target`, and `result`. At the same time these pointcut functions act as filters corresponding to the type of the context variable. For instance `args` in the example above filters all join points having an argument of type `const char*`.

2.2.1 Introductions

The second type of advice supported by AspectC++ are the introductions. Introductions are used to extend program code and data structures in particular. The following example extends two classes each by an attribute and a method.

Example: introductions

```
pointcut shapes() = "Circle" || "Polygon";

advice shapes() : bool m_shaded;
advice shapes() : void shaded(bool state) {
  m_shaded = state;
}
```

Like an ordinary advice declaration an introduction is introduced by the keyword `advice`. If the following pointcut is a name pointcut the C++ declaration following the token ":" is introduced in the classes and namespaces described by the pointcut. Introduced code can then be used in normal program code like any other function, attribute, etc. Advice code in introductions has full access rights regarding to program code at a join point, i.e. a method introduced in a class has access even to private members of that class.

A special kind of introduction is the base class introduction. It is used to extend the base class list of one or more classes, e.g. to let classes provide a special abstract interface. In the first line of the following example it is made sure that every class with a name that ends with "Object" is derived from a class `MemoryPool`. This class may implement an own memory management by overloading the `new` and `delete` operators. Classes that inherit from `MemoryPool` must redefine the pure virtual method `release` that is part of the

implemented memory management. This is done in the second line for all classes in the pointcut.

Example: base class introduction

```
advice "%Object" : baseclass(MemoryPool);
advice "%Object" : virtual void release() { ... }
```

2.2.2 Advice Ordering

If more than one advice affects the same join point it might be necessary to define an order of advice execution if there is a dependency between the advice codes (“aspect interaction”). The following example shows how the precedence of advice code can be defined in AspectC++.

Example: advice ordering

```
advice execution("void send(...)") : order("Encrypt", "Log");
```

If advice of both aspects (see 2.3) `Encrypt` and `Log` should be run when the function `send(...)` is executed this order declaration defines that the advice of `Encrypt` has a higher precedence. More details on advice ordering and precedence can be found in section 7 on page 22.

2.3 Aspects

The aspect is the language element of AspectC++ to collect introductions and advice code implementing a common crosscutting concern in a modular way. This put aspects in a position to manage common state information. They are formulated by means of aspect declarations as a extension to the class concept of C++. The basic structure of an aspect declaration is exactly the same as an usual C++ class definition, except for the keyword `aspect` instead of `class`, `struct` or `union`. According to that, aspects can have attributes and methods and can inherit from classes and even other aspects.

Example: aspect declaration

```
aspect Counter {
  static int m_count;
  Counting() : m_count(0) {}
```

```
  pointcut counted() = "Circle" || "Polygon";

  advice counted() : class Helper {
    Helper() { Counter::m_count++; }
  } m_counter;

  advice execution("% main(...)") : after() {
    cout << "Final count: " << m_count << " objects" << endl;
  }
};
```

In this example the count of object instantiations for a set of classes is determined. Therefore an attribute is introduced into the classes described by the pointcut incrementing a global counter on construction time. By applying advice code for the function `main` the final count of object instantiations is displayed when the program terminates.

This example can also be rewritten as an abstract aspect that can for instance be archived in an aspect library for the purpose of reuse. It only require to reimplement the pointcut declaration to be pure virtual.

Example: abstract aspect

```
aspect Counter {
  static int m_count;
  Counting() : m_count(0) {}

  pointcut virtual counted() = 0;
  ...
};
```

It is now possible to inherit from `Counter` to reuse its functionality by reimplementing `counted` to refer to the actual pointcut expression.

Example: reused abstract aspect

```
aspect MyCounter : public Counter {
  pointcut counted() = derived("Shape");
};
```

2.3.1 Aspect Instantiation

By default aspects in AspectC++ are automatically instantiated as global objects. The idea behind it is that aspects can also provide global program properties and therefore have to

be always accessible. However in some special cases it may be desired to change this behavior, e.g. in the context of operating systems when an aspect shall be instantiated per process or per thread.

The default instantiation scheme can be changed by defining the static method `aspectof` resp. `aspectOf` that is otherwise generated for an aspect. This method is intended to be always able to return an instance of the appropriate aspect.

Example: aspect instantiation using `aspectof`

```
aspect ThreadCounter : public Counter {
  pointcut counted() = "Thread";

  advice counted() : ThreadCounter m_instance;

  static ThreadCounter *aspectof() {
    return tjp->target()->m_instance;
  }
};
```

The introduction of `m_instance` into `Thread` guarantees that every thread object has an instance of the aspect. By calling `aspectof` it is possible to get this instance at any join point which is essential for accessing advice code and members of the aspect. For this purpose code in `aspectof` has full access to the actual join point in a way described in the next section.

2.4 Runtime Support

2.4.1 Support for Advice Code

For many aspects access to context variables may not be sufficient to get enough information about the join point where advice code was activated. For instance a control flow aspect for a complete logging of function calls in a program would need information about function arguments and its types on runtime to be able to produce a type-compatible output.

In AspectC++ this information is provided by the members of the class `JoinPoint` (see table below).

types:	
`Result`	result type
`That`	object type
`Target`	target type

`AC::Type`	encoded type of an object
`AC::JPType`	join point types
static methods:	
`int args()`	number of arguments
`AC::Type type()`	typ of the function or attribute
`AC::Type argtype(int)`	types of the arguments
`const char *signature()`	signature of the function or attribute
`unsigned id()`	identification of the join point
`AC::Type resulttype()`	result type
`AC::JPType jptype()`	type of join point
non-static methods:	
`void *arg(int)`	actual argument
`Result *result()`	result value
`That *that()`	object refered to by `this`
`Target *target()`	target object of a call
`void proceed()`	execute join point code
`AC::Action &action()`	`Action` structure

Table 1: API of class `JoinPoint` available in advice code

Types and static methods of the `JoinPoint` API deliver information that is the same for every advice code activation. The non-static methods deliver information that differ from one activation to another. These methods are accessed by the object `tjp` resp. `thisJoinPoint` which is of type `JoinPoint` and is always available in advice code, too.

The following example illustrates how to implement a re-usable control flow aspect using the `JoinPoint` API.

Example: re-usable trace aspect

```
aspect Trace {
  pointcut virtual methods() = 0;

  advice execution(methods()) : around() {
    cout << "before " << JoinPoint::signature() << "(";
    for (unsigned i = 0; i < JoinPoint::args(); i++)
      printvalue(tjp->arg(i), JoinPoint::argtype(i));
    cout << ")" << endl;
    tjp->proceed();
```

```
        cout << "after" << endl;
      }
    };
```

This aspect weaves tracing code into every function specified by the virtual pointcut redefined in a derived aspect. The helper function `printvalue` is responsible for the formated output of the arguments given at the function call. After calling `printvalue` for every argument the program code of the actual join point is executed by calling `proceed` on the `JoinPoint` object. The functionality of `proceed` is achieved by making use of the so-called actions.

2.4.2 Actions

In AspectC++ an action is the statement sequence that would follow a reached join point in a running program if advice code would not have been activated. Thus `tjp->proceed()` triggers the execution of the program code of a join point. This can be the call or execution of a function. The actions concept is realized in the `AC::Action` structure. In fact, `proceed` is implemented as `action().trigger()` so that `tjp->proceed()` may also be replaced by `tjp->action().trigger()`. Thereby the method `action()` of the `JoinPoint` API returns the actual action object for a join point.

3 Match Expressions

This section provides sample match expressions for matching types, attributes, and functions.

3.1 Type Matching

`"unsigned long"`
: matches the C++ built-in scalar type `unsigned long int`

`"% *"`
: matches pointers to any class or named C++ data type

3.2 Namespace and Class Matching

`"Chain"`
: matches the class, struct, or union `Chain`

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

3.3 Attribute Matching

`"Chain* Chain::next"`
: matches the attribute `next` of class `Chain` having type `Chain*` (pointer to `Chain`)

`"% Chain::%"`
: matches any attribute of class `Chain`

3.4 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 returns any type

`"void %(int,%)"`
: matches any function having exactly two parameters (from which the first one must be `int`) and returning `void`

4 Predefined Pointcut Functions

On the following pages a complete list of the pointcut functions supported by AspectC++ is presented. For every pointcut function it is indicated which type of pointcut is expected as argument(s) and of which type the result pointcut is. Thereby “N” stands for name pointcut and “C” for code pointcut. The optionally given index is an assurance about the type of join point(s) described by the result pointcut[4].

4.1 Types

base(*pointcut*) N→$N_{C,F}$
: returns all base classes of classes in the pointcut

derived(*pointcut*) N→$N_{C,F}$
: returns all classes in the pointcut and all classes derived from them

[4] C, C_C, C_E, C_S, C_G: Code (any, only *Call*, only *Execution*, only *Set*, only *Get*); N, N_N, N_C, N_F, N_T: Names (any, only *Namespace*, only *Class*, only *Function*, only *Type*)

Example: type matching

A software may contain the following class hierarchy.

```
class Shape { ... };
class Point : public Shape { ... };
...
class Rectangle : public Line, public Rotatable { ... };
```

With the following aspect a special feature is added to a designated set of classes of this class hierarchy.

```
aspect Scale {
  pointcut scalable() =
    (base("Rectangle") && derived("Point")) || "Rectangle";

  advice "Point" : baseclass("Scalable");
  advice scalable() : void scale(int value) { ... }
};
```

The pointcut describes the classes `Point` and `Rectangle` and all classes derived from `Point` that are direct or indirect base classes of `Rectangle`. With the first advice `Point` gets a new base class. The second advice adds a corresponding method to all classes in the pointcut.

4.2 Control Flow

cflow(*pointcut*) N→C

captures join points occuring in the dynamic execution context of join points in the pointcut

Example: function call matching

The following example demonstrates the use of the **cflow** pointcut function.

```
class Constant {
  int m_value;
public:
  Constant(int v) : m_value(v) {}

  operator int() { return m_value; }
  int get_value() { return m_value; }
```

```
};

int main() {
  Constant year(2003);
  ...
  int y = year.get_value();
  ...
  return 0;
}
```

The function `get_value` may be deprecated and a warning shall be displayed if it is still used in the program. A corresponding aspect using the **cflow** pointcut function is shown below.

```
aspect Deprecated {
  pointcut functions() = call("% Constant::get_value()") &&
                         cflow(execution ("% main(...)"));

  advice functions() : before() {
    cout << "Warning: Use of get_value() is deprecated." << endl;
  }
};
```

The pointcut describes all the calls to the function `get_value` that come from the function `main` and any function called in `main` and any function called in a function called in `main` and so on. The advice provides the warning message for every join point in the pointcut. The message is displayed before `get_value` is called.

4.3 Scope

within(*pointcut*) N→C

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

Example: matching in scopes

```
aspect Logger {
  pointcut calls() =
    call("void transmit()") && within("Transmitter");

  advice calls() : around() {
    cout << "transmitting ... " << flush;
```

```
    tjp->proceed();
    cout << "finished." << endl;
  }
};
```

This aspect inserts code logging all calls to `transmit` that are within the methods of class `Transmitter`.

4.4 Functions

call(*pointcut*) N→C_C

Provides all join points where a named entity in the pointcut is called. The pointcut may contain function names or class names. In the case of a class name all calls to methods of that class are provided.

execution(*pointcut*) N→C_E

provides all join points referring to the implementation of a named entity in the pointcut. The pointcut may contain function names or class names. In the case of a class name all implementations of methods of that class are provided.

Example: function matching

The following aspect weaves debugging code into a program that checks whether a method is called on a null pointer and whether the argument of the call is null.

```
aspect Debug {
  pointcut fct() = "% MemPool::dealloc(void*)";
  pointcut exec() = execution(fct());
  pointcut calls() = call(fct());

  advice exec() && args(ptr) : before(void *ptr) {
    assert(ptr && "argument is NULL");
  }
  advice calls() : before() {
    assert(tjp->target() && "'this' is NULL");
  }
};
```

The first advice provides code to check the argument of the function `dealloc` before the function is executed. A check whether `dealloc` is called on a null object is provided by the second advice. This is realized by checking the target of the call.

4.5 Context

that(*type pattern*) N→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→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

args(*type pattern*, ...) (N,...)→C

receives a list of type patterns and filters all methods or attributes with a matching signature

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

Example: context matching

4.6 Algebraic Operators

pointcut **&&** *pointcut* (N,N)→N, (C,C)→C

intersection of the join points in the pointcuts

pointcut **||** *pointcut* (N,N)→N, (C,C)→C

union of the join points in the pointcuts

! *pointcut* N→N, C→C

exclusion of the join points in the pointcut

Example: combining pointcut expressions

5 Advice Declarations

This section provides a summary of the AspectC++ placement functions allowed in advice declarations.

before(...)

the advice code is executed before the join points in the pointcut

after(...)

the advice code is executed after the join points in the pointcut

around(...)
: the advice code is executed in place of the join points in the pointcut

baseclass(*classname*)
: a new base class is introduced to the classes in the pointcut

Example: advice placement

6 JoinPoint API

The following sections provide a complete description of the `JoinPoint` API.

6.1 Types

`Result`
: result type of a function

`That`
: object type (object initiating a call)

`Target`
: target object type (target object of a call)

Example: type usage

6.2 Functions

`static AC::Type type()`
: returns the encoded type for the join point conforming with the C++ ABI V3 specification[5]

`static int args()`
: returns the number of arguments of a function for call and execution join points

`static AC::Type argtype(int number)`
: returns the encoded type of an argument conforming with the C++ ABI V3 specification

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

[5] www.codesourcery.com/cxx-abi/abi.html#mangling

`static unsigned int id()`
: returns a unique numeric identifier for this join point

`static AC::Type resulttype()`
: returns the encoded type of the result type conforming with the C++ ABI V3 specification

`static AC::JPType jptype()`
: returns a unique identifier describing the type of the join point

Example: static function usage

`void *arg(int number)`
: returns a pointer to the memory position holding the argument value with index `number`

`Result *result()`
: returns a pointer to the memory location designated for the result value or 0 if the function has no result value

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

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

`void proceed()`
: executes the original join point code in an around advice by calling `action().trigger()`

`AC::Action &action()`
: returns the runtime action object containing the execution environment to execute the original functionality encapsulated by an around advice

Example: non-static function usage

7 Advice Ordering

7.1 Aspect Precedence

AspectC++ provides a very flexible mechanism to define aspect precedence. The precedence is used to determine the execution order of advice code if more than one aspect

affect the same join point. The precedence in AspectC++ is an attribute of a join point. This means that the precedence relationship between two aspects might vary in different parts of the system. The compiler checks the following conditions to determine the precendence of aspects:

order declaration: if the programmer provides an order declaration, which defines the precedence relationship between two aspects for a join point, the compiler will obey this definition or abort with a compile-time error if there is a cycle in the precedence graph. Order declarations have the following syntax:

`advice` *pointcut-expr* : **`order (`** *high*`,` *...low* **`)`**

The argument list of `order` has to contain at least two elements. Each element is a pointcut expression, which describes a set of aspects. Each aspect in a certain set has a higher precedence than all aspects, which are part of a set following later in the list (on the right hand side). For example '`("A1"||"A2","A3"||"A4")`' means that `A1` has precedence over `A3` and `A4` and that `A2` has precedence over `A3` and `A4`. This order directive does *not* define the relation between `A1` and `A2` or `A3` and `A4`. Of course, the pointcut expressions in the argument list of `order` may contain named pointcuts and even pure virtual pointcuts.

inheritance relation: if there is no order declaration given and one aspect has a base aspect the derived aspect has a higher precedence than the base aspect.

7.2 Advice Precedence

The precedence of advice is determined with a very simple scheme:

- if two advice declarations belong to different aspects and there is a precedence relation between these aspects (see section 7.1 on the facing page) the same relation will be assumed for the advice.
- if two advice declarations belong to the same aspect the one that is declared first has the higher precedence.

7.3 Effects of Advice Precedence

Only advice predecence has an effect on the generated code. The effect depends on the kind of join point, which is affected by two advice declarations.

Class Join Points

Advice on class join points can extend the attribute list or base class list. If advice has a higher precedence than another it will be handled first. For example, an introduced new

base class of advice with a high precedence will appear in the base class list on the left side of a base class, which was inserted by advice with lower precedence. This means that the execution order of the constructors of introduced base classes can be influenced, for instance, by order declarations.

The order of introduced attributes also has an impact on the constructor/destructor execution order as well as the object layout.

Code Join Points

Advice on code join points can be `before`, `after`, or `around` advice. For `before` and `around` advice a higher precedence means that the corresponding advice code will be run first. For `after` advice a higher precedence means that the advice code will be run later.

If `around` advice code does not call `tjp->proceed()` or `trigger()` on the action object no advice code with lower precedence will be run. The execution of advice with higher precedence is not affected by `around` advice with lower precedence.

For example, consider an aspect that defines advice[6] in the following order: BE1, AF1, AF2, AR1, BE2, AR2, AF3. As described in section 7.2 on the page before the declaration order also defines the precedence: BE1 has the highest and AF3 the lowest. The result is the following advice code execution sequence:

1. BE1 (highest precedence)
2. AR1 (the indented advice will only be executed if `proceed()` is called!)
   (a) BE2 (before AR2, buts depends on AR1)
   (b) AR2 (the indented code will only be executed if `proceed()` is called!)
      i. original code under the join point
      ii. AF3
3. AF2 (does not depend on AR1 and AR2, because of higher precedence)
4. AF1 (run after AF2, because it has a higher precedence)

A Grammar

The AspectC++ syntax is an extension to the C++ syntax. It adds three new keywords to the C++ language: `aspect`, `advice`, and `pointcut`. Additionally it extends the C++ language by advice and pointcut declarations. In contrast to pointcut declarations, advice declarations may only occur in aspect declarations.

[6] BE is `before` advice, AF `after` advice, and AR `around` advice

class-key:

`aspect`

declaration:

pointcut-declaration

advice-declaration

member-declaration:

pointcut-declaration

advice-declaration

pointcut-declaration:

`pointcut` *declaration*

pointcut-expression:

constant-expression

advice-declaration:

`advice` *pointcut-expression* **:** *order-declaration*

`advice` *pointcut-expression* **:** *declaration*

order-declaration:

`order (` *pointcur-expression-seq* `)`

List of Examples

Index