Documentation:
AspectC++ Language Reference

pure-systems GmbH

Matthias Urban

and Olaf Spinczyk

Version 1.10, October 14, 2012

[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[1]. Detailed information about the AspectC++ compiler **ac++** can be looked up in the AspectC++ Compiler Manual.

AspectC++ is an aspect-oriented extension to the C++ language[2]. It is similar to AspectJ[3] 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 a set of (statically) known program entities like types, attributes, functions, variables, or namespaces. All name pointcuts are based on match expressions. A match expression can be understood as 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 "..."

[1] Sorry, but the Programming Guide is not written yet :-(

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

[3] http://www.eclipse.org/aspectj/

matches any number of parameters in a function signature or any number of scopes in a qualified name. A match expression is a quoted string.

Example: match expressions (name pointcuts)

`"int C::%(...)"`
: matches all member functions of the class C that return an `int`

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

`"% printf(const char *, ...)"`
: matches the function `printf` (defined in the global scope) having at least one parameter of type `const char *` and returning any type

`"const %& ...::%(...)"`
: matches all functions that return a reference to a constant object

Match expressions select program entities with respect to their definition scope, their type, and their name. A detailed description of the match expression semantics follows in section 3 on page 16. The grammar which defines syntactically valid match expressions is shown in appendix B on page 36.

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`

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

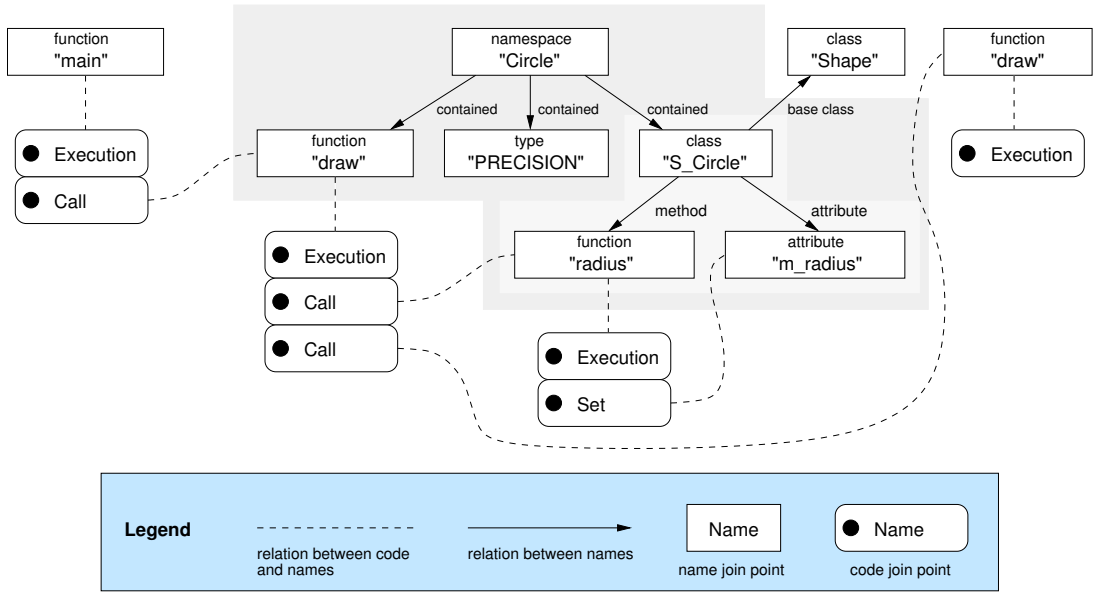

Figure 1: join points and their relations

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. A **construction** join point means the class specific instruction sequence executed when an instance is created. In analogy, a **destruction** join point means the object destruction.

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 expres-

sion 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 2.4. 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 Slices

A *slice* is a fragment of a C++ language element that defines a scope. It can be used by advice to extend the static structure of the programm. For example, the elements of a class slice can be merged into one or more target classes by introduction advice. The following example shows a simple class slice declaration.

Example: class slice declaration

```
slice class Chain {
  Chain *_next;
public:
  Chain *next () const { return _next; }
};
```

2.3 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*`.

2.3.1 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() : slice class {
  bool m_shaded;
  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 slice declaration following the token ":" is introduced in the classes and aspects 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.

Slices can also be used to introduce new base classes. 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" : slice class : public MemoryPool {
```

```
    virtual void release() = 0;
}
```

2.3.2 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 2.4) `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 8 on page 33.

2.4 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;

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

  advice counted() : slice struct {
    class Helper {
```

```
      Helper() { Counter::m_count++; }
    } m_counter;
  };

  advice execution("% main(...)") : after() {
    cout << "Final count: " << m_count << " objects" << endl;
  }
};
... and at an appropriate place
#include "Counter.ah"
int Counter::m_count = 0;
```

In this example the count of object instantiations for a set of classes is determined. Therefore, an attribute `m_counter` 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;
  Counter() : 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");
};
```

2.4.1 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.

2.5 Runtime Support

2.5.1 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.

2.5.2 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 equivalent to `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

Match expressions are a used to describe a set of statically known program entities in an AspectC++ program. They can either be match expressions for functions or for types. A class is seen as a special kind of type in this context.

For function matching a match expression is internally decomposed into the function type pattern, the scope pattern, and the name pattern.

Example: type, scope, and name parts of a function match expression

```
"const % Puma::...::parse_% (Token *)"
```

This match expression describes the following requirements on a compared function name:

name: the function name has to match the name pattern `parse_%`

scope: the scope in which the function is defined has to match `Puma::...::`

type: the function type has to match `const %(Token *)`

For classes and other types this decomposion is not necessary. For example, the type name "`Puma::CCParser`" is sufficient to describe a class, because this is the same as the class name.

If an entity matches all parts of the match expression, it becomes an element of the set, which should be defined by the match expression.

The grammar used for match expression parsing is shown in appendix B on page 36. The following subsections separately describe the name, scope, and type matching mechanisms. Note, that name and scope matching is used for matching of function names as well as matching of named types like classes.

3.1 Name Matching

3.1.1 Simple Name Matching

Name matching is trivial as long as the compared name is a normal C++ identifier. If the *name pattern* does *not* contain the special wildcard character %, it matches a name only if it is exactly the same. Otherwise each wildcard character matches an arbitrary sequence of characters in the compared name. The wildcard character also matches an empty sequence.

Example: simple name patterns

`Token`	only matches `Token`
`%`	matches any name
`parse_%`	matches any name beginning with `parse_` like `parse_declarator` or `parse_`
`parse_%_id%`	matches names like `parse_type_id`, `parse_private_identifier`, etc.
`%_token`	matches all names that end with `_token` like `start_token`, `end_token`, and `_token`

3.1.2 Operator Function and Conversion Function Name Matching

The name matching mechanism is more complicated if the pattern is compared with the name of a conversion function or an operator function. Both are matched by the name pattern `%`. However, with a different name pattern than `%` they are only matched if the pattern begins with "`operator `". The pattern "`operator %`" matches any operator function or conversion function name.

C++ defines a fixed set of operators which are allowed to be overloaded. In a name pattern the same operators may be used after the "`operator `" prefix to match a specific operator function name. Operator names in name patterns are not allowed to contain the wildcard character. For ambiguity resolution the operators `%` and `%=` are matched by `%%` and `%%=` in a name pattern.

Example: operator name patterns

`operator %`	matches any operator function name (as well as any conversion function name)
`operator +=`	matches only the name of a `+=` operator
`operator %%`	matches the name of an `operator %`

Conversion functions don't have a real name. For example, the conversion function `operator int*()` defined in a class `C` defines a conversion from a `C` instance into an object of type `int*`. To match conversion functions the name pattern may contain a type pattern after the prefix "`operator `". The type matching mechanism is explained in section 3.3.

Example: conversion function name patterns

`operator %`	matches any conversion function name
`operator int*`	matches any name of a conversion that converts something into an `int*` object
`operator %*`	matches any conversion function name if that function converts something into a pointer

3.1.3 Constructors and Destructors

Name patterns cannot be used to match constructor or destructor names.

3.1.4 Scope Restrictions

In a match expression a name pattern can optionally be prefixed by a scope pattern. A scope pattern (see section 3.2) is used to describe restrictions on the definition scope of matched entities. If no scope pattern is given, a compared function or type has to be defined in the global scope to be matched.

3.2 Scope Matching

Restrictions on definition scopes can be described by *scope patterns*. This is a sequence of name patterns (or the special *any scope sequence* pattern `...`), which are separated by `::`, like in `Puma::...::`. A scope pattern always ends with `::` and should never start with `::`, because scope patterns are interpreted relative to the global scope anyway[4]. The definition scope can either be a namespace or a class.

A scope pattern matches the definition scope of a compared function or type if every part can successfully be matched with a corresponding part in the qualified name of the definition scope. The compared qualified name has to be relative to the global scope and should not start with `::`, which is optional in a C++ nested-name-specifier. The special `...` pattern matches any (even empty) sequence of scope names.

Example: scope patterns

`...::`	matches any definition scope, even the global scope
`Puma::CCParser::`	matches the scope `Puma::CCParser` exactly
`...::%Compiler%::`	matches any class or namespace, which matches the name pattern `%Compiler%`, in any scope
`Puma::...::`	matches any scope defined within the class or namespace `Puma` and `Puma` itself

3.3 Type Matching

3.3.1 The Match Mechanism

C++ types can be represented as a tree. For example, the function type `int(double)` is a function type node with two children, one is an `int` node, the other a `double` node. Both children are leaves of the tree.

[4] This restriction is also needed to avoid ambiguities in the match expression grammer: Does "`A :: B :: C(int)`" mean "`A ::B::C(int)`" or "`A::B ::C(int)`"?

The types used in match expressions can also be interpreted as trees. As an addition to normal C++ types they can also contain the % wildcard character, name patterns, and scope patterns. A single wildcard character in a type pattern becomes a special *any type node* in the tree representation.

For comparing a type pattern with a specific type the tree representation is used and the *any type node* matches an arbitrary type (sub-)tree.

Example: type patterns with the wildcard character

`%`	matches any type
`void (*)(%)`	matches any pointer type that points to functions with a single argument and a `void` result type
%*	matches any pointer type

3.3.2 Matching of Named Types

Type patterns may also contain name and scope patterns. They become a *named type node* in the tree representation and match any union, struct, class, or enumeration type if its name and scope match the given pattern (see section 3.1 and 3.2).

3.3.3 Matching of "Pointer to Member" Types

Patterns for pointers to members also contain a scope pattern, e.g. `% (Puma::CSyntax::*)()`. In this context the scope pattern is mandatory. The pattern is used for matching the class associated with a pointer to member type.

3.3.4 Matching of Qualified Types (`const`/`volatile`)

Many C++ types can be qualified as `const` or `volatile`. In a type pattern these qualifier can also be used, but they are interpreted restrictions. If no `const` or `volatile` qualifier is given in a type pattern, the pattern also matches qualified types[5].

[5] Matching only non-constant or non-volatile types can be achieved by using the operators explained in section 4.7 on page 27. For example, `!"const %"` describes all types which are not constant.

Example: type patterns with `const` and `volatile`

`%`	matches any type, even types qualified with `const` or `volatile`
`const %`	matches only types qualified by `const`
`% (*)() const volatile`	matches the type of all pointers to functions that are qualified by `const` and `volatile`

3.3.5 Handling of Conversion Function Types

The result type of conversion functions is interpreted as a special *undefined* type in type patterns as well as in compared types. The *undefined* type is only matched by the *any type* node and the *undefined type* node.

3.3.6 Ellipses in Function Type Patterns

In the list of function argument types the type pattern `...` can be used to match an arbitrary (even empty) list of types. The `...` pattern should not be followed by other argument type patterns in the list of argument types.

3.3.7 Matching Virtual Functions

The *decl-specifier-seq* of a function type match expression may include the keyword `virtual`. In this case the function type match expression only matches virtual or pure virtual member functions. As `const` and `volatile`, the `virtual` keyword is regarded as a restriction. This means that a function type match expression without `virtual` matches virtual and non-virtual functions.

Example: type patterns with `virtual`

`virtual % ...::%(...)`	matches all virtual or pure virtual functions in any scope
`% C::%(...)`	matches all member functions of C, even if they are virtual

3.3.8 Matching Static Functions

Matching static functions works similar as matching virtual functions. The *decl-specifier-seq* of a function type match expression may include the keyword `static`. In this case the function type match expression only matches static

functions in global or namespace scope and static member functions of classes. As `const` and `volatile`, the `static` keyword is regarded as a restriction. This means that a function type match expression without `static` matches static and non-static functions.

Example: type patterns with `static`

`static % ...::%(...)`	matches all static member and non-member functions in any scope
`% C::%(...)`	matches all member functions of C, even if they are static

3.3.9 Argument Type Adjustment

Argument types in type patterns are adjusted according to the usual C++ rules, i.e. array and function types are converted to pointers to the given type and `const`/`volatile` qualifiers are removed. Futhermore, argument type lists containing a single `void` type are converted into an empty argument type list.

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[6].

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

[6] 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*) C→C

captures join points occuring in the dynamic execution context of join points in the pointcut. Currently the language features being used in the argument pointcut are restricted. The argument ist not allowed to contain any context variable bindings (see 4.6) or other pointcut functions which have to be evaluated at runtime like **cflow**(*pointcut*) itself.

Example: control flow dependant advice activation

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

```
class Bus {
  void out (unsigned char);
  unsigned char in ();
};
```

Consider the class `Bus` shown above. It might be part of an operating system kernel and is used there to access peripheral devices via a special I/O bus. The execution of the member functions `in()` and `out()` should not be interrupted, because this would break the timing of the bus communication. Therefore, we decide to implement an interrupt synchronization aspect that disables interrupts during the execution of in() and out():

```
aspect BusIntSync {
  pointcut critical() = execution("% Bus::%(...)");
  advice critical() && !cflow(execution("% os::int_handler()")) :
  around() {
    os::disable_ints();
    tjp->proceed();
    os::enable_ints();
  }
};
```

As the bus driver code might also be called from an interrupt handler, the interrupts should not be disabled in any case. Therefore, the pointcut expression exploits the **cflow()** pointcut function to add a runtime condition for the advice activation. The advice body should only be executed if the control flow did not come from the interrupt handler `os::int_handler()`, because it is not interruptable by definition and `os::enable_ints()` in the advice body would turn on the interrupts too early.

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 Object Construction and Destruction

construction(*pointcut*) N→C$_{Cons}$

all join points where an instance of the given class(es) is constructed. The construction joinpoint begins after all base class and member construction join points. It can be imagined as the execution of the constructor. However, advice for construction join points work, even if there is no constructor defined explicitly. A construction join point has arguments and argument types, which can be exposed or filtered, e.g. by using the **args** pointcut function.

destruction(*pointcut*) N→C$_{Des}$

all join points where an instance of the given class(es) is destructed. The destruction join point ends before the destruction join point of all members and base classes. It can be imagined as the execution of the destructor, although a destructor does not to be defined explicitly. A destruction join point has an empty argument list.

Example: instance counting

The following aspect counts how many instances of the class `ClassOfInterest` are created and destroyed.

```
aspect InstanceCounting {
  // the class for which instances should be counted
  pointcut observed() = "ClassOfInterest";
  // count constructions and destructions
  advice construction (observed ()) : before () { _created++; }
  advice destruction (observed ())  : after ()  { _destroyed++; }
public:
  // Singleton aspects can have a default constructor
  InstanceCounting () { _created = _destroyed = 0; }
private:
  // counters
  int _created;
  int _destroyed;
};
```

The implementation of this aspect is straightforward. Two counters are initialized by the aspect constructor and incremented by the construction/destruction advice. By defining `observed()` as a pure virtual pointcut the aspect can easily be transformed into a reusable abstract aspect.

4.6 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

result(*type pattern*) N→C
returns all join points where the result object of a call/execution is an instance of a type matched by the type pattern

args(*type pattern*, ...) (N,...)→C
The argument list of **args** contains type patterns that are used to filter all join points, e.g. calls to functions or function executions, with a matching signature.

Instead of the type pattern it is also possible here to pass the name of a variable to which the context information is bound (a **context variable**). In this case the type of the variable is used for the type matching. Context variables must be declared in the argument list of **before()**, **after()**, or **around()** and can be used like a function argument in the advice body.

The **that()** and **target()** pointcut functions are special, because they might cause a run-time type check. The **args()** and **result()** functions are evaluated at compile time.

Example: context matching

4.7 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 Slices

This section defines the syntax and semantics of slice declarations. The next section will describe how slices can be used by advice in order to introduce code. Currently, only class slices are defined in AspectC++.

5.1 Class Slice Declarations

Class slices may be declared in any class or namespace scope. They may be defined only once, but there may be an arbitrary number forward declarations. A qualified name may be used if a class slice that is already declared in a certain scope is redeclared or defined as shown in the following example:

```
slice class ASlice;
namespace N {
 slice class ASlice; // a different slice!
}
slice class ASlice { // definition of the ::ASlice
  int elem;
};
slice class N::ASlice { // definition of the N::ASlice
  long elem;
};
```

If a class slice only defines a base class, an abbreviated syntax may be used:

```
slice class Chained : public Chain;
```

Class slices may be anonymous. However, this only makes sense as part of an advice declaration. A class slice my also be declared with the `aspect` or `struct` keyword instead of `class`. While there is no difference between class and aspect slices, the default access rights to the elements of a struct slice in the target classes are public instead of private. It is forbidden to declare aspects, pointcuts, advice, or slices as members of a class slice.

Class slices may have members that are not defined within the body of a class slice declaration, e.g. static attributes or non-inline functions:

```
slice class SL {
  static int answer;
```

```
  void f();
};
//...
slice int SL::answer = 42;
slice void SL::f() { ... }
```

These external member declarations have to appear after the corresponding slice declaration in the source code.

6 Advice

This section describes the different types of advice offered by AspectC++. Advice be categorized in advice for join points in the dynamic control flow of the running program, e. g. function call or executions, and advice for static join points like introductions into classes.

In either case the compiler makes sure that the code of the aspect header file, which contains the advice definition (if this is the case), is compiled prior to the affected join point location.

6.1 Advice for Dynamic Join Points

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

6.2 Advice for Static Join Points

Static join points in AspectC++ are classes or aspects. Advice for classes or aspects can introduce new members or add a base class. Whether the new member or base class becomes **private**, **protected**, or **public** in the target class depends on the protection in the advice declaration in the aspect.

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

introduction declaration
: a new attribute, member function, or type is introduced

Introduction declarations are only semantically analyzed in the context of the target. Therefore, the declaration may refer, for instance, to types or constants, which are not known in the aspect definition, but only in the target class or classes. To introduce a constructor or destructor the name of the aspect, to which the introduction belongs, has to be taken as the constructor/destructor name.

Non-inline introductions can be used for introductions of static attributes or member function introduction with separate declaration an definition. The name of the introduced member has to be a qualified name in which the nested name specifier is the name of the aspect to which the introduction belongs.

7 `JoinPoint` API

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

7.1 API for Dynamic Join Points

The JoinPoint-API for dynamic join points can be used within the body of advice code.

7.1.1 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

7.1.2 Functions

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

specification[7]

`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, ...)

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

`static const char *filename()`
: returns the name of the file in which the join-point (shadow) is located

`static int line()`
: the number of the line in which the join-point (shadow) is located

`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

[7] http://www.codesourcery.com/cxx-abi/abi.html\#mangling

`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

7.2 API for Static Join Points

The JoinPoint-API for static join points can be used within the definition of a slice and describes the state of target class *before* the introduction took place. It is accessed through the built-in type `JoinPoint` (e.g. `JoinPoint::signature()`) and provides the following functions, types, and constants:

`static const char *signature()`
: returns the target class name as a string

`BASECLASSES`
: number of baseclasses of the target class

`BaseClass<I>::Type`
: type of the I^{th} baseclass

`BaseClass<I>::prot, BaseClass<I>::spec`
: Protection level (AC::PROT_NONE /PRIVATE /PROTECTED /PUBLIC) and additional specifiers (AC::SPEC_NONE /VIRTUAL) of the I^{th} baseclass

`MEMBERS`
: number of attributes of the target class

`Member<I>::Type, Member<I>::ReferredType`
: type of the I^{th} attribute of the target class

`Member<I>::prot, Member<I>::spec`
: Protection level (see BaseClass<I>::prot) and additional attribute specifiers (AC::SPEC_NONE /STATIC /MUTABLE)

`static ReferredType *Member<I>::pointer(T *obj=0)`
returns a typed pointer to the I^{th} attribute (obj is needed for non-static attributes)

`static const char *Member<I>::name()`
returns the name of the I^{th} attribute

8 Advice Ordering

8.1 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.

8.2 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 8.1 on the preceding 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.

8.3 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[8] in the following order: BE1, AF1, AF2, AR1, BE2, AR2, AF3. As described in section 8.2 the declaration

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

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 four new keywords to the C++ language: `aspect`, `advice`, `slice`, 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.

class-key:
`aspect`

declaration:
pointcut-declaration
slice-declaration
advice-declaration

member-declaration:
pointcut-declaration
slice-declaration
advice-declaration

pointcut-declaration:
`pointcut` *declaration*

pointcut-expression:
constant-expression

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

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

slice-reference:
`slice` **::**$_{opt}$ *nested-name-specifier*$_{opt}$ *unqualified-id* **;**

slice-declaration:
`slice` *declaration*

B Match Expression Grammar

Match expression in AspectC++ are used to define a type pattern and an optional object name pattern to select a subset of the known program entities like functions, attributes, or argument/result types. The grammer is very similar to the grammer of C++ declarations. Any rules, which are referenced here but not defined, should be looked up in the ISO C++ standard.

match-expression:
match-declaration

match-id:
`%`
nondigit
match-id `%`
match-id nondigit
match-id digit

match-declaration:
match-decl-specifier-seq$_{opt}$ *match-declarator*

match-decl-specifier-seq:
match-decl-specifier-seq$_{opt}$ *match-decl-specifier*

match-decl-specifier:
nested-match-name-specifier$_{opt}$ *match-id*
cv-qualifier
match-function-specifier

`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

match-function-specifier:
`virtual`
`static`

nested-match-name-specifier:
match-id `::` *nested-match-name-specifier*$_{opt}$
`...` `::` *nested-match-name-specifier*$_{opt}$

match-declarator:
direct-match-declarator
match-ptr-declarator match-declarator

abstract-match-declarator:
direct-abstract-match-declarator
match-ptr-declarator abstract-match-declarator

direct-match-declarator:
match-declarator-id
direct-match-declarator `(` *match-parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
direct-match-declarator `[` *match-array-size* `]`

direct-abstract-match-declarator:
direct-abstract-match-declarator `(` *match-parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
direct-abstract-match-declarator `[` *match-array-size* `]`

match-array-size:
`%`
decimal-literal

match-ptr-operator:

`*` *cv-qualifier-seq*$_{opt}$

`&`

nested-match-name-specifier `*` *cv-qualifier-seq*$_{opt}$

match-parameter-declaration-clause:

`...`

match-parameter-declaration-list$_{opt}$

match-parameter-declaration-list `,` `...`

match-parameter-declaration-list:

match-parameter-declaration

match-parameter-declaration-list `,` *match-parameter-declaration*

match-parameter-declaration:

matct-decl-specifier-seq match-abstract-declarator$_{opt}$

match-declarator-id:

nested-match-name-specifier$_{opt}$ *match-id*

nested-match-name-specifier$_{opt}$ *match-operator-function-id*

nested-match-name-specifier$_{opt}$ *match-conversion-function-id*

match-operator-function-id:

`operator %`

`operator` *match-operator*

match-operator: one of

```
new  delete   new[]   delete[]
+    -    *    /    %%   ^    &      |    ~    !    =    <    >
+=   -=   *=   /=   %%=  ^=   &=     |=   <<   >>   >>=  <<=  ==
!=   <=   >=   &&   ||   ++   --     ,    ->*  ->   ()   []
```

match-conversion-function-id:

`operator` *match-conversion-type-id*

match-conversion-type-id:

match-type-specifier-seq match-conversion-declarator$_{opt}$

match-conversion-declarator:

match-ptr-operator match-conversion-declarator$_{opt}$

List of Examples

Index