AspectC++ Quick Reference

Concepts

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

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

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

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

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

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

match expression
Match expressions are strings containing a search pattern.

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

Aspects

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

aspect *A* **{ ... };**
defines the aspect *A*

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

Advice Declarations

advice *pointcut* : **before**(...) {...}
the advice code is executed before the join points in the pointcut

advice *pointcut* : **after**(...) {...}
the advice code is executed after the join points in the pointcut

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

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

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

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

Pointcut Expressions

Type Matching

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

`"% *"`
matches any pointer type

Namespace and Class Matching

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

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

Function Matching

`"void reset()"`
matches the function *reset* having no parameters and returning `void`

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

`"% ...::%(...)"`
matches any function, operator function, or type conversion function (in any class or namespace)

`"% ...::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`

`"static % ...::%(...)"`
matches any static member or non-member function

Variable Matching

`"int counter"`
matches the variable *counter* of type `int`

`"% guard"`
matches the global variable *guard* of any type

`"% ...::%"`
matches any variable (in any class or namespace)

`"static % ...::%"`
matches any static member or non-member variable

Template Matching†

`"std::set<...>"`
matches all template instances of the class *std::set*

`"std::set<int>"`
matches only the template instance *std::set<int>*

`"% ...::%<...>::%(...)"`
matches any member function from any template class instance in any scope

Predefined Pointcut Functions

Functions / Variables

call(*pointcut*) $N \rightarrow C_C$‡‡
provides all join points where a named and user provided entity in the *pointcut* is called.

builtin(*pointcut*)‡ $N \rightarrow C_B$
provides all join points where a named built-in operator in the *pointcut* is called.

execution(*pointcut*) $N \rightarrow C_E$
provides all join points referring to the implementation of a named entity in the *pointcut*.

construction(*pointcut*) $N \rightarrow C_{Cons}$
all join points where an instance of the given class(es) is constructed.

destruction(*pointcut*) $N \rightarrow C_{Des}$
all join points where an instance of the given class(es) is destructed.

get(*pointcut*) $N \rightarrow C_G$
provides all join points where a global variable or data member in the *pointcut* is read.

set(*pointcut*) $N \rightarrow C_S$
provides all join points where a global variable or data member in the *pointcut* is written.

ref(*pointcut*) $N \rightarrow C_R$
provides all join points where a reference (reference type or pointer) to a global variable or data member in the *pointcut* is created.

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

Control Flow

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

Types

base(*pointcut*) $N \rightarrow N_{C,F}$
returns all base classes resp. redefined functions of classes in the *pointcut*

derived(*pointcut*) $N \rightarrow N_{C,F}$
returns all classes in the *pointcut* and all classes derived from them resp. all redefined functions of derived classes

Scope

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

member(*pointcut*) N→N
maps the scopes given in *pointcut* to any contained named entities. Thus a class name for example is mapped to all contained member functions, variables and nested types.

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 or other access 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 or other access join point is an instance of a type described by the *type pattern*

args(*type pattern*, ...) (N,...)→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,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*

JoinPoint-API for Advice Code

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 [type]
object type (object initiating a call or entity access)

Target [type]
target object type (target object of a call or entity access)

Entity [type]
type of the primary referenced entity (function or variable)

MemberPtr [type]
type of the member pointer for entity or “void *” for nonmembers.

Result [type]
type of the object, used to *store* the result of the join point

Res::Type, Res::ReferredType [type]
result type of the affected function or entity access

Arg<i>::Type, Arg<i>::ReferredType [type]
type of the i^{th} argument of the affected join point (with $0 \leq i < ARGS$)

ARGS [const]
number of arguments

Array [type]
type of an accessed array

Dim<i>::Idx, Dim<i>::Size [type], [const]
type of used index and size of the i^{th} dimension (with $0 \leq i < DIMS$)

DIMS [const]
number of dimensions of an accessed array or 0 otherwise

JPID [const]
unique numeric identifier for this join point

JPTYPE [const]
numeric identifier describing the type of this join point (*AC::CALL*, *AC::BUILTIN*, *AC::EXECUTION*, *AC::*CONSTRUCTION, *AC::*DESTRUCTION, *AC::GET*, *AC::SET* or *AC::REF*)

Runtime Functions and State

static const char ***signature**()
gives a textual description of the join point (type + name)

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

static int **line**()
the source code line number in which the joinpoint shadow is located

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

Entity ***entity**()
returns a pointer to the accessed entity (function or variable) or 0 for member functions or builtin operators

MemberPtr **memberptr**()
returns a member pointer to entity or 0 for nonmembers

Result ***result**()
returns a typed pointer to the result value or 0 if there is none

Arg<i>::ReferredType ***arg***<i>*()
returns a typed pointer to the i^{th} argument value (with $0 \leq i < ARGS$)

void ***arg**(*int* i)
returns a pointer to the i^{th} argument memory location ($0 \leq i < ARGS$)

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

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

Array ***array**()
returns a typed pointer to the accessed array

Dim<i>::Idx **idx***<i>*()
returns the value of the i^{th} used index

Runtime Type Information

static AC::Type **resulttype**()
static AC::Type **argtype**(*int* i)
return a C++ ABI V3†† conforming string representation of the result type / argument type of the affected function

JoinPoint-API for Slices

The JoinPoint-API is provided within introduced slices by the built-in class **JoinPoint** (state of target class *before* introduction).

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

That [type]
The (incomplete) target type of the introduction

BASECLASSES [const]
number of baseclasses of the target class

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

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

MEMBERS [const]
number of attributes of the target class

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

Member<I>::prot, Member<I>::spec [const]
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

Example (simple tracing aspect)

```
aspect Tracing {
  advice execution("% Business::%(...)") : before() {
    cout << "before " << JoinPoint::signature() << endl;
} };
```

Reference sheet corresponding to AspectC++ 2.0, March 8, 2016. For more information visit `http://www.aspectc.org`.

† support for template instance matching is an experimental feature

‡ This feature has limitations. Please see the AspectC++ Language Reference.

†† `https://mentorembedded.github.io/cxx-abi/abi.html#mangling`

‡‡ C, C_C, C_B, C_E, C_{Cons}, C_{Des}, C_G, C_S, C_R: Code (any, only *Call*, only *Builtin*, only *Execution*, only object *Construction*, only object *Destruction*, only *G*et, only *S*et, only *R*ef)

N, N_N, N_C, N_F, N_V, N_T: Names (any, only *Namespace*, only *Class*, only *Function*, only *V*ariables, only *Type*)