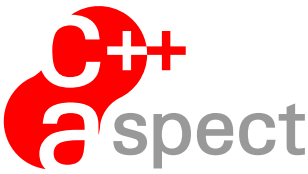


Documentation:
AspectC++ Programming Guide

The AspectC++ Developers

Version 2.5

January 16, 2026

Contents

1 Introduction to AspectC++

AspectC++ is an aspect-oriented extension to the C++ programming language. Every valid C++ program, from version C++98 to C++23 and beyond, is also a valid AspectC++ program. Beside the language extension, there is the AspectC++ compiler for translating AspectC++ programs into binary machine code just like C++ programs. This document introduces the core concepts of the AspectC++ language.

2 Join Points

The join point is the key concept in AspectC++. A join point is an identifiable location in the program source code, such as a function call or the assignment to a program variable. Join points provide the interface between C++ and AspectC++ code; that is, join points are the only places where we can apply AspectC++ code. This section describes all categories of join points needed to use AspectC++ effectively.

2.1 Dynamic Join Points

Most join points refer to the *dynamic* execution of a program at runtime. For example, a function call that takes place at runtime provides such a dynamic join point. AspectC++ can capture the control flow of the program that approaches the respective join point and you can apply advice there.

Furthermore, dynamic join points have context information associated with them. For instance, the join point of a function call provides the function arguments and, for member functions, the target object. Section 5.2 shows how to capture and make use of this context information.

2.1.1 Function Call and Execution

AspectC++ supports two types of join points for functions: call and execution join points. The execution join point is on the function body itself and covers all the statements within that function body. The following example shows the execution joint point for the function `draw()`:

```
class Square {
public:
  void draw() {
    /* ... */
  }
};
```

draw() *execution* join point

Every execution join point is associated with the fully qualified name of the respective function, such as `Square::draw()` in the aforementioned example. Pure virtual functions do not have any function body and, thus, do not provide execution join points either.

The call join point, however, typically occurs elsewhere in the program, that is, at those locations where the respective function is invoked. The following code snippet shows a call join point for the function `draw()`:

```
int main() {
  Square s;
  s.draw();
}
```

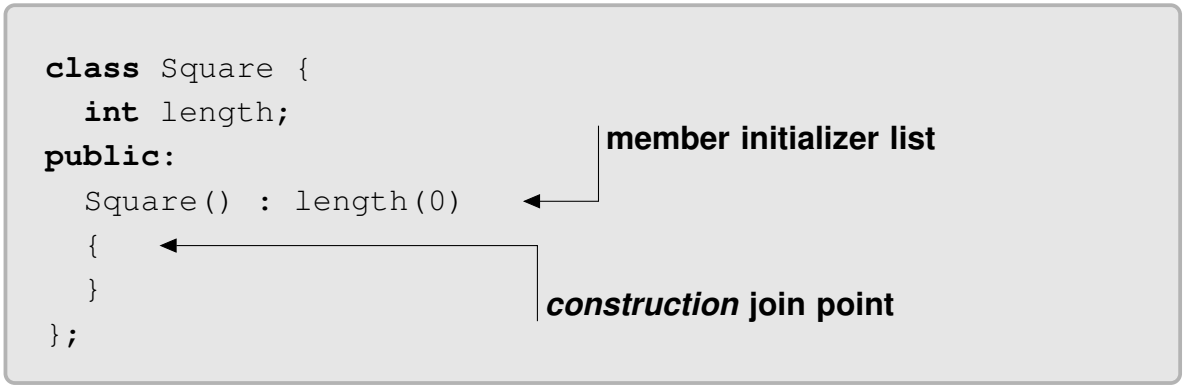
draw() *call* join point

Every call join point is associated with two fully qualified names: the name of the caller function and the name of the callee function. In the aforementioned code snippet, `Square::draw()` is the callee function and `main()` is the caller function. The name of the caller function can be used to select call join points within the `main()` function only, for example, and thus to filter out call join points elsewhere. Calls via functions pointers do not provide any call join point, as the name of the callee function cannot be determined at compile time.

In contrast to execution join points, pure virtual functions do provide call join points. AspectC++ also supports call join points for callee functions defined in external program libraries that have not been compiled with AspectC++.

2.1.2 Constructor and Destructor

Join points for class constructors are much like execution join points for member functions. The construction join point is on the constructor body itself and covers all the statements within that constructor body. The following example shows the construction joint point for a user-defined constructor:



The construction join point is located after the member initializer list; that is, the member initializer list is executed prior to the construction join point. Implicit default constructors and (implicit) copy constructors provide construction join points as well.

When a class instance is to be deleted eventually, the class destructor gets executed. The class destructor, either implicitly defined or user-defined as shown in the following code snippet, provides a destruction join point:

Both construction and destruction join points are on their respective constructor and destructor bodies. There is no call join point for constructors and destructors in AspectC++.

2.1.3 Variable Access

AspectC++ supports join points on read and write accesses to member variables and global variables. This means that there are no join points for local variables within functions. For the member and global variables, AspectC++ supports three types of join points: get, set, and ref join points. The get join point captures read access, as shown in the following example for the member variable `length`:

```
class Square {
  int length;
public:
  bool is_empty() {
    return length == 0;
  }
};
```

length *get* join point

Likewise, the set join point refers to write access to a variable. The following code snippet shows a set join point for assignment to the global variable `count`:

```
int count = 0;

int main() {
    count = 1;
}
```

initialization of a variable

count *set* join point

Note that the initialization of a global variable or member variable does not provide any set join point. The reason is that in C++, global variables are pre-initialized at once before the `main()` function is called. This pre-initialization is no event in the dynamic execution of the program and, thus, there is no dynamic join point either.

Every get and set join point is associated with the fully qualified name of the respective variable, such as `Square::length` and `count` in the above examples. As such, the variable has be accessed by name to provide a get or set join point. In other words, there is no get or set join point on indirect access via pointer or reference without the variable name. But, member access via object pointer or object reference do provide

get and set joint points, because the variable name is part of the access expression, as shown in the next example:

```
class Square {
  int length;
public:
  static Square* create() {
    Square* s = new Square;
    s->length = 0;
    return s;
  }
};
```

length *set* join point

For advanced dealing with pointers and references, AspectC++ supports the ref join point. The ref join point captures two cases: First, obtaining the address of a member or global variable provides a ref join point. Second, the initialization or binding of a reference to such a variable provides another ref join point. The following code snippet illustrates both cases:

```
int count = 0;

int main() {
    int* pointer = &count;
    int& reference = count;
}
```

count *ref* join points

The ref join point is particularly useful to restrict aliasing of variables, for example, to make sure that there are no references to a specific variable, so that get and set join points capture each read and write access to that variable, respectively.

Every ref join point is associated with the fully qualified name of the referenced variable. The get, set, and ref join points are furthermore associated with the name of function where the variable access takes place, much like the caller function for call join points. In the previous example, both ref join points are associated with the variable name `count` and the function name `main`. That function name can be used

to filter out join points in other functions, for instance.

2.1.4 Built-in Operators

Since AspectC++ 2.0, there is experimental support for call join points on C++ built-in operators. The fully qualified name of the callee function is the operator name in this case, such as the comparison `operator==`, the division `operator/`, and so on. However, there are many limitations with this experimental feature (see AspectC++ Language Reference, Section 4.5). Use it with care!

2.2 Static Join Points

Beside the various dynamic join points described in the previous section, AspectC++ also supports the static join point for user-defined data types, such as classes and structs. The static join point is on the class and struct definition, respectively. It refers to the program structure and is therefore not executable. The following piece of code shows the static join point for the class `Square`:

```
class Square {
  /* ... */
};
```

***static* join point**

Every static join point is associated with the fully qualified name of the respective data type, such as `Square` in the previous example. The static join point can be used to specify advice for introducing additional member variables, member functions, nested classes, and base classes into the respective data type.

3 Aspects

The aspect is the central language element of AspectC++ to organize crosscutting concerns in a modular way. Aspects can be understood as an extension of the C++ class concept. That is, you can define an aspect much like you would define a class, except for using the keyword `aspect` instead of `class`. As such, aspect share many features of C++ classes:

- Aspects can have member variables.
- Aspects can have member functions, constructors, and destructors.
- Aspects can inherit from classes.
- Aspects are abstract if they declare or inherit a pure virtual member function.

Furthermore, aspects provide three additional features over classes:

- Aspects can inherit from other aspects.
- Aspects are typically instantiated automatically.
- Aspects have pointcut definitions or advice or both (see Section 4 and 5).

Let's ignore pointcut definitions and advice for now and let's defer these concepts to later sections to focus on aspect design guidelines here: First, the AspectC++ developers consider the definition of an aspect in a separate header file as good design practice. Second, aspect header files always end with the `.ah` extension by convention. Third, aspect header files need include guards much like any other C++ header file. The example in Figure 1 illustrates the definition of an aspect conforming to these rules.

3.1 Instantiation

The previous section mentioned that aspects are instantiated automatically. This means that you typically don't create an object for an aspect manually. Aspects usually implement concerns that crosscut the whole program, and such aspects therefore need to exist for the whole program lifetime. Thus, the AspectC++ compiler automatically creates a single instance with static lifetime (i.e. one global object) for each aspect by default. Abstract aspects with pure virtual member functions are never instantiated. In almost all cases, you don't have to care about instantiation at all.

However, you can still change the default instantiation scheme if needed, for example, to create one object for an aspect per thread (see Section 5 for an example).

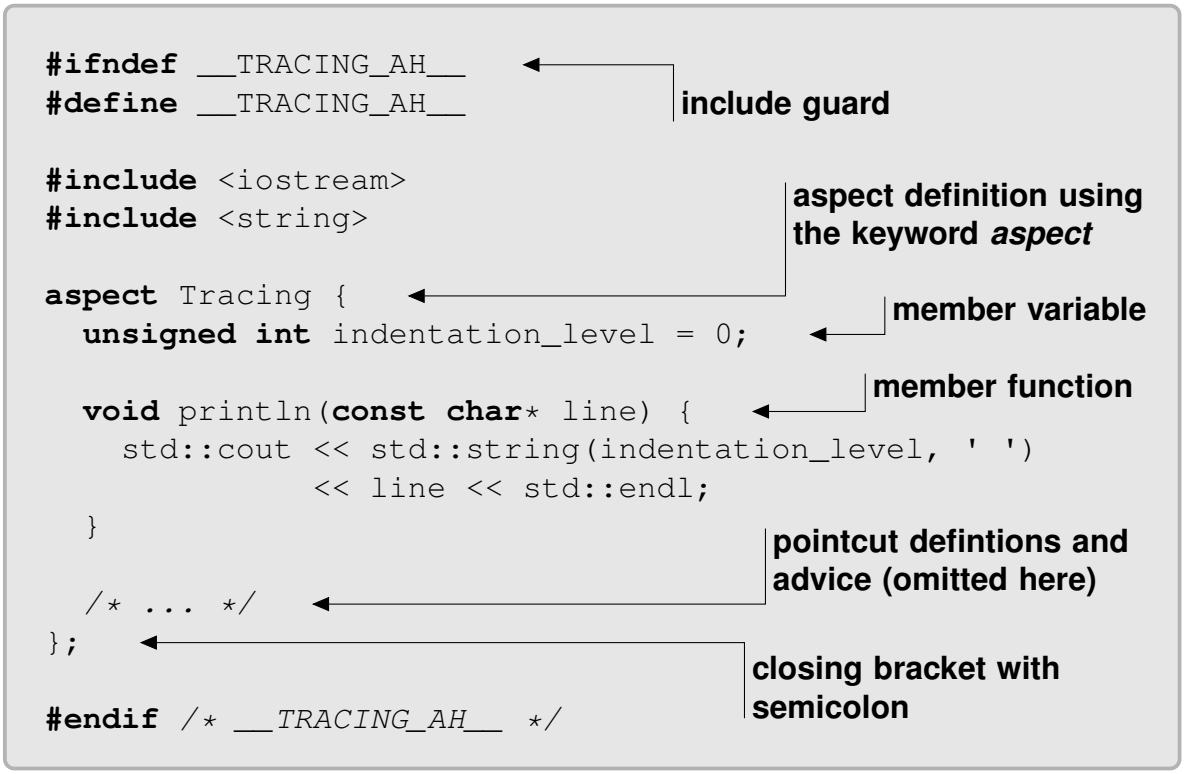


Figure 1: Aspect definition in the separate header file `Tracing.ah`

3.2 Abstract Aspects

An aspect is abstract if it declares or inherits a pure virtual member function or a pure virtual pointcut (see next section). Just like abstract classes, abstract aspects cannot be instantiated. Other aspects can inherit from abstract aspects, and if a derived aspect defines all inherited pure virtual functions and pointcuts, the AspectC++ compiler instantiates the derived aspect eventually.

The reason for abstract aspects is code reuse of common functionality. Imagine you want to apply the aspect previously shown in Figure 1, but rather than printing to the standard output stream, you want it to print to the standard *error* output stream. The previously shown aspect can be rewritten as an abstract aspect with a pure virtual `println` function as shown in the following code snippet:

```
aspect Tracing {
protected:
  unsigned int indentation_level = 0;

  virtual void println(const char* line) = 0;

  /* ... */
};
```


The actual implementation of that `println` function can now be defined in a separate aspect for the respective output stream. For example, the following piece of code shows the derived aspect that prints to the standard *error* output stream:

```
aspect ErrorStreamTracing : public Tracing {

  void println(const char* line) override {
    std::cerr << std::string(indentation_level, ' ')
              << line << std::endl;
  }
};
```


The derived aspect implements only one function and reuses all the functionality of the abstract base aspect. In this example, the abstract aspect allows the developer to define the printing functionality later and without code duplication. This design principle is particularly useful for aspects that allow for customization, such as aspects that are part of a program library.

4 Pointcuts

Pointcuts describe, or rather capture, a set of join points in the program. Given a pointcut, you can apply advice (see Section 5) on the captured join points to extend the program there. For example, you can insert additional program statements before the execution of the join points.

AspectC++ supports both *anonymous* and *named* pointcuts. You can define anonymous pointcuts at places where a pointcut expression is expected, for example, in line with the specification of advice (see Section 5). Named pointcuts, however, are defined much like functions. Typically, a named pointcut is defined as a member of an aspect, although it could be defined anywhere in the aspect header file where a function declaration is allowed. You can consider a named pointcut as a function that returns a set of join points, and you can reuse that pointcut in different parts of your aspect. A named pointcut is introduced by the keyword `pointcut` using the following syntax:

```
pointcut name(parameter-list) = pointcut-expression;
```

Let's take a look at an example:

```
aspect CaptureDraw {
  pointcut draw() = "void Square::draw()";
};
```

named pointcut and match expression

The draw pointcut has an empty parameter list and is defined as a member of the shown aspect. The right hand side of the pointcut definition refers to the function `Square::draw()` that has, in turn, no parameters and that returns void. The pointcut expression on the right hand side is a *match expression*. It matches the fully qualified name `Square::draw()` and can be used to capture all join points involving that function. The next section describes the match expression syntax in detail.

A special pointcut expression is the value of 0, which is used to declare a named pointcut as pure virtual. Just like the declaration of a pure virtual function, such a pointcut must be declared using the keyword `virtual` as shown in the following code snippet:

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

Aspects with pure virtual pointcuts are abstract and thus not instantiated (see Section 3.2). It is up to derived aspects to define the pure virtual pointcuts and, thereby, to provide the pointcut expressions for capturing the relevant join points.

4.1 Match Expressions

Match expressions are used to define pointcut expressions as shown in the previous section. They describe a set of (statically) known program entities such as types/-classes, variables, functions, or namespaces. 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 "..." 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.

Examples for match expressions

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

`"%List"`
: matches any namespace, 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

`"int* Puma::parsed_%"`
: matches any pointer variable to the type `int` whose name starts with `parsed_` and which is defined the scope `Puma`, which in turn can either be a namespace, a class, or a struct

Match expressions select program entities with respect to their definition scope, their type, and their name. The wildcard patterns turn match expressions into a powerful tool for selecting many join points at once, but you have to take care that you do not select too many – or too few – join points.

4.2 Attributes

Attributes allow you to define pointcut expressions without any match expression. You can select any program entity, such as a namespace, a class, a function, and a variable by applying one or several attributes to its definition. The following example shows the annotation of the function `is_empty()` with the C++ standard attribute `[[nodiscard]]`, which primarily instructs the compiler to issue a warning when the return value is not used:

```
[[nodiscard]] bool is_empty() {
  return length == 0;
}
```

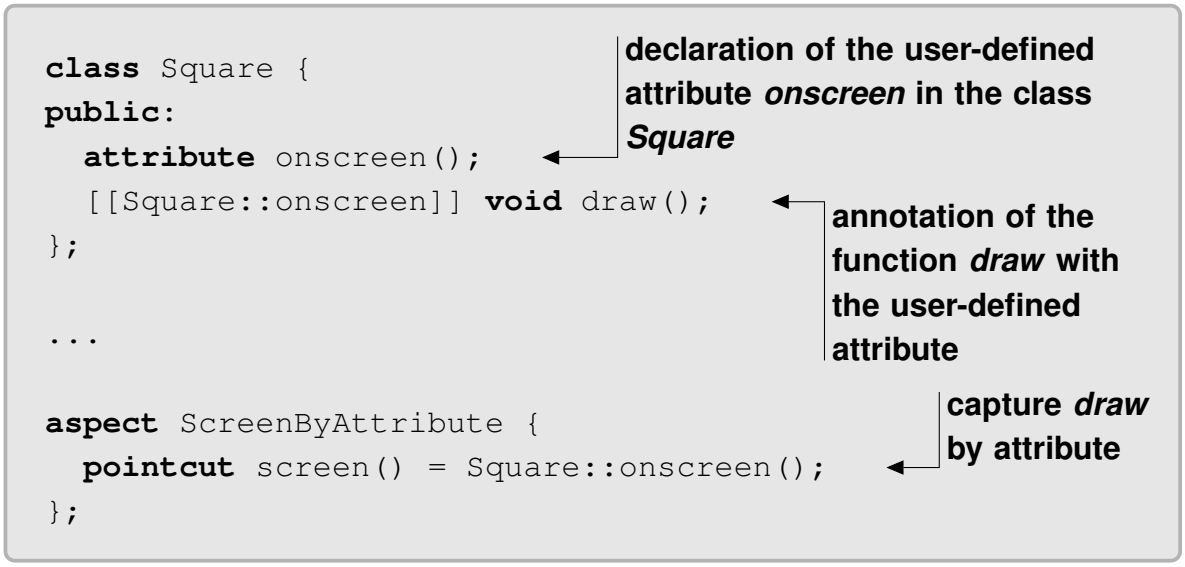
function annotation by attribute *nodiscard*

Given such a function annotation, you can now specify pointcut expressions that describe all such annotated functions. AspectC++ supports the declaration of attribute-based pointcut expressions using the `attribute` keyword and an empty parameter list as shown in the following code fragment:

```
attribute nodiscard();
```

After that declaration, the attribute can be used exactly like a named pointcut expression. In this specific example, the attribute declaration must take place in the global namespace, because the attribute `[[nodiscard]]` is already defined by the C++ standard there.

In addition to predefined standard attributes, AspectC++ supports user-defined attributes, which must be declared inside namespaces, classes, or aspects to avoid name conflicts with the standard attributes. A user-defined attribute has to be referenced later by its fully-qualified name. The following example shows the declaration and annotation of a function with a user-defined attribute:



In the shown example, the pointcut `screen()` captures all join points involving the function `Square::draw()`, because that function is annotated with the user-defined attribute. If multiple functions are annotated that way, the pointcut `screen()` captures their join points, too. Likewise, you can apply attributes on program variables as well.

Attributes can also be used to annotate classes and namespaces, as shown in the following code snippet, where the attribute name is indicated by three periods:

```
namespace [[...]] Shapes {
  class [[...]] Square;
}
```

In this case, the attribute goes in front of the identifier for the respective namespace and class. If there are multiple declarations of the same class, function, or variable, the attribute must be present at the very first declaration. The same rule applies to multiple namespace definitions.

4.3 Pointcut Operators

Pointcut expressions, such as match expressions and attribute-based pointcut expressions, can be combined by using the algebraic operators “&&”, “||”, and “!”.

- The binary || operator produces the *union set* of two pointcut expressions, that

is, it selects all join points that match either of the pointcuts or both.

- The binary && operator produces the *intersection set* of two pointcut expressions, that is, it selects only the join points that match both pointcuts.
- The binary ! operator produces the *complement* of the pointcut, that is, it selects all join points that do not match the given pointcut.

Examples for pointcut operators

`"% Square::%(...)" || "% Rectangle::%(...)"`
describes all member functions of both classes `Square` and `Rectangle`

`"% Square::%(...)" && !"void Square::draw()"`
describes all member functions of the class `Square` except `Square::draw()`

`"%List" && !"LinkedList"`
describes the set of classes with names that end with `List` but that are not named `LinkedList`

`"LinkedList" && "% ...::%(...)"`
describes all member functions of the class `LinkedList`

4.4 Predefined Pointcut Functions

All the match expressions you have seen so far in this chapter describe only sets of static program entities such as classes, structs, functions, and variables. In particular, match expressions do not capture the dynamic join points discussed in Section 2.1. Thus, we need additional language support. AspectC++ therefore provides a wide range of predefined pointcut functions, as shown in the following sections.

4.4.1 Code Pointcuts to Capture Dynamic Join Points

A code pointcut is a pointcut expression that captures a set of dynamic join points. AspectC++ supports several predefined pointcut functions that convert a pointcut expression into such a code pointcut. Table 1 presents a list of dynamic join points and shows the corresponding pointcut functions to capture the respective join points.

Dynamic join point	Predefined pointcut function
Function call	`call(pointcut expression)`
Function execution	`execution(pointcut expression)`
Constructor execution	`construction(pointcut expression)`
Destructor execution	`destruction(pointcut expression)`
Variable read access	`get(pointcut expression)`
Variable write access	`set(pointcut expression)`
Variable aliasing	`ref(pointcut expression)`
Built-in operators	`builtin(pointcut expression)`

Table 1: List of predefined pointcut functions for mapping a pointcut expression to the dynamic join points described in Section 2.1

For example, you can capture all function calls to any member function of the `Square` class by using the following pointcut expression:

```
call("% Square::%(...)")
```

Similarly, to capture all read accesses to boolean member variables of precisely that class, you can use:

```
get("bool Square::%")
```

If you are interested in both read and write accesses to precisely those member variables, you can apply the aforementioned union operator as shown below:

```
get("bool Square::%") || set("bool Square::%")
```

Finally, the execution of the respective class constructors would be captured by the following pointcut expression:

```
construction("Square")
```

Each of these predefined pointcut functions requires a suitable pointcut expression as a parameter. In particular, the call and execution pointcut functions require a pointcut expression that precisely describes program functions. Likewise, the get, set, and ref pointcut functions require a pointcut expression referring to program variables only. A notable exception is that the AspectC++ compiler automatically expands pointcut expressions that describe scopes, such as classes or namespaces, to their respective member functions and variables. For example, the following pointcut expression captures all function calls to any member function of the `Square` class:

```
call("Square")
```

This pointcut expression avoids wildcard symbols and is essentially a shorter version of first pointcut expression shown in the beginning of this section.

4.4.2 Control-flow and Scope-based Pointcuts

As mentioned in Section 2.1.1, each call join point is associated with both the name of the callee function and the name of the caller function. The call pointcut function described in the previous section filters the join points according to the callee function. AspectC++ supports another predefined pointcut function that filters on the caller

Description	Predefined pointcut function
Join points in the lexical scope	`within(pointcut expression)`
Join points in the control flow at runtime	`cflow(code pointcut)`

Table 2: List of predefined pointcut functions for scope-based and control-flow matching

function: The *within* pointcut function captures all join points that occur inside the lexical scope of the provided pointcut expression. In other words, it selects all join points in the function bodies where the name of the function is described by the pointcut expression.

The within pointcut function captures call, get, set, ref, and builtin-operator join points (see Section 2.1). As such, it typically returns a mixture of these join points and should be used in intersection with the desired kind of join point. For example, the following pointcut expression captures all function calls that take place in the body of the main function:

```
call("% ...::%(...)") && within("int main(...)")
```

Similar to all the aforementioned pointcut functions, the AspectC++ compiler evaluates the within pointcut function at compile time. This is because the compiler can easily identify the join points by just examining the function bodies. The join points statically occur there.

For dynamic relations between join points at runtime, AspectC++ provides the *cflow* pointcut function. It takes a code pointcut as a parameter and it captures join points in the control flow of that code pointcut. This means that it captures any join point *once* the control flow of the program has approached another join point, which is specified by the parameter. Consider, for example, that you want to capture the constructor execution of the class `Square`, but only for global and static objects, which are constructed prior to the main function. Thus, you want to exclude all join points in the control flow of the execution of the main function, as provided by the following pointcut expression:

```
construction("Square") && !cflow(execution("int main(...)"))
```

Description	Predefined pointcut function
Join points where the implicit C++ `this` pointer refers to an object that is compatible to the pointcut expression	**`that`**`(pointcut expression)`
Join points where the *target* object of a member function call or member variable access is compatible to the pointcut expression	**`target`**`(pointcut expression)`
Join points on functions whose return type is described by the pointcut expression	**`result`**`(pointcut expression)`
Join points on functions whose parameter types are described by the comma-separated list of pointcut expressions	**`args`**`(list of pointcut expressions)`

Table 3: List of predefined pointcut functions for filtering based on the execution context

The AspectC++ compiler evaluates the cflow pointcut function at runtime and, therefore, tracks the dynamic control flow of the program. The cflow pointcut function is only safe to use in single-threaded programs, because the current AspectC++ implementation of cflow is not thread-safe, yet.

Table 2 summarizes the cflow and within pointcut functions. We recommend using the within pointcut function when possible, because it is evaluated at compile time.

4.4.3 Context-dependent Pointcuts

AspectC++ supports four predefined pointcut functions that depend on the join-point context, such as the types of function arguments and the return type. This also includes the implicit `this` pointer argument of member functions. Table 3 shows a brief description of the four context-dependent pointcut functions.

The *that* pointcut function captures any join point where the implicit C++ `this` pointer refers to an object that is compatible to the provided match expression. This could be

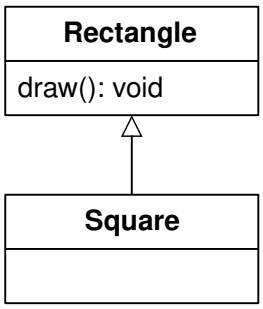


Figure 2: The class `Square` inherits from the class `Rectangle`, illustrated as UML class diagram

any dynamic join point, such as function and constructor execution. Consider the class hierarchy in Figure 2, where the class `Square` inherits from the class `Rectangle`. The execution of the draw member function, the implicit constructor, and the implicit destructor are captured by the following pointcut expression:

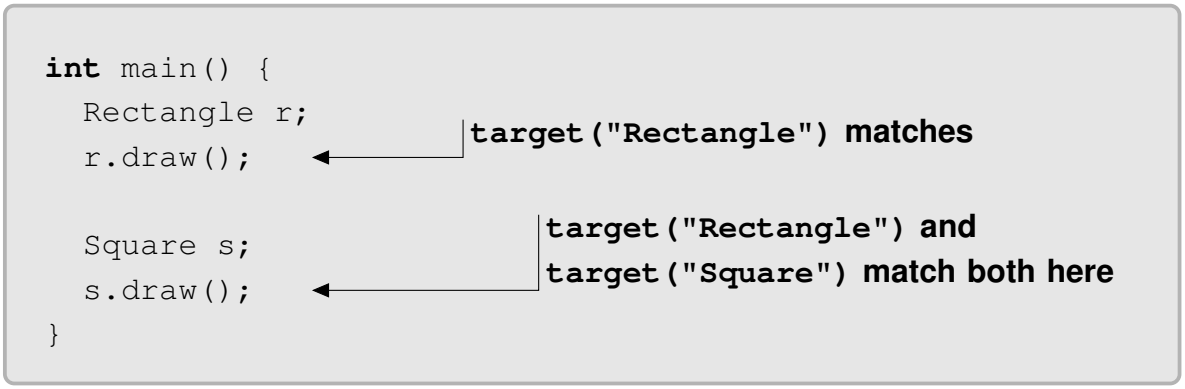
```
that("Rectangle")
```

This is because in each case, an object of `Rectangle` type is involved. Now consider the following pointcut expression:

```
that("Square")
```

This pointcut expression captures the implicit constructors and destructor of the `Square` class, respectively, and it also captures the execution of the inherited draw member function if it actually draws an instance of the `Square` class. On execution of the draw member function, the AspectC++ compiler inserts a runtime type check to find out whether the implicit C++ `this` pointer actually refers to a `Square` object. The pointcut function not only captures function execution, construction, and destruction join points. It also captures function call, get, set, ref, and builtin-operator join points where the caller object is compatible to the provided match expression.

The *target* pointcut function filters on the runtime type of the callee object for member-function calls. Likewise, it captures get, set, and ref join points on member variables whose containing object is compatible to the provided match expression. Consider the class hierarchy in Figure 2 once again and look at the two function calls in the following code snippet:



The first function call uses an object of type `Rectangle` and, thus, it is captured by the pointcut expression **`target`**`("Rectangle")`. Likewise, the second function call uses an object of type Square and, therefore, it is captured by the pointcut expression **`target`**`("Square")`. However, because a square is a rectangle, the former pointcut expression also captures the second function call. In the previously shown code snippet, the AspectC++ compiler can evaluate the target pointcut function at compile time. If the compiler cannot evaluate the final type of the callee object at compile time, it automatically inserts a runtime type check just as it does for the that pointcut function.

The two aforementioned pointcut functions, that and target, both filter on the implicit C++ `this` pointer. In addition, the *args* pointcut function filters on the explicit function parameters. As such, the pointcut function expects a comma-separated list of match expressions that each describe a data type. The args pointcut function selects only those join points where a function is involved that has exactly the same number of parameters as the pointcut function.

Examples for the *args* pointcut function

`args``("int")`
: captures any function with exactly one parameter of type `int`

`args``("%")`
: captures any function with exactly one parameter of any type

`args``("int", "%")`
: captures any function with exactly two parameters, the first of which is of type `int`

Furthermore, AspectC++ supports a pointcut function for filtering on the return type of functions: the *result* pointcut function. It takes a match expression as a parameter and captures any join point where the function's return type is described by the provided match expression. For example, you can capture any function call and execution join point on functions that return a pointer to a `Square` object by the following pointcut expression:

```
result("Square*")
```

The AspectC++ compiler evaluates both the args and result pointcut functions at compile time, meaning that no runtime type checks are carried out.

4.4.4 Subclass and Member Pointcuts

The *derived* pointcut expression allows specifying expressions that are robust with regard to subtyping. Consider the class hierarchy in Figure 2 once again and assume you want to capture all join points on the base class `Rectangle` and its derived class. You can accomplish your goal simply by using the following pointcut expression:

```
derived("Rectangle")
```

This is convenient because you do not have to worry about all the subclasses that may exist, either now or in the future. If someone implements a new subclass, the derived pointcut function captures any join point there automatically. You can also pass a match expression describing functions to the derived pointcut function as shown in the following example:

```
derived("void Rectangle::show()")
```

In this case, the pointcut expression captures any join point on the show function, either on the definition in the base class Rectangle, or on redefinitions in subclasses.

Finally, AspectC++ provides the *member* pointcut function. It takes a match expression as a parameter and returns all functions, variables, and nested types declared in the scopes described in the provided match expression. The main purpose of the member pointcut function is to capture nested classes. Table 4 summarizes both the member and derived pointcut function.

Description	Predefined pointcut function
Returns all classes provided in the pointcut expression plus all classes derived from them, in addition to all member functions provided in the pointcut expression plus all redefinitions of these member functions in derived classes	**derived**(pointcut expression)
Returns all functions, variables, and nested types declared in the scopes described in the pointcut expression	**member**(pointcut expression)

Table 4: List of predefined pointcut functions for subclasses and members (including nested types)

5 Advice

Pointcuts, as explained in the previous section, capture locations in the program *"where"* you can apply advice code. As such, the advice code defines the specific action of *"what"* to do there. Thus, the advice is the essential language concept of AspectC++ for making practical use of pointcuts. For dynamic join points in the execution of a program at runtime, advice is similar to a member function. You specify program statements that shall be executed one after the other on the respective join point.

5.1 Before, After, and Around Advice

When you apply advice to one of the dynamic join points described in Section 2.1, you can specify whether the advice shall be executed *before* or *after* the program reaches the respective join point. Alternatively, you can specify that the advice code shall be executed as a replacement for the code at the captured join point.

5.1.1 The Before Advice

Advice that takes place prior to the execution of some dynamic join point is introduced by the keyword `advice` using the following syntax:

```
advice code-pointcut :  before() { /* advice body */ }
```

Because the advice syntax expects a code pointcut, you have to use one of the predefined pointcut functions for dynamic join points (see Section 4.4.1). The keyword `before` specifies that the advice body, which contains the program statements, is executed prior to the dynamic join point. An advice has to be a member of an aspect, just like a member function. Let's take a look at an example:

```
#include <iostream>

aspect SquareLogger {
  advice execution("void Square::draw()") :
  before() {
    std::cout << "Log: Started drawing a square";
    std::cout << std::endl;
  }
};
```

code pointcut on *draw()* execution

advice body

This aspect contains one piece of advice that uses the predefined `execution` pointcut function to capture the execution of the `draw()` function by match expression. Both statements in the advice body are executed prior to any execution of the captured function. In this example, the before advice implements logging to the standard output stream. Typically, the before advice is also used for checking and enforcing preconditions for the captured code sections.

5.1.2 The After Advice

The syntax for defining the after advice is similar to the before advice, except that the keyword `after` is used. For example, the following aspect implements logging *after* the execution of the `draw()` function, that is, just before that function returns:

```
#include <iostream>

aspect SquareLogger2 {
  advice execution("void Square::draw()") :
  after() {
    std::cout << "Log: Finished drawing a square";
    std::cout << std::endl;
  }
};
```

code pointcut on *draw()* execution

advice body

Beside logging, this kind of advice is typically used for checking and enforcing post-conditions. As described in Section 4.4.1, other predefined pointcut functions can be used to apply advice *after* constructor/destructor execution, function calls, and variable accesses. The same applies to the aforementioned before advice and to the around advice as described in the following section.

5.1.3 The Around Advice

The around advice differs from the previously mentioned kinds of advice in that it completely bypasses the captured join point. This means that program code at the specific join point will not be executed by default when an around advice is applied. However, you can still execute the bypassed code if desired. AspectC++ provides the built-in function `tjp->proceed()` for executing code that was replaced by the around advice. By using that built-in function, code can be added both before and after the join point with a single advice. In the following example, the around advice implements logging (❶) before the execution of the captured `draw()` function, (❷) proceeds with the replaced code, and (❸) implements logging afterwards. Another typical use case for the around advice is executing the `tjp->proceed()` function in a try/catch block to handle exceptions transparently.

```
#include <iostream>

aspect SquareLogger3 {
  advice execution("void Square::draw()") :
  around() {
    std::cout << "Log: Started drawing a square"
              << std::endl;
    tjp->proceed();
    std::cout << "Log: Finished drawing a square"
              << std::endl;
  }
};
```

replace the execution of the *draw()* function by the advice body

❶ logging before

❷ execute the *draw()* function

❸ logging after

5.2 Accessing Context

The previously shown advice bodies with their program statements look similar to the bodies of member functions (except for the built-in `tjp->proceed` function that is only available in the around advice). Much like member functions, advice can also have parameters, referred to as *context variables*. You can use such context variables for passing data from the join point to the advice body. AspectC++ supports four kinds of context information that correspond to the four context-dependent pointcut functions listed in Table 3 on page 21: *that*, *target*, *args*, and *result*. Either of these pointcut functions can be used for binding context variables to the specific context information at the respective join point and, thus, for making that information available in the advice body.

5.2.1 That

AspectC++ provides access to the implicit C++ `this` pointer by the pointcut function *that*. It exposes the involved object for execution join points on member functions, construction join points, and destruction join points. You can declare a context variable of a pointer or reference type alongside the definition of advice, and you can bind that context variable by using the pointcut function *that*. The binding might involve a runtime type check if the underlying type inherits from a base class as explained in

Section 4.4.3.

The following example shows the declaration of the context variable `square` as a reference to a `Square` object in the parentheses after the `before` keyword (❶). At the same time, the pointcut function *that* binds the context variable to the implicit C++ `this` pointer (❷). It thereby captures any join point where the implicit C++ `this` pointer refers to an object that is compatible to the type of the context variable. In addition, the advice definition only captures the execution of the `draw()` member function because of the intersection operator (`&&`). As a result, the context variable `square` can be used in the advice body (❸) for accessing the object connected to the `draw()` member function as if it was a local variable.

```
#include <iostream>

aspect SquareLogger {
  advice execution("void Square::draw()") &&
  that(square) :
  before(Square& square) {
    std::cout << "Log: Drawing a square with length "
              << square.length
              << std::endl;
  }
};
```

❶ declaration of the context variable *square*

❷ binding of the context variable

❸ using the context variable *square* for accessing the data member *length*

The pointcut function *that* can also be used in intersection with the call, get, set, and ref pointcut functions (see Section 4.4.1). In these cases, it exposes the caller object, that is, the implicit C++ `this` pointer *prior* to the captured function call or variable access. In summary, the pointcut function *that* always provides access to the current C++ `this` pointer on any dynamic join point where that pointer is available.

5.2.2 Target

As explained in the previous section, the pointcut function *that* provides information on the caller object on call and variable-access join points. On such join points, the pointcut function *target* exposes the *callee* object likewise. The following example is similar to the previous one, but the shown advice captures the *call* join point on

the `draw()` function and, thus, binds the context variable `square` by the pointcut function *target* to the callee object.

```
#include <iostream>

aspect SquareLogger {
  advice call("void Square::draw()") &&
  target(square) :
  before(Square& square) {
    std::cout << "Log: Drawing a square with length "
              << square.length
              << std::endl;
  }
};
```

❷ binding of the context variable

❶ declaration of the context variable *square*

❸ using the context variable *square* for accessing the data member *length*

If you want to access both the caller and callee objects in the advice body at the same time, you can use the pointcut functions *that* and *target* in conjunction with two different context variables. The following code snippet exemplarily shows the declaration of two different context variables as a comma-separated list in the advice definition. Either context variable is bound by another pointcut function. As a result, the advice captures all call join points on the member function `draw()` that occur within some member function of the type `Screen`. Both the caller and callee object are thereby accessible in the advice body via references.

```
aspect SquareDrawingFromScreen {
  advice call("void Square::draw()") &&
  within("% Screen::%(...)") &&
  that(screen) &&
  target(square) :
  before(Screen& screen, Square& square) {
    /* ... */
  }
};
```

binding of *screen* (caller object)

binding of *square* (callee object)

declaration of two context variables

5.2.3 Args

The previous sections showed how advice code can get access to the C++ `this` pointer, which is an implicit argument of member functions. AspectC++ also provides access to the explicit function arguments via the *args* pointcut function. This pointcut function is particularly useful for call, execution and construction join points. It allows you to pass function arguments to the advice body by value. The following example shows the class `Rectangle` with a `set_size` member function that has two integer parameters `x` and `y`. The example also shows an aspect with call advice for that member function. In the definition of the advice, there is also a declaration of two context variables of integer type (❶). At the same time, the *args* pointcut function (❷) binds both context variables to the two arguments of the captured `set_size` function in the specified order. To achieve this, the types of the context variables and function parameters must be identical. Finally, both context variables can be used in the advice body (❸).

```
#include <iostream>

class Rectangle {
  int x, y;
public:
  void set_size(int x, int y) {
    this->x = x; this->y = y;
  }
};

aspect RectangleLogger {
  advice call("void Rectangle::set_size(...)") &&
  args(x, y) :                                    ❷ binding
  before(int x, int y) {                          ❶ declaration of the context variables
    std::cout << "Log: Setting rectangle size to x="
              << x << " and y=" << y              ❸ using the context variables x and y
              << std::endl;
  }
};
```

Note that the *args* pointcut function expects exactly the same number of parameters as the captured function provides. For instance, if you only want to read the first argument `x` in the previous example, you nevertheless have to use the *args* pointcut function with two parameters. However, you do not need to bind every argument to a context variable. You can simply provide match expressions for unbound arguments, as shown in the following examples:

Examples for the *args* pointcut function

`args``(x, y)`
: captures any function with exactly two parameters and binds both to the context variables x and y

`args``(x, "int")`
: captures any function with exactly two parameters (the second has to be an integer) and binds only the first argument to the context variable x

`args``("%", y)`
: captures any function with exactly two parameters (the first can be of any type) and binds only the second argument to the context variable y

5.2.4 Result

Last but not least, the return value of a function is exposed by the *result* pointcut function on call and execution join points. In addition, it exposes a variable's value on get join points. In either case, you must use the *result* pointcut function only in combination with after or around advice, and in the latter case, you must use the bound context variable only after `tjp->proceed`. This is because the return value of a function is only defined once the function has returned. Using it prior to that causes undefined behavior.

The next example picks up the factory function `create()` that returns a pointer to a `Square` object as described in Section 2.1.3. The example shows an after advice that declares a context variable with exactly the same type as the return value of the captured function (❶). That context variable is bound to the function's return value (❷). Therefore, the advice body gains access to the returned value and prints the memory address where it points to (❸).

```
#include <iostream>

aspect SquareFactoryLogger {
  advice execution("Square* Square::create()") &&
  result(square) :
  after(Square* square) {
    std::cout << "Log: Created a square at address "
              << square
              << std::endl;
  }
};
```

❷ binding

❶ declaration of the context variable

❸ using the context variable *square* that points to the returned object

5.2.5 Context Variables in Named Pointcuts

Section 4 introduced named pointcuts using the following syntax:

```
pointcut name(parameter-list) = pointcut-expression;
```

However, the parameter list has not been used at all, yet. Its sole purpose is the passing of context variables, as illustrated in the following example:

```
#include <iostream>

aspect SquareFactoryLogger2 {
  pointcut factory(Square* s) =
    execution("Square* Square::create()") &&
    result(s);

  advice factory(square) :
  after(Square* square) {
    std::cout << "Log: Created a square at address "
              << square << std::endl;
  }
};
```

named pointcut *factory* with one context variable as parameter

binding of the context variable

use of the named pointcut

declaration of the context variable

The shown example is semantically equivalent to the example in the previous section. It differs syntactically in that it defines the named pointcut `factory` with one context variable as parameter. The context variable is still declared alongside the definition of advice, but it is then passed as argument to the named pointcut `factory` that in turn binds it to the return value. This language feature of AspectC++ supports the reuse of the same pointcut by multiple pieces of advice in order to avoid repeated advice definitions.

5.3 Slice Introductions

In addition to advice for dynamic join points at runtime, you may want to extend the static program structure at compile time. This section shows advice for introducing additional members (variables, functions, and types) and base classes into the program's data structures in a crosscutting way.

5.3.1 Member Introduction

AspectC++ supports the definition of class fragments that can be inserted transparently into other C++ classes. Class fragments introduce new member variables, member functions, and nested types. Consider, for example, the aforementioned class `Square` that only contains the single member variable `length` (see Section 2.1.2).

Now, suppose you want to draw multiple squares with different colors. Thus, you want a square object to store its individual color in a separate member variable. Likewise, the same member variable is potentially needed for similar classes such as `Circle`, `Hexagon`, and so on. Therefore, the member variable for the color can be declared in a reusable class fragment. We recommend defining such a class fragment in a separate aspect header file, with separate include guards much like any other aspect header file (see Section 3). The following example shows the aspect header file `ColorIntroduction.ah` (include guards omitted) that defines a class fragment with the same name using the keyword `slice`.

ColorIntroduction.ah

```
enum Color { red, green, blue };

slice class ColorIntroduction {
private:
  Color color = Color::red;
public:
  Color getColor() const { return color; }
  void setColor(Color c) { color = c; }
};
```

definition of the class fragment using the keyword *slice*

member variable

member functions

The class fragment differs syntactically from a regular C++ class by only the additional keyword `slice`. As such, a class fragment contains member variables, member functions, nested types, and possibly base classes. However, a class fragment cannot be instantiated directly. There must be an aspect and advice for introducing the class fragment into a regular C++ class, which in turn can be instantiated.

For example, the following aspect header file `ColorExtension.ah` shows an aspect that defines advice for introducing the aforementioned class fragment into the class `Square`.

ColorExtension.ah

```
#include "ColorIntroduction.ah"

aspect ColorExtension {
  advice "Square" : slice ColorIntroduction;
};
```

introduction into the target class *Square*

First, the header file that defines the respective class fragment needs to be included. After that, the aspect `ColorExtension` specifies advice for the static join point on the class `Square` by using a match expression in quotes. The `slice` keyword after the colon allows the introduction of the class fragment specified after it. In other words, the AspectC++ compiler inserts the member variable and functions of the class fragment into the target class `Square` as if they had been declared there manually.

Likewise, AspectC++ supports the introduction of static member variables and member functions. Consider, as another example, a class `Screen` where you want to apply the singleton design pattern. The following class fragment declares a static member function that turns any class into a singleton.

Singleton.ah

```
slice class Singleton {
public:
  static Singleton& getInstance();
};

slice Singleton& Singleton::getInstance() {
    static Singleton instance;
    return instance;
}
```

definition of the class fragment

declaration of a static member function

definition of the static member function

The name of the class fragment is `Singleton` and its static member function seemingly returns a reference to such an object. However, because class fragments cannot be instantiated directly as mentioned earlier, the name `Singleton` actually refers to the type of the respective class where this fragment is introduced. Thus, if it is applied to the class `Screen`, the static member function returns a reference to a `Screen` object. Likewise, if it is applied to any other class, the function returns a reference to the respective class type. The name of the class fragment identifies the type of the receiving class in a generic way.

To give an example for non-inline member functions, the shown class fragment contains only a declaration of its member function. The function itself is later defined non-inline using the the keyword `slice`. The AspectC++ compiler takes care of introducing that function into the translation units as necessary. In particular, the AspectC++ compiler introduces the function into the same translation unit where the first regular non-inline function of the target class is defined already.

Finally, to apply the singleton class fragment to the aforementioned class `Screen`, you also need an aspect and advice as exemplarily shown in the following piece of code:

SingletonDesignPattern.ah

```
#include "Singleton.ah"

aspect SingletonDesignPattern {
  advice "Screen" : slice Singleton;
};
```

apply the singleton design pattern to the class *Screen*

5.3.2 Introduction of Base Classes

As mentioned in the previous section, class fragments can inherit from base classes. Once such a class fragment is introduced into a regular class, the regular class inherits from the class fragment's base classes. If you only want to modify the class hierarchy without introducing additional members, you can omit the definition of the class fragment and you can use an abbreviated syntax. For example, consider the following class `Drawable` that declares a pure virtual `draw()` function as an interface:

Drawable.h

```
class Drawable {
public:
  virtual void draw() = 0;
};
```

You can introduce that class as a base class into the previously mentioned class `Square` by defining the following aspect and advice:

DrawableInterface.ah

```
#include "Drawable.h"

aspect DrawableInterface {
  advice "Square" : slice class : public Drawable;
};
```

In short, the advice specifies that the class `Square` shall inherit from `Drawable`. By the way, the class `Square` already implements the `draw()` function (see Section 2.1.1) and thereby satisfies the introduced interface.

5.4 Ordering of Advice

In a large software system, it is possible that multiple pieces of advice affect the same join point. Thus, it is necessary to control the order in which the AspectC++ compiler applies the advice on a particular join point. For example, consider a class implementing a restaurant, with member functions for entering and exiting the restaurant.

Restaurant.h

```
#include <iostream>

class Restaurant {
public:
  void enter() { std::cout << "Entering" << std::endl; }
  void exit() { std::cout << "Exiting" << std::endl; }
};
```

Now, assume that customers want to hand their jackets in at the cloakroom after entering the restaurant; prior to exiting, they want to return their jackets. The following aspect implements these activities at the cloakroom.

Cloakroom.ah

```
#include <iostream>

aspect Cloakroom {
  advice execution("void Restaurant::enter()") : after(){
    std::cout << "Visiting cloakroom" << std::endl;
  }

  advice execution("void Restaurant::exit()") : before(){
    std::cout << "Visiting cloakroom" << std::endl;
  }
};
```

In addition, consider that the restaurant provides seating. Thus, a customers takes a seat after entering the restaurant, and leaves it prior to exiting. These activities can be implemented exemplarily by the following aspect.

Seat.ah

```
#include <iostream>

aspect Seat {
  advice execution("void Restaurant::enter()") : after(){
    std::cout << "Taking a seat" << std::endl;
  }

  advice execution("void Restaurant::exit()") : before(){
    std::cout << "Leaving the seat" << std::endl;
  }
};
```

Finally, let's create a simple test program to exercise the interaction of the two aspects.

TestRestaurant.cpp

```
#include "Restaurant.h"

int main() {
  Restaurant r;
  r.enter();
  r.exit();
}
```

Running that test program after compiling it with AspectC++ (and both aspects) could produce the following text output:

```
Entering
Taking a seat
Visiting cloakroom
Visiting cloakroom
Leaving the seat
Exiting
```

The order of advice is implicitly *implementation-defined* unless specified otherwise. AspectC++ supports advice for specifying an explicit order by using the keyword `order` with the following syntax:

```
advice pointcut : order(list of pointcuts);
```

The pointcut expression on the left side of the colon refers to the join points on which the order shall be specified. The list of pointcut expressions on the right side defines the precedence of aspects that apply to those join points. You typically provide a list of match expressions that describe the involved aspects. In the previous example, a typical behavior at a restaurant (i.e., visiting the cloakroom prior to taking a seat) can be specified exemplarily by the following aspect.

RestaurantCoordination.ah

```
aspect RestaurantCoordination {
  advice execution("void Restaurant::%()") :
  order("Seat", "Cloakroom");
};
```

The aspect *Seat* shall have higher precedence

Running the aforementioned test program after compiling it with all the previously shown aspects produces a more sensible text output:

```
Entering
Visiting cloakroom
Taking a seat
Leaving the seat
Visiting cloakroom
Exiting
```

The latter aspect defines the aspect `Seat` to have higher precedence than `Cloakroom`. In general, aspect precedence implies the following rules:

- For before advice, the advice of a high-precedence aspect is executed *prior* to that of a low-precedence aspect.
- For after advice, the advice of a high-precedence aspect is executed *later* than that of a low-precedence aspect.
- For around advice, the advice of a high-precedence aspect *encloses* the advice of a low-precedence aspect. Only if the high-precedence aspect invokes `tjp->proceed()`, advice of the low-precedence aspect is executed.
- For slice introductions (see Section 5.3), the class fragment of a high-precedence aspect is inserted first.

The previously shown order advice specifies a partial order between the two aspects `Seat` and `Cloakroom` only. If further aspects apply to the same join points, their order is still implementation-defined. You can use pointcuts with wildcard expressions (see Section 4.1) for specifying a global order on a join point. For example, the following order defines the aspect `Cloakroom` to have always the lowest precedence:

```
order(!"Cloakroom", "Cloakroom");
```

Likewise, the aspect `Seat` can be defined to always have the highest precedence as follows:

```
order("Seat", !"Seat");
```

If you specify advice for conflicting precedence, for example, two different aspects with both the highest precedence for a certain join point, the AspectC++ compiler provides an error message at compile time to indicate the impossibility of ordering.