1. Об этой книге

Этот документ предназначен для ознакомления с элементами AspectC++. Он глубоко описывает использование и назначение каждого элемента в сопровождении с примерами. Этот документ предназначен для опытных программистов. Пошаговое описание даётся в книге AspectC++ Programming Guide, которая, к сожалению ещё не написана. Полную информацию об AC++, который является компилятором AspectC++, вы можете получить из книги AC++ Compiler Manual.

AspectC++ – это аспектно-ориентированное добавление в C++. Хотя он и похож на AspectJ, он соответствует природе языка C++, являясь в некоторых местах совсем другим. Первая часть этого документа – введение в AspectC++. Детальное описание каждого элемента AspectC++ изложено во второй части документации.

2. Основные понятия

2.1. Разделительные точки (pointcuts)

Аспекты в AspectC++ реализуют подход пересечения модулей. С этой точки зрения наиболее важными элементами AspectC++ являются разделительные точки (pointcuts). Разделительная точка (pointcut) описывает набор соединительных точек (join points) с помощью вычисления состояния аспекта, который вступит в силу. Таким образом, каждая соединительная точка (join point) может ссылаться на функцию, атрибут, тип, переменную или на точку, откуда соединительная точка (join point) обратится, чтобы например событие достигло заданной позиции кода. Разделительные точки (pointcuts) вычисляются во время компиляции или во время выполнения программы в зависимости от их типов.

2.1.1. Сопоставимые выражения (match expressions)

В AspectC++ существуют два типа разделительных точек (pointcuts): кодовые разделительные точки (code pointcuts) и именованные разделительные точки (name pointcuts). Именованные разделительные точки (name pointcuts) описывают набор (статически) известных программных сущностей, таких как типы, атрибуты, функции, переменные или пространства имён. Все именованные разделительные точки (name pointcuts) основываются на сопоставимых выражениях. Сопоставимое выражение (match expression) может восприниматься, как шаблон поиска. В таком шаблоне поиска специальный символ “%” распознаётся, как групповой символ (wildcard) для имён или частей отличительных признаков. Специальный символ последовательности “…” заменяет некоторое количество параметров в функции или некоторую область видимости в уточненном имени. Сопоставимое выражение (match expression) – это строка, ссылающаяся на выражение.

Примеры: сопоставимые выражения (match expressions) (именованные соединительные точки (name pointcuts))

"int C::%(…)"
Сопоставимо любому методу класса C, который возвращает значение типа int
"%List"
Сопоставимо любой структуре, классу, объединению или перечислению, чьё имя заканчивается на "List"
"% printf (const char*, ...)"
Сопоставимо функции printf (в глобальной области видимости), первый параметр которой имеет тип const char* и возвращающей значение любого типа
"const %& …::%(…)"
Сопоставимо всем функциям, которые возвращают ссылку на константный объект

Сопоставимые выражения (match expressions) выбирают программные сущности с учётом их области видимости, типа и имени. Детальное описание семантики сопоставимых выражений (match expressions) приведено в разделе 3 на странице 16. Грамматика, определяющая корректность сопоставимых выражений (match expressions) показана в приложении B.

2.1.2. Выражения, состоящие из разделительных точек (pointcut expressions)

Другой тип разделительных точек (pointcuts) – кодовые разделительные точки (code pointcuts), описывающие пересечение наборов точек под контролем потока программы. Кодовые разделительные точки (code pointcuts) могут ссылаться на вызов функции или исполнение какой-либо её точки. Они могут быть созданы только с помощью именованных разделительных точек (name pointcuts), так как все соединительные точки (join points), поддерживаемые AspectC++, должны определяться одним именем в списке. Это достигнуто вызовом предопределённых функций разделительных точек (pointcut functions) в выражениях, состоящих из разделительных точек (pointcut expressions), ожидающих разделительную точку (pointcut) в качестве аргумента. Пример такой функции – within (pointcut), которая фильтрует все соединительные точки, содержащие функции или классы в данной разделительной точке (pointcut).

Именованные и кодовые разделительные точки (name and code pointcuts) могут комбинироваться с помощью алгебраических операторов "&&", "||", и "!".

Примеры: выражения, состоящие из разделительных точек (pointcut expressions)

```
"%List" && !derived  ("Queue" )
```
Описывает набор классов с именами, заканчивающимися на “List”, и не наследуемых от класса Queue

```
call ( "void draw ( )" ) && within ( "Shape" )
```
Описывает набор вызовов функции draw, находящихся внутри методов класса Shape

2.1.3. Типы соединительных точек (join points)

В AspectC++ существуют два типа соединительных точек (join points), соответственно двум поддерживаемым типам разделительных точек (pointcuts). Основываясь на коротких фрагментах кода, необходимо найти различия и взаимоотношения этих двух типов точек соединения.

```
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;
}
```

Кодовые соединительные точки (code join points) используются для формирования кодовых разделительных точек (code pointcuts) и именованных соединительных точек (name join points) (т.е. имён), которые используются для формирования именованных разделительных точек (name pointcuts). Схема 1 показывает некоторые соединительные точки (join points) фрагмента кода и то, как они соотносятся.

Каждое выполнение соединительной точки (join point) ассоциируется с именем исполняемой функции. Чисто виртуальные функции (pure virtual functions) не являются исполняемыми. Таким образом, код для выполнения соединительных точек (join points) никогда не будет запущен для этого типа функции, хотя вызов соединительной точки (join point) с целевой функцией (target function) такого типа вполне допустим.

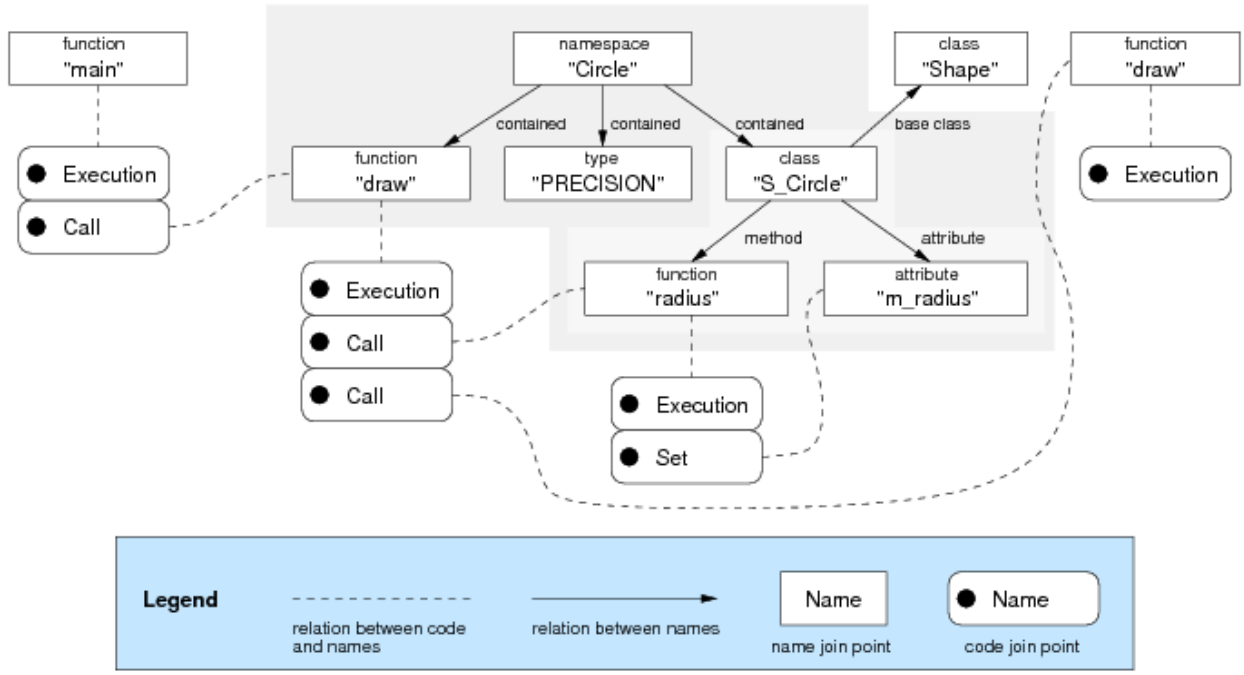

Схема 1.

Каждый вызов соединительной точки (join point) ассоциируется с двумя именами: именем источника и целевой функции (target function) вызова функции. Так как возможны многократные вызовы функций с одинаковой функцией, каждое имя функции может ассоциироваться со списком вызовов соединительных точек (join points). Конструирование (construction) соединительной точки (join point) означает специфическую последовательность инструкций, при которых объект создаётся. Аналогично созданию, уничтожение соединительной точки (join point) означает уничтожение объекта.

2.1.4. Объявления разделительных точек (pointcuts)

В AspectC++ предусмотрена возможность именовать выражения из разделительных точек (pointcut expressions) с помощью объявлений. Это делает возможным многократное использование выражений из разделительных точек (pointcut expressions) в различных местах программы. Они допустимы там, где допустимы объявления C++. В связи с этим обычные имена C++ подходят и для объявления разделительных точек (pointcuts).

Разделительная точка (pointcut) объявляется с помощью ключевого слова pointcut.

Пример: объявление разделительной точки (pointcut)

```
pointcut lists ( ) = derived ("Lists");
lists сейчас может использоваться в любом месте программы вместо derived ("Lists").
```

Кроме того, объявление разделительных точек (pointcuts) может использоваться для объявления чисто виртуальных разделительных точек (pure virtual pointcuts). Это даёт возможность абстрактного аспекта (aspect) многократного использования. Такие аспекты обсуждаются в секции 2.4. Синтаксически объявления чисто виртуальных разделительных точек (pure virtual pointcuts) аналогично объявлению обычных разделительных точек (pointcuts), но с добавлением ключевого слова virtual, причем вместо выражения, состоящего из разделительных точек (pointcut expression) надо написать "0".

Пример: объявление чисто виртуальной разделительной точки (pure virtual pointcut)

```
pointcut virtual methods () = 0;
methods теперь должен быть переопределён в каждом производном аспекте с 0 на
ссылку на корректное выражение из разделительных точек (pointcut expression).
```

2.2. Контекстные фрагменты (slices)

Контекстный фрагмент (slice) – это фрагмент элемента языка C++, который определяет контекст. Он может использоваться, как фрагмент для расширения статической структуры программы. Например, элементы класса, являющегося контекстным фрагментом, (slice class) могут сливаться в одном или нескольких целевых классах (target classes), как внесенные фрагменты. Приведенный ниже пример показывает объявление простого класса, являющегося контекстным фрагментом (slice class).

Пример: объявление класса, являющегося контекстным фрагментом (slice class)

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

2.3. Фрагментированный код (advice code)

Для кодовой соединительной точки (code join point), так называемый фрагмент кода (advice code) может быть границей (bound). Фрагмент кода (advice code) может быть понят, как действие, активируемое аспектом, когда надлежащий код соединительной точки (join point) в программе достигается. Активация фрагмента кода (advice code) может случиться до, после или и до, и после того, как код соединительной точки (join point) будет достигнут. Элемент языка AspectC++, чтобы создавать фрагменты кода (specify advice code) – это объявление фрагментов (advice declaration). Оно вводится с помощью ключевого слова advice, следующего в определении выражений из разделительных точек, определяя (pointcut expressions) где и на основании каких ситуаций фрагмент кода (advice code) должен быть активирован.

Пример: определение фрагмента (advice declaration)

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

Фрагмент кода :before (), сопровождающийся выражением из разделительных точек (pointcut expression) означает, что фрагмент кода (advice code) должен быть активирован до того, как кодовая соединительная точка (code join point) будет достигнута. Также здесь возможно использование :after (), что означает выполнение после достижения кодовой соединительной точки (code join point). Соответственно :around () означает, что фрагмент кода (advice code) будет выполнен вместо (instead) кода, описываемого кодовой соединительной точкой (code join point). В окружённом фрагменте (around pattern) фрагмент кода может явно порождать исключение программного кода в соединительной точке (join point) следовательно этот фрагмент кода может быть выполнен до и после соединительной точки. Не предусмотрено специальных прав доступа к фрагменту кода (advice code) относительно кода программы в соединительной точке (join point).

Помимо чистого описания соединительных точек (join points), разделительные точки (pointcuts) также могут связывать переменные с контекстной информацией соединительной точки (join point). Так, например фактическое значение аргумента вызова функции может быть доступным для фрагмента кода (advice code).

Пример: объявление фрагмента с доступом к контекстной информации

```
  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;
  }
```

В примере сверху разделительная точка (pointcut) new_user определяется со включенной в неё контекстную переменную name, которая присоединяется к ней. Это значит, что значение типа const char* доставляется каждый раз, когда соединительная точка (join point), описанная в разделительной точке (pointcut) new_user достигается. Функция разделительной точки (pointcut function) args используется в выражении из разделительных точек (pointcut expression), освобождая все соединительные точки (join points), в программе, где тип аргумента const char* используется. Поэтому args (name) доступно с выполнением соединительной точки (join point), привязывающей name к первому и единственному параметру функции login.

Объявление фрагмента в примере выше следующего объявления разделительной точки (pointcut declaration) связывает выполнение фрагмента кода с событием, когда соединительная точка (join point), описанная в new_user достигается. Контекстная переменная, которая содержит фактическое значение параметра достижения соединительной точки (join point) обязана быть объявлена как формальный параметр before, after или around. Этот параметр может использоваться во фрагменте кода (advice code) обыкновенный параметр функции.

Рядом с функцией разделительной точки (pointcut function) args, привязывающиеся контекстные переменные исполняются с помощью that, target и result. В одно и то же время эти функции разделительных точек (pointcut functions) действительны, как фильтры, соответствующие типу контекстной переменной. Например args в примере сверху фильтрует все соединительные точки, имеющие параметр типа const char*.

2.3.1. Представления (introductions)

Второй тип фрагментов, поддерживаемый AspectC++ – это представления (introductions). Представления (introductions) используются, чтобы конкретизировать код программы и структуры данных. Нижестоящий пример расширяет два класса: каждый посредством атрибута или метода.

Пример: представления (introduction)

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

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

Как и обыкновенное определение шаблона (advice), представление (introduction) вводится с помощью ключевого слова advice. Если следующая разделительная точка (pointcut) является именованной разделительной точкой (name pointcut), то определение класса, являющегося контекстным фрагментом (slice declaration), следующее за символом ":" помещается в классы и аспекты (aspects), описываемые точкой разделения (pointcut). Внедрённый код может в дальнейшем использоваться в нормальном коде программы, как любая другая функция, атрибут и т.д. Фрагмент кода (advice code) в представлениях (introductions) имеет полные права доступа к коду программы в соединительной точке (join point), т.е. метод, внедрённый в класс, имеет чёткий доступ к приватным (private) членам того класса.

Контекстные фрагменты (slice) могут также быть использованы для внедрения новых базовых классов. В первой строке нижеприведённого примера делается верным то, что каждый класс, имя которого заканчивается на "%Object" наследуется из класса MemoryPool. Этот класс может реализовывать управление собственной памятью с помощью перегрузки операторов new и delete. Классы, наследуемые из MemoryPool обязаны переопределять чисто виртуальный метод release, который является частью реализовываемого управления памятью. Это достигается во второй строке для всех классов в точке разделения (pointcut).

Пример: представление (introduction) базового класса

```
advice "%Object" : slice class : public MemoryPool  {
  virtual void release () = 0;
}
```

2.3.2. Последовательности фрагментов (advice ordering)

Если более чем один фрагмент воздействуют на одну и ту же соединительную точку (join point), может быть необходимым объявление последовательности вызовов фрагментов (advice order execution), если есть зависимость между фрагментами кода (advice codes) ("аспектных зависимостей" (aspect interaction)). Нижестоящий пример показывает, как предшествование фрагмента кода (advice code) может быть определено в AspectC++.

Пример: последовательности фрагментов (advice ordering)

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

Если фрагменты обоих аспектов (aspects) (смотри 2.4) Encrypt и Log должны быть выполнены, когда функция send(…) вызвана, эта последовательность объявлений объявляет, что фрагмент Encrypt имеет большее предшествование (precedence). Более детально последовательности фрагментов (advice ordering) и предшествование (precedence) описываются в секции 8 на странице 30.

2.4. Аспекты (aspects)

Аспект (aspect) – это элемент языка AspectC++, созданный для сбора представлений (introductions) и фрагментированного кода (advice code), наследующего общие перекрёстные отношения в модульном пути. Это кладёт аспекты (aspects) в категорию управления информацией об общем состоянии. Они формулируются с помощью возможностей объявлений аспекта (aspect) как расширение понятия классов C++. Базовая структура объявления аспекта (aspect) в точности совпадает со структурой объявления класса C++, исключая лишь то, что ключевое слово aspect ставится вместо class, struct или union. В соответствии с этим, аспекты (aspect) могут иметь атрибуты и методы и также могут наследоваться из классов или других аспектов.

Пример: объявление аспекта (aspect)

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

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

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

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

В примере количество экземпляров для набора классов определено. Следовательно атрибут, внесённый в класс, описываемый разделительной точкой (pointcut) инкрементирует глобальный счётчик во время создания. С помощью использования фрагментированного кода (advice code) для функции main финальное количество созданий объектов отображается на экране, когда программа завершается.

Этот пример может быть переписан, как абстрактный аспект (aspect), который может например быть запакован в аспектной библиотеке (aspect library) с целью повторного использования. Он нуждается в переопределении разделительной точки (pointcut), чтобы быть чисто виртуальным.

Пример: абстрактный аспект (aspect)

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

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

Сейчас возможно наследование из Counter для повторного использования его функциональности с помощью переопределения counted, чтобы он ссылался на реальное выражение из разделительных точек (pointcut expression).

Пример: повторно использованный абстрактный аспект (aspect)

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

```
};
```

2.4.1. Реализация аспектов (aspects)

При настройках по умолчанию аспекты (aspects) в AspectC++ автоматически реализуются как глобальные объекты. Идея, стоящая за этим состоит в том, что аспекты (aspects) могут также обеспечивать глобальные свойства программы и поэтому обязаны быть всегда общедоступными. Несмотря на это в некоторых специальных случаях может понадобиться изменить это поведение, например в контексте операционных систем, когда аспект (aspect) должен быть реализован процессом или потоком.

Схема реализации по умолчанию может быть изменена с помощью определения статического метода aspectof соответственно aspectOf, который иначе сгенерирован для аспекта (aspect). Этот метод предназначен, чтобы всегда мочь возвратить экземпляр соответствующего аспекта (aspect).

Пример: реализация аспекта (aspect) использующая aspectof

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

  advice counted ()  : ThreadCounter m_instance;

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

Введение m_instance во Thread гарантирует, что каждый объект потока имеет экземпляр аспекта (aspect). С помощью вызова aspectof возможно получить этот экземпляр в любой соединительной точке (join point), которая имеет доступ к фрагментированному коду (advice code) и членам аспекта (aspect). Для этой цели код в aspectof имеет полный доступ к фактической соединительной точке (join point) в пути, описанном в следующем разделе.

2.5. Поддержка во время выполнения

2.5.1. Поддержка для фрагментированного кода (advice code)

Для многих аспектов (aspects) доступ к контекстным переменным может быть недостаточным для получения достаточной информации о соединительной точке (join point), где фрагментированный код (advice code) был активирован. Например аспект (aspect), контролирующий потоки для регистрации вызовов функций в программе нуждается в информации про аргументы функций и их типы во время выполнения, чтобы быть способным производить совместимый по типам вывод.

В AspectC++ эта информация доставляется членами класса JoinPoint (смотри таблицу ниже).

типы:	
Result	тип возвращаемого значения
That	тип объекта
Target	тип указателя
AC::Type	закодированный тип объекта
AC::JPType	тип соединительных точек (join points)

статические методы:	
int args ( )	количество аргументов

AC::Type type ( )	тип функции или атрибута
AC::Type argtype (int)	типы аргументов
const char *signature ( )	сигнатура функции или атрибута
unsigned id ( )	идентификатор соединительной точки (join point)
AC::Type resulttype ( )	тип возвращаемого значения
AC::JPType jptype ( )	тип соединительной точки (join point)

не статические методы	
void *arg (int)	актуальный аргумент
Result *result ( )	возвращаемое значение
That *that ( )	объект, ссылающийся на this
Target *target ( )	целевой объект вызова
void proceed ( )	вызов кода соединительной точки (join point)
AC::Action &action ( )	структура Action

Таблица 1: Программный интерфейс класса JoinPoin доступный в фрагментированном коде (advice code)

Типы и статические методы программного интерфейса класса JoinPoint дают возможность получить информацию, которая одинакова для каждой активации фрагментированного кода (advice code). Не статические методы дают возможность получить информацию, которая может быть разной в разных активациях. Доступ к этим методам можно получить с помощью объекта tjp ,соответствующего thisJoinPoint и имеющего тип JoinPoint. Они также всегда доступны внутри фрагментированного кода (advice code).

Нижестоящий пример демонстрирует, как реализовать многократно используемый контролируемый полёт аспекта (aspect), использующего программный интерфейс класса JoinPoint.

Пример: повторно используемый отслеживающий аспект (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;
  }
};
```

Этот аспект (aspect) выполняет отслеживание кода в каждой функции, заданной с помощью виртуальной разделительной точки (pointcut), переопределённой в производном аспекте (aspect). Вспомогательная функция printvalue отвечает за форматированный вывод аргументов, передаваемых в вызове функции. После вызова printvalue для каждого аргумента программный код актуальной соединительной точки (join point) выполняется с помощью вызова proceed на объекте JoinPoint. Функциональность proceed достигается с помощью использования так называемых действий (actions).

2.5.2. Действия (actions)

В AspectC++ действие (action) – это последовательность операторов, которая будет обеспечивать достижение соединительной точки (join point) во время выполнения программы, если фрагментированный код (advice code) не будет активирован. Таким образом tjp->proceed () провоцирует выполнение программного кода соединительной точки (join point). Это может быть вызов или исполнение функции. Понятие действия (actions) реализовано в структуре AC::Action. Фактически proceed эквивалентно action ().trigger (), так что tjp->proceed () может быть заменено с помощью tjp->action ().trigger (). Таким образом метод action () программного интерфейса класса JoinPoint возвращает актуальный объект действия (action) для соединительной точки (join point).

3. Сопоставимые выражения (match expressions)

Сопоставимые выражения (match expressions) используется для описания набора статически известных программных сущностей в программе, написанной на AspectC++. Они могут сопоставлять выражения функций или типов. Класс виден, как специальный вид типа в этом контексте.

Для сопоставляемой функции сопоставимое выражение (match expression) внутренне разложено на составные части внутри шаблона типа функции, шаблона области видимости и шаблона имени.

Пример: тип, область видимости и имя частей сопоставимых выражений функции

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

Это сопоставимое выражение (match expression) описывает нижестоящие требования к сравнимому имени функции:

имя: имя функции должно сопоставляться имени шаблона parse_%

область видимости: область видимости в функции, которая определяется должно сопоставляться Puma::…::

тип: тип функции должен сопоставляться const %(Token *)

Для классов и других типов эта декомпозиция не требуется. Например имя типа “Puma::CCParser” является достаточным для описания класса, так как это то же самое, что и имя класса.

Если сущность сопоставляется всем частям сопоставимого выражения (match expression), она становится элементом набора, который должен быть определён с помощью сопоставимого выражения (match expression).

Грамматика, использующаяся для сопоставимых выражений (match expressions) показана в приложении B на странице 33. Нижестоящие секции отдельно описывают механизм сопоставления имени, области видимости и типа. Выражение, чьё сопоставление имени и области видимости используется для сопоставления имёни функции, так же может использоваться, как сопоставление именованных типов ,например, классов.

3.1. Сопоставление имени

3.1.1. Простое сопоставление имени

Сопоставление имени также тривиально, как сравнение имени со стандартным идентификатором C++. Если шаблон имени (name pattern) не содержит специального

группового символа (wildcard character) %, то он сопоставляется имени, только если оно является в точности таким же. Иначе каждый групповой символ (wildcard character) сопоставляется произвольной последовательности символом в сравниваемом имени. Групповой символ (wildcard character) также может сопоставляться пустой последовательности.

Пример: простые шаблоны имён

Token	сопоставляется только Token
%	сопоставляется любому имени
parse_%	сопоставляется только имени, начинающемуся на parce_, например parce_declarator или parce_
parce_%_id%	сопоставляется именам, как parce_type_id или parce_type_identifier
%_token	сопоставляется любому имени, заканчивающемосю на _token, например start_token, end_token или _token

3.1.2. Операторные функции и преобразования сопоставления имени функции

Механизм сопоставления имён является более запутанным, если шаблон сравнивается с именем функции преобразования типа или с операторной функцией. Имена обоих этих типов функций сопоставляются с помощью шаблона имени %. С отличными от % шаблонами имени они сопоставляются, только если шаблон начинается с "operator ". Шаблон "operator %" сопоставляется любому имени операторной функции или функции приведения типов.

C++ определяет фиксированный набор операторов, которые могут быть перегружены. В шаблоне имени одинаковые операторы могут быть использованы после префикса "operator " чтобы сопоставляться особому типу операторной функции. Имена операторов в шаблонах имён не могут содержать группового символа (wildcard character). Для устранения неоднозначности операторы % и %= сопоставляются с помощью % % и % %= в шаблоне имени.

Пример: шаблоны имён операторов

operator %	сопоставляется любому имени операторной функции (также, как и люьое имя функции приведения типов)
operator +=	сопоставляется только имени оператора +=
operator %	сопоставляется имени оператора operator %

Функции приведения типов не имеют реальных имён. Например функция приведения типов operator int* (), определённая в классе C определяет приведение типа класса C к типу объекта int*. Для сопоставления функциям приведения типов, шаблон имени должен содержать шаблон типа после префикса "operator ". Механизм сопоставления типов разъясняется в секции 3.3.

Пример: шаблоны имён функций приведения типов

operator %	сопоставляется любому имени функции приведения типов
operator int*	сопоставляется любому имени функции приведения типов, которая конвертирует что-то в тип объекта int*
operator %*	сопоставляется имени функции приведения типов, если эта функция конвертирует что-то в тип указателя

3.1.3. Конструкторы и деструкторы

Шаблоны имён не могут использоваться для сопоставления имён конструкторов или деструкторов.

3.1.4. Ограничение области видимости

В сопоставлении выражений перед шаблоном имени может (но не обязательно) стоять шаблон области видимости. Шаблон области видимости (смотри секцию 3.2) используется для описания ограничений в определении области видимости при сопоставлении сущностей. Если никакой шаблон области видимости не добавляется, сравниваемая функция или тип обязан быть определён в глобальной области видимости, чтобы быть сопоставимым.

3.2. Сопоставление области видимости

Ограничения на определение области видимости могут быть описаны с помощью шаблоны области видимости. Это последовательность шаблонов имён (или специальный шаблон любой последовательности области видимости…), которые разделяются с помощью ::, например Puma::…::. Шаблон области видимости всегда заканчивается на :: и никогда не должен начинаться с ::, так как шаблоны области видимости везде интерпретируются относительно глобальной области видимости. Определение области видимости может быть как пространством имён, так и классом.

Шаблон области видимости сопоставляется определению области видимости сравнимой функции или типу, если каждая его часть может успешно сопоставиться соответствующей части в уточненном имени определённой области видимости. Сравниваемое уточнённое имя обязано быть связана с глобальной областью видимости и не должна начинаться со знака ::, который является дополнительным вложенным спецификатором имени в C++. Специальный шаблон … сопоставляется любой (даже пустой) последовательности имён области видимости.

Пример: шаблоны областей видимости

…::	сопоставимо любой, даже глобальной области видимости
Puma::CCParser::	сопоставимо только области видимости Puma::CCParcer
…::%Compiler%::	сопоставимо любому пространству имён или классу в любой области видимости, чьё имя соответствует шаблону %Compiler%
Puma::…::	сопоставимо любой области видимости, определённой внутри пространства имён или класса Puma

3.3. Сопоставление типов

3.3.1. Механизм сопоставления

Типы C++ могут быть представлены, как дерево. Например тип функции int (double) – узел с двумя дочерними записями: одна из них – узел с типом int, а другая – узел с типом double. Обе дочерние записи – листья дерева.

Типы, использующиеся в сопоставлении выражений (match expressions) могут также быть интерпретированы, как деревья. Как добавление к нормальным C++ типам, они могут содержать групповой символ (wildcard character) %, шаблоны имени и шаблоны области видимости. Отдельно стоящий групповой символ (wildcard character) в шаблоне типа становится специальным узлом любого типа в изображении дерева.

Для сравнения типа шаблона со специфическим типом используется изображение дерева, и узел любого типа сопоставляется любому типу дерева или поддерева.

Пример: шаблоны типов с групповым символом (wildcard character)

%	сопоставимо любому типу
void (*) (%)	сопоставимо любому типу указателю на функцию, которая принимает один аргумент и возвращает значение типа void
%*	сопоставимо любому типу указателя

3.3.2. Сопоставление именованных типов

Шаблоны типов могут состоять из шаблонов имён и шаблонов области видимости. Они становятся узлами именованного типа в изображении дерева и сопоставляются любому типу объединения, структуры, класса или перечисления (enum), если его имя и область видимости сопоставима данному шаблону (смотри секции 3.1 и 3.2).

3.3.3. Сопоставление типов-указателей на член

Шаблоны для указателей на член тоже содержат шаблон области видимости, например % (Puma::CSyntax::*)( ). В этом контексте шаблон области видимости является обязательным. Шаблон используется для сопоставления класса, ассоциируемого с указателем на тип члена.

3.3.4. Сопоставление уточнённых типов

Многие C++ типы могут быть уточнены с помощью const или volatile. В шаблоне типа эти квалификаторы также могут использоваться, но тогда они будут лишь ограничениями. Если ни квалификатор const, ни квалификатор volatile не содержится в шаблоне типа, то этот шаблон также сопоставим и уточнённым типам. Сопоставление только типам, не являющимся const или volatile может быть достигнуто с помощью операторов разъяснения в секции 4.7 на странице 26. Например !"const %" описывает все не константные типы.

Пример: шаблоны типов с const и volatile

%	сопоставим любому типу, даже типами, уточнённым const или volatile
const %	сопоставим только типам, уточнённым с помощью const
% (*)( ) const volatile	сопоставим всем типам указателей на функции, уточнённым с помощью const и volatile

3.3.5. Оперирование типами функций приведения типов

Возвращаемое значение функций приведения типов интерпретируется, как специальный неопределённый тип в шаблонах типов также хорошо, как и в сравниваемых типах. Неопределённый тип сопоставим только с помощью узла любого типа и узла неопределённого типа.

3.3.6. Эллипсисы в шаблонах типов функций

Шаблон … в списке типов аргументов функции может использоваться для сопоставления произвольному (даже пустому) списку типов. Шаблон … не должен следовать после шаблонов типов аргументов в списке типов аргументов.

3.3.7. Сопоставление виртуальных функций

Последовательность определения спецификаторов типа функции для сопоставления выражения (match expression) может содержать ключевое слово virtual. В этом случае тип функции для сопоставления выражения (match expression) сопоставляется только

виртуальным и чисто виртуальным функциям членам. Как const и volatile, ключевое слово virtual задаёт ограничение, а это значит, что тип функции для сопоставления выражений (match expressions) без virtual сопоставляется как виртуальным, так и не виртуальным функциям.

Пример: шаблоны типов с virtual

virtual % ….::% (…)	сопоставимо всем виртуальным и чисто виртуальным функциям в любой области видимости
% C::% (…)	сопоставимо всем функциям членам C, даже если они виртуальны

3.3.8. Вычисление типов аргументов

Типы аргументов в шаблонах типов вычисляются соответственно обычным правилом C++, то есть массивы и типы функций конвертируются в указатели, чтобы данный тип и квалификаторы const/volatile были уничтожены. К тому же списки типов аргументов, состоящие из единственного типа void конвертируются в пустые списки типов аргументов.

4. Предопределённые функции разделительных точек (pointcut functions)

На следующих страницах представляется полный список всех функций разделительных точек, поддерживаемых в AspectC++. Для каждой функции разделительной точки (pointcut function) указывается какой тип разделительной точки (pointcut) ожидается в качестве аргумента, и какого типа будет результирующая разделительная точка (pointcut). В связи с этим "N" ставится в качестве именованной разделительной точки (name pointcut), а "C" в качестве кодовой разделительной точки (code pointcut). Факультативно дающийся индекс является гарантированным типом соединительной (соединительных) точки (точек), описанной (описанных) в результирующей разделительной точке (pointcut). C, Cc, Ce, Cs, Cg: кодовые (только Вызов, только Исполнение, только Установка, только Получение); N, Nn, Nc, Nf, Nt: именованные (только Пространство имён, только Класс, только Функция, только Тип).

4.1. Типы

base (pointcut) N→Nc,f

Возвращает все базовые классы классов в разделительной точке (pointcut).

derived (pointcut) N→Nc,f

Возвращает все классы в разделительной точке (pointcut) и все классы, производные от них.

Пример: сопоставление типов

Программное обеспечение может содержать нижестоящую иерархию классов

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

Вместе с нижестоящим аспектом (aspect) специальные особенности добавляются в назначенный набор классов этой иерархии классов.

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

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

Разделительная точка (pointcut), описывающая классы Point и Rectangle и все классы производные от Point, которые являются прямыми или косвенными базовыми классами Rectangle. С первым фрагментом (advice) Point получает новый базовый класс. Второй фрагмент (advice) добавляет соответственный метод во все классы в разделительной точке (pointcut).

4.2. Управление потоком (control flow)

cflow (pointcut) C→C

Собирает соединительные точки (join points) залегающие в динамическом контексте выполнения соединительных точек (join points) в разделительной точке (pointcut). В настоящее время языковые особенности используются в ограниченных разделительных точках (pointcuts), в качестве аргументов. Аргументу не позволяется содержать какие-либо привязки к контекстным переменным (смотри 4.6) или другие функции разделительных точек (pointcut functions), как cflow (pointcut), которые обязаны быть определены во время выполнения.

Пример: управление потоком в зависимости от активации фрагмента (advice)

Нижестоящий пример демонстрирует использование функции разделительной точки cflow (pointcut).

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

Рассмотрим класс Bus, показанный выше. Он может быть частью ядра операционной системы и использоваться там, чтобы управлять доступом внешних устройств через специальную шину ввода-вывода. Выполнение функций членов in () и out () не должно прерываться, потому что это прекратит синхронизацию связи шины. Поэтому мы решили реализовать аспект (aspect), управляющий синхронизацией, который отменяет прерывания, пока методы in () и out () выполняются:

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

В качестве драйвера шины код может также быть вызван из дескриптора прерывания, прерывания не должны отменяться в любом случае. Таким образом выражение из разделительных точек (pointcut expression) использует функцию разделительной точки (pointcut function) cflow (), чтобы добавить условие времени выполнения для активации фрагмента (advice). Тело фрагмента должно исполнятся только если управление потоком (control flow) не пришло из дескриптора прерывания os::int_handler (), потому что оно не прерывно по определению и os::enable_ints () в теле фрагмента (advice) будет крутиться в прерывании слишком преждевременно.

4.3. Область видимости

within (pointcut) N→C

Фильтрует все соединительные точки (join points), которые в функциях или классах в разделительной точке (pointcut)

Пример: управление потоком в зависимости от активации фрагмента (advice)

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

  advice calls () : around ()  {
    cout << "transmitting …" << flush;
    tjp->proceed ();
    cout << "finished." << endl;
  }
};
```

Этот аспект (aspect) вставляет код, регистрации всех вызовов transmit, которые содержат методы класса Transmitter.

4.4. Функции

call (pointcut) N→Cc

Предоставляет все соединительные точки (join points), где именованные сущности в разделительной точке (pointcut) вызываются. Разделительная точка (pointcut) может содержать имена функций или имена классов. В случае имени класса все вызовы методов этого класса предоставляются.

execution (pointcut) N→Ce

Предоставляет все соединительные точки (join points), ссылающиеся на реализацию именованных сущностей в разделительной точке (pointcut). Разделительная точка (pointcut) может содержать имена функций или имена классов. В случае имён классов все реализации методов класса предоставляются.

Пример: сопоставление функций

Нижестоящий аспект (aspect) переплетает отладочный код в программе, которая проверяет не вызван ли метод на нулевом указателе и не является ли аргумент вызова нулевым.

```
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");
  }
};
```

Первый фрагмент (advice) предоставляет код для проверки аргумента функции dealloc перед выполнением функции. Проверка вызван ли dealloc на нулевом объекте предоставляется вторым фрагментом (advice). Это реализовано с помощью проверки целевого объекта вызова.

4.5. Создание и уничтожение объектов

construction (pointcut) N→Ccons

Все соединительные точки (join points) где экземпляры данных классов конструируются. Конструирование соединительной точки (join point) начинается после всех базовых классов и членов составляющих соединительных точек (join points). Это может быть представлено, как выполнение конструктора. Однако фрагмент (advice) для конструированной соединительной точки (join point) работает, даже если никакой конструктор не определён явно. Создание соединительной точки (join point) имеет аргументы и типы аргументов, которые могут быть доступны или фильтрованы, например с помощью использования функции разделительной точки (pointcut function) args.

destruction (pointcut) N→Cdes

Все соединительные точки (join points) где экземпляры данных классов уничтожаются. Уничтожение соединительной точки (join point) заканчивается перед уничтожением соединительных точек (join points) членов и базовых классов. Это может быть представлено как выполнение деструктора, если даже деструктор не определён явно. Уничтожение соединительной точки имеет пустой список аргументов.

Пример: подсчёт экземпляров

Нижестоящий аспект (aspect) подсчитывает, как много экземпляров класса ClassOfInterest создано и уничтожено.

```
aspect InstanceCounting  {
  // класс для которого экземпляры должны быть подсчитаны
  pointcut observed () = "ClassOfInterest";
  // конструкторы и деструкторы подсчёта
  advice construction (observed ()) : before ()  { _created++;  }
  advice destruction (observed ()) : after ()  { _destroyed++;  }
public:
  // аспекты-одиночки (singleton aspects) могут иметь конструктор по умолчанию
  InstanceCounting ()  { _created = _destroyed = 0; }
private:
  // счётчики
  int _created;
  int _destroyed;
};
```

Реализация этого аспекта (aspect) довольно простая. Два счётчика инициализируются конструктором аспекта (aspect) и инкрементируются с помощью фрагмента (advice) конструктора/деструктора. С помощью определения observed () как чисто виртуальной разделительной точки (pointcut) этот аспект (aspect) может быть с лёгкостью трансформирован абстрактный аспект (aspect) многократного использования.

4.6. Контекст

that (type pattern) N→C

Возвращает все соединительные точки (join points), где текущий C++ указатель this ссылается на объект, который является экземпляром типа, совместимого с типом, описываемым шаблоном типа.

target (type pattern) N→C

Возвращает все соединительные точки (join points), где целевой объект вызова является экземпляром типа, совместимого с типом, описываемым шаблоном типа.

result (type pattern) N→C

Возвращает все соединительные точки (join points), где возвращаемый объект вызова/выполнения является экземпляром типа, описываемого шаблоном типа.

args (type pattern, …) N→C

Список аргументов args содержит шаблоны типов, которые используются для фильтрации всех соединительных точек (join points), например вызовов функций или их исполнений с сопоставимой сигнатурой.

Взамен шаблону типа также возможна передача имени переменной, для которой контекстная информация является ограничением (контекстную переменную). В этом случае тип переменной используется для сопоставления типов. Контекстные переменные обязаны быть определены в списке аргументов before (), after () или around () и могут быть использованы как аргументы функции в теле фрагмента (advice).

Функции разделительных точек (pointcut functions) that () и target () особые, потому что они могут основываться на проверке типов во время выполнения. Функции args () и result () вычисляются во время компиляции.

4.7. Алгебраические операторы

pointcut && pointcut (N,N)→N, (C,C)→C

Пересечение соединительных точек (join points) в разделительных точках (pointcuts).

pointcut || pointcut (N,N)→N, (C,C)→C

Объединение соединительных точек (join points) в разделительных точках (pointcuts).

!pointcut N→N, C→C

Исключение соединительных точек (join points) в разделительной точке (pointcut).

5. Контекстные фрагменты (slices)

Эта секция определяет синтаксис и семантику объявлений контекстных фрагментов (slices). Следующая секция опишет то, как контекстные фрагменты (slices) могут быть использованы фрагментом (advice) в последовательности для внедрения кода. В настоящее время только классы, являющиеся контекстными фрагментами, (slice classes) определены в AspectC++.

5.1. Объявление классов, являющихся контекстными фрагментами (slice classes)

Классы, являющиеся контекстными фрагментами (slice classes) могут быть объявлены в контексте любого класса или пространства имён. Они могут быть определены только единожды, но там могут быть произвольные число готовых определений. Уточнённое имя может быть использовано, если класс, являющийся контекстным фрагментом, (slice class) уже объявлен в точном контексте, переобъявлен или определён так, как показано в нижестоящем примере:

```
slice class ASlice;
namespace N  {
  slice class Aslice; // Другой контекстный фрагмент (slice)
}
slice class ASlice  { // определение ::ASslice
  int elem;
};
slice class N::ASlice  { // определение N::ASlice
  long elem;
};
```

Если класс, являющийся контекстным фрагментом, (slice class) определяет только базовый класс, может быть использован сокращённый синтаксис:

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

Классы, являющие контекстными фрагментами, (slice classes) могут быть безымянными. Однако это воспринимается только, как часть объявления фрагмента (advice). Классы, являющиеся контекстными фрагментами, (slice classes) могут быть объявлены вместе с ключевыми словами aspect или struct вместо слова class. Пока там не существует различий между классами, являющимися контекстными фрагментами, (slice classes) и аспектами, являющимися контекстными фрагментами, (aspect slice), стандартные правила доступа к элементам структуры, являющейся контекстным фрагментом, (struct slice) в целевых классах являются публичными (public) вместо приватных (private). Запрещено определять аспекты (aspects), разделительные точки (pointcuts), фрагменты (advices) или контекстные фрагменты (slices) в качестве членов класса, являющегося контекстным фрагментом (slice class).

Классы, являющиеся контекстными фрагментами, (slice classes) могут иметь члены, которые не определены в их теле, например статические атрибуты или не подставляемые (inline) функции:

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

Эти внешние объявления членов обязаны фигурировать после соответствующего объявления контекстного фрагмента (slice) в исходном коде.

6. Фрагменты (advices)

Эта секция описывает различные типы фрагментов (advices) предложенные AspectC++. Фрагменты (advices) распределяются на фрагменты (advices) для соединительных точек (join points) в динамическом контроле потока (control flow) выполняющейся программы, как-то: вызовы или выполнение функций, и на фрагменты для статических соединительных точек (join points), например внесение в классы.

В каждом случае компилятор гарантирует, что код заголовочного файла аспекта (aspect), который содержит определение фрагмента (advice definition) (если это случай), компилируется прежде в ячейку (location) затронутой соединительной точки (join point).

6.1. Фрагменты (advices) для динамических соединительных точек (join points)

before (…)

Код фрагмента выполнится перед соединительными точками (join points) в разделительной точке (pointcut)

after (…)

Код фрагмента выполнится после соединительных точек (join points) в разделительной точке (pointcut)

around (…)

Код фрагмента выполнится и после, и перед соединительными точками (join points) в разделительной точке (pointcut)

6.2. Фрагменты (advices) для статических соединительных точек (join points)

Статические соединительные точки в AspectC++ являются классами или аспектами (aspects). Фрагменты (advice) для классов или аспектов (aspects) могут внедрять новые члены или добавлять базовый класс. Новый член или базовый класс становится закрытым (private), открытым (public) или защищённым (protected) в целевом классе, зависимо от защиты в объявлении фрагмента (advice) в аспекте (aspect).

baseclass (classname)

Новый базовый класс внедряется в классы в разделительной точке (pointcut).

introduction declaration

Новый атрибут, функция-член или тип внедряется.

Объявления внедрений являются только семантически анализируемыми в контексте цели. Поэтому объявление может ссылаться, например, на типы или константы, которые не известны в определении аспекта (aspect), но только в целевом классе или классах. Для внедрения конструктора или деструктора имя аспекта (aspect), к которому внедрение принадлежит, обязано быть взято как имя конструктора/деструктора.

Не подставляемые (non-inline) внедрения могут быть использованы для внедрения статических атрибутов или внедрения функции члена с разделяемым объявлением и определением. Имя подставляемого члена обязано быть уточнённым именем, в котором вложенный спецификатор имени является именем аспекта (aspect), к которому внедрение принадлежит.

7. ИПП (Интерфейс прикладного программирования) соединительных точек (join point API)

Следующие секции предоставляют полное описание ИПП соединительных точек (join point API).

7.1. Типы

Result

Результирующий тип функции.

That

Тип объекта (объекта-инициатора вызова)

Target

Тип целевого объекта (целевой объект вызова)

7.2. Функции

static AC::Type type ()

Возвращает закодированный тип для соединительной точки (join point) в соответствии со спецификацией C++ ABI V3 (смотри www.codesourcery.com/cxx-abi/abi.html#mangling).

static int args ()

Возвращает количество аргументов функции для вызова или исполнения соединительной точки (join point).

static AC::Type argtype (int number)

Возвращает закодированный тип аргумента в соответствии со спецификацией C++ ABI V3.

static const char *signature ()

Дает текстовое описание соединительной точки (join point) (имя функции, имя класса,...).

static unsigned int id ()

Возвращает уникальный номер, идентифицирующий эту соединительную точку (join point).

static AC::Type resulttype ()

Возвращает закодированный тип результирующего значения в соответствии со спецификацией C++ ABI V3.

static AC::JPType jptype ()

Возвращает уникальный идентификатор, описывающий тип соединительной точки (join point).

void *arg (int number)

Возвращает указатель на позицию в памяти, хранящую значение аргумента с индексом number.

Result *result ()

Возвращает указатель на позицию в памяти соответствующую результирующему значению или 0, если функция не имеет результирующего значения.

That *that ()

Возвращает указатель на объект, инициирующий вызов или 0, если это статический метод или глобальная функция.

Target *target ()

Возвращает указатель на объект, который является целевым или 0, если это статический метод или глобальная функция.

void proceed ()

Выполняет исходный код соединительной точки (join point) в окружении фрагмента (advice) с помощью вызова action ().trigger ().

AC::Action &action ()

Возвращает действующий объект времени выполнения, содержащий исполнительную среду, чтобы выполнить исходную функциональность, защищённую с помощью фрагмента (advice) around.

8. Порядок фрагментов (advices)

8.1. Предшествование аспекта (aspect)

AspectC++ обеспечивает очень гибкий механизм определения предшествования аспекта (aspect). Предшествование используется для определения порядка исполнения фрагментов кода (advice code), если больше чем один аспект (aspect) влияет на одну и ту же соединительную точку (join point). Предшествование в AspectC++ является атрибутом соединительной точки (join point). Это означает, что зависимость предшествования между двумя аспектами (aspects) может изменяться в различных частях системы. Компилятор проверяет следующие условия, чтобы определить предшествования аспектов (aspects):

Объявление последовательности: если программист обеспечивает определение последовательности, которая определяет зависимость предшествования между двумя аспектами (aspects) для соединительной точки (join point), компилятор подчинится этому определению или вылетает с ошибкой времени компиляции, если это цикл в графе предшествования. Объявление последовательности имеет следующий синтаксис:
advice pointcut-expr: (high, …low)

Список параметров order обязан содержать в списке два элемента. Каждый элемент является выражением из разделительных точек (pointcut expression), которое описывает набор аспектов (aspects). Каждый аспект (aspect) в точном наборе имеет более высокое предшествование, чем все аспекты (aspects), являющиеся частью набора, который следует позже в списке (справа). Например ’("A1" | | "A2, "A3" | | "A4")’ означает, что A1 имеет предшествование перед A3 и A4 и также, что A2 имеет предшествование перед A3 и A4. Эта директива последовательности не определяет зависимость между A1 и A2 и между A3 и A4. Конечно выражение из разделительных точек (pointcut expression) в списке аргументов может содержать именованные разделительные точки (named pointcuts) и даже чисто виртуальные разделительные точки (pointcuts).

Наследующиеся зависимости: если никакой порядок не задаётся и один аспект (aspect) имеет базовый аспект (aspect), производный аспект (aspect) имеет более высокое предшествование, чем базовый аспект (aspect).

8.2. Предшествование фрагментов (advices)

Предшествование фрагмента (advice) определяется по очень простой схеме:

- Если два объявления фрагментов (advices) принадлежат разным аспектам (aspects) и существует зависимость предшествования между этими аспектами (aspects) (смотри секцию 8.1) такая же зависимость будет принята для этих фрагментов (advices)
- Если два объявления фрагментов (advices) принадлежат одному и тому же аспекту (aspect), то объявленный ранее имеет более высокое предшествование.

8.3. Результаты предшествования фрагментов (advices)

Только предшествование фрагментов (advices) оказывает влияние на генерируемый код. Влияние зависит от типа соединительной точки (join point), которая зависит от двух объявлений фрагментов (advices).

Классические соединительные точки (class join points)

Фрагменты (advices) на классических соединительных точек (class join points) могут расширять список атрибутов или список базовых классов. Если у фрагмента (advice) более высокое предшествование, чем у другого, он будет обработан первым. Например, внедрение нового базового класса фрагментом (advice) с более высоким предшествованием будет возникать в списке базовых классов левее базового класса, который был добавлен фрагментом (advice) с более низким предшествованием. Это значит, что на последовательность выполнения конструкторов добавленных базовых классов можно повлиять например с помощью объявления последовательности.

Последовательность добавленных атрибутов также влияет на последовательность выполнения конструкторов/деструкторов также, как размещение объектов.

Кодовые соединительные точки (code join points)

Фрагменты (advices) на кодовых соединительных точках (code join points) могут быть before, after или around фрагментом (advice). Для before и around фрагментов (advices) означает, что код соответствующего фрагмента (advice) ,будет выполнятся первым. Для

after фрагмента более высокое предшествование означает, что код фрагмента (advice) будет выполнятся позже.

Если код фрагмента around не вызывает tjp->proceed () или trigger () в действующем объекте, то никакой код фрагмента (advice) с более низким предшествованием не будет выполнен. Выполнение фрагмента (advice) с более высоким предшествованием не зависит от around фрагмента (advice) с более низким предшествованием.

Например, рассмотрим аспект (aspect), который определяет фрагменты (advices) в следующей последовательности: BE1, AF1, AF2, AR1, BE2, AR2, AF3 (BE – это before фрагмент (advice), AF – это after фрагмент (advice), AR – это around фрагмент (advice)). Как и написано в секции 8.2. на предыдущей странице, объявление последовательности также определяет предшествование: BE1 имеет самое высокое и AF3 самое низкое. В результате мы получили следующую последовательность выполнений:

1. BE1 (самое высокое предшествование)
2. AR2 (фрагменты (advices) с отступом будут выполнены только если proceed () вызвано!)
    a) BE2 (перед AR2, но зависит от AR1)
    b) AR2 (код с отступом будет выполнен только если proceed () вызвано!)
        i. оригинальный код соединительной точки (join point)
        ii. AF3
3. AF2 (не зависит от AR1 и AR2, т.к. имеет более высокое предшествование)
4. AF1 (выполняется после AF2, так как имеет более высокое предшествование)