AOP++: A Generic Aspect-Oriented Programming Framework in C++[*]

Zhen Yao, Qi-long Zheng, and Guo-liang Chen

National High Performance Computing Center at Hefei,
Department of Computer Science and Technology,
University of Science and Technology of China
Hefei, Anhui 230027, China
yaozhen@ustc.edu, qlzheng@ustc.edu.cn, glchen@ustc.edu.cn

Abstract. This paper presents AOP++, a generic aspect-oriented programming framework in C++. It successfully incorporates AOP with object-oriented programming as well as generic programming naturally in the framework of standard C++. It innovatively makes use of C++ templates to express pointcut expressions and match join points at compile time. It innovatively creates a full-fledged aspect weaver by using template metaprogramming techniques to perform aspect weaving. It is notable that AOP++ itself is written completely in standard C++, and requires no language extensions. With the help of AOP++, C++ programmers can facilitate AOP with only a little effort.

1 Introduction

Aspect-oriented programming (AOP)[1] is a new programming paradigm for solving the code tangling problems of object-oriented programming (OOP) by separating concerns in a modular way. Many crosscutting concerns that cannot be expressed by entities such as *classes* or *functions* in OOP can be abstracted and encapsulated into *aspects*. The so called *aspect code* and *component code* can be decoupled cleanly instead of tangled together. Aspects can have influence upon the component code in many ways. For instance, aspects can change the static structure of the component code by *introductions*, as well as change the dynamic behaviour by *advices*. The influence is injected by the *aspect weaver*. First the weaver identifies points in component code where aspects are to be inserted, which are called *join points*. Several join points can form a *pointcut* expression which is declared by each aspect to specify its scope. Then the weaver weaves the aspect code into every join point of the component code.

In existing AOP systems such as AspectJ[13] for Java and AspectC++[7] for C++, the aspect code is often written in a meta level language which is different from (usually a superset of) the language used by the component code.

[*] The work of this paper is funded by Intel HiEd HPC Research Project under grant 4507146713, and Science Foundation of Anhui, China under grant 050420205.

That means AOP usually requires language extensions as well as special AOP-aware (pre-)compilers, therefore aspects and components cannot be expressed in a uniform manner.

Though both C++ and Java are object-oriented programming languages, C++ is very different from Java in many aspects. C++ has many characteristic language constructs such as template, multiple inheritance and operator overloading. They are not optional features of the language, but indispensable parts that make C++ a harmonious whole. Template mechanisms and subsequent generic programming (GP)[5][9] and template metaprogramming techniques[20][10] enable the application of generative programming[12] concepts in C++ to create active libraries[19] supporting multiple programming paradigms[14] including OOP, GP and even functional programming (FP)[2][3]. Different paradigms in C++ can cooperate with each other harmoniously to solve complex programming problems in a more natural manner.

The C++ language is so complicated that even some commercial C++ compilers fail to support all its features (especially the complex template mechanisms) well. The syntax and semantics of C++ is too complex to add new language extensions, especially significant radical extensions such as aspect-orientation. We should not expect an AOP-aware (pre-)compiler to behave better than professional C++ compilers at dealing with complex generic components. Even so, it would be less likely to persuade developers to learn and accept the language extensions and use a specific AOP-aware compiler instead of their favorite standard C++ compilers.

AOP++ presented in this paper is a generic aspect-oriented programming library/framework for C++ that adopts an approach which is quite different from that used by AspectJ or AspectC++. It is remarkable that the aspect weaver and all the aspect code are completely written in standard C++. No extra language extension is required, so no proprietary AOP-enabled compiler is ever needed. With the help of AOP++, C++ programmers can facilitate AOP with only a little effort.

AOP++ can be considered as an active library that defines a new domain-specific sublanguage for AOP and extends the C++ compiler's ability through its aspect weaver. It makes heavy use of complex template metaprogramming techniques to perform aspect weaving at compile time within the framework of standard C++.

AOP++ is tightly coupled with the C++ language. As a consequence, all language features, especially template mechanisms and generic programming are supported intrinsically by AOP++. That means it can apply AOP to the realm of GP paradigm such as STL containers and generic components from other modern C++ template libraries. In fact, the aspect code itself is written as C++ templates and is generic by nature.

The infrastructure of AOP++ is depicted in Fig. 1. It is remarkable that aspect weaving all takes place in standard C++ compiler by metaprograms. Details will be described in later sections.

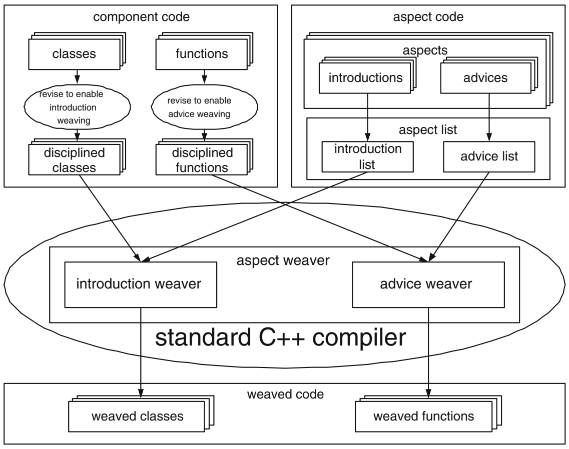

Fig. 1. Infrastructure of the AOP++ Framework

The rest of this paper is organized as follows: Section 2 gives an overview of the AOP++ framework, including its overall infrastructure, disciplines of the component code and definitions of pointcut expressions and introductions / advices / aspects; A typical example is presented in Sect. 3 to demonstrate the power of AOP++; Implementation approaches are explained in Sect. 4; Section 5 gives a brief discussion of related work; Finally, Section 6 summarizes the paper and gives some directions of the possible future work.

2 The AOP++ Framework

2.1 Overview

AOP++ is mainly composed of the following parts:

Pointcut Expression is an important building block of AOP++ for identifying join points in the component code. There are two categories of pointcut expression: *type pattern* represents a collection of types, which is used to specify the scope of introductions; while *method pattern* represents a collection of functions, which is used primarily to specify the scope of advices. Both type pattern and method pattern can be defined recursively. *Type operators* can be applied to existing ones to build up composite type patterns or method patterns.

Base Classes for Aspects are the implementation basis for the user-defined aspect code. They provide basic mechanisms for the aspect code to interact with the component code through a collection of reflection API.

Aspect Weaver is the core of AOP++. It does aspect weaving at compile time. Two weavers are included: the *introduction weaver* weaves introductions into user-defined classes, while the *advice weaver* weaves advices into user-defined functions.

2.2 Disciplines of the Component Code

In order to make AOP++ work, the component code must be written according to some simple disciplines.

2.2.1 Class Definitions

User-defined classes must be defined in a special way to enable introduction weaving, that is, to inherit from template class `aop::introd`. The declaration looks like this:

```
class SimpleClass : public aop::introd<SimpleClass> { /* ... */ };
```

If a user-defined class originally inherits from other class(es), for example:

```
class ComplexClass
    : public A, public B, private C { /* ... */ };
```

The disciplined code looks like this:

```
class ComplexClass : public aop::introd<ComplexClass>
    ::public_bases<A, B>::private_base<C> { /* ... */ };
```

2.2.2 Function Implementations

There are two ways to discipline function implementations to enable advice weaving in AOP++.

The common way is planting a local hook variable of type `aop::advice`, `aop::ctor_advice` or `aop::dtor_advice` in the function body, with function type (and name) provided as template parameters, while function arguments are provided to the constructor of the hook variable. Here are some examples:

```
void MyClass::mf(int i) {
  // do not forget the implicit "this" argument
  aop::advice<void (MyClass::*)(int), &MyClass::mf> hook(this, i);
  // original implementation code
}
void f(double d, const MyClass& c) {
  aop::advice<void (double, const MyClass&), f> hook(d, c);
  // original implementation code
}
MyClass::MyClass(int i) {
  aop::ctor_advice<MyClass(int)> hook(this, i);
  // original implementation code
}
MyClass::~MyClass() {
  aop::dtor_advice<MyClass()> hook(this);
  // original implementation code
}
```

There is another way to enable advice weaving for functions in dynamic linked libraries (or shared libraries). This style of advice weaving is also performed at compile time but is *pluggable* at runtime, we call it *pluggable advice weaving.* For example, a member function `void Dynamic::mf()`'s implementation in file "dynamic.cpp" is compiled into a dynamic linked library. The disciplined code can be put in another file named "dynamic_aop.cpp":

```
void Dynamic::mf()
{ return aop::dynamic_advice<void (Dynamic::*)(), &Dynamic::mf>(this); }
```

2.3 Type Pattern

Type pattern in AOP++ is a mechanism to represent collections of types at compile time.

A type pattern can be simply a type (atomic type pattern), or it can be defined recursively using `or_type`, `and_type`, `not_type`, `derived` or `const_or_not`, which are called type operators. For example, the type pattern that matches any type which is derived from `ClassA` or `ClassB` is

```
aop::or_type<aop::derived<ClassA>, aop::derived<ClassB> >
```

The same type pattern can be represented in AspectJ as `ClassA+ || ClassB+` and in AspectC++ as `derived(ClassA) || derived(ClassB)`.

There is also a special wildcard type pattern `any_type` which obviously matches any type, and a type pattern `null_type` which matches no type. In addition, AOP++ provides several predefined type patterns which are frequently used, such as `integral_type`, `class_type`, `arithmetic_type` and `abstract_type`, etc.

Type patterns in AOP++ have a peculiar capability that is lacking in other AOP frameworks to represent template types. When applying AOP to C++ libraries such as the STL, we need to represent concept like "`std::vector<T>` where `T` is any derived type of `MyClass`", and it is easy to write out the type pattern in AOP++:

```
std::vector<aop::derived<MyClass> >
```

AOP++ provides a mechanism called compile-time lambda expression[10] to presents type patterns which impose specific relationships between template parameters. For example, we can express the type pattern "`std::pair<T, T>` where the two `T`'s are the same." in AOP++ as follows:

```
std::pair<_, _>
```

Note that the predefined type patterns described previously all correspond to lambda type pattern expressions composed of type traits in the Boost Type Traits Library[4]. For instance, `aop::integral_type` and `boost::is_integral_type<_>` are equivalent type patterns.

AOP++ also provides a mechanism to do type pattern matching. Users can determine whether a type is contained in a type pattern expression at compile time by using `aop::matches` template:

```
aop::matches<TypePattern, Type>::value
```

`value` is of type `bool` that evaluates to `true` or `false` at compile time.

The introduction weaver uses a similar way to determine whether a class is in the scope of an introduction.

2.4 Method Pattern

Method pattern can be viewed as a special kind of type pattern that concerns with functions or overloaded operators.

AOP++ uses several helper class templates to wrap up functions to types, so as to manipulate them more easily.

Given a global function `f` and a member function `A::mf`:

```
void f(int i, std::string s);
void A::mf(int i, std::string);
```

The corresponding method patterns can be defined as following:

```
aop::method<void (int, std::string), f>
aop::method<void (A::*)(int, std::string), &A::mf>
```

Because of that there is no type to represent constructors or destructors directly in C++, AOP++ uses an alternative way to define method patterns for them. Given:

```
Point::Point(int x, int y);
Point::~Point();
```

Their corresponding method patterns are:

```
aop::ctor<Point (int, int)>
aop::dtor<Point ()>
```

In AspectJ, the same constructor can be represented as "`Point.new(int, int)`".

Wildcards can be used for method names in method patterns as well. The pattern below matches "any non-static non-const member function defined in class `MyClass` or any of its derived classes":

```
aop::methods<aop::any_type (aop::derived<MyClass>::*)(...)>
```

It is equivalent to "`* MyClass+.*(..)`" in AspectJ or "`% derived(MyClass)::%(...)`" in AspectC++.

It is possible to specify a collection of member functions with exactly the same name. The macro below defines a method pattern named `draw_methods` that matches "any non-static non-const member function named '`draw`' in class `Component` or any of its derived classes, which takes a reference to `Graphics` as its parameter and has no return value":

```
AOP_DEFINE_METHODS(void (derived<Component>::*)(Graphics&), draw,
                        draw_methods);
```

Similar to type patterns, method patterns can be combined by applying type operators `and_type`, `or_type` and so forth to form composite method patterns.

Type pattern and method pattern in AOP++ are simplified pointcut concepts. There is no distinct `call`, `execution`, `within` or `cflow` and so forth pointcuts in AOP++. AOP++ simply uses method pattern to represent the execution of functions.

2.5 Introduction

Introduction is used for modifying user-defined classes and their hierarchies. It changes the static structure of the component code at compile time. Introduction can introduce new base classes for user-defined classes, or add new members (member variables or member functions) to them.

A user-defined introduction in AOP++ is a class template which is derived from template class `aop::introd_base`. Extra base classes, member variables and member functions to be introduced into user-defined classes can be specified by just declaring base classes, member variables and member functions of the user-defined introduction class itself respectively.

An important part of user-defined introduction is an inner type named `scope`, which specifies the scope within which the introduction takes effects in terms of a type pattern.

For example, the code listed below shows how to declare a user-defined introduction named `MyIntrod` which introduces an extra base class, a member variable and a member function to the user-defined classes `A` and `B`:

```
template <typename Arg>
struct MyIntrod : aop::introd_base<Arg>
                    ::public_base<IntroducedBaseClass> {
  typedef aop::or_type<A, B> scope;

  int               introduced_member_variable;
  void              introduced_member_function();
  static const int introduced_static_member_variable = 0;
  static void      introduced_static_member_function();
};
```

There are also abstract introductions. An abstract introduction just leaves its scope definition empty, or holds pure virtual functions, waiting for derived introductions to complete the definition.

An introduction can be declared to "dominate" some other introductions by defining in its definition a type pattern named `dominates` which includes the dominated introductions as follows:

```
typedef aop::or_type<IntroductionA<Arg>, IntroductionB<Arg> > dominates;
```

That means, it will reside at a higher level than those dominated introductions in the introduction hierarchies generated by the introduction weaver (see Sect. 4 for detail).

2.6 Advice

Advice is used for defining additional code that should be executed at runtime. It changes the dynamic behaviour of the component code at runtime. However, advice weaving is done at compile time.

AOP++ currently supports three kinds of advices: *before*, *after* and *around.* Around advice is only available for pluggable advice weaving. All user-defined advices should be class templates that inherit from `aop::advice_base`. Inner type named `scope` is a method pattern specifying the functions to be advised.

The before, after and around methods in advice are member functions, each takes the same parameter list as that of the method to be advised, or a tuple which wraps up all the arguments by reference as its only parameter in situations when the advised methods have different signatures, or even takes no argument and gets these arguments via member functions of its `advice_base`. AOP++ will automatically choose the correct way to pass the actual arguments to the before, after and around methods. These arguments can be read or even modified in the before, after and around methods, to perform extra work or change the behaviour of the advised functions.

Below is a simple example illustrating how to define an advice containing before and after methods for tracing any AOP++-enabled functions.

```
template <typename Arg>
struct TracingAdvice : aop::advice_base<Arg> {
  typedef aop::any_type scope;

  void before()
  { clog << "TracingAdvice::before " << this->method_name() << endl; }

  void after()
  { clog << "TracingAdvice::after  " << this->method_name() << endl; }
};
```

There are also abstract advices. An abstract advice just leaves its scope definition empty, or holds pure virtual `before` / `after` / `around` functions, waiting for derived advices to complete the definition.

An advice can be declared to dominate some other advices by defining in its definition a type pattern named `dominates` which includes the dominated advices as follows:

```
typedef aop::or_type<AdviceA<Arg>, AdviceB<Arg> > dominates;
```

That means, it will precede those dominated advices in the advice chain generated by the advice weaver (see Sect. 4 for detail).

2.7 Aspect

Like data and functions can be encapsulated into a class, several introductions and advices can be encapsulated into a single aspect to emphasize their logical relation. Figure 2 shows an aspect that adds synchronization support for generic containers. We take `std::vector` for example here. (Suppose we could revise the standard containers to make them AOP++-enabled.)

```
template <typename Arg>
struct vector_monitor
     : aop::aspect_base<Arg> {
typedef recursive_read_write_mutex Mutex;
typedef recursive_read_write_lock  Lock;

struct monitorable : aop::introd_base<Arg> {
  typedef std::vector<aop::any_type> scope;

  mutable Mutex mutex;
};

struct read_monitor : aop::advice_base<Arg> {
  typedef aop::methods<aop::any_type
   (std::vector<aop::any_type>::*)(...) const>
   scope;

  Lock lock;

  read_monitor()
      : lock(this->this_object->mutex)
  {}
  void before() { lock.read_lock();  }
  void after()  { lock.unlock();     }
};

struct write_monitor : aop::advice_base<Arg> {
  typedef aop::methods<aop::any_type
    (std::vector<aop::any_type>::*)(...)>
    scope;

  Lock lock;

  write_monitor()
      : lock(this->this_object->mutex)
  {}
  void before() { lock.write_lock(); }
  void after()  { lock.unlock();     }
};

typedef aop::type_list<
    monitorable,
    read_monitor,
    write_monitor> aspect_list;
};
```

Fig. 2. The Synchronized Aspect for `vectors`

We can also specify aspect precedence by defining a type pattern named `dominates` which includes the dominated aspects. If an aspect "dominates" another aspect, that means all introductions and advices in it dominate those in the other.

2.8 Reflection

It is important for aspects to have the ability interacting with the corresponding component code. AOP++ provides a rich reflection API through `introd_base` and `advice_base` for the purpose. Aspect programmers can access type information of the component code, get arguments of the method being advised and so on. The APIs can be divided into two categories — compile time reflection and runtime reflection.

2.9 Putting It All Together

Once the component code is correctly disciplined and all the introductions, advices and aspects are defined, we need to tell AOP++ which of them are expected to take effect on the component code:

```
namespace aop {
  typedef template_list<
            Introd1, Introd2, ...,
            Advice1, Advice2, ...,
            Aspect1, Aspect2, ...
          > aspect_list;
}
```

Only introductions / advices / aspects in the aop::aspect_list will be woven into the component code. Users can maintain different aspect_list configurations for different projects, and define a preprocessor macro AOP_ASPECTS to specify the header including the expected aspect_list. Aspect weaving can also be disabled by defining a null aspect_list or a preprocessor macro AOP_DISABLE.

3 Example: Implementing the Observer Pattern

In this section, we demonstrate how to use AOP++ to implement the famous *observer pattern* described in [8]. Given classes FigureElement, Point, Line and Canvas as in Fig. 3.

```
class FigureElement {
public:
  virtual void setXY(int, int) = 0;
  virtual ~FigureElement();
};

class Point : public FigureElement {
  int _x;
  int _y;
public:
  Point(int x, int y);
  void setXY(int x, int y);
  void setX(int x);
  void setY(int y);
  int x();
  int y();
};
```

```
class Line : public FigureElement {
  Point p1;
  Point p2;
public:
  Line(int x1, int y1, int x2, int y2);
  Line(const Point& p1, const Point& p2);
  void setXY(int x, int y);
  void setP1(const Point &p);
  void setP2(const Point &p);
};

class Canvas {
  std::list<FigureElement*> elements;
public:
  void addFigureElement(FigureElement* fe);
  ...
};
```

Fig. 3. Definition of FigureElement and Its Derived Classes

```
template <typename _Observer>
struct Subject {
  typedef _Observer Observer;
  typedef std::list<Observer*> ObserverList;

  void attach(Observer* obs) {
    observers.push_back(obs);
  }

  void detach(Observer* obs) {
    observers.remove(obs);
  }

  void notify() {
    typedef
      typename ObserverList::iterator
      iterator;

    for (iterator it = observers.begin();
```

```
         it != observers.end(); ++it) {
      (*it)->update(static_cast<
        typename Observer::Subject*>(this));
    }
  }

  virtual ~Subject() {}

private:
  ObserverList observers;
};

template <typename _Subject>
struct Observer {
  typedef _Subject Subject;

  virtual void update(Subject* subject) = 0;
  virtual ~Observer() {}
};
```

Fig. 4. Generic Components for the Observer Pattern

```
template <typename Arg>
struct MoveMethods {
  AOP_DEFINE_METHODS(void (aop::derived<FigureElement>::*)(int, int),
                                                setXY, setXY_method);
  AOP_DEFINE_METHODS(void (Point::*)(int),        setX,  setX_method);
  AOP_DEFINE_METHODS(void (Point::*)(int),        setY,  setY_method);
  AOP_DEFINE_METHODS(void (Line::*)(const Point&), setP1, setP1_method);
  AOP_DEFINE_METHODS(void (Line::*)(const Point&), setP2, setP2_method);

  typedef aop::or_type<
      setXY_method, setX_method, setY_method, setP1_method, setP2_method> scope;
};
```

Fig. 5. The Move Events of `FigureElements`

```
template <typename Arg>
struct SubjectObserverProtocol
     : aop::aspect_base<Arg> {

struct SubjectIntrod
     : aop::introd_base<Arg>
          ::public_base<Subject<Canvas> > {
  typedef FigureElement scope;
};

struct StateChangedAdvice
     : aop::advice_base<Arg> {
  typedef
      typename MoveMethods<Arg>::scope
      scope;

  void after()
  { this->this_object->notify(); }
};

struct ObserverIntrod
     : aop::introd_base<Arg>
     ::public_base<Observer<FigureElement> > {
  typedef Canvas scope;

  virtual void update(FigureElement* fe) {
```

```
    // update the canvas according to fe
  }
};

struct SubjectAddedAdvice
     : aop::advice_base<Arg> {
  AOP_DEFINE_METHODS(
      void (Canvas::*)(FigureElement*),
      addFigureElement, scope);

  typedef aop::advice_base<Arg> base;
  typedef typename base::arg_list arg_list;

  void after(arg_list& args) {
    // attach the Canvas to the FigureElement
    aop::arg<1>(args)
        ->attach(this->this_object);
  }
};

typedef aop::type_list<
    SubjectIntrod,
    StateChangedAdvice,
    ObserverIntrod,
    SubjectAddedAdvice> aspect_list;
};
```

Fig. 6. The Aspect for implementing the Observer Pattern

A `Canvas` holds a list of `FigureElement` objects and is responsible for rendering them. Now we want the `Canvas` to be notified to update its display whenever any one of its `FigureElement` is moved. Obviously the `Canvas` acts as the observer, while the `FigureElement` and its derived classes act as the subjects.

The first step is to discipline the above classes to be AOP++-enabled.

The second step is to define generic components supporting the observer pattern, which is shown in Fig. 4. This approach is similar to those used in the Loki Library[6].

Before writing aspects using AOP++, let's define the pointcut expression that denotes the move event of a `FigureElement` in Fig. 5.

Now the aspects can be written as in Fig. 6.

It is also possible to make the concrete subject and observer classes as well as the state-changed pointcut as template parameters so as to generalize the above `SubjectObserverProtocol` aspect to accommodate general situations. The generalized aspect is called a *parameterized aspect*.

Several other design patterns such as the *visitor pattern*[8] can also be applied in this manner. The generic components are similar to those in the Loki Library, while their integration with user-defined components is automatically done by reusable parameterized aspects in an aspect-oriented manner.

4 Implementation

AOP++ makes extensive use of template metaprogramming in its implementation.

Pointcut expressions including type patterns and method patterns are all represented using the C++ type system. Type patterns are defined by simple types and type operators, also with some predefined shortcuts for defining often-used type patterns. Method patterns are method pointers wrapped up in `method`, `methods`, `ctor` or `dtor` templates.

There are two aspect weavers in AOP++, the introduction weaver and the advice weaver. A mechanism similar to the template `aop::matches` in Sect. 2.3 is used to determine whether a join point is covered by the scope of an introduction or advice. The following is a simplified description of the work flow of the aspect weaver.

Introduction weaving takes place while defining the base class(es) for a user-defined type. It first filters all introductions (both standalone and those embedded in aspects) from the `aop::aspect_list`, then determines whether the scope of an introduction covers the current join point (that is, the user-defined class which is being defined). If the answer is yes, the introduction is relevant and will be woven into the definition of the class; otherwise it is simply discarded. Then all relevant introductions will be instantiated with the current join point in their template parameter using a class template similar to `Loki::GenLinearHierarchy` template[6], and form a linearized class hierarchy in which the introductions are lined up and inherit one another (the order will be influenced by the domination declarations of introductions) with the user-defined base classes at the top of the hierarchy while the class being defined at the bottom. Hence extra base classes, member functions and member variables are "injected" into user-defined classes.

Consider a class `C` defined as follows:

```
class C : public aop::introd<C>::public_bases<A, B> { /* ... */ };
```

Suppose three introductions `Introd1`, `Introd2` and `Introd3` are injected into class `C`. The resulting inheritance structure generated by our introduction weaver is shown in Fig. 7.

Advice weaving takes place while defining the extra hook variable (which uses the scope guard idiom) of a user-defined method. The weaver first filters all advices (both standalone and those embedded in aspects) from the

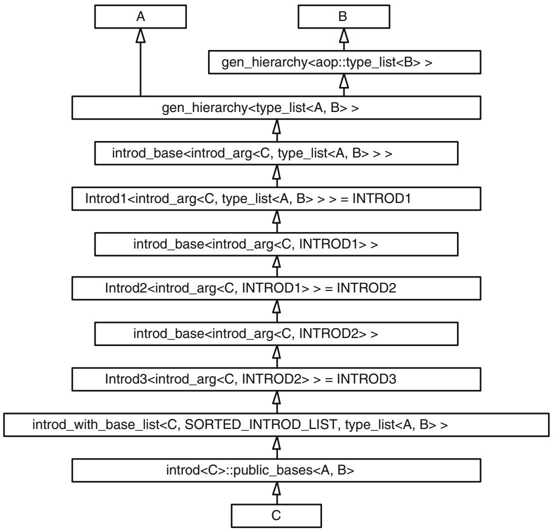
A
B
gen_hierarchy<aop::type_list<B> >
gen_hierarchy<type_list<A, B> >
introd_base<introd_arg<C, type_list<A, B> > >
Introd1<introd_arg<C, type_list<A, B> > > = INTROD1
introd_base<introd_arg<C, INTROD1> >
Introd2<introd_arg<C, INTROD1> > = INTROD2
introd_base<introd_arg<C, INTROD2> >
Introd3<introd_arg<C, INTROD2> > = INTROD3
introd_with_base_list<C, SORTED_INTROD_LIST, type_list<A, B> >
introd<C>::public_bases<A, B>
C

Fig. 7. The Generated Inheritance Structure of Class `C`

`aop::aspect_list` whose scope covers the current join point (that is, the function being defined), and creates a list of all the relevant advices (called an *advice chain*)in the corresponding `aop::(dynamic_)advice`, `aop::(dynamic_)ctor_advice` or `aop::(dynamic_)dtor_advice` classes. The constructor of the advice class will call the `around` and `before` member functions of every relevant advice one by one in order (the order will be influenced by the domination declarations of the advices), the destructor will call corresponding `after` member functions in reverse order.

The arguments of the method are passed to constructor of the hook variable, from which they will then be passed to every `before` / `after` / `around` member functions of the advices automatically in appropriate manner.

5 Related Work

AOP++ invents a brand-new approach to support aspect-oriented programming paradigm in C++. By using template metaprogramming techniques, AOP++ creates a full-fledged aspect-oriented programming framework which does not depend on any language extensions or privileged AOP-aware compilers. All concepts and components in AOP++ are all built on standard C++ constructs. This makes it easy to be accepted by C++ programmers.

AOP++ distinguishes itself from several previous approaches to simulate AOP in C++[16][17][21] by its characteristic aspect weaver.

The previous approaches [16] and [17] exploit techniques similar to *mixin-based programming*[18] to wrap up existing component with mixin layers, while in [21], the component class must be declared as a template with an extra "Aspect" template parameter. In all of them, the aspect user has to declare which aspects are desired for each class by defining an aspects list for every user-defined class *explicitly* and *manually.* the aspect list can be long and the manual definition is error-prone and cumbersome. Furthermore, all code that refers to the user-defined classes has to be changed to use the classes wrapped up with aspects explicitly. The most important crosscutting nature of AOP cannot be expressed.

On the contrary, aspect weaving in AOP++ is done *implicitly* and *automatically* at compile time behind the scenes by the aspect weaver according to the scope definition of each aspect. The component programmer need not concern about what aspects will be injected into which user-defined classes or functions at all.

6 Conclusion and Future Work

The main contributions of AOP++ are:

1. It innovatively makes use of C++ templates to express pointcut expressions and match join points at compile time.
2. It innovatively creates a full-fledged aspect weaver by using C++ template metaprogramming techniques to perform aspect weaving.
3. It successfully extends and applies AOP to the GP paradigm in standard C++. It bridges the gap between AOP and GP. This includes dual meaning: (1) GP techniques can be used in the aspect code and (2) AOP++ makes it possible to apply AOP to generic components in modern C++ template libraries.

Due to limitations of the C++ language itself, there are also some limitations of AOP++, such as that the weaved code is a structural and behavioural approximation to what is expected, but not exactly the same, and it is difficult to implement some advanced features such as join points for field access, privileged aspects, etc.

The main limitation of AOP++ is that it is not 100% transparent to the component programmer. Existing component code has to be revised according to some disciplines in order to enable AOP++ to act on them, though the disciplines are simple and straightforward, and may be applied automatically by a simple pre-processor. We believe that this does not really contradict the philosophy of AOP which demands the separation of the component code and the aspect code. The reason is that the revision is done *just once* in a uniform manner regardless of whatever aspects will be woven later. The disciplines are the keys for bridging the component code and the aspect code. The benefit we achieved is that no language extensions are imposed upon programmers. In addition, all the client code which uses the component code needs not to be changed to enjoy the benefit of AOP++.

Our future work includes:

1. Further enhancement of AOP++, including more dynamic features such as support for `cflow`. It is likely that they will introduce more runtime overhead upon the user program.

2. Investigate the possibility of integrating support for AOP++ in modern IDEs to facilitate the development of AOP programs.
3. Study the development model of the AOP paradigm and the interactions between aspects in AOP++, construct reusable generic AOP libraries for tracing, debugging, performance profiling, program visualization and verification, concurrent programming, etc.
4. Study the relation between AOP and other new programming paradigms such as *explicit programming*[15] and try to combine them in the framework of AOP++.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In Proceedings of ECOOP 1997. Springer-Verlag. (1997)
2. McNamara, B., Smaragdakis, Y.: Functional Programming in C++. In Proceedings of the ACM SIGPLAN ICFP 2000. (2000)
3. The Boost Lambda Library. http://www.boost.org/libs/lambda/
4. The Boost Type Traits Library. http://www.boost.org/libs/type_traits/
5. Austern, M.: Generic Programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc. (1998)
6. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley. (2001)
7. Spinczyk, O., Gal, A., Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In Proceedings of TOOLS 2002. (2002)
8. Gamma, E., Helm. R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. (1994)
9. Stepanov, A., Lee, M.: The Standard Template Library. HP Technical Report HPL-94-34. (1995)
10. The Boost C++ Metaprogramming Library. http://www.boost.org/libs/mpl/
11. The Boost Array Library. http://www.boost.org/libs/array/
12. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley. (2000)
13. AspectJ. Home Page. http://www.aspectj.org/
14. Coplien, J.: Multi-Paradigm Design for C++. Addison-Wesley. (1998)
15. Bryant, A., Catton, A., Volder, K., Murphy, G.: Explicit Programming. In Proceedings of AOSD 2002. (2002)
16. Diggins, C.: Aspect-oriented programming & C++. Dr Dobbs Journal 29 (8) (2004)
17. Gal, A., Lohmann, D., Spinczyk, O.: Aspect-Oriented Programming with C++ and AspectC++. Tutorial held on AOSD 2004. (2004)
18. Smaragdakis, Y., Batory, D.: Mixin-Based Programming in C++. In Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers. Springer-Verlag. (2001)
19. Veldhuizen, T., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98). SIAM Press. (1998)
20. Veldhuizen, T.: Using C++ template metaprograms. C++ Report. 7(4). (1995)
21. Sunder, S., Musser, D.: A Metaprogramming Approach to Aspect Oriented Programming in C++. MPOOL'04 (ECOOP 2004)