An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming

István Zólyomi, Zoltán Porkoláb, and Tamás Kozsik

Department of Computer Science, Eötvös Loránd University
Pázmány Péter sétány 1/D H-1117 Budapest, Hungary
{scamel,gsd,kto}@elte.hu

Abstract. Families of independent classes, where each class represents a separate, orthogonal concern are highly attractive for implementing collaboration-based design. However, required subtype relationship between such families cannot be expressed in many programming languages. This paper presents a framework to handle collaborating groups of classes using template metaprogramming based on standard C++ features in the style of `Loki::Typelist`. Our solution provides tailor-made implicit conversion rules between appropriate groups, inclusion polymorphism and a tool for dynamic binding.

1 Introduction

In this paper an extension to the subtyping mechanism of C++ is presented. Subtyping based on inheritance (subclassing) is known to cause many problems. (See e.g. Bruce in [3] for an extensive discussion of such flexibility and type-safety problems). In spite of these problems, most popular object-oriented languages, such as C++, Java, C# and Eiffel use subtyping provided by inheritance. Our extension to subtyping in C++ will not be an exception, as it will be based on multiple inheritance.

Subtyping is explicit in the aforementioned languages: the subtype relationship must be explicitly indicated in the type definitions. For obvious reasons, the subtype relation is made transitive, by defining it as the reflexive and transitive closure of the declared subclassing properties appearing in the type definitions. Multiple inheritance with a disjunctive subtype relation is highly attractive for implementing collaboration-based designs [12]. Each particular class from a family of collaborating classes represents a separate, orthogonal concern. In the same time, the client code must be separated from the knowledge about the exact structure of the family of classes. This client should be able to refer to a subset of supertypes of the collaborating classes.

The subtype relation in C++ and similar languages does not have language support for disjunctivity with respect to multiple inheritance. To clarify disjunctivity look at the classic example given by Stroustrup in [14]. It describes two orthogonal concerns. One is a hierarchy of vehicles with the base class `Vehicle`

F. Pfenning and Y. Smaragdakis (Eds.): GPCE 2003, LNCS 2830, pp. 209–227, 2003.

and derived classes, e.g. `Car` and `Truck` (see figure 1). Here a class like `Car` is a subtype of `Vehicle`; the functionality of `Vehicle` is a subset of `Car`'s. The other concern represents the aspect of an emergency vehicle which e.g. has priority in intersections and is under radio control of a dispatcher. This concern is implemented in class `Emergency`. Cars like a policecar or trucks like a fireengine should have the functionality provided by class `Emergency`, therefore classes `PoliceCar` and `FireEngine` should be subtypes of class `Emergency` and either `Car` and `Truck` respectively. The functionality of `PoliceCar` and `FireEngine` is the union of the functionalities of the collaborating classes. The subset relation between functionalities should be closed under union (disjunction). Assume we have a client code handling generic emergency vehicles as a collaboration of classes `Vehicle` and `Emergency`. The class `EmergencyVehicle` should be a supertype for classes `PoliceCar` and `FireEngine`, but implementing it using multiple inheritance we lose the language support for the subtype relationship between `EmergencyVehicle` and `PoliceCar` (marked with dashed line on figure 1), e.g. no automatic conversion is possible.

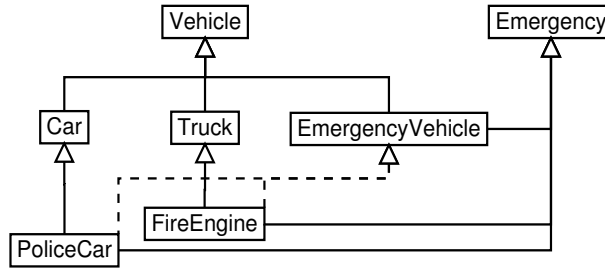

Fig. 1. Class hierarchy for the Vehicle example.

This paper presents a framework that supports a natural way to handle collaborating groups of classes. The framework consists of templates that provide tailor-made implicit conversions between two appropriate groups of collaborating classes in the spirit of our previous example: the class `PoliceCar` defined as `FAMILY_2(Car, Emergency)` is automatically converted to `FAMILY_2(Emergency, Vehicle)`. (Note that the order of the classes in the the groups is irrelevant and we could also apply a hierarchy of emergency levels derived from `Emergency`.) As a part of the increased support for inclusion polymorphism, our framework also provides a tool for dynamic binding of method calls.

2 Applicability

In this section we present a few examples to show our motivation and underline the advantages of our approach. An important advantage of this technique is that we can create our software components with less unnecessary dependencies so we can increase the maintainability of C++ code.

In this paper we will follow through a single example originally introduced by Harold Ossher in [15]. We write an application manipulating expressions consisting of literals and operators. Different manipulations are orthogonal concerns like evaluating, displaying and syntax-checking. We can express `Operator` as a collaboration of classes `OpDisplay`, `OpEval` and `OpCheck` (see figure 2). Moreover we can express `Plus` as the collaboration of `PlusDisplay`, `PlusEval` and `PlusCheck`.

```
typedef FAMILY_3(OpDisplay, OpEval, OpCheck) Operator;
typedef FAMILY_3(PlusDisplay, PlusEval, PlusCheck) Plus;
```

Collaborating classes are themselves in inheritance (and subtype) relation: `PlusDisplay` is subtype of `OpDisplay`, `PlusEval` is subtype of `OpEval`, etc. We would like to have the subtype relation between the collaborating `Plus` and `Operator` classes and this is exactly what our `FAMILY` construct can provide. Remember that if Plus and Operator were created by ordinary multiple inheritance, then `Plus` would not be a subtype of `Operator`. This is the example we will refer through the article.

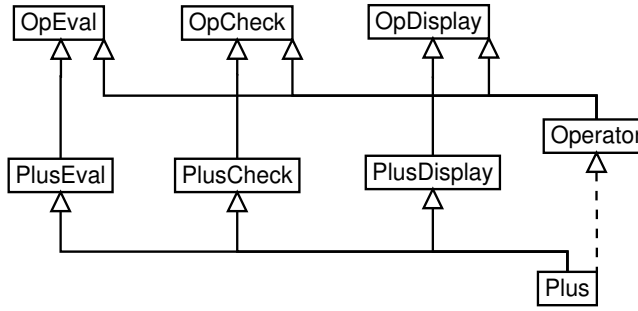

Fig. 2. Class hierarchy for the Operator example.

The next example is related to Grid-computing. Assume that we have Compute Service nodes in a grid offering different computational resources. When a client is started, a Resource Broker will find a Compute Service node where it can be executed. The requirements of the clients can be specified with types (interfaces). A client can run on a node if the required type is a supertype of the offered type. A set of building blocks for constructing the required resources can be `BasicTrigonometry` (supporting sin, cos, tan), `AdvancedTrigonometry` (it is a subtype of `BasicTrigonometry` with sh, ch, arcsin, cotan), `ComplexNumbers`, `ComplexTrigonometry`, (child of `AdvancedTrigonometry` and `ComplexNumbers`), `Derivation`, `Integration` and `DifferentialEquations` (subtype of `Derivation` and `Integration`).

A client can specify its requirements as `FAMILY_2(AdvancedTrigonometry, Derivation)`. The Resource Broker can choose a node offering either `FAMILY_2(`

`AdvancedTrigonometry, Derivation)` or `FAMILY_3(ComplexTrigonometry, Derivation, Integration)` or any similar.

To achieve similar flexibility without our framework one should define all possible combinations of the building blocks declaring the appropriate subtype relationships. This would be very cumbersome to write and impossible to maintain, especially if new computational libraries were to be introduced.

3 Class Hierarchy

A typical group of collaborating classes involves classes usually unrelated to each other. When designing the class hierarchy, we cannot find a common superclass. What is more, introducing such an artificial base class would be misleading both conceptually and technically (it would allow implicit conversions not deducible from the conceptual model). Therefore we did not choose an object-oriented but a generative method to express the required relationships.

The fundamental element in the implementation is structural conversion between groups of collaborating classes. Implementing structural conversion we have to provide a structure holding our collaborating classes together. This may not seem a problem at all because C++ has multiple inheritance, so any combination of classes can be created. However, for implementing the appropriate conversions, we also need the exact map of the inheritance hierarchy of the participating classes. Thus we have to implement a framework that provides all required inheritance information.

A natural way to implement such a framework is to assemble the required structure of classes in a controlled way. Thinking in generative programming terms, we can use template parameters to specify the components of the structure to be built. We want a template class that inherits from all parameter types. Using an ordinary `Mixin` class a naive implementation can be given:

```
template <class Left, class Middle, class Right>
class Family3 : public Left, public Middle, public Right {};
```

This example has an important limitation: the number of components is "hardwired". Extending this implementation philosophy our template should repeatedly be rewritten to support all possible numbers of collaborating classes.

An arbitrary long list of types can be specified using a single template parameter. Introduced by Alexandrescu[1] in [1] a template framework was designed to handle this kind of type lists. We could have written our own template classes, but why to reinvent the wheel? `Loki::Typelist` is a useful, versatile tool at hand and we will rely on it when building our structures.

[1] Though similar typelists were also discovered by Jaaki Järvi and they are also introduced in [6] both before [1], it was Alexandrescu who provided a comprehensive library and framework with it.

3.1 Overview of Loki::Typelist

The only role of this class is to hold compile-time information as a list of types in a single class. Its definition is very simple:

```
template <class T, class U>
struct Typelist {
    typedef T Head;
    typedef U Tail;
};
```

Basically this template is similar to a *trait* in that it contains static type information. An example of using this template can be given as:

```
template <class TypeInfo> class RefCountPtr {
    typedef typename TypeInfo::Head counterType;
    typedef typename TypeInfo::Tail pointeeType;
    ...
};

typedef Typelist<unsigned int, string> MyPtrTraits;
typedef RefCountPtr<MyPtrTraits> MyPtr;
```

One may think that `Typelist` can hold information of only two types. However, using a recursive approach, longer lists can also be written.

```
typedef Typelist< char, Typelist<signed char, unsigned char> >
        CharList;
```

Obviously, the head of the list can be simply referenced as `CharList::Head` when using such a list. To refer to the remaining types a longer expression is needed, but specifying classes like `CharList::Tail::Head` is almost as undesired as hardwiring the number of parameters. Fortunately it is possible to provide a better way.

By convention, a `Typelist` must end with the special type `NullType`. For compile-time algorithms it serves as `\0` does for C-strings: it marks the end of a list as an extremal element. Thus indexing and iterating over the list can be implemented without serious restrictions.

Because recursion makes defining a long list really annoying and error-prone, macros are written in Loki to ease this problem. Though we have to specify the exact length of the list and the types to use them, recursion-handling and adding `NullType` are not needed explicitly anymore:

```
#define TYPELIST_1(T1) Typelist<T1,NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
...

typedef TYPELIST_3(char, signed char, unsigned char) CharList;
```

3.2 Recursive Inheritance

`Loki::Typelist` has several recursive compile-time algorithms: random access, append, removal of duplicates, etc. All of them are based on partial template specialization (see [14] and [1]) utilizing `NullType` at the end of the list.

Our solution will be very similar to them from this point of view[2]. Now `Family` may have only one parameter type containing all required types to be assembled to a structure:

```
// --- Forward declaration
template <class List> class Family;

// --- Partial specialization for general lists
template <class Head, class Tail>
class Family< Typelist<Head,Tail> > :
    public Head, public Family<Tail>
{};

// --- Partial specialization for lists of one element
template <class Head>
class Family< Typelist<Head,NullType> > :
    public Head
{};
```

The forward declaration is only for safety purposes, there is no implementation for it. The two partial specializations have their own implementations, thus ensuring that a compile-time error will occur if the template is instantiated with any type other than a `Typelist`.

The first implementation inherits from both `Head` and `Family<Tail>`, but — because it is specialized for lists of only one element — no `Tail` remains for the second, therefore it inherits only from `Head`. A sample instance and an according class hierarchy may be seen below and in figure 3.

```
typedef Family< TYPELIST_3(OpDisplay, OpEval, OpCheck) Operator;
```

4 Instantiating Objects

After creating the class hierarchy we construct instances of classes. We need a constructor for our structure. As a consequence of using `Typelist` and recursive inheritance, during our compile time recursions we know nothing about the types in the list: we have only the atomic type `Head` and the remaining list `Tail` of any

[2] We could also use classes introduced in [1] like `Loki::GenScatterHierarchy` or `Loki::GenLinearHierarchy` to build our hierarchies, but they are needlessly complex for us. We do not need most of their functionalities so we would rather write a similar, but very simple class from scratch.

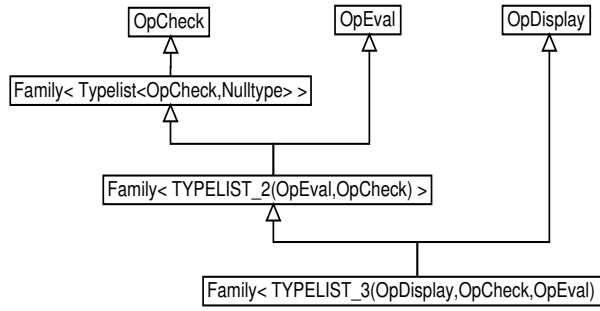

Fig. 3. A sample hierarchy of the `Family` template.

length. Without knowing all list elements, conventional constructor techniques cannot be used.

Our only possibility is continuing with the recursive technique by demanding two constructor parameters of types `Head` and `Family<Tail>`. Hereby our structure still remains quite simple as its base classes can be directly set using member initializer lists. (For a solution of the mixin constructor problem see [7]). Reasonably, we also want a copy constructor for easy initialization:

```
template <class Head, class Tail>
class Family< Typelist<Head,Tail> > :
    public Head, public Family<Tail>
{
public:
    // --- Type name shortcuts
    typedef Family<Tail> Rest;
    typedef Family< Typelist<Head,Tail> > MyType;

    // --- Copy constructor
    Family(const MyType& mt) : Head(mt), Rest(mt) {}

    // --- "Recursive" constructor
    Family(const Head& head, const Rest& rest) :
        Head(head), Rest(rest) {}
};

template <class Head>
class Family< Typelist<Head,NullType> > :
    public Head
{
public:
    // --- All in one constructor
    Family(const Head& head) : Head(head) {}
};
```

After defining the required constructors we are able to create our first collaboration group.

```
typedef Family< TYPELIST_2(PlusEval, PlusCheck) > PlusCalc;
typedef Family< TYPELIST_3(PlusDisplay, PlusEval, PlusCheck) > Plus;

// --- Create a 3-in-1 object
PlusEval add; PlusCheck checkParams; PlusDisplay show("+");
PlusCalc calculate(add, checkParams);
Plus sum(show, calculate);
```

Unfortunately a temporary object `sum` has to be used for passing a second parameter to the constructor. For a longer list more temporary objects are needed and longer member initialization is done. Though one could write this in a shorter form without variables[3], it is still not an appropriate solution:

```
Plus sum( PlusDisplay("+"), PlusCalc(PlusEval(), PlusCheck()) );
```

Even for this form of instantiation we have to embed constructor calls. It is very similar to the typelist definition problem, where macros were written to linearize the recursion. We can generate macros for our classes and use them in almost the same form as `TYPELIST_X`:

```
// --- Sample macro definitions for a family of 3 classes
#define FAMILY_3(T1,T2,T3) Family< TYPELIST_3(T1,T2,T3) >

#define FAMILYVAR_3(name, T1, P1, T2, P2, T3, P3) \
    FAMILY_3(T1,T2,T3) name (P1, FAMILY_2(T2,T3) (P2,P3) )

// --- Using a default constructor
FAMILY_3(PlusDisplay, PlusEval, PlusCheck) sum;

// --- Using object for initialization
FAMILYVAR_3(sum, PlusDisplay, PlusDisplay("+"),
    PlusEval, PlusEval(), PlusCheck, PlusCheck());
```

In the last example above `sum` stands for the name of the variable followed by the parameter types and the actual parameters. Though a default constructor was still not introduced for the `Family` template, it can be easily implemented for both specializations. We need only two lines of code to be added next to the copy constructors:

```
Family(): Head(), Rest() {}   //for general lists
Family(): Head() {}           //for a list of one element
```

[3] This form of constructor calls could be parsed as a function declaration with pointer-to-function type parameters, see [9].

5 Structural Conversions

We have already got class families, but they are still not related to each other. To provide the required subtype relations we need a structural conversion between adequate groups of classes. Unlike the creation of the hierarchy and the objects, we have lots of different ways to implement the conversion.

5.1 Do It in the Naive Way

At first glance the most simple way is a conversion function iterating over all types in the list and convert them to their appropriate base types. Following our compile-time recursion technique, this can be made in two steps: recursively converting the tail of the list first and simply converting the head type at last. We can make use of partial template specialization to implement this function. Because partial template specialization is allowed only for classes, we introduce an auxiliary `Converter` class. We do not want to create any instances of the class `Converter`, hence our function `convert` will be static:

```
// --- Forward declaration
template <class ToList, class FromList> struct Converter;

// --- Partial specialization for general lists
template <class ToHead, class FromHead,
          class ToTail, class FromTail>
struct Converter< Typelist<ToHead,ToTail>,
                  Typelist<FromHead,FromTail> >
{
    typedef Family< Typelist<ToHead,ToTail> > ToType;
    typedef Family< Typelist<FromHead,FromTail> > FromType;

    static ToType convert(const FromType& from)
    {
        // --- Recursion to the rest of list
        Family<ToTail> toTail =
            Converter<ToTail,FromTail>::convert(from);

        // --- Conversion FromHead -> ToHead
        return ToType(from, toTail);
    }
};


// --- Partial specialization for lists with one element
template <class To, class From>
struct Converter< Typelist<To,NullType>,
                  Typelist<From,NullType> >
```

```
{
    // --- Simple conversion From -> To
    static To convert(const From& f) {
        return f;
    }
};


// --- An example using the Converter class
typedef TYPELIST_3(OpEval, OpCheck, OpDisplay) ToList;
typedef TYPELIST_3(PlusEval, PlusCheck, PlusDisplay) FromList;

Family<FromList> sum;

Family<ToList> expr = Converter<ToList,FromList>::convert(sum);
```

This conversion became quite complex, but would be worth the hard work if it was all we need. Unfortunately it is far from that for its highly limited usability. The solution above supports only lists of classes of the same length and the same ancestor-descendent order. When adding a new element to `FromList` or when changing the order of elements in either lists, an error will occur during compilation.

There are still other problems with this construction. Firstly, there is a serious hidden inefficiency in the conversion. The construction of a `Family` object requires time linear in the length of the list of types. This is optimal because each type in the list has to be initialized. Inefficiency comes in sight during conversion: in every call of `convert()` an object is instantiated to return the result of the actual conversion step, which has a linear cost itself. Multiplied with the number of function calls, the total conversion cost becomes $O(n^2)$.

Secondly, there is still an explicit call of a function needed for the conversion. We intend our framework to work completely transparent (i.e. structural conversions should work without explicit function calls).

5.2 Template Constructors

All problems above can be solved by using a different approach. Firstly, we do not assume that the type we want to convert is a `Family` which is accessible by iterating over only `ToList` and ignoring `FromList`. Consequently, the object to be converted can be any user type created by multiple inheritance.

To avoid explicit function calls, we can use either a simple conversion operator or a constructor. Neither of them is suited for recursive solutions. Difficulties arise at generality, because not a single specific conversion is needed: we want all possible conversions to be supported. Both functions have to be implemented as members, so our `Converter` is of no use anymore, we must change to template functions.

To solve the cost problem, we need a radical change in our approach, because a conversion function returning a constructed result by value is not acceptable.

We can avoid copying the temporary results by initializing the adequate part of the resulting structure directly within the converted object.

We need a neat trick to solve the united needs of the two approaches above. Using template constructors may sound a little weird, but constructors are functions, thus they provide a possible solution. Even the essence of our previous conversion may remain the same, only its frame changes. Consider the following code:

```
template <class Head, class Tail>
struct Family< Typelist<Head,Tail> > :
    public Head, public Family<Tail>
{
    typedef Family<Tail> Rest;
    typedef Family< Typelist<Head,Tail> > MyType;

    // --- Good old constructor
    Family(const Head& head, const Rest& rest) :
        Head(head), Rest(rest) {}

    // --- Conversion and copy constructor
    template <class FromType>
    Family(const FromType& from) : Head(from), Rest(from) {}
};

template <class Head>
struct Family< Typelist<Head,NullType> > : public Head
{
    typedef Family< Typelist<Head,NullType> > MyType;

    // --- All-in-one constructor
    Family(const Head& head) : Head(head) {}
};
```

This code is even more simple. It is not obvious to see, but it follows the same conversion steps using the member initialization lists.

To examine this code, see the following example:

```
class Plus: public PlusEval,public PlusCheck,public PlusDisplay{};
Plus sum;
FAMILY_2(OpEval, OpCheck) calculate = sum;
```

As explained above, the converted object does not have to be an instance of `Family`, a hand made type will do. We can see that no temporary objects were used. The time cost of the conversion is linear since we iterate over the list elements only once.

The conversion is implicit, we did not use any explicit function calls. Since it is implemented as a constructor call, the compiler will silently do these kinds of conversions whenever needed, which greatly improves the ease of use.

Still one additional feature for our class may be desirable: an `operator=` to copy our objects. It can be easily implemented following our previous recursive techniques, similarly to our template constructor:

```
// --- Implementation for general list
template <class FromType>
MyType& operator = (const FromType& from) {
    Head::operator= (from);

    // --- recursive call
    Rest::operator= (from);
    return *this;
}

// --- Implementation for lists of one element
MyType& operator = (const Head& from) {
    Head::operator= (from);
    return *this;
}

// --- Example using our previous object
FAMILY_3(MinusEval, MinusCheck, MinusDisplay) minus;
calculate = minus;
```

6 Dynamic Binding

Now conversion works fine, providing coercion polymorphism for our class families. Like all default conversions in C++, our conversion is done by value, not by reference. We would also like to use dynamic binding, so we will follow the C++ way: build our smart pointer and reference classes to support it.

6.1 Smart Pointers

How can we write a smart pointer? The usual way is to have a referring member (mostly pointer but it also can be a reference) and overloaded operators like `operator->` which do some additional work (e.g. reference counting).

The natural extension of our previous recursive technique would produce an inheritance hierarchy using referring types for bases. We cannot inherit from either pointers or references so a pointer data member for each type in the list will be used instead. We build the following structure:

```
template <class Head, class Tail>
class FamilyPtr< Typelist<Head,Tail> > : public FamilyPtr<Tail>
{
    Head* head;
    ...
};
```

```
template <class Head>
class FamilyPtr< Typelist<Head,NullType> >
{
    Head* head;
    ...
};
```

Thus the structure becomes linear. An example (for the same list of types as shown in figure 3) can be seen in figure 4.

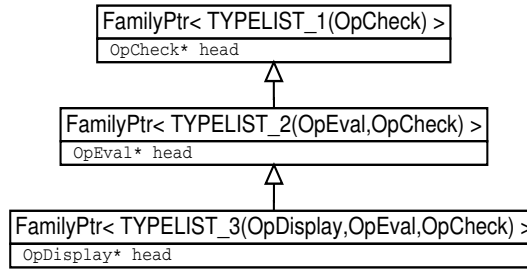

Fig. 4. A sample hierarchy of the `FamilyPtr` template resulting from the same example as that of figure 3.

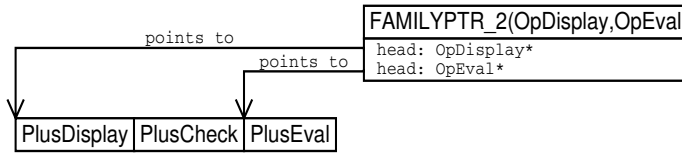

Fig. 5. A sample initialization of pointer members in `FamilyPtr`.

When constructing a `FamilyPtr` instance, every pointer member is set to the adequate part of the referred composite object. (See figure 5).

```
// --- FamilyPtr general implementation for Typelists
template <class Head, class Tail>
class FamilyPtr< Typelist<Head,Tail> > : public FamilyPtr<Tail>
{
    Head* head;
public:
    typedef FamilyPtr<Tail> Rest;
    typedef FamilyPtr< Typelist<Head,Tail> > MyType;

    FamilyPtr() : Rest(), head() {}
```

```
    // --- Conversion and copy constructor
    template <class FromType>
    FamilyPtr(FromType& from) : Rest(from), head(&from) {}

    // --- Simple cast operators
    operator Head& () const { return *head; }
    operator Head* () const { return head; }
};

template <class Head>
class FamilyPtr< Typelist<Head,NullType> >
{
    Head* head;

public:
    typedef FamilyPtr< Typelist<Head,NullType> > MyType;

    // --- Simple default and copy constructors
    FamilyPtr() : head() {}
    FamilyPtr(Head& from) : head(&from) {}

    // --- Simple cast operators
    operator Head& () const { return *head; }
    operator Head* () const { return head; }
};
```

Most of this code should not be a surprise by now, only the cast operators are really new. Because `Head` is not an ancestor any more, we need a workaround for keeping the possibility of conversion to `Head`. Overloading `operator->` to return pointer `head` will not do what we need: always the `operator->` of the most derived type would be called because of hiding, no better match would be searched for in base types.

Cast operators can provide a solution for this problem. Instead of overloading operators like `operator->`, we can directly convert our object to a pointer. Unfortunately this solution also has a drawback. By language definition automatic conversions are disabled on the left side of operators like `operator->` , so an explicit cast is needed to the required type. See the following example:

```
FAMILY_3(PlusDisplay, PlusEval, PlusCheck) sum;
FAMILYPTR_3(OpDisplay, OpEval, OpCheck) exprPtr(sum);

// --- Function call with explicit cast
static_cast<OpDisplay*>(exprPtr)->show();

// --- In longer form with implicit cast
OpDisplay *displayPtr = exprPtr;
displayPtr->show();
```

This explicit cast can be uncomfortable, but it also can be useful for removing the possibility of ambiguous expressions. It is still not an advantage though, but we could find no better way so we will have to live together with this limitation.

We should also implement a copy operator for our pointers which will be quite similar to the one of `Family`. A surprising explicit cast is needed though, because a conversion `T` → `U` with the cast operators of `T` does not imply that `T*` → `U*` is also possible, because builtin pointer types do not have their own cast operators.

```
template <class Head, class Tail>
class FamilyPtr< Typelist<Head,Tail> > : public FamilyPtr<Tail>
{
    template <class FromType>
    MyType& operator = (FromType& from) {
        // --- No builtin FamilyPtr<T>* -> T* conversion
        // --- allowed so explicit conversion needed
        head = & static_cast<Head&>(from);
        Rest::operator= (from);
        return *this;
    }
};

template <class Head>
class FamilyPtr< Typelist<Head,NullType> >
{
    MyType operator = (Head& from) {
        head = &from;
        return *this;
    }
};
```

As a result, dynamic binding and all conversion facilities of `Family` are provided by our pointers as structures and pointers are completely interchangeable[4]. No `operator*` is needed to dereference the pointers, so all conversions are possible between pointers and structures:

```
FAMILY_3(PlusDisplay, PlusCheck, PlusEval) sum;
FAMILYPTR_3(PlusDisplay, PlusCheck, PlusEval) plusPtr = sum;

FAMILY_3(OpDisplay, OpCheck, OpEval) expr;
FAMILYPTR_3(OpDisplay, OpCheck, OpEval) exprPtr = sum;

exprPtr = plusPtr;     // --- pointer -> pointer conversion
exprPtr = sum;         // --- object  -> pointer conversion
expr = plusPtr;        // --- pointer -> object  conversion
```

[4] This property is not necessarily an advantage and may need further revision.

When a `FamilyPtr` object is a left value, no objects are copied, only pointer members are set to adequate addresses. This greatly improves speed compared to `Family` which copies whole objects by value.

Apart from the need of the explicit cast to the required builtin pointer type, `FamilyPtr` is able to substitute `Family` much more efficiently. Why would we still use `Family` then? We do not have to, but we had better to do. It is designed to contain several objects and handle them together, without it we would have tons of separate objects and a chaotic resource management. We suggest creating collaborating objects in one `Family` object and then making all conversions using adequate `FamilyPtr` types.

6.2 Smart References

Usually references are preferred to pointers. Using them we can do the same as with pointers without essential changes. The only difference is that pointer members are changed to references, thus no address operator (`operator&`) is needed for initialization and copy. This has a few consequences:

- No default constructor can be made (references have to be initialized immediately when created).
- When using `operator=`, not pointers are set, but whole objects are copied, which may cause loss of dynamic type information. This means that dynamic binding is done only when initializing the reference. (E.g. we have a reference of type `Vehicle&` initialized to a `Car` object. If an object of `SportsCar` is copied to the referenced object using `operator=`, no dynamic binding is provided).

Because its implementation is almost like that of `FamilyPtr`, we omit the necessary program code[5].

7 Robustness and Limitations

An important advantage of our solution is that it relies only on standard C++ template features. Similarly to STL no extension to C++ is required, so (theoretically) any number of collaborating classes could be used with any standard compliant compiler. Effectively compilers do not follow the C++ standard and have finite resources. We have tested some compilers and had the following results:

[5] Source code of our whole framework can be downloaded from http://gsd.web.elte.hu/publications/

Compiler	# of classes	Cause of limitation
g++ 3.2	45	Macro parameter limit in precompiler
g++ 2.96	17	Recursion depth limit reached
Intel 7.1	25	Unacceptable compile time on a PIII 750MHz processor, it seems that exponential resources are consumed
VC++ .NET	0	Partial template specializations are not implemented
Borland 6.0	0	Loki does not compile
OpenWatcom 1.0	0	Loki does not compile

Though the number of classes in a family is limited, it's not a serious limitation because at most a dozen of classes are practically enough to express most design issues. Our solution is also completely type safe which is gained by relying on builtin compiler rules and language features. Our structures are converted class by class, so the compiler supports all types of required conversions. On the other hand, a compile time error will arise whenever an invalid conversion is done.

Though our conversions are working properly, they still have a few issues in consequence of certain C++ language rules. They are as follows:

- All required conversions are supported, but sometimes even conversions that were not intended are also introduced. They are valid but undesirable results of a class design error. Just imagine the class `std::string` in the source list and `std::istringstream` in the target list. Now an error would be expected, but `std::istringstream` has a constructor with a string parameter as an initial buffer value, so the conversion is completely legal. Defining our constructors `explicit` does not help here because explicit constructor calls are made in `Family` during conversion in the member initializer lists. It can be avoided only by careful class design.
- If a type is added to a list more than once or repeated inheritance occurs in the user object to be converted, a compilation error will arise for type ambiguity.
- If a required method of the class is not present (e.g. there is no copy constructor, default constructor or the conversion to a base class is explicitly forbidden), the code based on the non-existent feature will not compile.

They are not results of our construct, only consequences of the C++ language standard.

8 Related Work

Several discussions were made on the applicability of C++ mixins/mixin layers to solve similar problems (see e.g. Smaragdakis and Batory in [12]). They had

shortages either not to cover the whole problem addressed by us or inconveniences at usage.

In languages with structural subtyping (see e.g. [4]) the same problem does not arise: structural subtyping is disjunctive. Our tailor-made conversions can be considered as a tool to make the subtype relation of C++ closer to structural subtyping. We increase the flexibility of subtypes, but "without the loss of the semantic information that hierarchies of type names provide". The goal of Muckelbauer and Russo in [10] is similar, but they start from a language with structural subtypes and propose the addition of "semantic attributes" to types. This way they can express the same semantic dependencies as the inheritance-based type hierarchies for the different concerns in our approach.

Advanced techniques for separation of concerns such as Multi-Dimensional Separation of Concerns [11], Aspect-Oriented Programming [8] or Composition Filters [2] can also aid collaboration-based design. However, these techniques are aiming to solve problems different from the one this paper brought on. For example, in AspectJ [5] or in AspectC++ [13] we can extend a hierarchy of classes with orthogonal concerns (implemented as aspects), preserving the subtype relationships, but this extension applies to whole classes and not to individual objects.

9 Conclusions

C++ supports explicit subtyping based on multiple inheritance. This subtyping mechanism is not flexible enough to express certain subtype relationships which are necessary for implementing collaboration-based designs. Our framework extends the possibilities of the subtyping mechanism in C++: it allows families of collaborating classes to be subtypes of other families. The framework makes the subtype relation disjunctive with respect to multiple inheritance.

In the implementation of our framework we have used standard template metaprogramming tools such as `Loki::Typelist`. If the programmer expresses families of collaborating classes with the help of the templates defined in our framework, (s)he can make use of coercion polymorphism using class `Family` or inclusion polymorphism with class `FamilyPtr`.

References

1. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
2. Lodewijk Bergmans, Mehmet Aksit. Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.
3. Kim B. Bruce. Foundations of Object-Oriented Languages. The MIT Press, Cambridge, Massachusetts (2002)
4. Luca Cardelli. Structural Subtyping and the Notion of Power Type. Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, January 1988. pp 70-79.

5. Eclipse. The aspectj project. http://www.eclipse.org/aspectj/
6. Krzysztof Czarnecki, Ulrich W. Eisenecker. Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
7. Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. Presented at the GCSE2000 Workshop on C++ Template Programming.
8. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241, June 1997.
9. Scott Meyers. Effective STL. Addison-Wesley (2001). pp. 33-35.
10. Patrick A. Muckelbauer, Vincent F. Russo. Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems. USENIX Conference on Object-Oriented Technologies (COOTS), http://www.usenix.org/publications/library/proceedings/coots95/
11. Harold Ossher, Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center. http://www.research.ibm.com/hyperspace/Papers/tr21452.ps
12. Yannis Smaragdakis, Don Batory. Mixin-Based Programming in C++. In proceedings of Net.Object Days 2000 pp. 464-478
13. Olaf Spinczyk, Andreas Gal, Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, February 18-21, 2002. http://www.aspectc.org/download/tools2002.ps.gz
14. Bjarne Stroustrup. The C++ Programming Language Special Edition. Addison-Wesley (2000)
15. Harold Ossher, Peri Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. IBM Research Report 21452, April, 1999. IBM T.J. Watson Research Center. http://www.research.ibm.com/hyperspace/Papers/tr21452.ps