Template Base Delegation

Ted Law
tedlaw@vnet.ibm.com

IBM Software Solutions Toronto Laboratory
844 Don Mills Road
North York, Ontario M3C 1V7
Canada

ABSTRACT

When a class derives from a template instantiation, the base class is called a template base class. Non-template base classes are more commonly used, but do not take into account the specific needs of the derived class. The base class can provide better service; it can be customized by becoming a template instantiation using the derived class and/or other classes as arguments. In this way, the template base class can tailor-make itself for the derived class, taking advantage of the knowledge of its type. For example, the template base class can type its methods appropriately for the derived class. Furthermore, inheriting publicly from a template base class makes it possible for the derived class to delegate part of its public type-specific interface to the base. This paper explores the advantages of template base delegation.

1 Introduction

In C++ object-oriented programming, objects frequently delegate responsibilities to other objects to enhance modularity, reuse, and extendability[5]. Responsibilities can also be abstracted into the base class to increase sharing among derived classes. Usually the type of the base class is fixed. Consequently, the signature of the methods and the types of the data members of the base class must be fixed. By turning the base class into a template instantiation, it can be customized according to the type of the derived class.

To illustrate, Fig. 1(a) shows two classes `D1` and `D2` inheriting from a common non-template base class `B`. Fig. 1(b) shows the technique of template base delegation, where `D1` and `D2` inherit from template base classes `B_<D1,...>` and `B_<D2,...>`, respectively.

This is a powerful technique that can be usefully applied to a variety of domain areas.

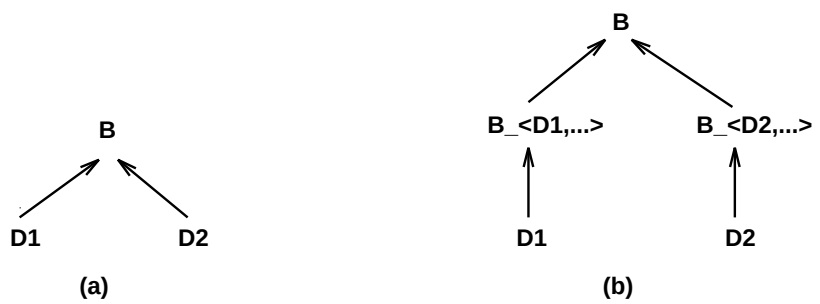

Figure 1: Using template instantiations as bases

Section 2 demonstrates how template base delegation can be used to provide an alternative to the container class implementation of lists. This simple example is intended for illustrative purposes. Section 3 builds on this example, explaining how the technique can be usefully applied to the area of object-modeling.

2 A Simple Example of Template Base Delegation

Many introductory text books on C++ use List<T> as an example for source code reuse[5]. List<T> is called a container class template.

A limitation of embedding a type T object in ListItem<T> is that the type T object can belong to at most one List<T>. This limitation can be readily overcome by using List<T*> instead of List<T>. See Fig. 2.

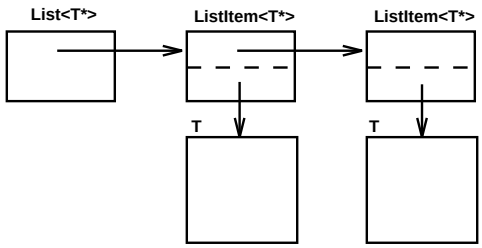

Figure 2: List<T*>

The number of List<T*>s that can contain the same T object can now vary dynamically. There is no predefined limit on the number of List<T*>s to which a T object can belong. The cost of this feature is the allocation cost of ListItem<T*> and the extra level of indirection to get to the T object. This cost is more expensive than we need because the number of lists that an object needs to be on is usually fixed at compile time.

Consider the ER diagram[1] in Fig 3, which shows the relationships in a Company-Employee example. In this example, Manager is a semantic *subtype* of Employee.

Assuming the two relationships are implemented using simple linked lists, Fig. 4 represents one possible runtime instance of some information.

In each Employee object there are two next pointers: one for chaining all the employees of the same company; the other for chaining all the employees under the supervision of the same manager, who himself is also an employee.

Since this application has two almost identical linked lists, there is an opportunity for source code sharing that can be realized by wrapping each next pointer in a base class. However, to distinguish between the two different next pointers, we can turn the base into a template called FanTip_[2][3] and parameterize it by the kind of the relationship that is being represented. Similarly, the head pointers at the beginning of the lists are wrapped into a template called FanHead_. Together they are called the Fan base class templates. For readers interested in more detail, the source code for this example is given in the Appendix.

[1] Entity-Relationship Modeling is a popular semantic data modeling technique in the database area. [5]

[2] The shape of a hand fan is used to name these templates. In the example, a FanHead_ is the head of a simple linked list, and a FanTip_ is an item on the list.

[3] The naming convention used here:

a. Data members are always postfixed by an underscore.

b. An underscore at the end of a class name indicates that the class is meant only to be used as a base of another class. It is not stand-alone.

c. For function parameters, references are used for objects that must exist, and pointers are used if they may exist.

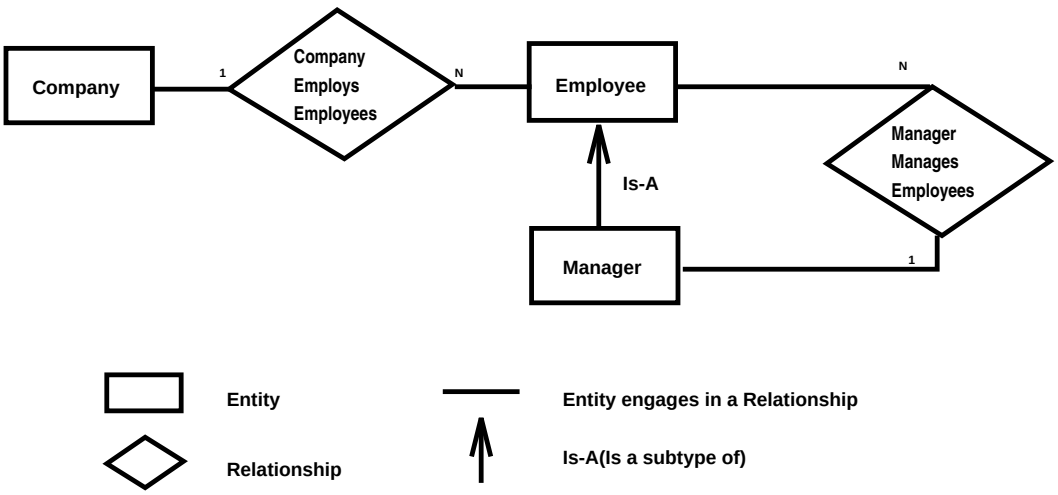

Figure 3: ER diagram of the `Company-Employee` example

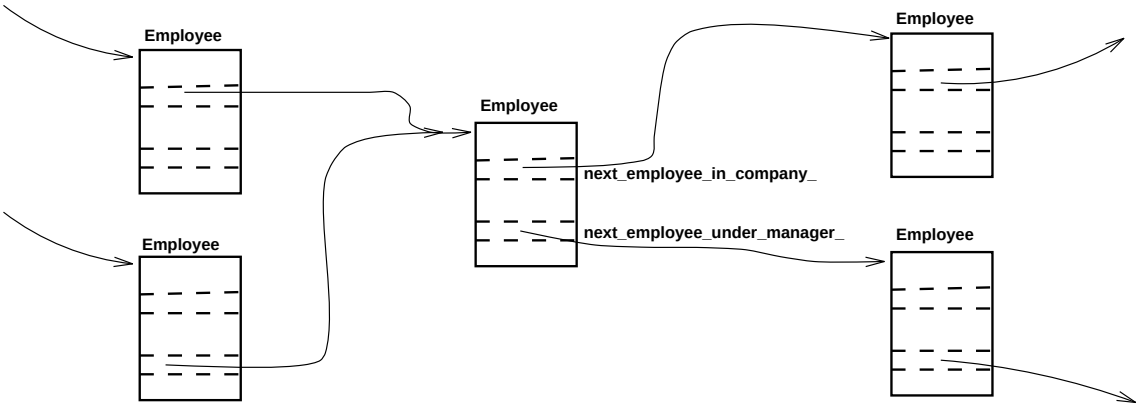

Figure 4: A runtime instance of multiple linked lists

Both `FanHead_` and `FanTip_` take three arguments: the type of the 1 side, the type of the N side, and the type of the 1-N relationship. The template class definitions are written so that the class definitions of the template arguments are not needed. In fact, the definitions of `CompanyEmploys Employees`[4] and `ManagerManagesEmployees` do not need to exist unless they are required for other reasons (for example, to carry an attribute of the relationship).

Since `FanHead_` and `FanTip_` know the type of the derived class, their interfaces can be typed accordingly. For example, `FanHead_` defines an `add_tip()` method, which takes a parameter of type `Tip&` (the second template argument). The client class `Company` customizes its base by inheriting from `FanHead_<Company,Employee,CompanyEmploysEmployees>`. Thus, the inherited `add_tip()` method will expect an `Employee&` as a parameter.

In addition to the advantages of source code sharing and strong typing discussed above, using the `Fan` template bases also makes the definitions of the derived classes more declarative. The template bases *declare* what relationships the derived class can engage in using the inheritance clause of a `C++` class declaration:

[4] The convention is that the name of a relationship has the encoding, Noun1VerbNoun2, for example `Company EmploysEmployees`. The "direction" of the relationship is always from Noun1 to Noun2. Noun1 is always on the outgoing side, and Noun2, incoming.

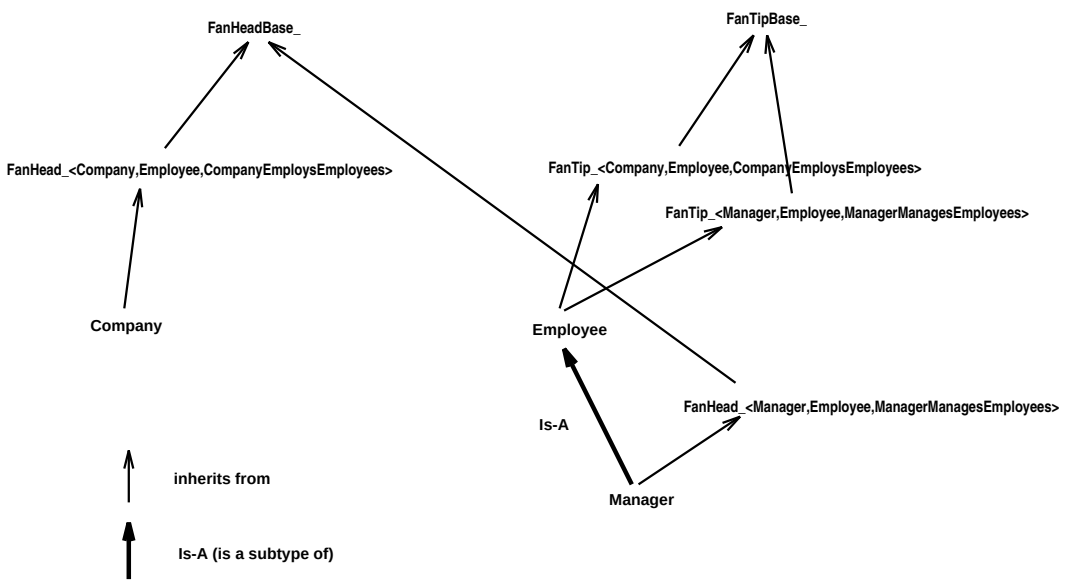

Figure 5: Inheritance hierarchy of the Company-Employee example

```
class Employee :
  protected FanTip_<Company,Employee,CompanyEmploysEmployees>,
  protected FanTip_<Manager,Employee,ManagerManagesEmployees>
{
    ...
};
```

Protected inheritance is used because `Manager`, a subtype of `Employee`, may need to access the methods which `Employee` inherits from its `FanTip_` bases. Public inheritance is not used in this case because `Employee` is not, semantically, a subtype of `FanTip_<Company,Employee,Company EmploysEmployees>`.

Typically, the binary code for every instantiation of a template is generated. To avoid generating an undue amount of binary code, we can make `FanHead_` inherit from a non-template base class `FanHeadBase_`, and `FanTip_` from `FanTipBase_`. There is no need for these non-template bases to realize the aforementioned advantages (source code sharing, strong typing, and declarative programming). However, factoring most of the code out of the template classes and into their non-template bases means that only one copy of the binary code needs be generated. After the bulk of the code is moved into these non-template bases, only pointer adjustment code, resulting from up/down casting, is left in the templates. When all the template code is inlined, the same efficiency can be obtained as any C equivalent. Thus storage and runtime efficiency can be added to the list of advantages of applying the technique of template base delegation to the problem of implementing linked lists.

If we compare the `Fan` templates with the conventional `List` container template, we will see that `List` makes no assumption about its client class, while `FanHead_` and `FanTip_` each assumes that it is inherited by its own client class. Using this assumption, `FanHead_` and `FanTip_` can embed, respectively, the `first_tip_` and the `next_tip_` pointers into objects of their client classes.

In summary, template base delegation allows a base class to be customized according to the derived class. That customization can be on the signatures of the methods or on the types of the data members. It gives the base class a chance to know exactly who is inheriting from it. Using template

bases is also more declarative, directing the reader of the source code to the actual intentions of the programmer. The example shown in this section is a simple application of the technique. The author has found this technique to be quite useful in the area of object-modeling, which is explained next.

3 A Real Life Example in Object Modeling

In many C++ database-like applications, there is a C++ class, called an *entity class*, used to represent each type of entities in the real world. And there is a C++ class, called a *relationship class*, used to represent each type of relationships in the real world. Furthermore, the application code needs to inquire into the possible relationships among these types of entities and their attributes at run time. This information is often called the (conceptual) schema of the application. To make the schema information available, we represent the schema at run time using a network of objects called *descriptors*. We use a unique *entity descriptor* object to represent each entity class, and a unique *relationship descriptor* object for each relationship class. A challenging problem is to ensure the consistency between the schema information as manifested in the source code and the explicit representation of that schema at run time.

Manually maintaining a schema generation routine or table according to the schema used in the source code is error prone. It is easy to change one but forget to update the other.

Consider the inheritance hierarchy shown in Fig. 6 for the same conceptual schema as was used in the previous example (Fig. 3).

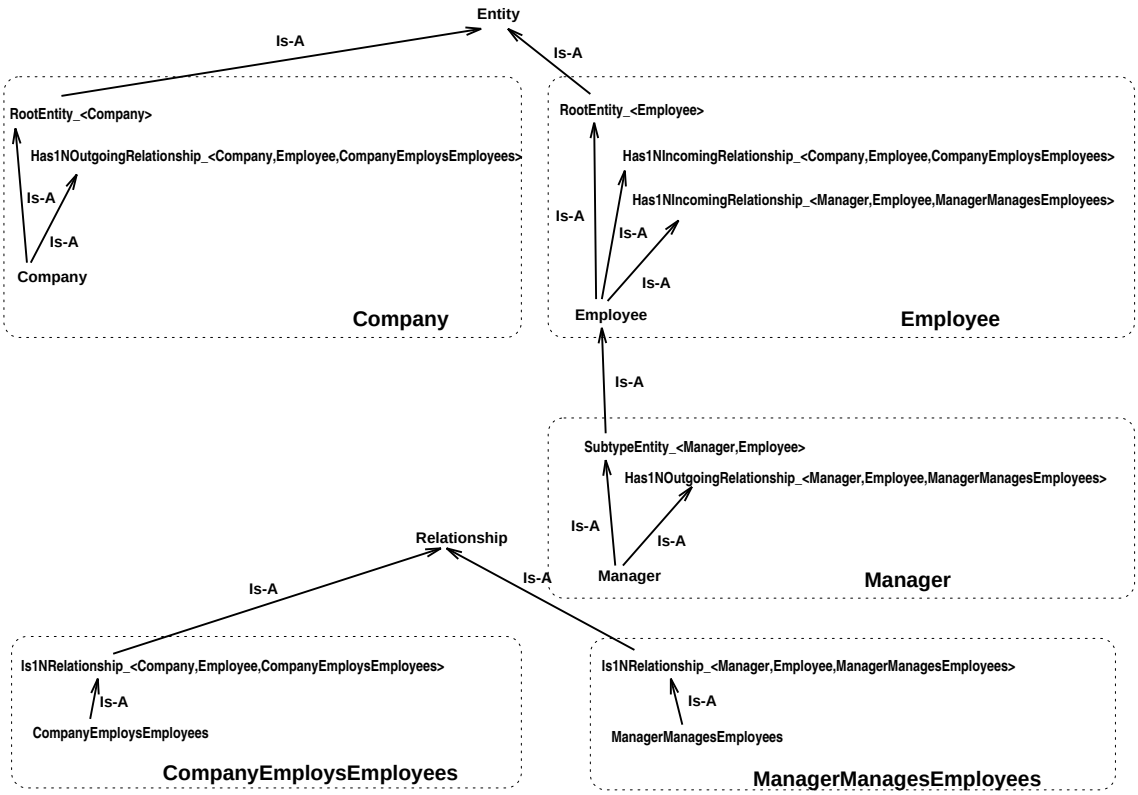

Figure 6: Inheritance hierarchy when using template bases to capture schema information

The fact that `Company` and `Employee` engage in a 1-N `CompanyEmploysEmployees` relationship is captured by `Company` inheriting from the base class `Has1NOutgoingRelationship_<Company, Employee,CompanyEmploysEmployees>`, and `Employee` inheriting from the base class `Has1NIncom ingRelationship_<Company,Employee,CompanyEmploysEmployees>`.

The fact that `Manager` is a subtype of `Employee` is captured by `Manager` inheriting from `SubtypeEntity_<Manager,Employee>`. The template `SubtypeEntity_` is written so that it derives publicly from its second template argument. This is a new kind of customization that allows the client to specify to the template base whom to inherit from, so that the intended subtype hierarchy in the application is preserved.

Every inheritance shown in Fig. 6 is public. Since the public interface of a class is the union of the public interface of all of its public bases, we can look at an entity class together with its public template bases as forming an "expanded" public interface for that semantic entity. This is shown by the dotted rectangles in the figure. In a way, each entity class have delegated part of its public interface to its template bases. Alternatively, each template base can be viewed as representing a particular role that the semantic entity can play. Note that, in the conceptual schema shown in Fig 3, there are three kinds of semantic entities, `Company`, `Employee` and `Manager`, where `Manager` is a subtype of `Employee`; and there are two kinds of relationships, `CompanyEmploysEmployees` and `ManagerManagesEmployees`. These facts will be captured by the template bases. Other helper classes can also be added to the hierarchy as required without affecting the representation of this schema.

3.1 Building the Schema Representation

Since template bases have been used to "declare" the various pieces of the schema, this information can be used to construct the schema's runtime representation automatically.

There is a static "init" object defined for each instantiation of `RootEntity_`, `SubtypeEntity_`, and `Is1NRelationship_`. Each init object uses the type information supplied to the template base to specify how a piece of the schema should be built.

The construction of the schema representation begins by chaining these "init" objects together to form a single list. This chaining can be easily achieved during static initialization by the constructor of a common base class, `Init_`, for these static init objects.

The rest of the construction process is carried out when a static member function `Init_::initialize()` is called sometime after `main()` is entered.[5] In this function, several phases may be involved. During each phase, a pass is made through the list of init objects, calling a different virtual method for each phase. After each phase is finished, the next phase begins, until all the phases are carried out.

There are two important phases.

The `build_descriptor_phase` is the phase for allocating all descriptor objects. For example, when the `build_descriptor_phase` virtual method of the init object of `RootEntity_<Employee>` is called, it will allocate an entity descriptor object to represent the entity class `Employee`. The `build_descriptor_phase` virtual method of the init object of `Is1NRelationship_<Company,Employee, CompanyEmploysEmployees>` will allocate a relationship descriptor object to represent the relationship `CompanyEmploysEmployees`. When this phase is finished, all descriptor objects will be created.

During the `link_schema_phase`, the descriptor objects are connected to represent the relationships among them. For example, when the `link_schema_phase` virtual method of the init object of `Is1N Relationship_<Company,Employee,CompanyEmploysEmployees>` is called in this phase, it will link the entity descriptor objects for `Company` and `Employee` to the relationship descriptor object for `CompanyEmploysEmployees`. Also, when the `link_schema_phase` virtual method of the init object of `SubtypeEntity_<Manager,Employee>` is called, it will link the entity descriptor object for `Manager` and that for `Employee` to represent the subtype relationship between the two.

[5] Because `Init_::initialize()` will call some virtual methods in the translation units that define the static init objects, the objects are guaranteed by the language to be properly initialized.

3.2 Overriding Pure Virtuals

Template base delegation can be used to supply similar concrete implementations for pure virtual functions declared in an abstract base class. Suppose there is a pure virtual function `entity_descriptor()`, declared in the abstract base class `Entity`, which is to be overridden by each entity class to return its own descriptor object.

Using the conventional way, each entity class will have to provide an overriding virtual function. This will likely involve:

1. Declaring the virtual function `entity_descriptor()` in the definition of the entity class.
2. Declaring the descriptor object, or a pointer to it, as a static data member.
3. Defining the virtual function `entity_descriptor()`, which returns the static data member.

Alternatively, an entity class can delegate this responsibility to a template base, either `RootEntity_<...>` or `SubtypeEntity_<...>`. For example, instead of the class `Employee` overriding the pure virtual, its template base, `RootEntity_<Employee>`, can supply a concrete implementation. Similarly, `SubtypeEntity_<Manager,Employee>` can do the same for `Manager`. The template base will ensure that an appropriate override is provided for the virtual function.

Using the conventional way, the number of source lines required to override the virtual is small, but they must be repeated for each entity class. In comparison, using template base delegation, there is a constant number of lines for the base template definition, but the number of additional source lines for each addition entity class is equal to 1, that is, the one used to declare the template base.

The real advantage, however, is that the implementation of the overriding virtuals is centralized at the base template definition. If we need to change the implementation, say from declaring the descriptor object as a static data member to declaring a pointer to it, there is only one place to change in the source.

It may be argued that using macros can achieve the same results. However, using macros generally means losing the ability to debug into the macro expansions.

In comparison, the definitions of the entity classes using template base delegation are more readable, because more code is moved into the template bases.

3.3 Uniform Access

Building a "uniform-access" interface using these template bases and descriptor objects is possible. Uniform-access means essentially a general accessor method, `get_related_entities(...)`, which takes a relationship descriptor and returns a list of related entities with the specified relationship. A uniform access interface is especially useful for writing schema-independent client code, for example, a GUI (Graphical User Interface).

4 Template Base Delegation as a Design Tradeoff

Template base delegation can be viewed as a way of writing library classes. Because these library templates need to have the ability to customize themselves, they usually require more experience to write and may be more difficult to maintain. In a way, we are making the life of the client class writer easier by making the life of the base template writer more miserable. Whether this tradeoff makes sense depends on the number of client classes and client class writers versus the number of base templates and base template writers, as well as the relative complexity of the classes.

5 Conclusion

Some of the advantages of template base delegation are as follows:

1. The definition of the derived class can become more declarative.
2. The interface and data members of the base class can be customized for the derived class.
3. The derived class can specify to the template base whom to inherit from, thus preserving the application's inheritance hierarchy.
4. Runtime efficiency can be enhanced without compromising on code sharing or strong typing.
5. The derived class can inherit from two or more instantiations of the same base class template.

The technique was illustrated using the `Fan` templates as an alternative to the traditional `List` container templates. A practical application of the technique in the area of object modeling was also demonstrated.

6 Acknowledgements

Dave Penny and Sam Wong have provided many good insights and suggestions for this paper.

An anonymous referee suggested the use of `((void)sizeof(T))` to ensure that `class T` is defined before any downcast to `T*`. This will catch a common programming error which occurs when an upcast or downcast requires a non-zero pointer adjustment. If the definition of the derived class has not been seen by the compiler, the compiler will erroneously treat the cast as to an unrelated type without generating the required pointer adjustment code.

References

[1] Rebecca Wirfs-Brock and Brian Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. *Proceedings OOPSLA*, SIGPLAN Notices, Vol 24, #10, October 1989.

[2] Bjarne Stroustrup. *The C++ Programming Language: Second Edition.* Addison-Wesley, Reading, Massachusetts, 1991.

[3] P. Chen. "The Entity-Relation Model - Toward a Unified View of Data" ACM Transaction on Database Systems, March 1976.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual* Addison-Wesley, Reading, Massachusetts, 1990.

[5] James Coplien. *Advanced C++ Programming Styles and Idioms* Addison-Wesley, Reading, Massachusetts, 1992.

7 Appendix

Source for the `Fan` templates (some details omitted):

```
//  If a use of the following macro causes the compiler to
//  fail, the definition of class T has not been seen but is
//  required.  Try rearrange your source code so that the
//  definition of class T comes before the use of this macro.
//
#define ASSERT_CLASS_IS_DEFINED(T)      ((void)sizeof(T))

//  If a use of the following macro fails to compile, then
//  H is not inheriting from FanHead_<H,T,R>.
#define ASSERT_HEAD_INHERITS_FROM_FANHEAD(H,T,R)        \
        { FanHead_<H,T,R>*    dummy = (H*)0; }
...
```

```
class FanHeadBase_ {
  public:
    add_tip(FanTipBase_& tip);
    unsigned num_tips() const;
    ...
  protected:
    FanHeadBase_() : first_tip_(0) {}
    FanTipBase_*                        first_tip_;
};

class FanTipBase_ {
  protected:
    FanTipBase_() : head_(0), next_tip_(0) {}
    FanHeadBase_*                       head_;
    FanTipBase_*                        next_tip_;
};

template<class Head, class Tip, class Relationship>
class FanTip_;          // Forward declaration

template<class Head, class Tip, class Relationship>
class FanHead_ : protected FanHeadBase_ {
    friend class FanTip_<Head,Tip,Relationship>;
  public:
    FanHeadBase_::num_tips;
    add_tip(Tip& tip);
    ...
};

template<class Head, class Tip, class Relationship>
class FanTip_ : protected FanTipBase_ {
    friend class FanHead_<Head,Tip,Relationship>;
  public:
    Head* head() const {
        ASSERT_CLASS_IS_DEFINED(Head);
        ASSERT_HEAD_INHERITS_FROM_FANHEAD(Head,Tip,Relationship);
        return (Head*)(FanHead_<Head,Tip,Relationship>*) head_;
    }
    ...
};
```