Implementing One-to-Many Relations in C++

—

Xavier Nodet
xavier.nodet@gmail.com

—

August 2, 2010

Abstract

This paper presents a memory-efficient implementation of One-to-Many relations in C++. Automating relations relives the programmers from tedious and error-prone manual management of back-pointers and notifications when objects are inserted or removed from a relation, or destroyed altogether. This implementation only uses a minimal amount of memory: a container of pointers in the owner, and one pointer in each owned object.

Thanks to the use of the powerful mechanisms of C++ templates, these relations are perfectly type-safe, without the need for down-casting, very flexible, and often as efficient as hand-written code.

1 Introduction

The need to manage relations between objects is at the core of many business applications: objects refer to each other with various semantics, with or without inverse links, for various durations.

A very common case is the One-to-Many relation: a `Owner` object has relationships with several `User` objects. The owner must know which objects it is related to, and the users have an inverse link to their owner. When a user is added to or removed from its owner, both links should be updated. When the owner is destroyed, all the owned objects are also destroyed. Such relations are extremely common, and often implemented with ad-hoc code.

This paper proposes a set of classes to implement those One-to-Many relations with a minimal amount of code to be written in the `Owner` and `User` classes, and performance equivalent to hand-written code, both in terms of memory and CPU.

2 One-to-Many relations

For the purpose of this paper, a One-to-Many relation between classes `Owner` and `User` is defined as the following:

- An `Owner` instance refers to one or more instances of `User`.
- A given instance of `User` can appear at most once in any `Owner`.
- A `User` refers to 0 or 1 instance of `Owner`.
- A `User` refers to an `Owner` if and only if this `Owner` has the address of `User` in its list.

3 Preliminary implementation in C++

The core idea of the proposed implementation is that each relation is represented by two classes `RelationOwner` and `RelationUser`, one for the

`Owner`, and one for the `User`. For this preliminary implementation, an instance of `User` is necessarily owned by one instance of `Owner`. The `RelationOwner` class stores pointers to the instances of `User` which it has a relationship with, and the `RelationUser` class stores a pointer to its owner. The classes `Owner` and `User` inherit from `RelationOwner` and `RelationUser`. This is all that's needed to implement the relation, but programmers may choose to hide the API provided by `RelationOwner` and `RelationUser` through methods of `Owner` and `User` delegating to their base class.

Listing 1: A preliminary implementation of `RelationOwner`

```
template <class Owner,class User>
class RelationUser;

template <class Owner, class User>
class RelationOwner {
  friend class RelationUser<Owner, User>;
public:
  typedef std::set<User*> Container;
  typedef
    typename Container::const_iterator iterator;
public:
  ~RelationOwner();
  iterator begin() const {return _users.begin();}
  iterator end() const {return _users.end();}
private:
  void attach(User* u) {_users.insert(u);}
  void detach(User* u) {_users.erase(u);}
private:
  Container _users;
};

template <class Owner, class User>
RelationOwner<Owner,User>::~RelationOwner() {
  Container temp;
  swap(_users, temp);
  for (iterator it (temp.begin());
       it != temp.end(); ++it) {
    delete *it;
  }
}
```

You will find in listing 1 a preliminary version of the `RelationOwner` class. It defines a container to hold pointers to instances of `User`, and methods to attach a `User` to its `Owner` and detach it, and iterate over the content of the container.

Note the destructor that first swaps the `_owned` container with an empty one before deleting all the objects that it owns. Iterating on a container that's not the one that stores all the pointers to the users is necessary because, when deleted, the users will try to remove themselves from this container. Copying would be less efficient in itself, and erasing from an empty container is quicker...

Listing 2: A preliminary implementation of `RelationUser`

```
template <class Owner, class User>
class RelationUser {
public:
  explicit RelationUser(Owner* o)
    : _owner(o)
  { _owner->attach(user()); }
  ~RelationUser() { _owner->detach(user()); }
private:
  User* user() {return static_cast<User*>(this);}
private:
  Owner* _owner;
};
```

The `RelationUser` class is presented in listing 2. The interesting thing to notice is the implementation of the `user()` method, that returns the address of the `User` object. It uses the Curiously Recurring Template Pattern and this was suggested to me on Stack Overflow (see [Fa09]). The CRTP idiom *occurs when the base class* `RelationUser` *is templated on the type of its derived class* `User` (see [CRTP]). Indeed, we need the `user()` method to return the address of the `User` instance, not of its `RelationUser` base class. See section 7.3 for a complete discussion on this topic.

4 A simple example

Here is now an example of use:

Listing 3: Example of use

```
class User;
class Owner;

class Named {
public:
  Named(const std::string& name)
    : _name(name) {}
  const std::string& name() const {
    return _name;
```

```
  }
  ~Named() {
    std::cout << "Object '" << _name
              << "' destroyed" << std::endl;
  }
private:
  std::string _name;
};

class Owner
  : public RelationOwner<Owner, User> {
public:
  void show();
};

class User
  : public RelationUser<Owner, User>
  , public Named {
public:
  User(Owner* owner, const std::string& name)
    : RelationUser<Owner,User>(owner)
    , Named(name)
  {}
};

void Owner::show() {
  std::cout << "Users of Owner: ";
  for (Owner::iterator it (begin());
       it != end(); ++it) {
    std::cout << (*it)->name() << " ";
  }
  std::cout << std::endl;
}


void test() {
  Owner* owner (new Owner());
  User* user1 (new User(owner, "1"));
  User* user2 (new User(owner, "2"));
  User* user3 (new User(owner, "3"));
  owner->show();

  delete user2;
  owner->show();

  delete owner; // All users deleted
}
```

And here is the printed output:

```
Users of Owner: 1 2 3
User '2' destroyed
Users of Owner: 1 3
User '1' destroyed
User '3' destroyed
```

The net result is that just by inheriting from those classes, users get automatic management of ownership.

5 Adding functionalities

5.1 Several relations to the same class

The most pressing issue with the code above is that it is not possible to add two or more relations from one class to another... Suppose the users wants to implement a link between two objects, with a 'preceding' and 'following' node for each link. To define the `Node` class that should hold the `Links`, one would need to inherit twice from the `RelationOwner<Node,Link>`, but this is clearly impossible.

The solution is to add a *type marker* template parameter to the `RelationOwner` template class, e.g. an empty struct. Once two different types `Preceding` and `Following` have been defined, the `Node` can inherit from both `RelationOwner<Node,Link,Preceding>` and `RelationOwner<Node,Link,Following>`, which are different classes from the compiler's point of view, even if the rest of their definition is exactly the same. Providing a default value allows programmers that don't need this feature to simply ignore it.

Listing 4: Integer template parameters

```
template <class Owner, class User,
          class RelId = void>
class RelationOwner {
    ...
};

template <class Owner, class User,
          class RelId = void>
class RelationUser {
  typedef typename
    RelationOwner<Owner,User,RelId> RelOwner;
public:
  RelationUser(Owner* owner)
    : _owner(owner)
  {
    _owner->RelOwner::attach(user());
  }
```

```
  ~RelationUser() {
    owner()->RelOwner::detach(user());
  }
   ...
};

struct Preceding {};
struct Following {};

class Owner
  : public RelationOwner<Owner,User,Preceding>
  , public RelationOwner<Owner,User,Following> {
  ...
};
```

Such a template parameter is also added to `RelationUser` so that each `RelationUser` instantiation can refer to its `RelationOwner` counterpart: now that the owner may have several `RelationOwner` base classes, it has several `attach()` and `detach()` methods. This implies that `RelationUser` must be able to access the correct method: the one that corresponds to the other side of the relation, but not another. Calls to those methods must thus be qualified, which implies to know the exact type of `RelationOwner`.

5.2 Flexibility using policies

Another issue is that storing the users in an `std::set`, which is the default, is not necessarily always the best choice. It may be the case that the same user should appear several times in the same relation. Or that, on the contrary, it can never be the case that a user is inserted twice, and an `std::vector` would be a better choice.

As defined by Andrei Alexandrescu in [Al01], *a* policy *defines a class interface or a class template interface. [They] have much in common with traits, but differ in that they put less emphasis on types and more emphasis on behavior.*

5.2.1 Policy for container of users

A policy can define the type of the container to use in the `RelationOwner` template, and methods to insert and remove objects from this container:

Listing 5: Policies for container of users

```
template <class L>
struct SetValuesPolicy {
  typedef std::set<L*> Container;
  static void insert(Container& c, L* l) {
    c.insert(l);
  }
  static void remove(Container& c, L* l) {
    c.erase(l);
  }
};

template <class L>
struct VectorNoCheckValuesPolicy {
  typedef std::vector<L*> Container;
  static void insert(Container& c, L* l) {
    c.push_back(l);
  }
  static void remove(Container& c, L* l) {
    c.erase(std::remove(c.begin(), c.end(), l),
            c.end());
  }
};

template <class Owner, class User,
          class RelId = void,
          class Policy = SetValuesPolicy<User> >
class RelationOwner {
public:
  typedef typename Policy::Container Container;
public:
  void attach(User* owned) {
    Policy::insert(_users,owned);
  }
  void detach(User* owned) {
    Policy::remove(_users,owned);
  }
  ...
private:
  typename Policy::Container _users;
};
```

The `Policy` template parameter is, by default, the `SetValuesPolicy<User>` class. This class defines a `Container` type, and methods `insert()` and `remove()`. Another class can define those four members so that an `std::vector` is used, as in `VectorNoCheckValuesPolicy`, or any other data-structure.

5.2.2 Policy for deletion behavior

If a `User` instance is not necessarily tied to a `Owner` instance but can live on its own, another policy can be defined to determine the behavior of the `RelationOwner` when it is destroyed: should it also destroy all the users? Should it do something else?

Listing 6: Policy for deletion behavior

```
template <class L> struct DefaultEnder {
  void operator()(L* l) { delete l; }
};

template <class Owner, class User,
          class RelId = void,
          class Policy = SetValuesPolicy<User>,
          class UserEnder = DefaultEnder<User> >
class RelationUser {
public:
  void reset() {
    if (_owner) {
      _owner->detach(user());
      _owner = 0;
    }
  }
  ...
private:
  User* user() {return static_cast<User*>(this);}
  RelOwner* _owner;
};

template <...>
class RelationOwner {
  ...
public:
  ~RelationOwner() {
    Container temp;
    swap(_users, temp);
    for (iterator it (temp.begin());
         it != temp.end(); ++it) {
      static_cast<RelUser*>(*it)->reset();
      UserEnder()(*it);
    }
  }
  ...
private:
  typename Policy::Container _users;
};
```

By default, destruction of a `RelationOwner` instance triggers the destruction of all the users in this relation. But providing another `operator()` allows to choose another behavior. Note that the user is always 'detached': as the owner is being destroyed, one does not want to have pointers to it any more... A `static_cast` is needed for this call, so that the call to `detach()` is not ambiguous when `User` implements several relations..

5.3 Array-like API for access to users

It is very easy to provide an array-like API on `Owner`.

Listing 7: Array-like API

```
template <class L>
struct VectorValuesPolicy {
        ...
  static L* elem(const Container& c, size_t i) {
    return c[i];
  }
  static size_t nbElem(const Container& c) {
    return c.size();
  }
};

template <class RelationUser>
class RelationOwner {
  ...
  User* user(int i) const {
    return Policy::elem(_users,i);
  }
  size_t nbUsers() const {
    return Policy::nbElem(_users);
  }
  ...
};
```

What's interesting is the `Policy` class does not need to define those `elem` and `nbElem` methods as long as methods `user(size_t)` and `nbUsers()` on `RelationOwner` are not used. Those will be compiled only if used somewhere in the code, and this compilation will only succeed if `Policy` has the correct API. Users of set-based relations (for which the policy does not provide the API) will thus not be able to access elements with the array-like API, and won't have to wait for run-time before discovering this...

6 Example of use

Listing 8: Usage of the `RelationOwner` and `RelationUser` classes

```
#include "relations.h"

using namespace std;
```

```
class Named {
public:
  Named(const string& name)
    : _name(name)
  {}
  const string& name() const { return _name; }
  ~Named() {
    cout << "Object '" << _name
         << "' destroyed" << endl;
  }
private:
  string _name;
};

using namespace relations;


struct Preceding {};
struct Following {};

class User;
class Owner;

typedef RelationUser<Owner,User,Preceding,
  VectorNoCheckValuesPolicy<User> >
  PrecedingUser;
typedef RelationOwner<PrecedingUser>
  PrecedingOwner;

typedef RelationUser<Owner,User,Following>
  FollowingUser;
typedef RelationOwner<FollowingUser>
  FollowingOwner;

class Owner
  : public PrecedingOwner
  , public FollowingOwner
  , public Named {
public:
  typedef PrecedingOwner Preceding;
  typedef FollowingOwner Following;
  Owner(const string& name) : Named(name) {}
};

class User
  : public PrecedingUser
  , public FollowingUser
  , public Named {
public:
  typedef PrecedingUser Preceding;
  typedef FollowingUser Following;
  User(Owner* owner1,
       Owner* owner2,
       const string& name)
    : Preceding(owner1)
    , Following(owner2)
    , Named(name)
  {}
};

template <class Rel>
void printRel(typename Rel::_Owner* owner,
              const string& relName) {
  if (owner->Rel::begin() == owner->Rel::end()) {
    cout << owner->name()
         << " is not related as "
         << relName << " to any item";
  } else {
    cout << owner->name()
         << " is related as "
         << relName << " to ";
    for (Rel::iterator it (owner->Rel::begin());
         it != owner->Rel::end(); ++it) {
      cout << "'" << (*it)->name() << "' ";
    }
  }
  cout << endl;
}

void test() {
  Owner* owner1 (new Owner("owner1"));
  Owner* owner2 (new Owner("owner2"));
  User* user1 (new User(owner1, owner2, "user1"));
  User* user2 (new User(0, owner2, "user2"));
  printRel<Owner::Preceding>(owner1, "Preceding");
  printRel<Owner::Following>(owner1, "Following");
  printRel<Owner::Preceding>(owner2, "Preceding");
  printRel<Owner::Following>(owner2, "Following");

  cout << "Deleting "
       << owner1->name() << endl;
  delete owner1;
  printRel<Owner::Preceding>(owner2, "Preceding");
  printRel<Owner::Following>(owner2, "Following");
}
```

The output for this example is:

```
owner1 is related as Preceding to 'user1'
owner1 is not related as Following to any item
owner2 is not related as Preceding to any item
owner2 is related as Following to 'user1' 'user2'
Deleting owner1
owner2 is not related as Preceding to any item
owner2 is related as Following to 'user2'
```

7 Other approaches

7.1 The traditional approach

The usual way to manage those links is to have each object hold a pointer (or a container of pointers) to the object(s) that it may have a relation with. E.g. in a Belongs-To relation between Plant and ProductionLine classes, each Plant holds a list (or array, or any other container) of pointers to its ProductionLine instances, while the ProductionLine instances have a pointer to their Plant.

Each time a relation is implemented this way, some code must be added in both classes to manage this relation: setting the back-pointer when a new object is being referred to, resetting it when the object is no longer referred to, making sure that the pointer is removed from the container if the object referred to gets destroyed, and notifying or even destroying the objects referred to when the containing object gets destroyed. All this code gets repeated for each relation, which is bug-prone and decreases the cohesiveness of the classes. In particular, the destructor of `Owner` needs to be updated (so that it notifies the owned objects) each time a relation is added, which is a change that's very easy to forget.

7.2 Code generation

Generating the code to handle the relations from a description of the BOM is a possible solution. Compared to the traditional approach, there is much less risk of mistakes. But it requires that this description is available, and external tools generate the code.

7.3 Smart pointers

A smart pointer class can be devised with the goal of being used in the `User` class to refer to the `Owner` instance. Destruction of the smart pointer (when the `User` instance is destroyed) would trigger the notification of the `Owner` so that it removes the address of the `User` from its list. The main drawback of this approach is that there is no way for the smart pointer instance to know the address of the `User` instance it belongs to, other than getting it at construction time and storing it.

The proposed approach solves this problem efficiently. Recall the code to retrieve the address of the `User` instance from inside the `RelationUser` class:

Listing 9: Retrieving the address of the containing object using CRTP

```
template <class, class User>
class RelationUser {
        ...
  User* user() {return static_cast<User*>(this);}
};
```

The reasons this code works are the following:

- The compiler provides the `this` pointer, which is not of type `User*`, but of type `RelationUser<...>*`, as we're in a method of the latter.
- It is only at the point of instantiation (i.e. when the compiler actually encounters a call to `RelationUser<...>::user()`) that the compiler inserts the code of the method.
- The method is called in the destructor of `RelationUser` to notify the `Owner` of the destruction of an object it owns. This destructor is instantiated from the destructor of `User`. The `User` class is obviously known to the compiler at this point.
- As `User` is a template parameter of `RelationUser` the former is known to the compiler when instantiating the code for `RelationUser<...>::user()`. The compiler can thus generate the code to convert a pointer to the base class `RelationUser` into a pointer to the derived class `User`. Et voilà...

The fact that the compiler provides the `this` pointer, plus the fact that `User` inherits from `RelationUser` (something the compiler knows as `User` is a template parameter of `RelationUser`), allows to retrieve the address of the `User` object without incurring the cost of storing it. In terms of CPU, the implementation of the cast is a no-op if the `RelationUser` class is the first super-class of the `User` and simply the substraction of a constant in the other cases. Pretty cheap...

Now suppose that the `User` class stores, as a member, an instance of a smart pointer. There is

no way[1] this smart pointer would be able to retrieve the address of `User` without getting it from `User` itself and storing it. The inheritance solution proposed in this paper is thus more efficient in terms of memory.

7.4 Relationship Manager

One proposed solution to these issues, as in [Bu01], is to have all relations managed and stored into a 'central' `RelationhipManager` class. This approach increases the cohesiveness of the classes in the BOM, but does so at the expense of the additional memory needed to store the information: the `Owner` and `User` classes store a pointer to the relationship manager (instead of a pointer to the other object in the relation), but the latter must also use additional memory to hold all the information about which object has a relation with which. This approach is also less efficient in terms of CPU.

8 Conclusion

This paper shows an efficient implementation of One-to-Many relations in C++. Thanks to the template mechanisms, this implementation is type-safe, flexible and often as efficient as hand-written code.

9 Acknowledgements

Thanks to Kevlin Henney and Georges Schumacher for their encouragements and very useful comments...

References

[Al01] Andrei Alexandrescu. Modern C++ Design (Addison-Wesley, 2001)

[Bu01] Andy Bulka, Relationship Manager, http://www.andypatterns.com/index.php?cID=44

[CRTP] The Curiously Recurring Template Pattern: http://c2.com/cgi/wiki?CuriouslyRecurringTemplate

[Fa09] http://stackoverflow.com/questions/709790/how-can-i-know-the-address-of-owner-object-in-c/709996#709996

[1] Except if allowing non-portable code using undefined behavior

A Complete listing

```
#include <algorithm>
#include <iostream>
#include <vector>

namespace relations {

// The default Policy class used to determine what
// happens to the owned objects
// when the owner is destroyed.
struct DefaultEnder {
  template <class L>
  void operator()(L* l) { delete l; }
};

// A Policy class to be used to determine what
// happens to the owned objects when the owner is
// destroyed.  This one does nothing. Note that
// the back-pointer is set to 0 by the owner prior
// to the call to operator()
struct DoNothing {
  template <class L>
  void operator()(L*) {}
};

// A Policy class to specify the container to use
// in the owner: std::set
template <class L>
struct SetValuesPolicy {
  typedef std::set<L*> Container;
  static void insert(Container& c, L* l) {
    c.insert(l);
  }
  static void remove(Container& c, L* l) {
    c.erase(l);
  }
  static bool has(Container& c, L* l) {
    return c.find(l) != c.end();
  }
};

// Base class for Policy classes to specify
// std::vector to use in the owner.
template <class L>
struct VectorValuesPolicyBase {
  typedef std::vector<L*> Container;
  static void remove(Container& c, L* l) {
    c.erase(std::remove(c.begin(), c.end(), l),
            c.end());
  }
  static bool has(Container& c, L* l) {
    find(c.begin(), c.end(), l) != c.end();
  }
  // elem and nbElem should probably only be
  // provided for containers that implement those
  // operations in O(1)
  static L* elem(const Container& c, size_t i) {
    return c[i];
  }
  static size_t nbElem(const Container& c) {
    return c.size();
  }
};

// A Policy class to specify the container to use
// in the owner: an std::vector where insertion is
// checked to avoid inserting again the same object
template <class L>
struct VectorValuesPolicy
  : public VectorValuesPolicyBase<L>
{
  static void insert(Container& c, L* l) {
    Container::iterator it =
      find(c.begin(), c.end(), l);
    if (it == c.end()) { c.push_back(l); }
  }
};

// A Policy class to specify the container to use
// in the owner: an std::vector with no checking
// on insertion
template <class L>
struct VectorNoCheckValuesPolicy
  : public VectorValuesPolicyBase<L>
{
  static void insert(Container& c, L* l) {
    c.push_back(l);
  }
};


// Inherit from RelationOwner to give to your
// class the possibility to own objects in a
// one-to-many relation.
template <class RelUser>
class RelationOwner {
  friend RelUser;
public:
  typedef typename
    RelUser::_Owner          _Owner;
  typedef typename
    RelUser::_User           _User;
  typedef typename
    RelUser::_Policy         _Policy;
  typedef typename
    RelUser::_UserEnder      _UserEnder;
  typedef typename
```

```
    _Policy::Container          _Container;
  typedef typename
    _Container::const_iterator iterator;

  // Destructor of owner notifies all the owned
  // objects and then calls the user-provided
  // Policy class.
  ~RelationOwner() {
    _Container temp;
    swap(_users, temp);
    for (iterator it (temp.begin());
         it != temp.end(); ++it) {
      static_cast<RelUser*>(*it)->reset();
      _UserEnder()(*it);
    }
  }
  // All the remaining methods will typically be
  // wrapped into methods of the derived Owner
  // class to hide the implementation of the
  // relation
  bool has(_User* owned) const {
    return _Policy::has(_users,owned);
  }
  iterator begin()       const {
    return _users.begin();
  }
  iterator end()         const {
    return _users.end();
  }
  _User* user(int i)     const {
    return _Policy::elem(_users,i);
  }
  size_t nbUsers()       const {
    return _Policy::nbElem(_users);
  }

private: // Changes are made through the users
  void attach(_User* owned) {
    _Policy::insert(_users,owned);
  }
  void detach(_User* owned) {
    _Policy::remove(_users,owned);
  }

private:
  _Container _users;
};

// Base class for the objects owned in the
// relation:
// - Owner and User inherit from RelationOwner
//   and RelationUser.
// - The RelId marker type allows to distinguish
//   relations that would otherwise have the same
//   set of template parameters.
// - Policy allow to define the container used
//   in RelationOwner
// - UserEnder defines what happens to the owned
//   objects when the owner instance is destroyed
//
template <class Owner, class User,
          class RelId = void,
          class Policy = SetValuesPolicy<User>,
          class UserEnder = DefaultEnder >
class RelationUser {
  typedef Owner    _Owner;
  typedef User     _User;
  typedef Policy   _Policy;
  typedef UserEnder _UserEnder;
  typedef RelId    _RelationId;

  typedef RelationUser<Owner,User,RelId,
    Policy,UserEnder> _RelUser;
  typedef RelationOwner<_RelUser> _RelOwner;
  // Give access to the typedefs above
  friend _RelOwner;

public:
  explicit RelationUser(_Owner* owner)
    : _owner(0)
  { reset(owner); }
  ~RelationUser() { detach(); }
  _Owner* owner() const { return _owner; }
  bool hasOwner() const { return _owner != 0; }
  // Change owner
  void reset(_Owner* newOwner=0) {
    detach();
    _owner = newOwner;
    if (_owner) {
      static_cast<_RelOwner*>(_owner)
        ->attach(user());
    }
  }
private:
  _User* user() {
    return static_cast<_User*>(this);
  }
  void detach() {
    if (hasOwner()) {
      static_cast<_RelOwner*>(_owner)
        ->detach(user());
    }
  }
private:
  _Owner* _owner;
};

} // namespace
```