Adapters and Binders - Overcoming problems in the design and implementation of the C++-STL

Volker Simonis¶
Universität Tübingen

February 28, 2000

Abstract

The Standard Template Library [Lee-Step], [SGI-STL], had the reputation of being complex and hard to use. Because of its exhaustive use of the new C++ language features, most compilers weren't able to fully support this library. Nowadays, new compilers and the standardisation of C++ [ANSI-CPP] not only enable us to use the STL in its full strength, but also show us the drawbacks of a library which apparently was designed from scratch without the possibility of testing and using it. This paper will focus on the syntax and semantics of STL Adaptors and Binders. It will reveal some shortcomings in their C++ implementation and suggest solutions for the encountered problems.
Keywords: C++, STL, adapters, binders, template specialisation

1 Introduction

The STL is a new kind of library for many C++ users. Rather than being another class hierarchy of derived classes like so many other libraries which exploit mainly the object oriented features of the language, the STL sets its main focus on genericity. As a consequence, the main elements of the library, namely containers and algorithms, are coupled together by iterators. There are no sub-class relationships between the components. They can only work together by implementing a common interface. This is the characteristic of generic programming. Algorithms are designed for a special domain for which a set of requirements have to hold. Once the algorithms are implemented, they can be used with arbitrary input domains, as long as they fulfill the requirements.

The opportunity of implementing such kinds of generic libraries in C++ was a result of the introduction of template mechanisms into the language. Unfortunately, the intensive use of templates pushes even the newest C++-compilers to their limits. Especially error messages, which are inevitable during every development process, appear to be unreadable even to experienced C++ programmers, if they occur in multiply nested template instantiations. This may be one reason, why few big projects in the software industry are build on top of the STL until now. Another reason can be the fact that this library didn't really grew together with the language and the requirements of its users. Instead it was designed "from scratch", even before the language was ready to compile it[1].

2 Adapters

In the STL, adapters are used to build function objects out of ordinary functions or class methods. In fact, most of the predefined STL algorithms don't care about their argument being a function or a function object. However some standard algorithms, like for example `bind()` which we will discuss in more detail in the next section, rely on the fact that their argument is a function object. In this case a "pointer to function adapter" can be used to transform a function pointer into an function object.

Since algorithms invoke their function argument by simply calling the arguments function operator, it would not be possible for algorithms to call a member function for all their operands. In this case, a member function adapter can be

¶Fakultät für Informatik, Arbeitsbereich Computer Algebra, Sand 13, 72076 Tübingen, Germany, e-mail: simonis@informatik.uni-tuebingen.de

[1] It must be stated here that the STL is in fact older then C++ since it evolved from the ADA Generic Library [AGL]. The Ada Generic library in turn based on the ideas of its authors on generic programming which go back to the early '80th ([Kapur], [Musser]). In contrast to this, the C++ implementation of the STL was incorporated into the C++ standard at a relatively late stage of the standardization process.

used which transforms a member function into a general function object. Let's have a look at an example adapted from [Strou97](18.4.4.2):

```
void draw_all(list<Shape> &ls) {
  for_each(ls.begin(), ls.end(), mem_fun_ref(&Shape::draw));
}
```

Provided that `Shape` is a class which provides a `draw` method, the function `draw_all` will call `draw` for every `Shape` object inside its `list` argument. The adapter `mem_fun_ref` which is defined in `<functional>` as follows [ANSI-CPP](20.3.8#7):

```
template<class R, class T>
mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)());
```

transforms its argument, a pointer to a member function without arguments, into a function object of type `mem_fun_ref_t`. Note however that `mem_fun_ref_t` is a unary function and its function operator takes one argument, namely the object for which the method will be called. The definition of `mem_fun_ref_t` is [ANSI-CPP](20.3.8#5):

```
template<class R, class T>
struct mem_fun_ref_t : public unary_function<T, R> {
  explicit mem_fun_ref_t(R (T::*pm)());
  R operator() (T &x) const;
};
```

Because the called method will not be `const` in general, the object argument of the function operator cannot be `const` too. For `const` methods, an overloaded version of `mem_fun_ref` ([ANSI-CPP](20.3.8#13)) is available, which returns a function object of type `const_mem_fun_ref_t` ([ANSI-CPP](20.3.8#11)).

```
template<class R, class T>
struct const_mem_fun_ref_t : public unary_function<T, R> {
  explicit mem_fun_ref_t(R (T::*pm)() const);
  R operator() (const T &x) const;
};
template<class R, class T>
const_mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)() const);
```

In this case, the argument of the function operator can be constant, as it is common sense and good programming style in C++. All this details seem to be not very important, but they will turn out be crucial if adapters are to be combined with binders as we will see in the next section.

For now lets have a look at the remaining overloaded versions of the `mem_fun_ref` function. They take a pointer to a member function with one argument or a pointer to a constant member function with one argument respectively and return binary function objects ([ANSI-CPP](20.3.8#7,20.3.8#13)).

```
template<class R, class T, class A>
mem_fun1_ref_t<R, T, A> mem_fun_ref(R (T::*pm)(A));
template<class R, class T, class A>
const_mem_fun1_ref_t<R, T, A> mem_fun_ref(R (T::*pm)(A) const);
```

As in the previous case notice the fact that unary functions are transformed into binary function objects which await as additional argument the object for which the method will be invoked. Only the non-const version of `mem_fun1_ref_t` will be shown here ([ANSI-CPP](20.3.8#6)):

```
template<class R, class T, class A>
struct mem_fun1_ref_t : public binary_function<T, A, R> {
  explicit mem_fun_ref_t(R (T::*pm)(A));
  R operator() (T &x, A arg) const;
};
```

An additional difficulty with the member function adapters is the unintuitive pointer to member syntax. If the class `Shape` from the above example would define a second `draw` method which takes an `int` argument, we could not write

```
for_each(ls.begin(), ls.end(), mem_fun_ref(&Shape::draw));
```

anymore. Instead we had to explicitly qualify the method name with the desired type:

```
for_each(ls.begin(), ls.end(), mem_fun_ref((void (Shape::*)(void))&Shape::draw));
```

In this case a `typedef` could be a good alternative to simplify the expression.
Of course operators also can be used as arguments for member function adapters. Consider the following example:

```
struct Int {
  int _i;
  Int(int i) : _i(i) {}
  Int& operator++ ()    { ++_i; return *this; }            // prefix increment
  Int  operator++ (int) { Int i(*this); ++_i; return i; }  // postfix increment
};
void increment_all(vector<Int> &vi) {
  for_each(vi.begin(), vi.end(), mem_fun_ref((Int (Int::*)())&Int::operator++));
}
```

It uses the prefix increment oprator. But what if we absolutely want to use the postfix operator (though it is no difference in this context of course)? Let's try:

```
void increment_all_(vector<Int> &vi) {
  for_each(vi.begin(), vi.end(), mem_fun_ref((Int (Int::*)(int))&Int::operator++));
}
```

A compiler error will occur, reporting something like: "No match for call to `(mem_fun1_ref_t<Int, Int,int>) (Int&)`". Not quit what we expected but rather one of the confusing error messages mentioned above. What does it mean? We clearly adviced the compiler to take the postfix increment operator and this operator is a unary function, no matter if it's argument is used or not. We could even apply a default argument, it would be no difference, since it would not change the signature of the method. Consequently the `mem_fun_ref` version for unary methods is called which creates a binary function object of type `mem_fun1_ref_t`. The function operator of this object of course is binary too with a signature of type `Int (::*)(Int&, int)`. Therefor the call to a binary function demanded by the `for_each()` function cannot be resolved, resulting in the mentioned compiler error.

So in fact, after rethinking the situation we must admit that the compiler is not wrong, nor is this a problem of the STL. Rather this is a C++ problem because C++ allows us to hide the real signature and thus the type of functions. This can be done either by supplying them with default arguments or by convention, like this is the case with the postfix increment operator. A library based on algebraic rules like the STL, merciless unhides such inaccuracies in the language definition. But the situation is not hopeless. After having understood the problem we can fix it with the help of another construct from the STL - namely binders.

3 Binders

Binders can be used to bind one argument of a binary function object to a fixed value thus transforming it into an unary function. Therewith we could write the broken version of `increment_all` in the following way:

```
void increment_all(vector<Int> &vi) {
  for_each(vi.begin(), vi.end(), bind2nd(mem_fun_ref((Int (Int::*)(int))&Int::operator++), 999));
}
```

By binding the integer argument of `mem_fun1_ref_t`'s function operator (its 2nd argument) to 999, we get a unary function object, exactly as expected by `for_each()`. Notice that in this case it is unimportant to which value the argument is bound, since the call to `mem_fun1_ref_t`'s function operator will be resolved into a call to `Int`'s postfix increment operator which ignores its argument anyway.

This looks quit satisfying, unfortunately it will depend solely on your compiler if this will compile and run. In any case, according to the C++ standard it should definitely not. To understand why, we must take a closer look on `bind2nd()`'s implementation ([ANSI-CPP](20.3.6.4)):

```
template <class Operation, class Arg2>
inline binder2nd<Operation> bind2nd(const Operation& op, const Arg2& x) {
  return binder2nd<Operation>(op, typename Operation::second_argument_type(x));
}
```

bind2nd() expects it's argument to be a binary function object derived from binary_function which is declared as well as unary_function in the standard header <functional>. These structs only define the types of their arguments and their result type for later use. For your convenience, here the definition of binary_function ([ANSI-CPP](20.3.1)):

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

Notice how bind2nd() uses this type information to cast the argument of type Arg2 into the type of the function objects second argument. This is crucial in order to call the constructor of binder2nd if the types don't match exactly and there's no standard conversion available.

As you would expect, binder2nd, the result returned by the call to bind2nd(), is a unary function object derived from unary_function. It has the following definition ([ANSI-CPP](20.3.6.3)):

```
template <class Operation>
class binder2nd : public unary_function<typename Operation::first_argument_type,
                                        typename Operation::result_type> {
protected:
  Operation op;
  typename Operation::second_argument_type value;
public:
  binder2nd(const Operation& x, const typename Operation::second_argument_type& y)
      : op(x), value(y) {}
  typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const { return op(x, value); }
};
```

It stores the binary function object and the second argument. The function operator is defined in such a way that it takes only one argument. It then calls the function operator of the stored function object with its argument and the stored second argument which is fixed for the life time of the binder2nd object.

The problem that arises now consists in the fact that the function operator of binder2nd declares its argument with a const modifier, thus assuring not to alter its value. This is not unusual and considered to be good programming style in C++. But it is fatal in the case where bind2nd() is applied to the result of mem_fun_ref(), because mem_fun_ref() creates a binary function of which the first argument is an object for which a certain method has to be called. Of course it is impossible to assume constness for this object unless the method which will be called is declared to be constant. In general, this will not hold. On the contrary, for this case the STL defines two special structs const_mem_fun_ref_t and const_mem_fun1_ref_t. The only difference between them and the corresponding non-const structs mem_fun_ref_t and mem_fun1_ref_t is the fact that the argument taken by the function operator is declared to be const in the former ones (see section 2, on page 2).

But in the general case, binder2nd will store a mem_fun1_ref_t as its operation parameter and it will try to call mem_fun1_ref_t's function operator with a const modified object as first argument. This call will result in compiler error because it discards the constness of the object passed as the first argument to mem_fun1_ref_t's function operator.

No general solution can be given for this problem. One possibility would be to relax the postconditions of binder2nd's function operator by removing the const modifier from its function argument. This would work fine for our example but it results in a STL not compatible with the standard anymore. Even worse, it could break existing code which relies on the argument of binder2nd's function operator to be constant.

So the only viable way is the specialization of binder2nd for a template argument of type mem_fun1_ref_t. Notice that this explicit specialization has to be declared before it's first use ([ANSI-CPP](14.7.3)). The specialization will look as follows:

```
template <class A, class B, class C>
class binder2nd<mem_fun1_ref_t<A,B,C> >
  : public unary_function<typename mem_fun1_ref_t<A,B,C>::first_argument_type,
                          typename mem_fun1_ref_t<A,B,C>::result_type> {
protected:
  mem_fun1_ref_t<A,B,C> op;
  typename mem_fun1_ref_t<A,B,C>::second_argument_type value;
public:
  binder2nd(const mem_fun1_ref_t<A,B,C>& x,
            const typename mem_fun1_ref_t<A,B,C>::second_argument_type& y)
      : op(x), value(y) {}
  typename mem_fun1_ref_t<A,B,C>::result_type
  operator()(typename mem_fun1_ref_t<A,B,C>::first_argument_type& x) const {
    return op(x, value);
  }
};
```

The only difference with respect to the general version is the missing `const` modifier for the function operators argument. With the declaration of `mem_fun1_ref_t` in mind, this can slightly be simplified into:

```
template <class A, class B, class C>
class binder2nd<mem_fun1_ref_t<A,B,C> > : public unary_function<B,A> {
protected:
  mem_fun1_ref_t<A,B,C> op;
  C value;
public:
  binder2nd(const mem_fun1_ref_t<A,B,C>& x, const C& y) : op(x), value(y) {}
  A operator()(B& x) const {
    return op(x, value);
  }
};
```

With this explicitly specialized version of `binder2nd` our example from the previous section will compile and run without errors. Hopefully, it will be incorporated into a future version of the STL. It is also interesting to mention Stroustrups book ([Strou97]) here. In section 18.4.4, in which he introduces binders, adapters and negators, he presents the following example:

```
void rotate_all(list<Shape*>& ls, int angle) {
  for_each(ls.begin(), ls.end(), bind2nd(mem_fun(&Shape::rotate), angle));
}
```

If we assume `rotate` to be a non-const method of `Shape` as defined in section 2.6 of his book, this example will not work as well, because of the reasons mentioned above. Notice that the only difference between `mem_fun()` and `mem_fun_ref()` is that `mem_fun()` creates a function object which expects an object pointer as argument whereas the function object created by `mem_fun_ref()` expects a reference to an object. To fix the problem in Stroustrups book, `binder2nd` has to be specialized for `mem_fun1_t` in the same way it was specialized for `mem_fun1_ref_t` whereby `mem_fun1_t` is the type of the object returned by `mem_fun()`.

3.1 Binders and references

Now that the problems which arose through the interaction of adapters and binders seemed to be solved, let's somewhat extend the example from section 2. We refine the class `Int` as follows:

```
struct Int {
  int _i;
  Int(int i) : _i(i) {}
  Int& operator++ ()    { ++_i; return *this; }            // prefix increment
```

```
  Int  operator++ (int) { Int i(*this); ++_i; return i; }  // postfix increment
  Int& add(Int i)       { _i+= i._i; return *this; }
};
```

with the only change being the additional method `add()` which adds its given argument to the actual `Int` object. Given this new definition, we can define a new function `increment_all_by()`:

```
void increment_all_by(vector<Int> &vi, Int i) {
  for_each(vi.begin(), vi.end(), bind2nd(mem_fun_ref((Int& (Int::*)(Int))&Int::add), i));
}
```

Due to our changes from the last section, this will work as expected. But if we decide to change the definition of `add()` into:

```
Int& add(Int& i)       { _i+= i._i; return *this; }
```

by converting its argument into a reference type, we will face new problems. Note however that this is common use and considered to be good programming style in C++, especially for big objects. So what's wrong? Trying to compile the function `increment_all_by()` will result in compiler error similar to: "reference to reference is not allowed".

First of all, a version of `mem_fun_ref()` is called, which is parametrized by a pointer to member of type "`Int& (Int::*)(Int&)`". It creates a function object of type `mem_fun1_ref_t< Int&, Int, Int&>`. `mem_fun1_ref_t` is derived from `binary_function` whereby the base class `binary_function` is used only in order to declare the types of the function arguments in a common way (see section 3 on page 4). In this case `binary_function` will be parametrized with "`<Int, Int&, Int&>`" resulting in `first_argument_type` being equal to `Int`, `second_argument_type` being equal to `Int&` and the `result_type` being equal to `Int&`.

`bind2nd()` in turn takes this `mem_fun1_ref_t` object as template parameter for the `binder2nd` object it will create. Unfortunately, the constructor of `binder2nd` takes as its second argument a reference of type `Operation::second_argument_type` where `Operation` is the template argument of `bind2nd()`, in this case `mem_fun1_ref_t<Int&, Int, Int&>`. But the second argument of `Operation` already has the type `Int&`. During the instantiation of `binder2nd` the compiler tries to get the reference of a reference to a `Int&` object which leads to the compiler error presented above.

Again this is no STL problem, but a problem of the C++ reference mechanism. While it is possible to take the address of an address in C++ and to apply the dereference operator as many times as necessary, this is not possible for references. As the compiler complained, references of references are not possible in C++. Again, there are two solutions for the problem. The first one is not to use references at all, but pass objects by value or use pointer to objects for big objects. This may be unacceptable under certain circumstances or simply impossible if the classes we use are encapsulated in third party libraries.

The remaining solution is again the specialization of classes already present in the STL, to fit our needs. In this case we need a special version of the class `binary_function`, which hides the reference type of it's second argument:

```
template <class Arg1, class Arg2, class Result>
struct binary_function<Arg1, Arg2&, Result> {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

With this definition of `binary_function` the type of the function objects second argument will be properly propagated to `binder2nd`, where a reference of it can be taken. Note that in other cases, it can be useful to have a specialization which hides the reference type of the first argument or from both arguments. So in fact a set of three specializations may be necessary to meet all requirements.

A third solution would be possible here as well, but it would require a change of the STL. The constructor of `binder2nd` may be changed to take its second argument by value and not as a reference.

Once again it is interesting here to have a look into Stroustrups book. In section 18.4.4.3 he brings the following example for pointer to function adapters:

```
struct Record { /* ... */ };

bool name_key_eq(const Record&, const Record) { return true; }
bool ssn_key_eq(const Record, const Record&) { return false; }
```

➥

```
void f(list<Record> lr) {
  typedef typename list<Record>::iterator LI;
  LI p = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(name_key_eq), "FOO"));
  LI q = find_if(lr.begin(), lr.end(), bind2nd(ptr_fun(ssn_key_eq), 123456));
}
```

Besides a small typo, `typename` is allowed only in templates ([ANSI-CPP](14.6#5)), this example will not work in general. This is because the two functions take theirs arguments by reference. Consequently, the types of the arguments of the function object created by `ptr_fun` will be defined as reference types as well (this again is done through simple derivation from `binary_function`). Thereupon any instantiation of `binder2nd` with one of these function objects created by `ptr_fun` will lead to a compilation error because it will force the compiler to create a reference from another reference.

4 Conclusion

Although standardized, it seems as if the C++ version of the STL is far from being complete and consistent. Although prized as a "generic" library, even its own components can't always be used together. This paper focused only on a small part of the STL, namely adapters and binders, but it revealed some serious problems in their implementation in C++. As shown, many of this problems can easily be solved by applying C++ techniques like template specialization.

One reason for this unsatisfying situation might be the absence of fully standard compliant compilers during the design of the library. As demonstrated, even Stroustrup, the "creator" of C++, uses the STL in his book in a canonical way, but without noticing that the real C++-STL would fail to meet his needs.

As a consequence, we must realize that such a complex library like the STL cannot be implemented from scratch in such a complex language like C++. Many problems in C++ itself like the asymmetry between functions and function objects or the dubious references need special workarounds in the library. Therefore a lot of user interaction is needed.

Using the STL still can significantly improve your productivity. But it assumes both, an in deep understanding of the library concepts behind the STL and an in deep understanding of the language concepts of C++. Used with this in mind, the remaining inconsistencies will hopefully be eliminated in the near future, resulting in a powerful and clean library.

5 Future work

It is desirable to analyze the whole C++-STL in order to remove remaining problems. For that a specification language like Tecton ([Tecton]) can be useful. First steps in this area have been made in [TecSTL].

Another possibility may be to change the semantics of the reference declarator in C++ in order to make it idempotent. That means a reference to a reference to a type `T` would be resolved to a reference to `T`, thus eliminating the "reference to reference is not allowed"-error. This could be done only in the special case, where the reference to a reference is introduced through a `typedef` or a template type argument. The same technique is applied in the C++ language in order to ignore repeated `const` or `volatile` qualifiers ([ANSI-CPP](7.1.5)).

References

[AGL] Musser David R. and Stepanov Alexander, *"Ada Generic Library"* Springer-Verlag, 1989

[ANSI-CPP] ANSI/ISO Standard *"International Standard ISO/IEC 14882 - Programming Language C++"*
American National Standards Institute (ANSI), 1998 at: ftp://research.att.com/dist/stdc++/WP/ (1996 draft)

[Kapur] Kapur D., Musser David R. and Stepanov Alexander, *"Operators and Algebraic Structures"*
Proc. of the Conf. on Func. Progr. Languages and Comp. Arch., Portsmouth, New Hampshire, October 1981

[Lee-Step] Lee, Meng and Stepanov, Alexander *"The Standard Template Library"*
at: http://www.cs.rpi.edu/ musser/doc.ps

[Musser] Musser David R. and Stepanov Alexander, *"A Library of Generic Algorithms in Ada"*
Proc. of the 1987 ACMSIGAda International Conference, Boston, December 1987

[SGI-STL] *"The STL Implementation of Silicon Graphics"* at: http://www.sgi.com/Technology/STL/

[Strou97] Stroustrup, Bjarne. *"The C++ Programming Language. Third edition."* Addison Wesley, 1997

[TecSTL] Musser David R. *"Tecton Description of STL Container and Iterator Concepts"* Wilhelm-Schickard-Institute für Informatik, University of Tübingen, 1998 (unpublished)

[Tecton] Kapur D., Musser David R. and Stepanov Alexander, *"Tecton: a framework for specifying and verifying generic system components"*, Rensselaer Polytechnic Inst., Computer Science Technical Report 92-20, 1992