

# STL and OO Don't Easily Mix

Dietmar Kühl  
Phaidros Software AG \*

29.September 2000

## Abstract

The STL is a powerful tool for many kinds of processing. Unfortunately, using polymorphic objects with the STL seems not to work: Polymorphic objects stored in STL containers either get sliced (i.e. only the base part is copied or assigned but not the derived part) or, when storing pointers to them instead, are not destroyed. Applying algorithms to such containers often results in the wrong thing or complex predicate objects are needed. This article shows how to overcome at least some of these problems using some adaptors and also outlines a possible implementation of STL for better integration with polymorphic objects. The improved integration just acknowledges the distinction between the object and the entity used to maintain it.

## 1 Introduction

Lets solve a rather simple problem: A set of of customers being either companies or persons is to be read from a file and to be printed in alphabetic order of the names. The customers are represented with `customer` being an abstract base class of the two concrete classes `company` and `person`. The class `customer` declares a virtual function `name()` and defines a "less than" operator for alphabetic comparison of two objects' name. In addition, it defines input and output operators for reading and writing the polymorphic objects. This is a model well suited to show typical problems with polymorphic objects when using the STL, i.e. the algorithm, containers, and iterators library from the standard C++ library.

Following the encouraging tests made on sets of ints to read, sort, and print them, a simple program should solve the task at hand:

```
std::ifstream          in("customer.txt");
std::istream_iterator<customer> beg(in), end;
std::vector<customer>   vec(beg, end);
std::sort(vec.begin(), vec.end());
```

---

\*e-mail: [dietmar\\_kuehl@yahoo.com](mailto:dietmar_kuehl@yahoo.com), <http://www.phaidros.com/>

```
std::copy(vec.begin(), vec.end(),
          std::ostream_iterator<customer>(std::cout, "\n"));
```

Since this example followed from the tests using `int` by simply replacing each `int` by `customer` there should be no problem. The compiler thinks differently and complains about `customer` being abstract. A solution seems to be using pointers to `customer` rather than objects which immediately requires that they are deleted after running through the loop. Also, it is no longer possible to just rely on the `sort()` function to select the right predicate for the comparison: Comparing two pointers with the "less than" operator does something rather useless (at least in this context) and does not at all call the lexicographic compare.

Using a bunch of function objects and with some trickery we could get the example to work but the code is rather complex even for such a basic problem. Taking it further and applying STL for things not that trivial really obfuscates the original advantage of STL, namely that it is easy to apply to arbitrary data structures: It is not at all easy to apply to polymorphic objects! Investigating what causes these problems helps in creating solutions to them, fostering integration of object orientation with STL.

## 2 Values and References

If the above example would not have used an abstract base as the value type for the container, the compiler would have accepted the code without complaint. Unfortunately, this is not the solution to the problem because the result would do the wrong thing: STL is oriented towards values and not towards references resulting in polymorphic objects being "sliced". That is, their dynamic type is not retained and only the base part of the objects is copied which is clearly not the right thing when polymorphic objects are to be stored.

Dealing with polymorphic objects always requires some form of indirection which is actually the only way to access objects in typical object-oriented programming languages with C++ being an exception. In C++ this means that a container of polymorphic objects would store some form of a pointer. Using plain pointers is probably not the right thing, at least not if the STL containers are used:

- Objects stored in the container have to be released somehow. Since the STL containers only destroy the value, i.e. the pointer, some entity unrelated to the container has to be in charge of the objects in the container. A possible solution to the lifetime management can be reference counted pointers.
- When using STL with values these are copied creating different instances of the objects. When pointers are used, copying creates a new pointer which refers to the same object as the original. Whether this is the right

thing somewhat depends on the specific use but STL idioms somewhat expect that real copies are made. Thus aliasing problems are rather likely: For example, just copying a container prior to modifying the elements does not suffice to retain the original state.

If aliasing problems are no concern, a reference counted pointer might be the right approach. Otherwise, a copying pointer can be used which creates copies of the referenced object with the correct dynamic type. This can even be done without any additional requirement beyond the objects being copy constructible: A copying pointer holds a pointer to an object derived from an auxiliary base capable of cloning the value. The only requirement is on the style objects of the copying pointer type are created: The constructor takes a reference to the concrete object type which is used to initialize a heap allocated object of a class derived from the auxiliary base.

This copying pointer implementation consists of three template classes:

- The copying pointer class `poly<T>` which is templated with some base class.
- An auxiliary abstract base class `poly_base<T>` which is templated with the same `T` as `poly<T>`: Each `poly<T>` holds a pointer to a `poly_base<T>`.
- An auxiliary `poly_object<T, S>` class which is derived from `poly<T>`. The additional template argument `S` is a class derived from `T`.

The basic idea is that a templated constructor of `poly<T>` allocates an object of type `poly_object<T, S>` to which a pointer is stored in the `poly<T>` object. On copying or assigning the copying pointer `poly<T>` the concrete object is copied. The implementation of this is pretty simple. Here is the auxiliary base class declaring the abstract methods used to clone and access objects:

```
template <typename T>
struct poly_base {
    virtual ~poly_base() {}           // support derivation
    virtual poly_base* clone() = 0;   // copy derived object
    virtual T& get_object() = 0;     // stored in derived
};
```

The base class defines a virtual destructor to allow deletion of derived objects using a pointer to a base object. In addition, it declares an abstract function `clone()` which allocates a copy of the object and an abstract function `get_object()` used to obtain a reference to the `T` object hold in the derived object. The derived class just implements these methods:

```
template <typename T, typename S>
struct poly_object: public poly_base<T> {
    poly_object(S const& s): object(s) {}
    poly_base<T>* clone() { return new poly_object(*this); }
```

```

    T& get_object() { return object; }
private:
    S object;
};

```

Each `poly_object<T, S>` stores an object of type `S` to which a reference is returned by the `get_object()` method, i.e. `S` is derived from `T`. The stored object is initialized using copy construction. The whole purpose of this class system is the method `clone()` which creates a copy of the correct type without requiring such a method for the type `T`: This method just allocates a new copy of `poly_object<T, S>` initialized by copy construction from `*this`.

These two classes are just auxiliary classes which are not used directly by the user. Instead, the user only uses the copying pointer:

```

template <typename T>
struct poly {
    poly(): ptr(0) {} // no object by default

    template <typename S> // creation from prototype
    poly(S const& s):
        ptr(new poly_object<T, S>(s)) {}

    poly(poly const& p): // create a cloned object
        ptr(p.ptr->clone()) {}

    ~poly() { delete ptr; } // release current object

    poly& operator= (poly const& p) { // assign cloned object
        poly_base<T>* tmp = p.ptr->clone();
        std::swap(ptr, tmp);
        delete tmp;
        return *this;
    }

    // Accessors to the hold object:
    T& operator*() const { return ptr->get_object(); }
    T* operator->() const { return &(ptr->get_object()); }
private:
    poly_base<T>* ptr;
};

```

Each `poly<T>` object is just a reference to an actual polymorphic object. On copying or assigning a `poly<T>` object, a new copy of the hold objects is created with the correct dynamic type. Such objects can be put into an STL container without any possibility of creating aliasing conflicts. The tricky part is the

templated constructor: This constructor takes an object used as prototype to construct the internal object. It is expected that the static type of this object matches the dynamic type. If this is not the case, the copied object will be sliced because the type is inferred from the static type.

A simple use of the copying pointer could look like this:

```
std::vector<poly<customer> > customers;
customers.push_back(company("Foo Inc. "));
customers.push_back(person("Sam Sample"));
```

This would create a `std::vector` with two `customers`. Of course, this is just one of many approaches to put polymorphic objects into a container and having them maintained automatically. This approach will be used in the remainder of the article as an example but the following discussion applies to other forms of smart pointers, too.

### 3 Running Algorithms

Now that `std::vector<poly<customer> >` can be used to hold the polymorphic objects, the next thing to be done is sorting and printing them. Just using

```
std::sort(vec.begin(), vec.end());
```

does not work because there is no less than operator defined for `poly<T>`. Similarly when trying to send the objects to the output there is not output operator. However, there is a rather simple approach to solve this: The corresponding operations can simply be implemented for `poly<T>` forwarding the requests to the referenced object:

```
template <typename T>
bool operator< (poly<T> const& p1, poly<T> const& p2) {
    return *p1 < *p2;
}
template <typename T, typename cT, typename traits>
std::basic_ostream<cT, traits>&
operator<< (std::basic_ostream<cT, traits>& out,
           poly<T> const& p) {
    return out << *p;
}
```

For the limited number of operators defined this is a viable approach giving a useful interface, too: The only change necessary to make the original code work is to use `poly<customer>` instead of just `customer`.

However, this does not work well with function objects operating on the base class: For example, if there would be a compare function for `customer`

objects rather than an operator, this would not really work. To use function objects with the container holding polymorphic objects, it is necessary to create an adaptor. Actually, the standard provides a set of adaptors for this: Giving a pointer to member function to the function `std::mem_fun()` returns a functor which can be applied to pointers. Unfortunately, this approach has two problems:

- It obviously only works with member functions and the standard library does not support a corresponding approach for non-member functions (`std::ptr_fun()` does not use pointers).
- It only works with raw pointers but not with smart pointers. Unfortunately, the `operator()()` of the adaptor class cannot simply be templated on the argument types because there is a typedef for the argument types.

Basically, what is missing to use some function object with iterators on a collection of smart pointers is additional dereferencing: The iterator has to be dereferenced once to get at the smart pointer and then once again to get at the held object. Below is a generic adaptor adding the necessary dereference to function objects. The implementation of the adaptor needs a few auxiliary classes which are used to deal with qualifiers on the types:

```
template <typename T, template <typename S> class Ptr>
struct ptr_traits {
    typedef Ptr<T> const type;
};
template <typename T, template <typename S> class Ptr>
struct ptr_traits<T const&, Ptr> {
    typedef Ptr<T> const type;
};
template <typename T, template <typename S> class Ptr>
struct ptr_traits<T&, Ptr> {
    typedef Ptr<T> type;
};
```

The template class `ptr_traits` is used to define a smart pointer type appropriate for being passed as argument to a function. The definition deals appropriately with type qualifiers: If the template argument is a reference, the reference part is removed from the type. In addition, the reference part together with a `const` qualifier is used to determine whether the smart pointer can be passed as `const` reference or has to be passed as non-`const` reference. The first template argument is the type referenced by the smart pointer and the second template argument is a template class defining the smart pointer type.

The argument types in the adaptor class are determined from the corresponding typedefs of the function object being adapted by just using the type defined by `ptr_traits` applied to these types with the smart pointer template as second argument. To be a well behaved functor, the adaptor also provides

the corresponding typedefs. The function call operator just calls the underlying function object after dereferencing the arguments:

```
template <template <typename T> class Ptr, typename Fct>
struct binary_deref_function
{
    // typedefs required to be a binary function object:
    typedef typename
        ptr_traits<typename Fct::first_argument_type, Ptr>::type
        first_argument_type;
    typedef typename
        ptr_traits<typename Fct::second_argument_type, Ptr>::type
        second_argument_type;
    typedef typename Fct::result_type      result_type;

    // constructor:
    binary_deref_function(Fct const& f = Fct()): fct(f) {}

    // function call operators:
    result_type operator()(first_argument_type& arg1,
                          second_argument_type& arg2)
        { return fct(*arg1, *arg2); }
    result_type operator()(first_argument_type& arg1,
                          second_argument_type& arg2) const
        { return fct(*arg1, *arg2); }

private:
    Fct fct;
};
```

The first template argument to this class is a template used to define the types of the arguments: This template argument is passed to the `ptr_traits` to obtain the argument types of the functor. Thus, the first template argument should name a smart pointer template class. The second template argument is the functor which is adapted for use with smart pointers.

To simplify creation of the function adaptor, a few functions creating objects of this type can be provided similar to the function `std::mem_fun`. For example:

```
template <template <typename T> class Ptr,
         typename Result, typename Arg1, typename Arg2>
binary_deref_function<Ptr,
    std::pointer_to_binary_function<Arg1, Arg2, Result> >
binary_deref_fct(Result (*fct)(Arg1, Arg2)) {
    return std::ptr_fun(fct);
}
```

The function provides a function object dereferencing its arguments prior to calling the binary function passed as argument. This can be used to sort a container `customers` with `poly<customer>` objects like this:

```
std::sort(customers.begin(), customers.end(),
          binary_deref_fct<poly>(compare));
```

For function objects returning types different from the elements in the sequence this approach works fairly well. For function objects returning the elements of the sequence as might be desirable e.g. for `std::transform()`, the above adaptor would not return the smart pointer to the object but rather the object itself. This problem can be avoided using another adaptor which appropriately changes the return type, too. Thus it would be desirable to have even better integration of smart pointers.

## 4 Three, Not Two, Involved Types

Using [smart] pointers in containers introduces the problem of having two objects representing one element: There is the pointer and the object pointed to. For operations adding elements to a container or to move elements within a container, the pointer has to be used to avoid slicing of objects. For predicates and functors the referenced object is used. The pointer which ends up as being the value type of the container is of no interest while the referenced objects are. That is, there are two roles involved in the operations: The role of the pointer which just holds an object and the role of the held object which is used with predicates, functors, etc.

For homogeneous containers with elements of a concrete type there is no such distinction: The value type is used when adding or moving elements as well as for predicates and functors. That is, the value type appears in both roles, being holder and held object at once <sup>1</sup>. To achieve better integration of pointers, algorithms should be able to distinguish between these two roles using the holder role when objects are moved, copied, etc. and using the held object role when accessing objects, applying predicates or functors, etc. The iterators passed to the algorithm would be used to access either the holder or the held object depending on what kind of operation is to be done. For example, let `poly_vector` be a template class used to maintain polymorphic objects e.g. by using the template class `poly` internally to maintain the objects:

```
// create a sample container:
poly_vector<customer> vec;
vec.push_back(person("Sam Sample"));
vec.push_back(company("Foo Inc."));
```

---

<sup>1</sup>Thanks to Daniel Johnson for pointing out the two different roles of value types in the current STL.

```

// use the holder role:
std::reverse(vec.begin(), vec.end());

// use the held object role:
std::find(vec.begin(), vec.end(), person("Sam Sample"));

```

To reverse the elements in the sequence, the holder objects have to be exchanged: Using the held object type to swap the elements would slice the elements in the above case. This does not occur if the holders are swapped. On the other hand, finding a match for the newly created object passed as third argument to `std::find()` has to inspect the held objects.

The idea is to use the held object as value type when using access functions like `operator*` on iterators even if the holder and the held object are different entities. However, the holder object is used when objects are moved or added, e.g. when `std::reverse()` swaps two elements. Of course, in cases where there is no distinction between the holder and the held object this does not make a difference. If there is a distinction between holder and held object as is the case when smart pointers are used, this allows moving the explicit dereferencing applied e.g. in the `binary_deref_function` functor into the container and iterator removing the need to explicitly deal with pointers when using inhomogeneous containers. On the other hand, some burden is put on the implementation of algorithms which potentially have to distinguish between the two roles.

An approach to the implementation of algorithms taking care of the two different roles is the use of functions from a traits class doing the appropriate operations: The default implementation just performs the operation on the value type while a specialized traits class can perform the operation using the holder objects. For example, the `std::reverse()` function could be implemented like this:

```

template <typename It>
void reverse(It begin, It end) {
    for (; begin != end && begin != --end; ++begin)
        holder_traits<It, It>::swap(begin, end);
}

```

The class `holder_traits` collects mutating functions which can be specialized to operate on the holder type. Two iterators are used as template arguments because to operate on the holder objects these objects have to be accessible from both iterators. Since the iterator types may differ, e.g. when swapping elements between collections of different type, the specializations can depend on two iterator types. The default implementation just uses the value type. For example, the default implementation of `swap()` would just swap the elements using `std::swap()`:

```

template <typename It1, typename It2>

```

```

void holder_traits<It1, It2>::swap(It1 it1, It2 it2) {
    std::swap(*it1, *it2);
}

```

Specializations of this `swap()` member would swap the holders pointed to by the iterator. That is, specializations would not dereference the iterator but rather use the data internal to the iterator to get access to the holder rather than to the held object.

Some implementations partially support this approach already in the form of using `std::iter_swap()` rather than `std::swap()` in some STL algorithms. Clearly, `iter_swap()` can be implemented in terms of the above traits class.

In addition to a method `swap()` for exchanging the contents of two elements, an `assign()` method would be available in the `holder_traits` to change the value: When assigning polymorphic values, it is necessary to change the type of the changed object if types do not match. This method would also be used when adding new elements using an inserting iterator. Other forms of adding elements are up the containers anyway, i.e. a container for polymorphic objects can do the right thing already when adding new elements.

If STL algorithms would be required to distinguish between the holder and the held object, use of collections with polymorphic objects could much more convenient. In addition to mere convenience, this can be used to change the handling of objects in a container without changing how the container is used. This can be important if moving objects turns out to be too expensive and instead of moving them, the container type is changed to one where only pointers are moved rather than the objects.

## 5 Conclusions

Although STL algorithms and containers don't seem to mix at all with polymorphic objects, it is actually not that bad. With a few adaptors and a little bit of trickery, typical tasks can be made reasonably easy. The trickery can be written by some advanced C++ users with the other just using them as tools. However, for full integration of polymorphic objects with the STL algorithms, additional guarantees about the implementation of the algorithms would be desirable. The major drawback of such guarantees would be additional restrictions on the implementations of algorithms making them marginally more complex.

## 6 References

1. Matthew H.Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, 1998.
2. Nicolai M.Josuttis. *The C++ Standard Library*. Addison-Wesley, Reading, 1999.