Heterogeneous, Nested STL Containers in C++

Volker Simonis and Roland Weiss

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
E-mail: {simonis,weissr}@informatik.uni-tuebingen.de

1 Motivation

The incentive to write a nested, heterogeneous container in C++ surfaced during the SUCHTHAT project [11]. Therein we are working on the implementation of a SuchThat compiler. The first prototype's back-end [14], as well as many of the other components, were implemented in Scheme [8]. One of Scheme's main advantages is the powerful list data structure, which can hold arbitrary data types[1]. This allows the user to build nested lists, e.g. to represent a parse tree or symbol table.

Our current focus is on merging Tecton [7] with SuchThat. Due to severe performance problems with our first prototype we have switched to C++ as implementation language. The STL provides basic containers that suit most simple needs and exhibit very good runtime behavior. The containers' major drawback for our purposes is the inability to hold objects of different types and that they do not support nesting.

We will show that exploitation of C++'s newest technologies, like templates and run-time type information (RTTI), leads to a powerful data structure based on the STL. We think that the different paradigms of generic, object oriented and functional programming, which often are seen as adversaries, can instead complement each other.

2 Approaches

We observed a trade off between syntactic elegance and runtime performance. This made us come up with two fundamentally distinct approaches.

The first one, the more conservative, relies on an abstract base class that provides polymorphic behavior with easy to use parameterized standard elements. The `nseq` class uses template template arguments (see [1], 14.3.3) for maximal flexibility.

The second approach builds on the semantics of chameleon objects [12]. It is outperformed by the first one regarding runtime but excels in usability.

[1] Of course, this holds true for any untyped language.

2.1 Specification of the Problem

We informally state with the following three requirements what we call a *heterogeneous, nested sequence S*.

1. Every STL sequence container should be applicable as underlying implementation container of *S* (flexibility property).
2. *S* should be able to hold arbitrary objects (heterogeneity property).
3. Any nested sequence *S* should be able to hold other nested sequences recursively (nesting property).

2.2 Classical Polymorphism

The well established way in C++ to provide polymorphic behavior uses inheritance. The heterogeneous container holds pointers to a base class and the C++ runtime system will dispatch methods based on the polymorphic type. Our base class `BaseElem` declares the virtual functions `BaseElem* clone()` and `BaseElem* create()` to support the virtual constructor idiom (see [2], 20.5).

Instead of letting the user write wrapper objects for every type he uses, we deliver the template class `Elem<>` that inherits from `BaseElem`. The signature is `template <class valT> class Elem : public BaseElem`. This wrapper class does all the tedious work a user usually has to do on his own: define constructors and destructors, as well as various auxiliary methods (e.g. I/O functions). She must only instantiate the template class. This works for basic types and classes, e.g. `Elem<int> i` or `Elem<string> s("test")`.

Let us examine the class `nseq`, which should fulfill the problem specification given in section 2.1. The heterogeneity property can be obtained by keeping pointers to the base class `BaseElem` in the sequence. If we want to comply to the nesting property, `nseq` must be derived from `BaseElem` itself. Furthermore, in order to use STL containers, `nseq` must also be a subclass of such a container. These considerations lead to this signature of a nested list:

```
class nlist : public list<BaseElem*>, public BaseElem;
```

This works fine, but it does not fulfill the flexibility property, because the implementation container is hard-coded. You cannot provide different containers, as the class `nlist` is a subclass of the STL container instantiation `list<BaseElem*>`. To gain the desired additional level of abstaction, we use template template arguments, a very novel C++ feature. `nseq` becomes a template class, whose template argument is a container, which is a template class itself. Therefore, a simple template would not suffice. The final class header for `nseq` reads:

```
template <template <class valT> class containerT>
class nseq : public containerT<BaseElem*>, public BaseElem;
```

The `clone()`-function has a boolean parameter `shallow`, which is of importance for nested sequences only. It controls if either a shallow or a deep copy of the container is made. A shallow copy creates a new container with pointers to all top-level elements.

A deep copy creates a new container, recursively holding copies of the containers and atoms in the source sequence.

Working with our `nseq` class is quite simple. An instance of a nested deque is created with `nseq<deque> nd`. We can use all of `deque`'s member functions to add elements to our container, e.g. `nd.push_back(new Elem<int>(4711))`. The sequence can be walked with the STL container's iterators, but only at the top-most level. You can recursively descend into nesting layers, if the member function `bool is_atom()` returns false, which is the case for elements that are nested sequences. When you walk a `nseq` and want to operate on the elements, you have to perform a dynamic cast on `BaseElem`. The following code example shows how the first level of a nested sequence is walked with the container provided iterator and every integer entry is replaced by its square power.

```
for (nseq<list>::iterator iter = nl.begin()); iter != nl.end(); ++iter)
  if (intp = dynamic_cast<Elem<int>*>(*iter))
    *intp = intp->getValue() * intp->getValue();
```

Figure1 compares the layout of `nseq<vector>` and `vector<int>`[2]. It shows that we get a memory overhead of two pointers (eight bytes) for every element, regardless of the wrapped object's type. The first one points to `BaseElem`. This indirection is needed to use C++'s polymorphic mechanism. The second pointer holds the address of the element's virtual table.

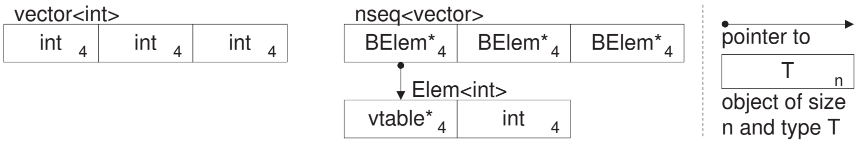

Figure1. Layout comparison of a standard STL container and a nested sequence.

2.3 Chameleon Objects

In [12] a new technique for providing a generic, type-safe wrapper class is presented. This goal is achieved through the unparameterized class `Value`. Contrary to the class itself, all its methods, like the constructor and a set of overloaded operators, are parametrized, i.e. template functions. Thus, any object of arbitrary type can be assigned to a `Value` object due to the parameterized assignment operator `template<class T> T operator=(const T&)`. In turn, a `Value` object can easily be reassigned to an object of its initial type because of the parameterized conversion operator `template<class T> T& operator T()`.

The `Value` class guarantees strict type safety by signaling any attempt to assign a `Value` object to another object of incorrect type by throwing an exception[3]. To achieve this functionality, the `Value` class keeps all information about the wrapped object in a private, static data member inside a member function, parameterized with the same type.

[2] We assume that sizeof(int) = 4 and sizeof(pointer) = 4, which is true for most contemporary 32bit architectures.

[3] Type identity is defined as name identity here.

Since all used data types are known at compile time, the compiler can instantiate the corresponding methods and data objects. In fact, for every data type used in conjunction with a `Value` object, a full set of operators and methods is instantiated by the compiler. The type checking of any operation concerning a `Value` object is performed by these methods at run time. Because of their ability to change their internal type at any time, `Value` class instances are called *chameleon objects*.

Given this `Value` class, we can now construct heterogeneous containers based on the standard STL containers by instantiating such a container for `Value` with `list<Value> polyCont`. Thereafter, objects of arbitrary type can be inserted into the container, e.g. `polyCont.push_back(12)`, `polyCont.push_back(0.9)` and `polyCont.push_back(string("hello"))`. Because, as stated above, `Value` objects can hold objects of any type, even containers can be inserted as elements into a `Value` parameterized container, thus obtaining nested containers.

The extraction of elements from the container is straightforward, too. If we know the the desired element's type, we can simply query it. Given the list `polyCont` from the previuos example, we can write `int i=polyCont.front()` to get the first element of the list. More care must be taken if the type of the desired element is unknown. Since overloading the `typeid()` operator is not allowed in C++ (see [1], 13.5), the `Value` class defines a method `typeId()`, which returns the type information for the currently wrapped object. With this information and `Value`'s parameterized method `template<class T> T& getValue()`, one can access every element of the nested, polymorphic container. This is shown in the following sample code:

```
// double sin(double); a function with this signature must exist
void apply(list<Value> &cont) {
  for (list<Value>::iterator it = cont.begin(); it != cont.end(); ++it) {
    if (it->typeId() == typeid(list<Value>)) apply((list<Value>)*it);
    else if (it->typeId() == typeid(int)) *it = (int)*it * (int)*it;
    else if (it->typeId() == typeid(double)) *it = sin(*it);
  }
}
```

3 Performance Tests

The results of the performance tests are presented in Figure2. We compared the original STL `list` and `vector` containers against their nested counterparts based on our implementations. The tests consisted of two parts, container creation and element access. They were all performed for `int` and `std::string` data types. The containers were filled in a loop using `push_back(T&)`. Access was measured by iterating over the created container and mutating its elements.

The charts show the overhead introduced by our containers for the flat, homogeneous case, where of course their additional features are not used. The price you have to pay for nesting and heterogeneity is a runtime penalty ranging from 1.3 to 1.9 for complex objects (`std::string` in our tests) and 2.9 to 13 for built in types (`int`).

Furthermore, we want to note that due to the extensive use of the `new` operator in our element classes, the tests depend heavily on the applied memory allocation scheme. Therefore our source code includes the smart memory allocator presented in [5], which speeds up the tests significantly compared to the default `new` operator.

Our classes make heavy use of new C++ language features like RTTI and templates. We were able to compile our code at the time of this writing with egcs 1.1 [3], the EDG front-end [4] and IBM VisualAge C++ 4.0 [6].

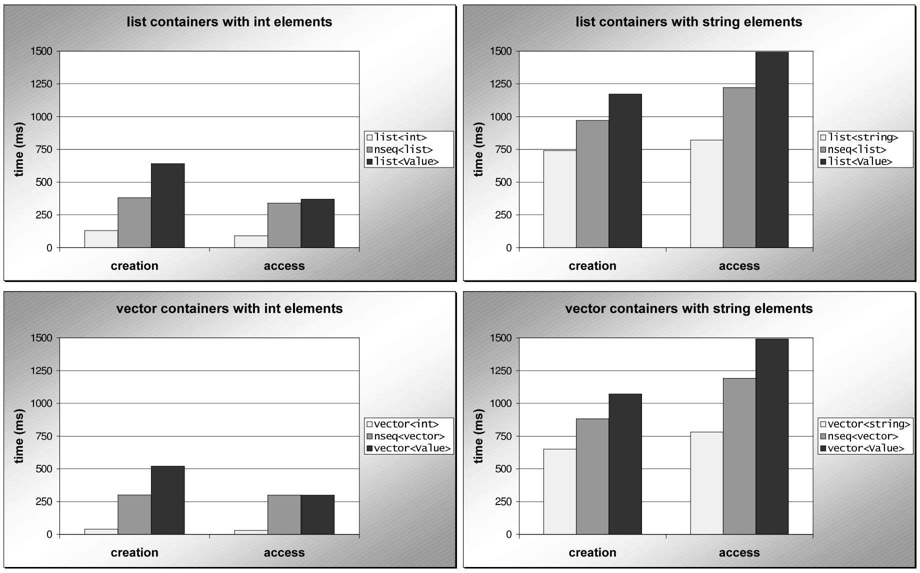

Figure2. All tests ran on a Pentium II, 333 MHz, 128MB machine under Windows NT 4 and were compiled with egcs 1.1.1. The container size is 400000 elements.

4 Results

We presented two distinct approaches to the problem of implementing a nested, heterogeneous container in C++. The classical one shows better runtime performance. Its overhead, compared to a standard STL container, arises from the pointer indirection, which is necessary for the polymorphic approach, and the virtual table pointer in the `Elem<>` class (see Figure1). This generic wrapper frees the user of boring work. One drawback is the pointer semantics, uncommon in the value semantics of STL containers. It also forces the user to handle most aspects of memory management.

The container based on chameleon objects offers syntactic elegance that equals untyped data structures, like those present in Scheme. Operations on elements are inherently type-safe, type violations are signalled by exceptions. All this is made possible by the seamless integration of STL containers with the `Value` class. It hides much of the details of casting and type checking from the user, which is still visible in our classical approach. No efforts must be taken by the user to adapt objects for storage in the nested container. The beauty of this approach is bought at the cost of increased runtime.

Our classes leave it up to the programmer to choose either faster executing code or more elegant source code.

5 Future Work

We currently focus on implementing a `flat_iterator`, which behaves like a simple sequence iterator and traverses all elements in a nested sequence in depth first order. This iterator enables us to use STL algorithms on our nested sequences.

Another interesting question will be the use of different memory allocators and the implementation of garbage collection ([10]) for the objects stored in the containers. We believe this task can be addressed efficiently and transparently for the user through the introduced element classes `Elem<>` and `Value`, respectively.

Finally, one can think of a reference counting mechanism, implemented also through the mentioned element classes, which can lead to a dramatical performance increase, especially in situations where deeply nested containers are heavily copied for read only purposes.

References

1. ANSI/ISO Standard: *Programming languages - C++*, ISO/IEC 14882, 1998.
2. Marshall Cline: *C++ FAQ LITE – Frequently Asked Questions*, `http://www.cerfnet.com/~mpcline/c++-faq-lite`.
3. Cygnus Solutions: *egcs project home page*, `http://egcs.cygnus.com`.
4. Edison Design Group: *The C++ Front End*, `http://www.edg.com/cpp.html`.
5. Sasha Gontmakher and Ilan Horn: *Efficient Memory Allocation*, Dr. Dobbs Journal, p. 116-119, January 1999.
6. IBM: *IBM VisualAge C++*, `http://www.software.ibm.com/ad/visualage_c++`.
7. D. Kapur and David R. Musser: *Tecton: a framework for specifying and verifying generic system components*, Rensselaer Polytechnic Institute Computer Science Technical Report 92-20, 1992.
8. Richard Kelsey, William Clinger, and Jonathan Rees (editors): *Revised^5 Report on the Algorithmic Language Scheme*, 1998.
9. David R. Musser, Atul Saini: *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, 1996.
10. Gor V. Nishanov, Sibylle Schupp: *Garbage Collection in Generic Libraries*, Proceedings of the International Symposium on Memory Management 1998, p. 86-97, Richard Jones (editor), 1998.
11. Sibylle Schupp: *Generic programming — SuchThat one can build an algebraic library*, Ph.D. thesis at the Wilhelm-Schickard-Institut für Informatik, Eberhard-Karls-Universität Tübingen, 1996.
12. Volker Simonis: *Chameleon Objects, or how to write a generic type safe wrapper class*, C++ Report Jan. 2000, SIGS Publications, `http://www-ca.informatik.uni-tuebingen.de/people/simonis/papers/chameleon/chameleon.htm`.
13. David Vandevoorde: *C++ Solutions*, Addison-Wesley Publishing Company, 1998.
14. Roland Weiss: *ScmToCpp: a configurable, intelligent back-end for SuchThat*, Internal Report at the WSI für Informatik, Universität Tübingen, 1998.