Exploring Template Template Parameters

Roland Weiss and Volker Simonis

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

Abstract The generic programming paradigm has exerted great influence on the recent development of C++, e.g., large parts of its standard library [2] are based on generic containers and algorithms. While templates, the language feature of C++ that supports generic programming, have become widely used and well understood in the last years, one aspect of templates has been mostly ignored: template template parameters ([2], 14.1). In the first part, this article will present an in depth introduction of the new technique. The second part introduces a class for arbitrary precision arithmetic, whose design is based on template template parameters. Finally, we end with a discussion of the benefits and drawbacks of this new programming technique and how it applies to generic languages other than C++.

1 Introduction

The C++ standard library incorporated the standard template library (STL) [15] and its ideas, which are the cornerstones of generic programming [14]. Templates are the language feature that supports generic programming in C++. They come in two flavors, class templates and function templates. Class templates are used to express classes parameterized with types, e.g., the standard library containers, which hold elements of the argument type. Generic algorithms can be expressed with function templates. They allow one to formulate an algorithm independently of concrete types, such that the algorithm is applicable to a range of types complying to specific requirements. For example, the standard `sort` algorithm without function object ([2], 25.3) is able to rearrange a sequence of arbitrary type according to the order implied by the comparison operator <. Of course, the availability of this operator is a requirement on the elements' type.

It is possible to use instantiated class templates as arguments for class and function templates, therefore one is able to write nested constructs like `vector<list<long> >`. So where does the need for template template parameters arise? Templates give one the power to abstract from an implementation detail, the types of the application's local data. Template template parameters provide one with the means to introduce an additional level of abstraction. Instead of using an instantiated class template as argument, the class template itself can be used as template argument. To clarify the meaning of this statement, we will look in the following sections at class and function templates that take template template parameters. Then we will present a generic arbitrary precision arithmetic implemented with template template parameters. Finally, the presented technique is discussed and effects on other generic languages are considered.

2 Class Templates

The standard library offers three sequence containers, `vector`, `list` and `deque`. They all have characteristics that recommend them for a given application context. But if one wants to write a new class called `store` that uses a standard container internally to store values, it is hard to choose the *perfect* container for all possible scenarios. This is exactly the situation where template template parameters fit in. The class designer can provide a default container, but the user can override this decision easily. Note that the user can not only use standard containers but also any proprietary container that conforms to the standard sequence container interface. Let us look at a code example that implements the class `store_comp` using object composition.

```
template < typename val_t,
  template <typename T, typename A> class cont_t = std::deque,
  typename alloc_t = std::allocator<val_t> >
class store_comp
{
  cont_t<val_t, alloc_t> m_cont; //instantiate template template parameter
public:
  typedef typename cont_t<val_t, alloc_t<val_t> >::iterator iterator;
  iterator begin() { return m_cont.begin(); }
  // more delegated methods...
};
```

The first template parameter `val_t` is the type of the objects to be kept inside the store. `cont_t`, the second one, is the template template parameter, which we are interested in. The declaration states that `cont_t` expects two template parameters `T` and `A`, therefore any standard conforming sequence container is applicable. We also provide a default value for the template template parameter, the standard container `deque`. When working with template template parameters, one has to get used to the fact that one provides a real class template as template argument, not an instantiation. The container's allocator `alloc_t` defaults to the standard allocator.

There is nothing unusual about the usage of `cont_t`, the private member `m_cont` is an instantiation of the default or user provided sequence container. As already mentioned, this implementation of `store_comp` applies composition to express the relationship between the new class and the internally used container. Another way to reach the same goal is to use inheritance, as shown in the following code segment:

```
template <typename val_t, ...>
class store_inh : public cont_t<val_t, alloc_t<val_t> > {};
```

The template header is the same as in the previous example. Due to the public inheritance, the user can work with the container's typical interface to change the store's content. For the class `store_comp`, appropriate member functions must be written, which delegate the actual work to the private member `m_cont`. The two differing designs of class `store` are summarized in Figure 1. The notation follows the diagrams in [9]. The only extension is that template template parameters inside the class' parameter list are typeset in boldface.

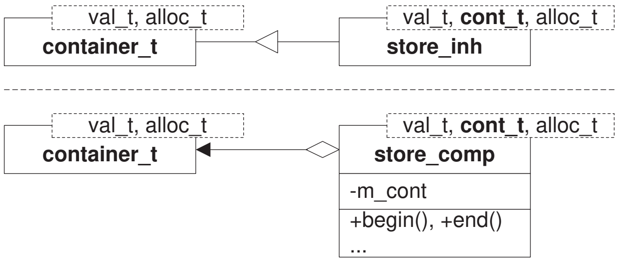

Figure1. Comparison of the competing designs of the `store` classes.

To conclude the overview, these code lines show how to create instances of the `store` classes:

```
store_comp<std::string, std::list> sc;
store_inh<int> si;
```

`sc` uses a `std::list` as internal container, whereas `si` uses the default container `std::deque`. This is a very convenient way for the user to select the appropriate container that matches the needs in his application area. The template template parameter can be seen as a container *policy* [1].

Now that we have seen how to apply template template parameters to a parameterized class in general, let us examine some of the subtleties.

First, the template template parameter – `cont_t` in our case – must be introduced with the keyword `class`, `typename` is not allowed ([2], 14.1). This makes sense, since a template template argument must correspond to a class template, not just a simple type name.

Also, the identifiers `T` and `A` introduced in the parameter list of the template template parameter are only valid inside its own declaration. Effectively, this means that they are not available inside the scope of the class `store`. One can instantiate the template template parameter inside the class body with different arguments multiple times, which would render the identifier(s) ambiguous. Hence, this scoping rule is reasonable.

But the most important point is the number of parameters of the template template parameter itself. Some of you may have wondered why two type parameters are given for a standard container, because they are almost exclusively instantiated with just the element type as argument, e.g., `std::deque<float>`.

In these cases, the allocator parameter defaults to the standard allocator. Why do we have to declare it for `cont_t`? The answer is obvious: the template parameter signatures of the following two class templates `C1` and `C2` are distinct, though some of their instantiations can look the same:

```
template <typename T> class C1 {};
template <typename T1, typename T2 = int> class C2 {};
C1<double> c1; // c1 has signature C1<double>
C2<double> c2; // c2 has signature C2<double, int>
```

In order to be able to use standard containers, we have to declare `cont_t` conforming to the standard library. There ([2], 23.2), all sequence containers have two template parameters.[1] This can have some unexpected consequences. Think of a library implementor who decides to add another default parameter to a sequence container. Normal usage of this container is not affected by this implementation detail, but the class `store` can not be instantiated with this container because of the differing number of template parameters. We have encountered this particular problem with the `deque` implementation of the SGI STL [23].[2] Please note that some of the compilers that currently support template template parameters fail to check the number of arguments given to a template template parameter instantiation.

The template parameters of a template template parameter can have default arguments themselves. For example, if one is not interested in parameterizing a container by its allocator, one can provide the standard allocator as default argument and instantiate the container with just the contained type.

Finally, we will compare the approach with template template parameters to the traditional one using class arguments with template parameters. Such a class would look more or less like this:

```
template <typename cont_t>
class store_t
{
  cont_t m_cont; // use instantiated container for internal representation
public:
  typedef typename cont_t::iterator iterator;            // iterator type
  typedef typename cont_t::value_type value_type;        // value type
  typedef typename cont_t::allocator_type allocator_type; // alloc type
  // rest analogous to store_comp ...
};
typedef std::list<int> my_cont; // container for internal representation
store_t<my_cont> st;            // instantiate store
```

We will examine the advantages and drawbacks of each approach. The traditional one provides an instantiated class template as template argument. Therefore, `store_t` can extract all necessary types like the allocator, iterator etc. This is not possible in classes with template template parameters, because they perform the instantiation of the internal container themselves.

---

[1] The C++ Standardization Committee currently discusses if this a defect, inadequately restricting library writers.

[2] The additional third template parameter was removed recently.

But the traditional approach was made applicable at all by the fact that the user provides the type with which the sequence container is instantiated. If the type is an implementation detail not made explicit to the user, the traditional approach doesn't work. See [21] for an application example with these properties. The ability to create multiple, different instantiations inside the class template body using the template template argument is also beyond the traditional approach:

```
cont_t<int, alloc_t> cont_1;
cont_t<val_t, std::allocator<val_t> > cont_2;
```

3 Function Templates

In the preceding section we showed that by application of template template parameters we gain flexibility in building data structures on top of existing STL container class templates. Now we want to examine what kind of abstractions are possible for generic functions with template template parameters. Of course, one can still use template template parameters to specify a class template for internal usage. This is analogous to the class `store_comp`, where object composition is employed.

But let us try to apply a corresponding abstraction to generic functions as we did to generic containers. We were able to give class users a convenient way to customize a complex data structure according to their application contexts. Transferring this abstraction to generic functions, we want to provide functions whose behavior is modifiable by their template template arguments.

We will exemplify this by adding a new method `view` to the class `store`. Its purpose is to print the store's content in a customizable way. A bare bones implementation inside a class definition is presented here:

```
template <template <typename iter_t> class mutator>
void view(std::ostream& os)
{
  mutator<iterator>()(begin(),end()); // iterator: defined in the store
  std::copy(begin(), end(), std::ostream_iterator<val_t>(os, " "));
}
```

Here, `mutator` is the template template parameter, it has an iterator type as template parameter. The `mutator` changes the order of the elements that are delimited by the two iterator arguments and then prints the changed sequence. This behavior is expressed in the two code lines inside the method body. The first line instantiates the `mutator` with the store's iterator and invokes the `mutator`'s application operator, where the elements are rearranged. In the second line, the mutated store is written to the given output stream `os`, using the algorithm `copy` from the standard library. The types `iterator` and `val_t` are defined in the store class.

The first noteworthy point is that we have to get around an inherent problem of C++: functions are not first order objects. Fortunately, the same workaround already applied to this problem in the STL works fine. The solution is to use function objects (see [15], chapter 8). In the `view` method above, a function object that takes two iterators as arguments is required.

The following example shows how to write a function object that encapsulates the `random_shuffle` standard algorithm and how to call `view` with this function object as the `mutator`:

```
// function object that encapsulates std::random_shuffle
template <typename iter_t>
struct RandomShuffle
{
  void operator()(iter_t i1, iter_t i2) { std::random_shuffle(i1, i2); }
};
// A store s must be created and filled with values...
s.view<RandomShuffle>(cout); //RandomShuffle is the mutator
```

There are two requirements on the template arguments such that the presented technique works properly. First, the application operator provided by the function object, e.g., `RandomShuffle`, must match the usage inside the instantiated class template, e.g., `store_comp`. The `view` method works fine with application operators that expect two iterators as input arguments, like the wrapped `random_shuffle` algorithm from the standard library.

The second requirement touches the generic concepts on which the STL is built. `RandomShuffle` wraps the `random_shuffle` algorithm, which is specified to work with random access iterators. But what happens if one instantiates the `store` class template with `std::list` as template template argument and calls `view<RandomShuffle>`? `std::list` supports only bidirectional iterators, therefore the C++ compiler must fail instantiating `view<RandomShuffle>`. If one is interested in a function object that is usable with all possible `store` instantiations, two possibilities exist. Either we write a general algorithm and demand only the weakest iterator category, possibly loosing efficiency. Or we apply a technique already used in the standard library. The function object can have different specializations, which dispatch to the most efficient algorithm based on the iterator category. See [4] for a good discussion of this approach. This point, involving iterator and container categories as well as algorithm requirements, emphasizes the position of Musser et. al. [16] that generic programming is requirement oriented programming.

Completing, we want to explain why template template parameters are necessary for the `view` function and simple template parameters won't suffice. The key point is that the `mutator` can only be instantiated with the correct iterator. But the iterator is only know to the `store`, therefore an instantiation outside the class template `store` is not possible, at least not in a consistent manner.

Overall, the presented technique gives a class or library designer a versatile tool to make functions customizable by the user.

4 Long Integer Arithmetic – An Application Example

Now we will show how the techniques introduced in the last two sections can be applied to a real world problem. Suppose you want to implement a library for arbitrary precision arithmetic. One of the main problems one encounters is the question of how to represent long numbers. There are many well known possibilities to choose from: arrays, single linked lists, double linked lists, garbage collected or dynamically allocated and freed storage and so on. It is hard to make the right decision at the beginning of the project, especially because our decision will influence the way we have to implement the algorithms working on long numbers. Furthermore, we might not even know in advance all the algorithms that we eventually want to implement in the future.

The better way to go is to leave this decision open and parameterize the long number class by the container, which holds the digits. We just specify a minimal interface where every long number is a sequence of digits, and the digits of every sequence have to be accessible through iterators. With this in mind, we can define our long number class as follows:

```
template<
  template<typename T, typename A> class cont_t = std::vector,
  template<typename AllocT> class alloc_t = std::allocator
>
class Integer {
  // ..
};
```

The first template template parameter stands for an arbitrary container type, which fulfills the requirements of a STL container. As we do not want to leave the memory management completely in the container's responsibility, we use a second template template parameter, which has the same interface as the standard allocator. Both template template parameters have default parameters, namely the standard vector class `std::vector` for the container and the standard allocator std::allocator for the allocator.

Knowing only this interface, a user could create `Integer` instances, which use different containers and allocators to manage a long number's digits. He even does not have to know if we use composition or inheritance in our implementation (see Figure 1 for a summary of the two design paradigms).[3]

In order to give the user access to the long number's digits, we implement the methods `begin()`, `end()` and `push_back()`, which are merely wrappers to the very same methods of the parameterized container. The first two return iterators that give access to the actual digits while the last one can be used to append a digit at the end of the long number. Notice that the type of a digit is treated as an implementation detail. We only have to make it available by defining a public type called `digit_type` in our class. Also we hand over in this way the type definitions of the iterators of the underlying containers. Now, our augmented class looks as follows (with the template definition omitted):

[3] We used composition in our implementation. The main reason was that we wanted to minimize the tradeoff between long numbers consisting of just one digit and real long numbers. Therefore, our `Integer` class is in fact a kind of union or variant record in Pascal notation of either a pointer to the parameterized container or a plain digit. The source code of our implementation is available at http://www-ca.informatik.uni-tuebingen.de/people/simonis/projects.htm.

```
class Integer {
public:
  typedef int digit_type;
  typedef typename cont_t::iterator iterator;
  iterator begin() { return cont->begin(); }
  iterator end() { return cont->end(); }
  void push_back(digit_type v) { cont->push_back(v); }
private:
  cont_t<digit_type, alloc_t> *cont;
};
```

With this in mind and provided addition is defined for the digit type, a user may implement a naive addition without carry for long numbers of equal length in the following way (again the template definition has been omitted):

```
Integer<cont_t, alloc_t>
add(Integer<cont_t, alloc_t> &a, Integer<cont_t, alloc_t> &b) {
  Integer<cont_t, alloc_t> result;
  typename Integer<cont_t, alloc_t>::iterator ia=a.begin(), ib=b.begin();
  while(ia != a.end()) result.push_back(*ia + *ib);)
  return result;
}
```

Based on the technique of iterator traits described in [5] and the proposed container traits in [4] specialized versions of certain algorithms may be written, which make use of the specific features of the underlying container. For example, an algorithm working on vectors can take advantage of random access iterators, while at the same time being aware of the fact that insert operations are linear in the length of the container.

5 Conclusions and Perspectives

We have shown how template template parameters are typically employed. They can be used to give library and class designers new power in providing the user with a facility to adapt the predefined behavior of classes and functions according to his needs and application context. This is especially important if one wants to build on top of already existing generic libraries like the STL.

With our example we demonstrate how template template parameters and generic programming can be used to achieve a flexible design. In contrast to usual template parameters, which parameterize with concrete types, template template parameters allows one to parameterize with incomplete types. This is a kind of *structural* abstraction compared to the abstraction over simple types achieved with usual template parameters. As templates are always instantiated at compile time, this technique comes with absolutely no runtime overhead compared to versions which don't offer this type of parameterization.

One has to think about the applicability of template template parameters, a C++ feature, to other programming languages. Generally, a similar feature makes sense in every language that follows C++'s instantiation model of resolving all type bindings at compile time (e.g., Modula-3 and Ada). Template

template parameters are a powerful feature to remove some restrictions imposed by such a strict instantiation model without introducing runtime overhead.

We measured our example with GCC 2.97 and two versions of the STL, namely the from SGI [23] and one from Rogue Wave [20]. Table 1 compares our `Integer` class with some widely available arbitrary precision libraries (GMP 3.1.1 [11], CLN 1.0.3 [12], NTL 4.1a [22] and Piologie 1.2.1 [24]). The tests have been done on a PentiumIII 667MHz Linux system using the PCL library [6].

bits	GMP	CLN	NTL	Piologie	Integer†	Integer‡	RW-Integer‡	Integer§	RW-Integer§
Addition of n-Bit numbers									
128	820	718	2.087	509	3.693	4.119	3.275	1.658	2.038
	3.846	4.375	6.080	3.290	6.186	6.723	7.186	4.411	4.014
1.024	1.001	871	2.649	1.210	16.769	18.025	7.671	3.078	2.974
	4.336	4.596	6.350	5.392	17.512	17.961	10.892	5.574	4.497
8.192	1.691	1.971	7.350	3.748	121.805	128.774	40.046	13.456	11.668
	5.317	7.986	12.603	8.601	104.750	113.095	45.748	14.231	11.241
65.536	8.174	10.505	43.530	23.491	960.955	1.015.816	278890	96.657	80.150
	32.217	48.811	65.176	52.254	844.618	952.695	339371	104.143	77.745
Multiplication of n-Bit numbers									
128	922	990	1.900	1.293	6.181	6.687	4.088	2.798	2.646
	6.450	4.513	6.972	5.461	11.744	12.674	11.601	9.682	6.743
1.024	8.815	13.740	28.837	34.383	71.585	72.311	52.074	40.234	32.564
	16.572	24.736	29.671	43.539	60.661	63.933	48.949	35.594	30.221
8.192	240.738	394.093	828.825	940.873	2.852.491	2.786.261	2.749.538	2.399.391	1.882.072
	243.438	523.903	671.630	926.693	1.853.809	1.971.483	18.67.466	1.715.800	1.488.350
65.536	5.477.327	13.370.805	22.798.590	28.939.305	167.792.489	163.149.346	171.090.108	151.754.471	118.611.246
	5.158.666	14.695.137	18.031.103	27.289.599	117.485.455	122.771.953	120.008.350	107.798.237	93.442.537

Table1. Performance comparison of our Integer class compared to other arbitrary precision libraries. While GMP is a C library with optimized assembler routines, all the other libraries are written in C++. The first line of every entry denotes the number of processor instructions while the second one indicates the number of processor cycles needed for one operation. Integer† stands for `Integer<slist>`, Integer‡ for `Integer<std::list>`, and Integer§ for `Integer<std::vector>`. The "RW-" prefix marks tests, which have been taken with the Rogue Wave STL in contrast to the other tests, which used the SGI STL.

The results of some tests with garbage collected containers using the Boehm-Weiser-Demers [7] collector have been not very promising. However the significant performance difference between the two STL versions we used indicate that this may be no fundamental problem, but a problem of bad compiler optimization and the orthogonal design of the SGI-STL containers and the plain Boehm-Weiser-Demers garbage collector. Therefor we plan further tests in the future using optimizing compiler and other collectors like TGC [18], [19], which address exactly this problems.

6 Compiler Support

One major problem in working with template template parameters is not a conceptual, but rather a practical one. Even now, three years after the publication of the ISO C++ standard, not all compilers implement this feature.

We were able to compile our examples only with the following compilers: Borland C++ V5.5 [3], Visual Age C++ V4.0 [13], Metrowerks V6.0 and all compilers based on the edg front-end V2.43 [8]. The snapshot versions after November 2000 of the Gnu C++ Compiler [10] also meet the requirements.

7 Acknowledgements

We want to thank Rüdiger Loos for his ideas on nested lists and long integers based on STL containers, which sparked our interest in template template parameters. We also want to thank the guys from EDG [8] for always supplying us with the newest version of their fabulous compiler and Martin Sebor from Rogue Wave for making available to us the latest version of their STL.

References

1. Andrei Alexandrescu. *Modern C++ Design*, Addison-Wesley Publishing Company, 2001.
2. ANSI/ISO Standard. *Programming languages - C++*, ISO/IEC 14882, 1998.
3. Borland, Inprise Corporation. *Borland C++ Compiler 5.5*, http://www.borland.com/bcppbuilder/freecompiler.
4. Matthew Austern. *Algorithms and Containers*, C++ Report, p. 44-47, 101 communications, July/August 2000.
5. Matthew Austern. *Generic Programming and the STL*, Addison-Wesley Publishing Company, 1999.
6. Rudolf Berrendorf, Bernd Mohr. *PCL - The Performance Counter Library*, http://www.fz-juelich.de/zam/PCL.
7. Hans Boehm. *Boehm-Weiser-Demers collector*, http://www.hpl.hp.com/personal/Hans_Boehm/gc.
8. Edison Design Group. *The C++ Front End*, http://www.edg.com/cpp.html.
9. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*, Addison-Wesley Publishing Company, 1995.
10. GCC steering committee. *GCC Home Page*, http://gcc.gnu.org.
11. Torbjorn Granlund. *GNU MP - The GNU Multi Precision Library*, http://www.swox.com/gmp.
12. Bruno Haible. *CLN*, http://clisp.cons.org/~haible/packages-cln.html.
13. IBM. *IBM VisualAge C++*, http://www.software.ibm.com/ad/visualage_c++.
14. Mehdi Jazayeri, Rüdiger G. K. Loos, David R. Musser (Eds.). *Generic Programming*, LNCS No. 1766, Springer, 2000.
15. David R. Musser, Gillmer J. Derge, Atul Saini. *STL Tutorial and Reference Guide: Second Edition*, Addison-Wesley Publishing Company, 2001.
16. David R. Musser, S. Schupp, Rüdiger Loos. *Requirement Oriented Programming*, in [14], p. 12-24, 2000.

17. Scott Meyers. *Effective C++ CD*, Addison-Wesley Publishing Company, 1999.
18. Gor V. Nishanov, Sibylle Schupp. *Garbage Collection in Generic Libraries*, Proc. of the ISMM 1998, p. 86-97, Richard Jones (editor), 1998.
19. Gor V. Nishanov, Sibylle Schupp. *Design and implementation of the fgc garbage collector*, Technical Report98-7, RPI, Troy, 1998.
20. Rogue Wave Software. *Rogue Wave STL (development snapshot)*, November 2000.
21. Volker Simonis, Roland Weiss. *Heterogeneous, Nested STL Containers in C++*, LNCS No. 1755 (PSI '99): p. 263-267, Springer, 1999.
22. Victor Shoup. *NTL*, http://www.shoup.net/ntl.
23. STL Team at SGI. *Standard Template Library Programmer's Guide*, http://www.sgi.com/Technology/STL.
24. Sebastian Wedeniwski. *Piologie*, ftp://ftp.informatik.uni-tuebingen.de/pub/CA/software/Piologie.