Annales Mathematicae et Informaticae
***38** (2011) pp. 75–86*
http://ami.ektf.hu

C++ Standard Template Library by infinite iterators*

Tamás Kozsik, Norbert Pataki, Zalán Szűgyi

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
e-mail: kto@pnyf.inf.elte.hu, patakino@elte.hu, lupin@ludens.elte.hu

Submitted August 31, 2010 Accepted December 8, 2011

Abstract

The C++ Standard Template Library (STL) is an essential part of professional C++ programs. STL is a type-safe template library that is based on the generic programming paradigm and helps to avoid some possible dangerous C++ constructs. With its usage, the efficiency, safety and quality of the code is increased.

However, professional C++ programmers are eager for some missing STL-related opportunities. For instance, infinite ranges are hardly supported by C++ STL. STL does not contain iterators that use a predicate during traversal. STL's design is not good at all from the view of predicates. In this paper we present some extensions of C++ STL that supports effective generic programming. We show scenarios where these extensions can be used pretty gracefully. We present the implementation of our infinite iterators.

Keywords: C++, STL, iterators

MSC: 68N15

1. Introduction

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [1]. In this way containers are defined as class templates and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different

*The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

containers [19]. C++ STL is widely-used inasmuch as it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), a large number of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible [2]. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with the existing containers. The expression problem [23] is solved with this approach.

Iterators bridge the gap between containers and algorithms. They provide a standard interface to the algorithms to access the elements of the containers. Iterators are distinguished based on their capabilities and a hierarchy is formed based on these categories [11], too. The following categories defined in the STL:

- input iterator: the elements are reachable sequentially and they are just readable for the algorithms.
- output iterator: the elements are reachable sequentially and they are just writeable for the algorithms.
- forward iterator: the elements are reachable sequentially and the algorithms can both read and write them.
- bidirectional iterator: the elements are reachable sequentially, but the algorithms can read them forward and backward too, and the elements are both readable and writeable. For example: the container `list` provides this kind of iterator.
- random access iterator: the elements are reachable in any order and the algorithms can read and write the elements. For example: the container `vector` provides iterators with these capabilities.

STL also includes adaptor types which transform standard elements of the library for a different functionality. There are iterator adaptors allowing to read an input stream or write an output stream. These iterator adaptors instead of access an existing element of a container read or write them to a stream. Other iterator adaptor allows to insert a new element into a container instead of access an existing one. These iterator adaptors are mainly input or output iterators [9].

Functor objects make STL more flexible as they enable the execution of user-defined code parts inside the library without significant overhead [10]. Basically, functors are usually simple classes with an `operator()`. Inside of the library, the `operator()`s are called to execute user-defined code snippets. This can called a function via pointer to functions or an actual `operator()` in a class. Functors are widely used in the STL inasmuch as they can be inlined by the compilers and they cause no runtime overhead in contrast to function pointers. Moreover, in case of functors extra parameters can be passed to the code snippets via constructor calls.

Functors can be used in various roles: they can define predicates when searching or counting elements, they can define comparison for sorting elements and properly searching, they can define operations to be executed on elements.

We can distinguish the functors by their behaviour. The functors with no arguments are called *generators*. The functors which return boolean values are *predicates*. We call those functors *unary or binary functor*, which has exactly one or two arguments.

The algorithms usually work on a range of input sequence. The range is defined by a pair of iterators. The first iterator of the pair referring to the beginning of the range and the second one referring to the end. The range is inclusive on the left and exclusive on the right. All containers has two member functions `begin` and `end`, which return an iterator of the first element of the container and a dummy iterator referring to the element after the last element in a container. As the range is exclusive on the right, the range `begin() ... end()` covers the whole container. Moreover the *end iterator* can be used as extremal value, such as the algorithm `find` returns it, when the searched element is not in the range.

The begin and the end of the range are handled specially for those iterators, which do not belong to a container. For example the `istream_iterator` reads elements from the standard input, and it reaches the end of its range when the next read is failed. (E.g.: it reaches the end of file.) `istream_iterator` created by constructor setting the source stream representing the beginning of the range, and the other created by the default constructor will be the end of the range.

The iterators are essential part of STL as they provide the input to the algorithms. Although all the containers provide different classes of iterators, and variety of iterator adaptors are exist in STL, there are several important functionality still missing. There is no support to filter the elements that the iterator traverses, there is no possibility to iterate over an integer range, the elements cannot be transformed by the iterator and at last but not at least there is not possible to work with infinite ranges.

There are ongoing researches to improve the iterator facility of STL (see section 7), no one of them supports infinite ranges.

In this paper we provide an *infinite iterator* type, which is able to generate an infinite sequence of elements. This feature is mainly supported by functional languages, such as Haskell. Functional languages are able to use infinite ranges and lazy evaluation [6]. C++ programmers are eager for these features, too [5, 7, 8, 13, 15, 16, 17]. While infinite sequence is widely used in functional programming realm, the *infinite iterator*s simplifies the initialization of containers, as well.

Our paper is organized as follows: we introduce *infinite iterators* in section 2, and we detaile our enhancements applied on C++0x in section 3. The implementation details about detecting the arity of functors and the stoppage of generation are discussed in section 4 and 5. In section 6 we discuss which kind of infinite ranges are supported by our library. The related works is detailed in section 7 and we conclude our results in section 8.

2. Infinite iterators

The `infinite_iterator`, like the `istream_iterator` of the STL, does not belong to any container, but generates a sequence of elements. Any increase on `infinite_iterator` generates the next element of the sequence. The generator strategy is provided by the user as a functor object. The `infinite_iterator` is an *input iterator*, thus the elements are only readable.

In the hereinafter example, we create an `infinite_iterator` which generates Fibonacci numbers. Then we write the first 10 Fibonacci number to the standard output.

```
struct Fib
{
    Fib() : a(0), b(1) {}
    int operator()()
    {
        int res = a + b;
        a = b;
        b = res;
        return res;
    }
private:
    int a;
    int b;
};

Fib fib;
infinite_iterator<Fib> ii(fib);

for( int i = 0; i < 10; ++i )
    std::cout << *ii++;
```

The struct `Fib` is a generator functor that generates the elements of the sequence. The `infinite_iterator` has one template argument: the type of the generator functor. The type of the generated element is deduced by the compiler of the signature of the functor's member function `operator()`. Its constructor receives only the functor object.

3. C++0x-based approach

The code in the previous subsection works fine, but the functor has to deal with the whole process of generating elements of the sequence. Besides calculation of the next element, it has to take care to save the previous elements which are playing role in computation of the following one. However, the process of saving previous

elements is mostly independent of the generated sequence. The only operative question is the number of the previous elements that are needed to compute the next one.

With the features of the new standard of C++, we can provide a more sophisticated `infinite_iterator` that makes easier of writing functors. Our new `infinite_iterator` is able to save the previous elements in a sequence and feed the functor with them. That way the functor only need to take care of computing the next element of sequence. The number of the saved previous elements depends on the number of the functor's `operator()` arguments. The constructor of `infinite_iterator` needs the same number of initial values of the sequence. On increase of `infinite_iterator`, the stored elements are passed to the functor as arguments.

The oldest will be the first argument, and the previously calculated will be the last one. Then the functor computes the next element and returns it.

See the code snippet below, which simplifies the example in previous subsection.

```
struct Fib
{
    int operator()(int a, int b) const
    {
        return a + b;
    }
};

Fib fib;
infinite_iterator<Fib> ii(fib);

for( int i = 0; i < 10; ++i )
    std::cout << *ii++;
```

Now the struct `Fib` needs to take care of the computation of the next element only. Every other is done by the `infinite_iterator`.

Our solution supports the lambda expressions, which are introduced by the C++0x [4]. Lambda expression is also accepted in place of functors [21]. The code snippet below shows the way to apply lambda expression with `infinite_iterator`.

```
auto fib = [](int a, int b){return a + b;};
infinite_iterator<decltype(fib)> ii(fib);
```

4. Specializing by the arity of functors

In C++0x realm the `infinite_iterator` is able to distinguish between the functors by their arity. The arity of a functor is the number of the arguments of its `operator()`. With nullary functor, the `infinite_iterator` does not save the previous elements. That case all the computation process is done by the functor, like

in the example in section 2. With unary, binary, trinary, etc. functors our iterator saves the previous one, two, three, etc. elements, and feeds the functor with these values on computation of the next element, like the example in subsection 3.

The code skeleton below shows the way we specializing `infinite_iterator` by the arity of functor. The template argument E is related to the stoppage of the generation of infinite sequence and it is detailed in 5.

```
template<class T>
struct infinite_iterator_base :
    std::iterator<
        std::input_iterator_tag,
        T>
{
  /* the common functionality
     is implemented here */
};

template<
    class G,
    class E = not_specified,
    class P = decltype(&G::operator())>
struct infinite_iterator
{
};

template<class G, class E, class T>
struct infinite_iterator<
    G,
    E,
    T (G::*)()> :
    infinite_iterator_base<T>
{
  /* specialization for nullary functor */
};

template<class G, class E, class T>
struct infinite_iterator<
    G,
    E,
    T (G::*)(T) const> :
      infinite_iterator_base<T>
{
 /* specialization for unary functor */
};
```

```
template<class G, class E, class T>
struct infinite_iterator<
    G,
    E,
    T (G::*)(T, T) const> :
      infinite_iterator_base<T>
{
 /* specialization for binary functor */
};

/* similarly for n-ary functor */
```

The struct `infinite_iterator_base` implements the common functionality of the different `infinite_iterator` specializations, hence these specializations are inherited from the `infinite_iterator_base`. Different specialization belongs to the functors with different arity from 0 to `MAX_ARITY`. `MAX_ARITY` is a preprocessor macro and it sets the upper limit of the supported functor arity. We generate the different specializations from a template using *Boost.Preprocessor* library [24]. Our solution is similar to the way that the *Boost.MPL* library [25] is implemented. While the different arities of functors require different functionalities, thus the general version of `infinite_iterator` is not used and its body remains blank.

5. Stoppage of generation

The `infinite_iterator`s generate an infinite sequence. However, in real problems a finite subsequence of elements is required. Our solution provides *end iterator* to determine a finite range of an infinite sequence. The *end iterator* can be created in two ways.

- *By a constructor with one integer argument:* The argument specifies the length of the finite subsequence.
- *By a constructor with a predicate as its argument.* With this kind of end iterator, the generation of the elements in an infinite range is stopped, when the predicate returns false for the currently generated element. Using this version of `infinite_iterator`, the type of the predicate must be specified as the second template argument during the instantiation of either *normal* or *end* iterator.

The example above fills two arrays of integers `(t, r)` with the numbers from 1 to 10. For array `t` the first kind of end iterator is used, while for array r, the second one is chosen.

```
struct ints
{
    int operator()(int a) const
```

```
    {
        return a + 1;
    }
};

struct pred
{
    pred(int i) : max(i) {}
    bool operator()(int a) const
    {
        return a < max;
    }
private:
    int max;
}

int t[10];
int r[10];

infinite_iterator<ints> tb(ints(), 0);
infinite_iterator<ints> te(10);

infinite_iterator<ints, pred> br(ints(), 0);
infinite_iterator<ints, pred> re(pred(11));

copy(tb, te, t);
copy(rb, re, r);
```

The `infinite_iterator` created by a default constructor is also an end iterator, however, this one represents a real infinite range, thus the generation of the elements needs to be stopped in other way.

6. Supported infinite ranges

While the programmer can apply any kind of functor object to our library, we support a large variety of infinite ranges. The only restriction is that the generated elements must be copy constructable and assignable. (STL also requires this concept.)

With the help of the predefined functors of STL, the most commonly used infinite ranges can be defined without writing any user defined functors. Infinite iterators utilized by:

- functor `binder2nd<plus<int> >(plus<int>(), 1)` with 0 as initial number generates the natural numbers

- functor `binder2nd<plus<int> >(plus<int>(), 2)` with 0 as initial number generates the positive even numbers
- functor `binder2nd<plus<int> >(multiplies<int>(), 2)` with 1 as initial number generates the powers of 2
- functor `plus<int>()` with 1 and 1 as initial numbers generates the Fibonacci numbers
- etc.

Although the usage of the functors provided by the STL covers the most commonly used ranges, our library provides additional functors allowing the user to define infinite ranges easier. Our functors are:

- `inc<T>` which increases an element by prefix `operator++`.
- `dec<T>` which decreases an element by prefix verb|operator–|.
- `constant<N>` which returns always N, thus it can be used to define infinite range of the same elements.

While it is possible to generate any kind of infinite range which elements are copy constructable and assignable, for effeciency reasons our solution mainly focuses on those ranges, where the next element can be determined by the finite number of previous elements. In the latter case the functor itself has to take care about all the generation process. It is a common design rule in STL that the inefficient methods are not supported, for example, there is no index operator for `list`, or `push_front` member function is missing in `vector`.

7. Related work

One known extension of the STL is the View Template Library, which provides a layer over the C++ Standard Template Library [14]. It consists of *views* that are container adaptors, and *smart iterators* that are a kind of iterators provided a different view onto the data that are pointed to by the iterator. Views wrap the container to be filtered and transformed the elements on which the view operates. These transformations and filterings are done during the execution, without taking effect the stored data in the container. The interface provided by the views is a container interface.

Although View Template Library provides *views* that filters the elements on a range its usage is limited to the containers. We cannot filter ranges defined by those iterators which are not belongs to the container. Thus a simple problem: to copy the odd numbers from the standard input to the standard output is not soluble.

The Boost Iterator Library [26] is an extension to the STL with a variety of useful iterator types. It contains two parts. The first one is a system of concepts

which extend the C++ standard iterator requirements. The second one is a framework of components for building iterators based on these extended concepts and includes several useful iterator adaptors, such as `filter_iterator`, which traverses only those elements, which satisfy a given requirement; `counting_iterator`, which generates a sequence of an elements; `function_input_iterator`, which invokes a nullary function on dereference operation, etc. The extended iterator concepts have been carefully designed so that old-style iterators can fit in the new concepts and so that new-style iterators will be compatible with old-style algorithms, although algorithms may need to be updated if they want to take full advantage of the new-style iterator capabilities. Several components of this library have been accepted into the C++ standard technical report.

Our solution unifies and extends the functionality of `counting_iterator` and `function_input_iterator` as it is able to accept an arbitray arity of functors. Besides the `infinite_iterator` is able to cooperate the other iterator adaptors of the Boost Library.

Our `infinite_iterator`s can be adapted by `filter_iterator` of Boost Library. It is useful, when only elements with a specific property are needed from an infinite sequence. Separating the condition of the specific property and the general generation method may highly simplify to define special infinite sequences. Let us suppose someone needs the infinite sequence of odd Fibonacci numbers. As the addition of last two odd Fibonacci numbers is not the next odd Fibonacci number, the generator functor has to deal with the even Fibonacci numbers as well. However, as the process of checking the number is odd is moved to `filter_iterator`, the generator functor can be a simple Fibonacci sequence generator as in 3.

The example below prints the first ten odd Fibonacci number to the standard output.

```
typedef infinite_iterator<Fib> inf_fib;
inf_fib ib(Fib(), 0, 1);

IsOdd pred;
boost::filter_itertator<IsOdd, inf_fib> fb(pred, ib);

for( int i = 0; i < 10; ++i )
{
    std::cout << *fb++;
}
```

8. Conclusion

C++ Standard Template Library is the most widely-used library based on the generic programming paradigm. It consists of handy containers and general, reusable algorithms. Iterators bridge the gap between containers and algorithms, so algorithms do not depend on the used container. Adaptors are also an impor-

tant part of the STL, which can change the behaviour of the STL components for special situations.

However, there are functionalities that are missing from the library. Although iterators play a main role in the library, the several features that are make the programmer work easier and fail-safe are missing or limitedly supported.

It this paper we have prompted that the infinite ranges have only a very limited support either in the STL itself or in the other third party libraries, too. We presented a comfortable extension for the STL which supports the usage of infinite ranges in a general way. With the support of the incoming new standard of C++ our library become an highly customizable, easy to use, library which is able to cooperate either the STL or the iterator extensions of the other third party libraries, too.

References

[1] Alexandrescu A., Modern C++ Design, *Addison-Wesley* (2001).

[2] Austern, M. H., Generic Programming and the STL: Using and Extending the C++ Standard Template Library, *Addison-Wesley* (1998).

[3] Buss, A., Fidel, Harshvardhan, Smith, T., Tanase, G., Thomas, N., Xu X., Bianco, M., Amato, N. M., Rauchwerger, L. The STAPL pView, *Proc. of Languages and Compilers for Parallel Computing (LCPL 2010), Lecture Notes in Comput. Sci.* **6548** (2011), 261–275.

[4] Järvi, J., Freeman, J., C++ Lambda Expressions and Closures, *Sci. Comput. Programming* **75(9)** (2010), 762–772.

[5] Király, R., Kitlei, R., Application of Complexity Metrics in Functional Languages, *Proc. of the 8th Joint Conference on Mathematics and Computer Science, Selected Papers*, 267–282.

[6] Kiselyov, O., Functional Style in C++: Closures, Late Binding, and Lambda Abstractions, *Proc. of the third ACM SIGPLAN international conference on Functional programming (ICFP '98)* (1998), p. 337.

[7] McNamara, B., Smaragdakis, Y., Functional Programming with the FC++ Library, *Journal of Functional Programming* **14(4)** (2004), 429–472.

[8] McNamara, B., Smaragdakis, Y., Functional Programming in C++, *Proc. of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00)*, 118–129.

[9] Meyers, S., Effective STL - 50 Specific Specific Ways to Improve Your Use of the Standard Template Library, *Addison-Wesley* (2001).

[10] Pataki, N., Advanced Functor Framework for C++ Standard Template Library, *Stud. Univ. Babeş-Bolyai, Inform.* **LVI(1)** (2011), 99–113.

[11] Pataki, N., Porkoláb, Z., Istenes, Z., Towards Soundness Examination of the C++ Standard Template Library, *Proc. of Electronic Computers and Informatics* (2006), 186–191.

[12] Pataki, N., Szűgyi, Z., Dévai, G., Measuring the Overhead of C++ Standard Template Library Safe Variants, *Electronic Notes in Theoret. Comput. Sci.* **264(5)** (2011), 71–83.

[13] Porkoláb, Z.: Functional Programming with C++ Template Metaprograms, *Proc. of Central European Functional Programming School (CEFP 2009), Lectures Notes in Comput. Sci.* **6299** (2010), 306–353.

[14] Powell, G., Weiser, M., A New Form of Container Adaptors, *C/C++ Users Journal* **18(4)**, (April 2000) 40–51.

[15] Sipos, Á., Pataki, N., Porkoláb, Z., Lazy Data Types in C++ Template Metaprograms, *Proc. of 6th International Workshop on Multiparadigm Programming with Object-Oriented Language 2007 (MPOOL' 07)* (2007).

[16] Sipos, Á., Porkoláb, Z., Pataki, N., Zsók V., Meta<Fun> – Towards a Functional-Style Interface for C++ Template Metaprograms, *Proc. of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007)* (2007), 489–502.

[17] Slodičák, V., Szabó Cs., Recursive Coalgebras in Mathematical Theory of Programming, *Proc. of the 8th Joint Conference on Mathematics and Computer Science, Selected Papers*, 385–394.

[18] Smetsers, S., van Weelden, A., Plasmeijer, R., Efficient and Type-Safe Generic Data Storage, *Electronic Notes in Theoret. Comput. Sci.*, **238(2)** (2009), 59–70.

[19] Stroustrup, B., The C++ Programming Language (Special Edition), *Addison-Wesley* (2000).

[20] Szűgyi, Z., Török, M., Pataki, N., Towards a Multicore C++ Standard Template Library, *Proc. of Workshop on Generative Technologies (WGT 2011)* (2011), 38–48.

[21] Szűgyi, Z., Török, M., Pataki, N., Kozsik, T., Multicore C++ Standard Template Library with C++0x, *Proc. of NUMERICAL ANALYSIS AND APPLIED MATHEMATICS ICNAAM 2011: International Conference on Numerical Analysis and Applied Mathematics*, 857–860.

[22] Tanase, G., Buss, A., Fidel, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Thomas, N., Xu, X., Mourad, N., Vu, J., Bianco, M., Amato, N. M., The STAPL Parallel Container Framework, *Proc. of ACM SIGPLAN Symp. Prin. Prac. Par. Prog (PPoPP 2011)* (2011), 235–246.

[23] Torgersen, M., The Expression Problem Revisited – Four New Solutions Using Generics, *Proceedings of European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Comput. Sci.* **3086** (2004), 123–143.

[24] `http://www.boost.org/doc/libs/1_46_1/libs/preprocessor/doc/`

[25] `http://www.boost.org/doc/libs/1_47_0/libs/mpl/doc/index.html`

[26] `http://www.boost.org/doc/libs/1_47_0/libs/iterator/doc/index.html`