SAFE ITERATOR FRAMEWORK FOR THE C++ STANDARD TEMPLATE LIBRARY

Norbert PATAKI
Department of Programming Languages and Compilers, Eötvös Loránd University,
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary,
e-mail: patakino@elte.hu

ABSTRACT

The C++ Standard Template Library is the flagship example for libraries based on the generic programming paradigm. The usage of this library is intended to minimize classical C/C++ errors, but does not warrant bug-free programs. Furthermore, many new kinds of errors may arise from the inaccurate use of the generic programming paradigm, like dereferencing invalid iterators or misunderstanding remove-like algorithms.

In this paper we present some typical scenarios that may cause undefined or weird behaviour. We present approaches that can be used for developing different safe iterators to avoid run-time errors. Some of these iterators are able to manipulate the container directly, hence they cannot result in undefined behaviour when an algorithm needs to add elements to the container or delete elements from the container. Our iterators are able to indicate if they are invalid. Algorithms' preconditions are evaluated with our iterators.

Keywords: *C++, STL, iterators, safety*

1. INTRODUCTION

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach. 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 containers [24]. C++ STL is widely-used because it is a very handy, standard C++ library that contains beneficial containers (like list, vector, map, etc.), a lot of algorithms (like sort, find, count, etc.) among other utilities.

The STL was designed to be extensible [13]. We can add new containers that can work together with existing algorithms. On the other hand, we can extend the set of algorithms with a new one that can work together with existing containers. Iterators bridge the gap between containers and algorithms [4]. The expression problem [26] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [1]. Adaptors can modify the interface of a container, transform streams into iterators, modify the behavior of functors etc.

However, the usage of C++ STL does not mean bugless or error-free code [8]. Contrarily, incorrect application of the library may introduce new types of problems [16].

One of the problems is, that the error diagnostics are usually complex, and very hard to figure out the cause of a program error [27,28]. Violation of the requirement of strict weak ordering in comparison functors also means strange bugs [10]. This results in incosistent containers at runtime.

Most of the properties are checked at compilation time. For example, the code does not compile if one uses sort algorithm with the standard list container, because the list's iterators do not offer random accessibility [12]. Other properties are checked at runtime [22]. For example, the standard vector container offers an `at` method which tests if the index is valid and it raises an exception otherwise [19].

Unfortunately, there is still a large number of some properties are tested neither at compilation-time nor at run-time. Observance of these properties are in the charge of the programmers. Let us consider the following code snippet:

```
std::vector<int> v;
int x;
//...
std::vector<int>::iterator i =
  std::lower_bound( v.begin(), v.end(), x );
```

The purpose of `lower_bound` is to find an element in an ordered range. It is a version of binary search, hence it has logarithmic complexity. We assume that we can find an element in a vector in logarithmic time because of the sortedness of the vector. However, it causes undefined result, if the vector is not ordered [21]. Implementations of these algorithms do not test if the range is sorted appropriately. Many STL algorithms expect ordered range: `equal_range`, `binary_search`, `set_difference`, etc.

Furthermore, sortedness of container is not enough. We must make sure that the same sorting function object is used for sorting and for searching. The following code snippet also results in undetermined behaviour:

```
std::vector<int> v;
int x;
//...
std::sort( v.begin(), v.end() );
std::vector<int>::iterator i =
  std::lower_bound( v.begin(), v.end(), x,
                    std::greater<int>() );
```

Other typical STL-related mistakes are related to iterator invalidation. This problem occurs when a container that is being processed using an iterator has its shape changed during the process, for example anything that causes a vector's reallocation (increase in the result of `capacity()`) will invalidate all iterators. When one use an invalid iterator also causes an undefined result [9]. Let us consider the following code:

```
std::vector<int> v;
```

```
//...

std::vector<int>::iterator i = v.begin();

// ...
// vector's capacity has been changed...
std::cout << *i;
```

When `*i` is referred, it causes undefined result because `i` has become invalid.

STL's `copy` and `transform` algorithm can be used to copy an input range of objects into a target range. These algorithms neither allocate memory space nor call any specific inserter method while coping elements. They assume that the target has enough, properly allocated elements where they can copy elements with `operator=`. Inserter iterators can enforce to use `push_back`, `push_front` or `insert` method of containers. But these algorithms cannot copy elements into an empty list, for instance. They do not know how to insert elements into the empty container. The following code snippet can be compiled, but it results in an undefined behaviour [20]:

```
std::list<int> li;
std::vector<int> vi;
v.push_back( 3 );

std::copy( vi.begin(),
           vi.end(),
           li.begin() );
```

However, there are some adaptors in the library to overcome this situation: `back_inserter` and `front_inserter` adaptors. On the other hand, they cannot change the elements of a container, only add new element to the container [15]. However, we have developed a technique that is able to emit compilation warnings if the usage of the `copy` algorithm may be erroneous [18].

Another common mistake is related to removing algorithms. The algorithms are container-independent, hence they do not know how to erase elements from a container, just relocate them to a specific part of the container, and we need to invoke a specific erase member function to remove the elements phisically. Therefore, for example the `remove` and `unique` algorithms do not actually remove any element from a container [15]. Let us consider the following code snippet:

```
std::vector<int> v;
for( int = 1; i <= 10; ++i )
  v[i] = i;

v[3] = v[5] = v[7] = 99;

std::remove( v.begin(), v.end(), 99 );

std::cout << v.size();
```

In contrast to the name of the algorithm, the size of the container is unchanged. The remaining elements have been moved to front of the container, but the tail is also unchanged. The result of this algorithm may be counter-intuitive at first time. The proper usage of the remove is called *erase-remove idiom*:

```
v.erase( std::remove( v.begin(),
                      v.end(),
                      99 ),
         v.end() );
```

Whereas C++ STL is pre-eminent in a sequential realm, it is not aware of multicore environment [3]. For example, the Cilk++ language aims at multicore programming. This language extends C++ with new keywords and one can write programs for multicore architectures easily. However, the language does not contain an efficient multicore library, just the C++ STL only which is an efficiency bottleneck in multicore environment. We develop a new STL implementation for Cilk++ to cope with the challenges of multicore architectures [25]. This new implementation can be a safer solution, too. Hence, our safety extensions will be included in the new implementation. However, the techniques presented in this paper concern to the original C++ STL, too.

In this paper we present extensions of the C++ STL, that is able to check iterators' validness at runtime. We also describe a technique that can use generic algorithms on sorted intervals in a safer way. Erasable and copy-safe iterators are introduced in order to overcome some typical mistakes.

This paper is organized as follows. In section 2 a modification in the related traits type is advised to be taken advantage of the following sections. We provide an approach for checking algorithms' preconditions in section 3. We argue for a solution to the iterator invalidation in section 4. Erasable iterators are introduced and present in section 5. We provide our copy-safe iterators in section 6. Finally, this paper concludes in section 7.

2. EXTENSION OF ITERATOR TRAITS

In this paper we argue for some new kinds of iterators that subserve the correct usage of the STL. This requires new traits to make the compilers able to make decisions about iterators'usage. The following empty types describes if an iterator is *erasable* and *precondition-safe*.

```
struct erasable{};
struct unerasable{};

struct precond_safe{};
struct precond_unsafe{};
```

Next, two new traits added to `iterator_traits`. A similar approach is available [18]. The first traits specifies if an iterator is erasable, the second one specifies if an iterator is precondition-safe. The default `iterator_traits` is the following:

```
template <class T>
struct iterator_traits
{
  typedef typename T::iterator_category
    iterator_category;
```

```
  typedef typename T::value_type
     value_type;
  typedef typename T::difference_type
     difference_type;
  typedef typename T::pointer
     pointer;
  typedef typename T::reference
     reference;
  typedef unerasable
     erasability;
  typedef typename T::precond_safety
     precond_safety;
};
```

The new kind of traits is defined as `erasability` and `precond_safety` type synonyms. As the default case shows, the ordinary iterators are not erasable and not precondition-safe. These traits can be set by the trivially modified `iterator` base class. In the case of erasable iterators, the erasibility tag must be set to `erasable`, and in the case of precondition safe iterators, the precondition safety property must be set to `precond_safe`. In every other case it must be set to the default.

3. PRECONDITIONS OF ALGORITHMS

The STL algorithms work well only if its preconditions are satisfied. Typically, the algorithm which require sorted input, check if the input is sorted neither at compilation-time nor at run-time. If this requirement is violated the result of algorithm is undefined.

We provide precondition-safe iterator adaptor to overcome this situation. The implementation is the following:

```
template <class T>
struct Precond_safe: T
{

  Precond_safe( T t ): T( t ) { }

  typedef precond_safe precond_safety;
};
```

The adaptor is based on the mixin technique, the type of the base class is the iterator type itself [23]. Thus, `Precond_safe` provides all the properties and operations just like the original iterator only the safety type is defined to `precondition_safe`. Algorithms can be overloaded on this type information. The following template method can be used to deduce the parameter:

```
template <class T>
Precond_safe<T> Precond( T t )
{
  return Precond_safe<T>( t );
}
```

To present this technique the safe implementation of `lower_bound` is shown.

The following is the type of exception to indicate the erroneous usage of the algorithm:

```
class not_sorted{};
```

The standard algorithm implementation checks if the iterator is precondition-safe:

```
template <class It, class T>
It lower_bound( It first,
                It last,
                const T& t )
{
  return lower_bound(
    first,
    last,
    t,
    typename
      iterator_traits<It>::
        precond_safety() );
}
```

The precondition-safe version checks the precondition. An exception is raised if the precondition fails, otherwise calls the original implemention:

```
template <class Iterator, class T>
Iterator lower_bound( Iterator first,
                      Iterator last,
                      const T& t,
                      precond_safe )
{
  if ( !std::is_sorted(first, last ) )
  {
    throw not_sorted();
  }

  return lower_bound( first,
                      last,
                      t,
                      precond_unsafe() );
}
```

The original version is the precondition-unsafe one:

```
template <class Iterator, class T>
Iterator lower_bound( Iterator first,
                      Iterator last,
                      const T& t,
                      precond_unsafe )
{
  // original implementation...
}
```

If the adapter is in-use, the safe implementation works. The conversion trivially works:

```
std::vector<int> v;
int x;
// ...
std::vector<int>::iterator i =
  std::lower_bound( Precond( v.begin() ),
                    Precond( v.end() ),
                    x );
```

The user-defined predicated version is straightforward, just an other template parameter must be used. This technique is able to check arbitrary precondition of arbitrary algorithm.

4. INVALID ITERATORS

In this section we present a technique that can be used to avoid the undefined behaviour of invalid iterators' usage. The technique is adaptable for all standard and nonstandard containers. Different containers invalidate iterators in different ways, however, this technique can be transformed to `list`, `deque` or other third party defined containers too. In a more sophisticated solution the invalidation behaviour should be parametrized. We present the technique as an extension of STL's vector template.

In our implementation the vector objects keep tracks their iterators which have a member to describe if the iterator is valid. When the vector reallocates itself, it sends a message to its iterators that they become invalid. If one accesses an element via an invalid iterator, then an exception is raised. Since STL always creates copies from the iterators, we have to keep them on the heap memory. We use the `shared_ptr` to avoid memory-leaks which is the part of the C++11, and it is the part of Boost library [15].

Let us consider the following code snippet:

```
template <class T,
          class Alloc = std::allocator<T>,
          bool debug = false>
class vector
{

  typedef ItCont
    std::list<shared_ptr<iterator_impl> >;


  T* p;
  int cap, s;

  ItCont iterators;

public:

  struct iterator_impl
  {
  private:
     bool isvalid;
     T* curr;
  public:
     iterator_impl( T* c ) : curr(c),
                             isvalid( true )
                             {}

     T& operator*()
     {
        if ( !isdebug )
          return *curr;

        if( isvalid )
          return *curr;
        else
          throw invalid_iterator();
     }

     iterator_impl& operator++()
     {
       ++curr;
       return *this;
     }

     iterator_impl operator++( int )
     {
       iterator_impl tmp( *this );
       ++curr;
       return tmp;
     }

     // ...

  };

  struct iterator:
   std::iterator<
     std::random_access_iterator_tag,
     T>
  {
    iterator_impl* p;

    // delegates
    // iterator_impl's operations
  };


private:
  void realloc()
  {
    cap*=2;
    T* t = new T[cap];
    std::copy( p, p + s, t );
    delete [] p;
    p = t;
  }

  void invalid()
  {
    for( typename ItCont::iterator it =
           iterators.begin();
         it != iterators.end();
         ++it)
    {
      (*it)->isvalid = false;
    }
  }

public:
  vector(): cap( 1 ), s( 0 )
  {
    p = new T[cap];
  }
```

```
  vector()
  {
    delete [] p;
  }

  void push_back( const T& a )
  {
    if ( s < cap )
      p[ s++ ] = a;
    else
    {
      realloc();
      invalid();
      push_back( a );
    }
  }

  iterator begin()
  {
    iterator_impl* x =
      new iterator_impl( p );
    iterators.push_back( x );
    return iterator( x );
  }

  iterator end()
  {
    iterator_impl* x =
      new iterator_impl( p + s );
    iterators.push_back( x );
    return iterator( x );
 }

  // ...

};
```

Of course, the testing can depend on a preprocessor macro or something else. Legacy STL-based codes can be easily transformed to use this vector container with extra checks. Just an extra parameter should be passed to the vector type. However, there is no trivial assignment and copy between an untested and tested vector container, but a special template copy constructor and assignment operator can be added.

Naturally, we can create a specialization for the safe and unsafe versions. This makes our implementation faster.

Similarly, we can create a safe iterator implementation that is able to pursue the vector's pointer. In this case, an exception is thrown when an iterator is referred which point at an erased element.

It is also should be considered if invalidation includes the end iterators. Also causes runtime problems if end iterators are dereferenced. It can be handled in an orthogonal way.

5. ERASABLE ITERATORS

In this section we present our approach to develop iterators that are able manipulate the container and remove elements from it [4].

First, we add a new inner class to the `vector` container. This class is called `erasable_iterator`: this is quite similar to the standard `iterator` class, but it has a *pointer* to the container and a new member function called `erase`. This method accesses the member functions of the container via the pointer. Only point is that the method has to avoid invalidation of the iterator. The container's member function `ebegin` returns an erasable iterator to the first element, and its method `eend` returns an erasable iterator to the end of the sequence, respectively.

```
typedef
  std::random_access_iterator_tag
  ran_acc_tag;

template <class T,
          class Alloc = std::alloc<T> >
class vector
{
  T* p;
  int s, cap;
  // usual vector's members, typedefs,
  // classes, operators

public:

  class iterator:
    public
      std::iterator<ran_acc_tag,
                    T>
  {
  protected:
     T* p;
     // usual operators...
  };

  class erasable_iterator: public iterator
  {
    vector<T, Alloc>* v;
  public:
    erasable_iterator( iterator i,
                       vector<T, Alloc>* vt ) :
      iterator( i ), v( vt ) { }

    void erase()
    {
      T* tmp = iterator::p + 1;
      v->erase( *this );
      iterator::p = tmp;
    }
  };

  erasable_iterator ebegin()
  {
    return erasable_iterator( begin(), this );
  }

  erasable_iterator eend()
  {
    return erasable_iterator( end(), this );
```

```
  }

};
```

This technique can be transformed to other containers, too.

However, the standard algorithms of the STL do not known the notation of erasable iterators. Thus, we have to write new algorithms that take advantage of this new kind of iterators. An algorithm can be decide if it uses erasable iterator based on the extended traits. In the case of erasable iterators the algorithm is able to use the `erase` method.

```
template <class It, class T>
It remove( It first,
           It last,
           const T& t )
{
  return remove( first,
                 last,
                 t,
                 typename
                   std::iterator_traits<It>
                      ::erasability() );
}
```

The erasable version can be implemented in the following way:

```
template <class Iter, class T>
void remove( Iter first,
             Iter last,
             const T& t,
             erasable )
{
  while( first != last )
  {
    if ( t == *first )
    {
      first.erase();
    }
    else
    {
      ++first;
    }
  }
  return first;
}
```

The version that uses unerasable iterators is the same as the original implementation.

6. COPY-SAFE ITERATORS

In this section we present our implementation of copy-safe [17] iterators.

This iterator type is also similar to `iterator` type of the container. This kind of iterator also has a *pointer* to the container. When a safe pointer is dereferenced (ie. its `operator*` is called, it can invoke the container's `push_back` method and add new element to the `vector` if necessary. Hence, if this kind of iterators is in use it causes no runtime problems if someone copies elements into an empty `vector`. Our implementation is able to detect if the client uses problematic iterators for copying ranges [18]. The container's member function `cbegin` returns a copy-safe iterator to the first element, and its method `cend` returns a copy-safe iterator to the end of the sequence, respectively.

```
template <class T,
          class Alloc = std::allocator<T> >
class vector
{
  // usual members, methods, typedefs, etc.

  class copy_safe_iterator: public iterator
  {
    vector<T, Alloc>* v;
  public:
    copy_safe_iterator( iterator i,
                        vector<T, Alloc>* vt ) :
      iterator( i ), v( vt ) { }

    T& operator*()
    {
      if ( *this == v->end() )
      {
        v->push_back( T() );
        iterator::p = &( v->back() );
      }
      return iterator::operator*();
    }
  };

  copy_safe_iterator csbegin()
  {
    return copy_safe_iterator( begin(), this );
  }

  copy_safe_iterator csend()
  {
    return copy_safe_iterator( end(), this );
  }

};
```

This technique can be transformed to other containers, too.

Modification of any algorithms is not necessary because these iterators can be work with standard copying algorithms, such as `copy` or `transform`. For instance, the following code snippet shows the usage that cannot be implemented in the original STL way:

```
std::vector<int> vi;
// ...
std::list<int> li;
// ...
std::copy( li.begin(),
           li.end(),
           vi.csbegin() );
```

This invokation of `copy` algorithm overwrites all the existing elements in the `vector`, and added more new elements to the vector, if necessary. To achieve this goal without copy-safe iterators is much more harder.

However, limitations can be mentioned with this approach. However, `vector` does not offer `push_front` method, the `copy_iterator` should be parametrized with strategy of adding new element to container. *Function objects* (also known as *functors*) make the library much more flexible without significant runtime overhead. They parametrize user-defined algorithms in the library, for example, they determine the comparison in the ordered containers or define a predicate to find.

The iterator always executes a check when it is dereferenced, it has runtime overhead. However, it guarantees safety, and original non-copier iterators are available, too. The runtime overhead should be measured [20].

7. CONCLUSION

STL is the most widely-used library based on the generic programming paradigm. STL increases efficacy of C++ programmers mightily because it consists of expedient containers and algorithms. It is efficient and convenient, but the incorrect usage of the library results in weird or undefined behaviour.

In this paper we present some examples that can be compiled, but at runtime their usage is defective. We argue for some new extensions to overcome these risky situations. New kind of iterators are presented as a solution. The limitations of these iterators are also discussed.

ACKNOWLEDGEMENT

The Project 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).

REFERENCES

[1] ALEXANDRESCU, A.: Modern C++ Design, Addison-Wesley, Reading, MA., 2001.

[2] AUSTERN, M. H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley, Reading, MA., 1998.

[3] AUSTERN, M. H. – TOWLE, R. A. – STEPANOV, A. A.: *Range partition adaptors: a mechanism for parallelizing STL*, ACM SIGAPP Applied Computing Review **4**, No. 1 (1996) 5–6.

[4] BECKER, T.: *STL & generic programming: writing your own iterators*, C/C++ Users Journal **19**, No. 8 (2001) 51–57.

[5] BICZÓ, M. – PÓCZA K. – FORGÁCS, I. – PORKOLÁB, Z.: *A New Concept of Effective Regression Test Generation in a C++ Specific Environment*, Acta Cybernetica **18**, No. 3 (2008) 408–501.

[6] CZARNECKI, K. – EISENECKER, U. W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading, MA., 2000.

[7] DAS, D. – VALLURI, M. – WONG, M. – CAMBLY, C.: *Speeding up STL Set/Map Usage in C++ Applications*, Lecture Notes in Computer Science **5119**, No. 1 (2008) 314–321.

[8] DÉVAI, G. – PATAKI, N.: *Towards verified usage of the C++ Standard Template Library*, In Proc. of The 10th Symposium on Programming Languages and Software Tools (SPLST) 2007, pp. 360–371.

[9] DÉVAI, G. – PATAKI, N.: *A tool for formally specifying the C++ Standard Template Library*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **31**, No. 1 (2009) 147–166.

[10] GREGOR, D. – JÄRVI, J. – SIEK, J. – STROUSTRUP, B. – DOS REIS, G. – LUMSDAINE, A.: *Concepts: linguistic support for generic programming in C++*, in Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006), pp. 291–310.

[11] GREGOR, D. – SCHUPP, S.: *STLlint: lifting static checking from languages to libraries*, Software - Practice & Experience **36**, No. 3 (2006) 225-254.

[12] JÄRVI, J. – GREGOR, D. – WILLCOCK, J. – LUMSDAINE, A. – SIEK, J.: *Algorithm specialization in generic programming: challenges of constrained generics in C++*, in Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2006), pp. 272–282.

[13] KOLLÁR, J. – PORUBÄN, J.: *Building Adaptive Language Systems*, INFOCOMP - Journal of Computer Science **7**, No. 1 (2008) 1–10.

[14] MATSUDA, M. – SATO, M. – ISHIKAWA, Y.: *Parallel Array Class Implementation Using C++ STL Adaptors*, Lecture Notes in Computer Science **1343**, No. 1 (1997) 113–120.

[15] MEYERS, S.: Effective STL - 50 Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley, Reading, MA., 2001.

[16] PATAKI, N.: *Advanced Functor Framework for C++ Standard Template Library*, Studia Univ. Babeş-Bolyai, Informatica **LVI**, No. 1 (2011) 99–113.

[17] PATAKI, N.: *Advanced Safe Iterators for the C++ Standard Template Library*, in Proc. of the Eleventh International Conference on Informatics, Informatics 2011, pp. 86–89.

[18] PATAKI, N. – PORKOLÁB, Z.: *Extension of Iterator Traits in the C++ Standard Template Library*, In Proc. of the Federated Conference on Computer Science and Information Systems, pp. 911-914.

[19] PATAKI, N. – PORKOLÁB, Z. – ISTENES, Z.: *Towards Soundness Examination of the C++ Standard Template Library*, In Proc. of Electronic Computers and Informatics, ECI 2006, pp. 186–191.

[20] PATAKI, N. – SZŰGYI, Z. – DÉVAI, G.: *Measuring the Overhead of C++ Standard Template Library Safe Variants*, Electronic Notes in Theoretical Computer Science **264**, No. 5 (2011) 71–83.

[21] PATAKI, N. – SZŰGYI, Z. – DÉVAI, G.: *C++ Standard Template Library in a Safer Way*, In Proc. of Workshop on Generative Technologies 2010 (WGT 2010), pp. 46–55.

[22] PIRKELBAUER, P. – PARENT, S. – MARCUS, M. – STROUSTRUP, B.: *Runtime Concepts for the C++ Standard Template Library*, In Proc. of the 2008 ACM Symposium on Applied Computing, pp. 171–177.

[23] SMARAGDAKIS, Y. – BATHORY, D.: *Mixin-Based Programming in C++*, Lecture Notes in Computer Science **2177**, No. 1 (2000) 164–178.

[24] STROUSTRUP, B.: The C++ Programming Language. Addison-Wesley, Reading, MA., 1999.

[25] SZŰGYI, Z. – TÖRÖK, M. – PATAKI, N.: *Multicore C++ Standard Template Library in a Generative Way*, Electronic Notes in Theoretical Computer Science. **279**, No. 3 (2011) 63–72.

[26] TORGERSEN, M.: *The Expression Problem Revisited – Four New Solutions Using Generics*, Lecture Notes in Computer Science **3086**, No. 1 (2004) 123–143.

[27] ZOLMAN, L.: *An STL message decryptor for visual C++*, C/C++ Users Journal **19**, No. 7 (2001) 24–30.

[28] ZÓLYOMI, I. – PORKOLÁB, Z.: *Towards a General Template Introspection Library*, Lecture Notes in Computer Science **3286**, No. 1 (2004) 266–282.

Received January 2, 2012, accepted March 28, 2012

BIOGRAPHY

Norbert Pataki was born on 26. 2. 1982. In 2006 he graduated (MSc) at the department of Eötvös Loránd University (ELTE), Budapest. He takes part many industrial projects during his PhD. In 2009 he has become assistant professor at Eötvös Loránd University. His research area includes programming languages (especially the C++ programming language), multicore programming, software metrics, and generative programming.