C++ Function Object Binders Made Easy

Jaakko Järvi *

Turku Centre for Computer Science
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
jaakko.jarvi@cs.utu.fi

Abstract. A novel argument binding mechanism that can be used with STL algorithm invocations is proposed. Without using any adaptors, binding can be applied directly to pointers to nonmember functions, pointers to const and nonconst member functions and STL function objects. The types and number of arguments in the functions to be bound can be practically arbitrary; argument list lengths up to few dozens of elements can be supported.
The unbound arguments are expressed as special placeholders in the argument list; they can appear for any argument position. Hence, binding sites preserve the resemblance to the function prototype of the underlying function, leading to simple and intuitive syntax.
Binding can be applied recursively. This results in a versatile function composition mechanism. The binding mechanism is efficient in the sense that it induces very little or no runtime cost.

1 Introduction

The Standard Template Library (STL) [1], now part of the C++ standard library [2], is a generic container and algorithm library. STL algorithms are function templates operating on container elements via iterators and *function objects*. Any C++ construct which can be called with the ordinary function call syntax is a function object. This includes pointers and references to nonmember functions and static member functions, and class objects with a function call operator. Nonstatic member functions are not function objects, but the STL includes a set of *function adaptors* for creating function objects from member functions.

Adaptable function objects are a subset of function objects. They are class objects which, in addition to the function call operator, define a certain set of types. New adaptable function objects can be created from adaptable function objects using function adaptors. For example, function objects can be projected by binding one argument to a constant. Some STL implementations [3] add function composition objects and other adaptor extensions to the standard. Superficially, this seems to provide a great amount of flexibility for adapting functions

* This work has been supported by the Academy of Finland, grant 37178. The author is grateful to Harri Hakonen and Daveed Vandevoorde for their valuable comments on the manuscript of this paper.

to satisfy commonly encountered needs. However, a closer look reveals that this is not the case.

The function objects accepted by STL algorithms are either *nullary*[1], *unary* or *binary*. Standard adaptors accept only unary or binary function objects. Consequently, all functions with more than two parameters, and all member functions having more than one parameter, cannot be passed as function objects to STL algorithms. To be able to pass such functions to STL algorithms, explicit function object classes must be written. Often this results in considerable programming overhead—especially when only a simple operation or algorithm needs to be supported. This inconvenience in turn discourages the use of an otherwise compact and intuitive functional programming style, and not infrequently the STL invocation is expanded by hand.

For example, consider the following two functions for computing values of exponential and Gaussian distributions:

```
double exponential(double x, double lambda);
double gaussian(double x, double mean, double standard_deviation);
```

The computation of the values of the exponential distribution for a set of points (in vector x) can be programmed as follows:

```
vector<double> x, result;    double lambda;
       ...
transform(x.begin(), x.end(), result.begin(),
          bind2nd(ptr_fun(exponential), lambda));
```

The ptr_fun wrapper creates an adaptable binary function object from the exponential function and bind2nd binds the second argument to lambda. The transform algorithm calls this unary function object for each element in x and places the result in result.

We should expect the code for computing Gaussian distribution values to be analogous. However, it turns out that an explicit function object class is needed:

```
vector<double> x, result;    double mean, std;
       ...
class gaussian_caller {
  double mean_, std_;
public:
  gaussian_caller(double mean, double std) : mean(mean_), std(std_) {};
  double operator()(double x) const { return gaussian(x, mean, std); }
}
transform(x.begin(), x.end(), result.begin(), gaussian_caller(mean, std));
```

This is a consequence of the ptr_fun adaptor only being defined for functions taking fewer than three arguments. To overcome this limitation, it is possible to define more ptr_fun templates to cover functions with longer argument lists.

[1] A function with no arguments.

Unfortunately, this is not enough, since the binder adaptors are defined for binary function objects only. Hence, n binders (bind1st–bindNth) would be needed for each argument list of length n. Even with all these $k^2/2$ templates, k being the longest allowed argument list length, the outcome is not particularly intuitive. It becomes difficult to see how the gaussian function is actually called:

```
transform(x.begin(), x.end(), result.begin(),
          bind2nd(bind3rd(ptr_fun(gaussian),std),mean));
```

This article proposes a more general binding mechanism. Instead of specifying the index of the argument to bind, special *placeholder* objects are used in the argument list. For example, the previous example becomes:

```
transform(x.begin(), x.end(), result.begin(), bind(gaussian, free1, mean, std));
```

The role of the placeholder free1 is to specify the varying argument to be left unbound. Other arguments are bound to the values given. The binder invocation syntax preserves a direct resemblance to the original function prototype, revealing instantly which parameters are bound. This mechanism bears similarities to the built-in binding mechanism of the programming language Theta [4], as well as with *agents*, a recently proposed extension to Eiffel [5]. The technique is also related to *currying*, the partial function application mechanism in the functional programming domain.

This article describes the functionality and design of the generic *Binder Library*, BL in the sequel. The BL allows argument binding in the above style for (almost) any C++ function. The library can be downloaded from the address *www.cs.utu.fi/BL*.

2 Binder library functionality

Binding is an operation which creates a k-argument function object, a *binder object*, from an n-argument *bindable function object*, such that $k \leq n$. Some of the arguments of the original function object are bound to fixed values, the remaining unbound arguments are called *free*.

2.1 Bindable function objects

Binding can be applied to nonmember functions and to static member functions, as well as to pointers to such entities. Pointers to const and nonconst member functions are bindable as well. Other function objects, i.e., class objects with a function call member operator, are bindable if they contain a specific set of member typedefs. For smooth integration with the STL, function objects adhering to the adaptable function object requirements of the STL are bindable. Constructors are not bindable.

The STL has special wrapper templates for creating adaptable function objects from nonmember function (ptr_fun) and member function (mem_fun, mem_fun_ref) pointers. They provide a uniform call syntax for different types of functions: a wrapped n argument member function can be called with $n+1$ arguments, where the first argument is a reference or pointer to the object whose member is called. In the BL such wrappers are not needed. (A similar mechanism is used but it is hidden from the client.) Hence, binding can be applied directly to pointers to member and nonmember functions.

2.2 Function argument list length

As explained in section 1, STL style binders cannot easily be applied to functions with more than two arguments. The binding mechanism using placeholders, in turn, does not have this restriction. However, the implementation requires a set of template definitions for each supported argument list length. This imposes a predefined upper limit for the number of allowed arguments. The limit is not particularly restrictive, since argument lists up to a few dozen elements can be supported—the size of the library grows linearly with the supported argument list length.

2.3 Argument binding

A binder object is created with a call to an overloaded generic bind function. The first argument in this call is the bindable function object, henceforth called the *target function*. The remaining arguments correspond to the argument list of the target function.

The argument list can contain free arguments, which are specified with placeholder objects. Since there are no STL algorithms accepting function objects with more than two arguments, the BL defines just two placeholder objects. These are called free1 and free2 and refer to the first and second argument of the function call operator in the resulting function object. Depending on which placeholders are used, this function call operator is either nullary, unary or binary.

Consider the following examples, where Op is a four-argument bindable function object callable with arguments of some types A, B, C and D. Further assume that a, b, c and d are variables of these types, respectively.

```
bind(Op, a, b, c, d);
bind(Op, a, free1, c, d);
bind(Op, free1, b, c, free2);
bind(Op, free2, free2, free1, free1);
```

The first line creates a nullary function object fixing all arguments. The second invocation of bind creates a unary function object taking one argument of type B. The third line results in a binary function object with the first argument of type A and second of type D. The last use of bind illustrates how several free arguments can be unified. In this case A and B as well as C and D must be compatible types. The call results in a function object taking two variables. The first is of type C, second of type A. An invocation with, say, c and a, results in a call Op(a, a, c, c) of the original function object. Therefore, a must be implicitly convertible to B and c to D.

To summarise, the argument list can contain an arbitrary number of both types of placeholders, but if it contains one or more free2 placeholders, it must contain at least one free1 placeholder. A violation of this rule leads to a compile time error. This is reasonable, since the omission of free1, while free2 is present, would result in a function object having the second argument but not the first.

2.4 Argument types

The parameter types of target functions can be arbitrary[2]. However, references to nonconst objects and to objects which cannot be copied require special treatment.

Arguments to bind functions are passed as reference to const. Thus, if a parameter of type reference to nonconst object is bound, the actual argument must be wrapped inside a special object, which creates a temporary const disguise for the argument. This is achieved with a simple function call. For example:

```
void up_and_down(int& i, int& j) { i++; j--;}
list<int> a_list;    int counter = 0;
        ...
for_each(a_list.begin(), a_list.end(), bind(up_and_down, ref(counter), free1));
```

The example decrements each element of a_list by one and increments counter by one n times, n being the number of elements in a_list. Since counter is bound to the nonconst reference parameter i, it is wrapped using the generic ref function. Although the parameter corresponding to the free argument is a reference to a nonconst type as well, it need not be wrapped. The wrapping mechanism is safe with respect to constness: a const reference cannot be wrapped.

The original motivation for requiring the use of the ref wrapper was the desire to avoid combinatorial explosion of template definitions. However, it is semantically beneficial as well. As the example above demonstrates, side effects in bound arguments quickly lead to code that is hard to comprehend. Wrapping forces the programmer to explicitly state that an argument is susceptible to side effects.

By default, the binder object stores copies of the bound arguments. If an argument is of a type which cannot be copied, another wrapper is needed: cref instructs the binder object to store a reference to the argument instead of a copy. Array types are exceptions. Although they can not be copied, wrapping is not necessary: a reference to const array type is stored by default. This ensures that string literals can be used directly as bound arguments. The cref wrapper is used similarly to the ref wrapper.

[2] Volatile qualified types are not supported in the current version of the BL.

2.5 Member functions

When binding member functions, the object for which the member function is to be called, is the first argument after the target function. This *object argument* can be a reference or pointer to the object; the BL supports both cases with a uniform interface. Similarly, if the object argument is free, the sequence can contain either pointers or references. For instance:

```
bool A::f(int); A a;
vector<int> ints;    vector<A> refs;    vector<A*> pointers;
      ...
find_if(ints.begin(), ints.end(), bind(&A::f, a, free1));
find_if(ints.begin(), ints.end(), bind(&A::f, &a, free1));
      ...
find_if(refs.befin(), refs.end(), bind(&A::f, free1, 1));
find_if(pointers.begin(), pointers.end(), bind(&A::f, free1, 1));
```

The first two calls to find_if are equivalent. In the first one, A::f is called using the .* -operator, whereas in the second the same member is called through the –>* -operator. The latter two find_if invocations both find the first A for which A::f returns true. The .* -operator is used in both cases, the library automatically dereferences object arguments of pointer types.

The call mechanism is safe with respect to constness. A nonconst member function can only be called via a reference or pointer to a nonconst object. This holds whether the object argument is bound or free. Note that the ref wrapper is not needed for the object argument. This is a deliberate design choice. It is understood that the object's state may be changed when a nonconst member function is called—there is no need to explicitly state it.

2.6 Function composition with binders

Binding can be applied recursively, thereby enabling function composition. Consider the following example:

```
void canvas::point(double x, double y, colour c);
canvas* canv;    vector<double> x, y;
for_each(x.begin(), x.end(), y.begin(),
         bind(&canvas::point, canv, free1, free2, black));
```

point is a function for drawing points on a canvas using some colour. The for_each invocation draws the points given by the coordinates in vectors x and y. Note that this two-sequence for_each algorithm is not part of the C++ standard library, it is however straightforward to write [6, p. 532].

Now, to represent the y coordinates in logarithmic scale, the standard library function double log(double) can be bound as follows:

```
for_each(x.begin(), x.end(), y.begin(),
         bind(&canvas::point, canv, free1, bind(log, free2), black));
```

The resulting binder object invokes canv−>point(*iter1, log(*iter2), black) at each iteration, where iter1 and iter2 are the iterators provided by for_each: they point to the elements of x and y. Hence, a nested binder object defers the target function call and acts as the inner function in a function decomposition.

The number of nested recursive bindings can be arbitrary. With respect to placeholders, the argument lists of all nested binders are treated as one concatenated list. This means that the requirements stated at the end of section 2.3 must hold for the concatenated argument list, not for any of the individual lists.

2.7 Nullary functions

For completeness, nullary functions can be bound as well. This may seem superfluous, but with function composition, it is a powerful feature. For instance, the following code creates a vector of 100 random number pairs:

```
vector<pair<int, int> > pvec;
generate_n(back_inserter(pvec), 100,
           bind(&make_pair<int, int>, bind(rand), bind(rand)));
```

3 Library design

This section describes the general library design and discusses the key programming techniques used in its implementation. Certain technical details have been omitted from the presented code for clarity.

To begin with, some parts of the binder library cannot be written generically with respect to the length of the target function parameter list. A set of templates are repeated with slight modifications for each supported argument list length. A great deal of this repetition can be avoided using *tuples*.

3.1 Tuples

The binder library uses a set of templates comprising a tuple abstraction [7]. These template definitions are rather intricate and not presented here. Nevertheless, their usage is intuitive.

The tuple template is basically a generalisation of the pair template in the standard library. It can be instantiated to contain an arbitrary number (up to some predefined limit) of elements of arbitrary types. E.g., tuple<int, string, A> is a valid tuple type, corresponding to a class having three member variables of types int, string and A. As is the case of standard pairs, tuples can be constructed directly or using make_tuple (cf. make_pair) helper templates:

```
tuple<int, string, A>(1, string("foo"), A());
make_tuple(1, string("foo"), A());
```

The elements can be accessed in a generic fashion with the syntax get<N>(x), where x is some tuple and N is an integral constant stating the index of the element. The get function template is a *template metaprogram* [8], which resolves a reference to the given element of a tuple at compile time.

The types of the tuple elements can be expressed generically as well: the expression tuple_element<N, Y>::type gives the type of the Nth element of a tuple type Y. This expression is a kind of a *type function* from the constant N and type Y to the element type.

The implementation is based on compile time lists [9, 10], which are recursively instantiated templates. Consider, for example, how the pair instantiation pair<int, pair<string, pair<A, nil> > > could represent the preceding tuple. The compile time lists in the BL are structurally similar, but they are not based on the pair template.

3.2 The bind function templates

The bind function templates define the binding interface. They are repeated for each argument list length. For example, the four-argument bind function is defined as:

```
template <class Target, class A1, class A2, class A3>
inline binder<Target, 3, tuple<A1, A2, A3> >
bind(Target fun, const A1& a1, const A2& a2, const A3& a3) {
  return binder<Target, 3, tuple<A1, A2, A3> >
    (fun, make_tuple(a1, a2, a3));
};
```

These functions group the actual arguments to the target function into a tuple and create a binder object. There are two overloaded bind function templates for each argument list length because nonconst target member functions must be handled differently.

3.3 Binder objects

An important goal in the design of the BL was to minimise the code repetition. This has considerably impacted the structure of the binder classes. Binder objects are instances of class types that fit in the four-level class hierarchy illustrated in Fig. 1. Each level in the hierarchy encapsulates some task orthogonal to the tasks in the other levels. This reduces the amount of partial specialisations, since only one property must be taken into consideration at a time. It also avoids the need to repeat the member and type definitions which are generic with respect to the argument list length.

Target function The base class target_function is a generalisation of the STL templates unary_function and binary_function:

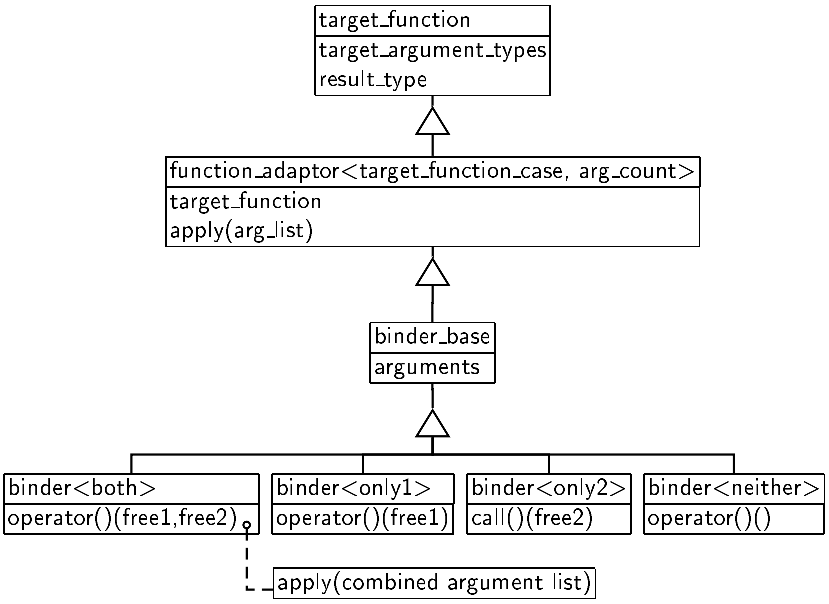

Fig. 1. Binder class hierarchy. The binder template arguments represent the intuitive interpretation of the specialisation criteria. The combined argument list in the call to the inherited apply function refers to the merged argument list where the placeholders have been substituted with the parameters of the function call operator.

```
template<class Args, class Result> struct target_function {
  typedef Args target_argument_types;
  typedef Result result_type;
};
```

It contains type definitions which specify the argument types and the result type of the target function. These type definitions are necessary in various type functions presented below. The same template covers all target function argument lengths, since the template is instantiated with Args substituted by a tuple type.

Function adaptors The purpose of the function adaptors is to unify the function invocation syntax for different types of functions. For each argument length, there are four specialisations of the function_adaptor template. They cover const and nonconst member functions, nonmember functions and other function objects.

Each specialisation contains an appropriately typed data member for storing the target function object and defines the apply function, which provides the uniform call syntax. Another task of the function adaptor templates is to decompose the type of the target function and forward the result and argument types to the target_function template.

For example, the primary template and the function adaptor for pointers to three-argument nonmember functions are defined as follows:

```
template <class Target, int ArgsCount> class function_adaptor;
      ...
template <class A1, class A2, class A3, class Result>
class function_adaptor<Result (*)(A1, A2, A3), 3>
  : public target_function<tuple<A1, A2, A3>, Result> {
  Result (*ptr)(A1, A2, A3);
public:
  explicit function_adaptor(Result (*x)(A1, A2, A3)) : ptr(x) {}
  Result apply(A1 a1, A2 a2, A3 a3) const { return (*ptr)(a1, a2, a3); }
};
```

Binder base The binder_base template has one member variable: the tuple containing the actual arguments of the bind call. This is the sole purpose of binder_base. One generic definition suffices for all argument lengths.

Binders The classes described in the preceding presentation are for internal use in the library, whereas the binder template provides the function call operator to be called from client code. For each target function argument list length, there are four bind specialisations. Which specialisation is instantiated depends on the composition of placeholders in the actuals of the bind invocation. For example, if free1 and free2 are both present, a specialisation defining a binary function call operator is instantiated.

The variables free1 and free2 are defined by the library and are of types placeholder<1> and placeholder<2> respectively. The objects themselves are not important but their types are. These types serve as tags that can be localised from the parameter lists using type functions.

The primary binder template is defined as follows:

```
template<class Target, unsigned int N, class Args,
         bool Free1 = find_free<1, Args>::value,
         bool Free2 = find_free<2, Args>::value>
class binder;
```

The first parameter is the target function type, the second the number of arguments in the target function and the third a tuple type representing the actual argument types deduced in the bind function template. The values of the two bool template parameters Free1 and Free2 indicate whether placeholder<1> or placeholder<2> types are present in Args. These values are deduced with the find_free type functions specified as default arguments.[3]

[3] In general, such use of default template arguments is a convenient technique. It can be used to specialise templates with respect to some property (of template arguments) that is not directly usable as a specialisation criterion.

One particular binder specialisation, for the case of a three-argument parameter list including both types of placeholders, is defined as follows:

```
template<class Target, class Args>
class binder<Target, 3, Args, true, true>
  : public binder_base<Target, 3, Args> {
public:
  typedef binder_base<Target, 3, Args> inherited;
  typedef typename inherited::target_argument_types TA;
  typedef typename deduce_free<1, TA, Args>::type first_argument_type;
  typedef typename deduce_free<2, TA, Args>::type second_argument_type;
  explicit binder(const Target& fun, const Args& a) : inherited(fun,a) {}
  typename inherited::result_type operator()
    (first_argument_type a, second_argument_type b) const {
      return apply( choose(get<1>(args), a, b),
                    choose(get<2>(args), a, b),
                    choose(get<3>(args), a, b));
  }
};
```

This covers the case where Free1 and Free2 template parameters both evaluate to true and thus the specialisation defines a binary function call operator. The result type of this function is the result type of the target function, which is defined in the inherited instantiation of target_function.

The argument types are more intricate because they must be deduced by comparing the types in the actual argument type tuple Args and the parameter types of the target function (inherited::target_argument_types). The type function deduce_free<N, Tuple1, Tuple2>::type defines these deductions. Slightly simplified, the first argument type is resolved by locating a placeholder<1> type from Tuple1 and then selecting the corresponding element in Tuple2. The second argument type is deduced in a similar way. These type definitions provide a function call operation with the correct prototype—the correct functionality is achieved with the choose function templates. Their task is to provide the right arguments to the inherited apply function, which invokes the target function. The args tuple contains the actual arguments—some of which are placeholders—of the bind call. Within each argument position, the choose templates select which argument is redirected to apply. For placeholders, a or b is chosen. Otherwise the bound argument, an element of the args tuple, is used.

The choose functions merely return one of their arguments. The types of the arguments determine which argument is returned. The code below shows the definitions of two particular choose function templates.

```
template<class T1, class T2, class T3>
inline T1& choose(T1& a, T2&, T3&) { return a; }

template<class T2, class T3>
inline T2& choose(const placeholder<1>& a, T2& b, T3&) { return b; }
```

3.4 Library design summary

In the interest of brevity, several important issues were omitted from the preceding description. The implementation of reference wrapping and of recursive binding was not described. The type functions were not shown in detail. How to achieve compatibility with STL function objects or ensure correct usage of free arguments were not addressed either. Crucial techniques for avoiding reference to reference situations in binder instantiations were not explained. Nevertheless, the library contains solutions to these technical aspects and provides the functionality described in section 2.

The library consists entirely of template definitions. It has a fixed as well as variable size part. The fixed size part is approximately 700 lines of code. The variable size part grows linearly with the maximum argument list length supported for target functions. Each supported argument list length requires approximately 100 lines of code and that code can be generated mechanically.

4 Performance considerations

An important goal in the design of the STL was to provide a high level of abstraction without sacrificing efficiency [1]. The BL is compatible with this objective. The functions in the BL only redirect arguments and function calls. Every function in the library is inlined so that commercial-grade compilers can eliminate any overhead arising from these redirections.

In various tests performed with the gcc C++ compiler (version 2.95)[11], the performance of STL algorithms using BL style binder objects was nearly identical to algorithms using standard STL binders. Furthermore, compared with code where the algorithms had been expanded manually to call the target functions directly, the performance was essentially the same. However, due to the extensive template instantiations, the compilation of code using the BL requires more resources than corresponding code that does not use the BL.

5 Conclusions

Using STL algorithms and function objects is a step towards adopting a functional programming style. However, only a subset of C++ function objects can be used with STL algorithms. Function object adaptors and binders in the STL provide some means to adapt functions for STL algorithms, but the solution is insufficient.

This article proposed a novel argument binding mechanism and a generic binder library based on this mechanism. Unbound arguments are specified directly in the argument lists as opposed to an index within the function name of the binder. This allows much greater flexibility in bind expressions. Arguments of nonmember functions, const and nonconst member functions, as well as STL function objects can be bound. The types and number of the arguments can be arbitrary.

The binding syntax is very simple and intuitive. In particular, no adaptors (cf. ptr_fun or mem_fun in STL) are required prior to binding. Moreover, the library supports recursive binding, which provides a versatile function composition mechanism.

The library is type safe. Type errors in bind invocations result in compile time errors. As a downside, these error messages are sometimes lengthy and difficult to interpret. The proposed binding mechanism does not induce any performance degradation, it is as efficient as the standard STL binding mechanism.

References

1. Stepanov, A. A., Lee, M.: The Standard Template Library. Hewlett-Packard Laboratories Technical Report HPL-94-34(R.1) (1994) www.hpl.hp.com/techreports.
2. International Standard, Programming Languages – C++. ISO/IEC:14882 (1998).
3. The SGI Standard Template Library. Silicon Graphics Computer Systems Inc. www.sgi.com/Technology/STL.
4. Liskov, B., Curtis, D., Day, M., Ghemawat S., Gruber R., Johnson, P., Myers A. C.: Theta Reference Manual, Preliminary version. Programming Methodology Group Memo 88 1995, MIT Lab. for Computer Science www.pmg.lcs.mit.edu/Theta.html.
5. Agents, iterators and introspection. Interactive Software Engineering Inc. Technology paper www.eiffel.com.
6. Stroustrup, B.: The C++ Programming Language - Third Edition. Addison-Wesley, Reading, Massachusetts 1997.
7. Järvi J.: Tuples and multiple return values in C++. submitted for publication, see TUCS Technical Report 249 (1999) www.tucs.fi/publications.
8. Veldhuizen, T.: Using C++ Template Metaprograms. C++ Report **7** (1995) 36–43.
9. Järvi J.: Compile Time Recursive Objects in C++. Proceedings of the TOOLS 27 conference (Beijing Sept. 1998) 66–77. IEEE Computer Society Press.
10. Czarnecki, K.: Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Ph.D. Thesis, Technische Universitt Ilmenau, Germany 1998.
11. The GNU Compiler Collection. www.gnu.org/software/gcc/gcc.html.