Towards better usability of component frameworks

Bo Simonsen

Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark
`bosim@diku.dk`

Abstract. The CPH STL is an enhanced version of the STL. During the development of the CPH STL we focused on the container part of the STL. Our goal is to provide several versions of individual STL containers, each providing different trade-offs and desired properties. We found that maintaining complete implementations of all variants would become a hazard for the future development of the library. Therefore, we designed component frameworks, where the concepts of the containers are factorized into smaller parts and most of them can vary independently. Component frameworks give the user an enormous flexibility which allows he or she to specify the desirable trade-offs and properties. We observed that flexibility and usability are hard to reconcile because of limitations in the C++ programming language. In this work we will study the usability problems, caused by these limitations, and we will also provide solutions for these problems. The key problems are that the user has to write a large declaration for using a container, and a component mismatch is likely to occur, i.e. the user gives incompatible components to the framework. Such a component mismatch results in unreadable error messages and the actual errors can be hard to correct. We solved the problem of large declarations by extending C++ with named template arguments and we applied C++ metaprogramming techniques to solve the problem of component mismatches. We believe that the solutions to the problems described in this work are relevant beyond the CPH STL.

Keywords. component frameworks, C++ language features, C++0x, named template arguments, template argument propagation, component mismatch.

1. Introduction

Adaptable component frameworks, in context of the STL, were introduced in our paper on the CPH STL vector implementation [18]. A *component framework* is a skeleton of a software component which is to be filled in with implementation-specific details in the form of policies. A *policy* [33] is the generic variant of a strategy used in the strategy design pattern [10, 11]. More details of how policies are integrated in the CPH STL can be found in [17, 18, 19][1]. The result of our work was a component framework for `vector`, which gave us a high level of parameterization, by template arguments, such

[1] [17] is included in [26].

that the user can select the desired properties (and trade-offs) of the `vector` container. The conceptual view of the vector framework is shown in Figure 1.

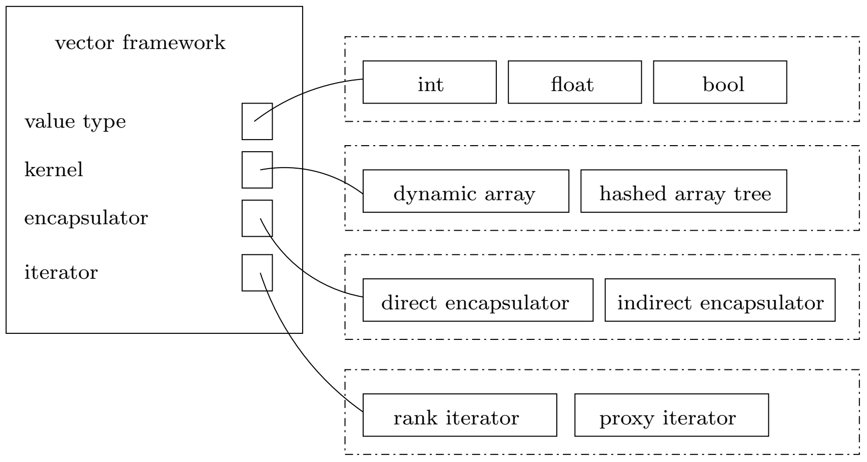

Figure 1. The conceptual view of the vector framework.

We factorized the `vector` concept into the following concepts: A *kernel* is a minimal implementation of a data structure. For `vector` this policy provides the member functions `grow`, `shrink`, and `access` which allow the framework to adjust the size of the kernel and to access elements stored in the kernel. We provide several different kernels with different trade-offs which include space efficiency and worst-case time complexity. An *encapsulator* is a storage policy which states how each element should be encapsulated. We provide three different encapsulators, an encapsulator which stores elements directly, an encapsulator which stores elements indirectly, and an encapsulator which stores elements doubly indirectly. In this context, indirectly means that the underlying array contains references to objects where each value is stored. Each encapsulator gives different properties with respect to referential integrity and strong exception safety. An *iterator* is an implementation of a random access iterator which interface is described in the C++ standard [6]. Because we provide different encapsulators and kernels, we need different kinds of iterator classes. Currently, we have a rank iterator and a proxy iterator. The rank iterator is used when elements are stored directly and the proxy iterator when elements are stored either indirectly or doubly indirectly. However, in the future more iterator classes may appear, since the framework is extendable.

Notice that these concepts are generic for most containers. We have also made a component framework for binary search trees [27]. The significant concepts in this framework is a balancing policy and a storage policy. The *balancing policy* (or the *balancer*) contains member functions for restoring the balance of a binary search tree after modifying operations are executed, and

Listing 1. An example of a configuration of the binary search tree framework.

```
typedef cphstl::set<int,
                    std::less<int>,
                    std::allocator<int>,
                    cphstl::tree<int,
                           int,
                           cphstl::unnamed::identity<int>,
                           std::less<int>,
                           std::allocator<int>,
                           cphstl::avl_tree_node<int, true>,
                           cphstl::avl_tree_balancer<
                                  cphstl::avl_tree_node<int, true>
                           >
                    >,
                    cphstl::node_iterator<
                        cphstl::avl_tree_node<int, true>,
                        false
                    >,
                    cphstl::node_iterator<
                        cphstl::avl_tree_node<int, true>,
                         true>
                 > C;
```

the *storage policy* contains member functions for adjusting and retrieving the value and the pointers. This work was carried out before we constructed the vector component framework. At the time when we developed the vector component framework we observed that the balancing policy was similar to the kernel concept and the storage policy was similar to the encapsulator concept.

An important property of the construction of the vector component framework is that the kernels and encapsulators can vary independently and they are interchangeable. For example, if the user desires a space-efficient container which provides referential integrity, he or she would configure the framework with the kernel `hashed_array_tree` and the encapsulator `indirect_encapsulator`. If the user later observes that he or she does not need referential integrity, he or she should simply change the encapsulator to `direct_encapsulator`.

Component frameworks give both developers and users an enormous flexibility, but they do also introduce problems. We found two significant problems: The user has to give many template arguments for using the framework, and the probability of a component mismatch is large. Such a component mismatch results in many lines of error messages produced by the compiler, where the actual error can be hard to find.

We will justify the claim that component frameworks are hard to use with an example, which is given in Listing 1. This example shows a configuration of the binary search tree framework. A *configuration* is an instance of the framework assembled with the user-selected policies. The container which is assembled with this configuration is an ordered container `set` which stores unique elements. In this configuration the balancing policy is the AVL-tree balancer [1] (`avl_tree_balancer`), the storage policy is a space-efficient node (`avl_tree_node`, the last template argument determines whether the node is space efficient or not), and the iterator is a generic iterator class for containers that are based on nodes (`node_iterator`). More details about the binary search tree framework can be found in [27]. More details on the architecture of the CPH STL can be found in [17, 19].

By analysing the code shown in Listing 1 we deduced the following observations:

– Several template arguments are given several times, for example, the type of the value which is `int` is given 11 times.
– The meaning of each template argument is not clear. For example, it is not obvious to an inexperienced user what `false` and `true` mean in the context of the iterator class `node_iterator`.
– The default arguments are not sufficient. Consider the template parameter list for a class $\mathcal{L} = \langle P_0, P_1, \ldots, P_n \rangle$. We assume that all template parameters have default arguments. If a user wants to override the default argument for P_n, he or she needs to give all template arguments because the order of the template arguments matters. This applies to our current example, if we just want to override the iterator class, we need to give the whole declaration.

In [18] we propose a partial solution to these problems by introducing predefined container classes. For predefined container classes we select the policies in advance such that the user should only give the type of the value and the type of the allocator as required by the C++ standard. For example, for `vector` we provide `compact_vector` which is realized by a space-efficient data structure. The predefined container classes cannot be the only way of using the component framework, simply because we desire the flexibility obtained by the current construction of the framework. Therefore we should support both variants of use. These two kinds of use are defined in [18] as *selective use*, where the user selects a predefined container class, and as *integrated use* where the user builds a component from smaller components (kernel, encapsulator, and iterator). We need a better way of writing the declarations for integrated use because of the problems identified earlier.

With these problems in mind, we can specify the requirements for the future declarations of integrated use: Each template argument should be given once, the meaning of each template argument should be clear, and overriding default arguments should not affect other default arguments in the template parameter list.

1.1 Contributions

The problem of writing proper container declarations is just one problem related to the use of component frameworks. In this work we will study other aspects related the use of component frameworks as well. More precisely, the main contributions of this work are:

- We describe how to implement selective use and how to improve the interaction between the user and the framework for realizing integrated use.
- We show how to detect a component mismatch and provide a solution which produces better error messages when a component mismatch is detected.

Further contributions of this work are:

- We discuss several elements from the upcoming revision of the C++ standard, which we found useful in the context of program-library development.
- We define the template argument propagation idiom and provide some ideas of its application.
- We provide a complete specification and implementation of our named template argument language extension such that others can use it.

2. Selective use

In this section we will consider several different approaches of how to implement selective use. We will consider a C-macro approach, and an approach based on inheritance. We will also consider several different language-feature proposals which have been accepted to the upcoming revision of the C++ standard.

We will use the `vector` component framework as an example in our study of how to provide selective use. The predefined container classes for the `vector` framework are the following:

`cphstl::vector<V, A, R, I, J>`: The parameterized `vector` class. The default template arguments ensure that the container class can be used as specified in the C++ standard; this container class is standard compliant[2] and it should be realized by a dynamic array [8] or a similar data structure.

`cphstl::fast_vector<V, A>`: A vector similar to `cphstl::vector` but the array is not contracted due to performance considerations. This implementation is similar to the one provided by GNU `libstdc++` [13].

`cphstl::safe_vector<V, A>`: A vector based on a regular dynamic array [8], or a similar data structure, with the safety extensions (strong exception safety and referential integrity) as described in [18].

`cphstl::compact_vector<V, A>`: A vector based on the hashed array tree [30] or a similar data structure. This container must provide a space overhead bounded by $O(\sqrt{n})$.

[2] The current `cphstl::vector` interface is not standard compliant because of the reference proxy (described in [18]). The reference proxy should be given as template argument to the framework, such that the vector framework can be used to produce a standard compliant vector.

The template parameter `V` is the type of the value and the template parameter `A` is the type of the allocator.

When finding solutions to problems in the area of library development we have several metrics to measure the quality of our solutions, including flexibility, maintainability, usability, and code reuse. Regarding the construction of predefined container classes, we are mostly concerned about code reuse. That is because, currently we have four different predefined container classes but more may appear in the future. At that time, maintaining several complete implementations of the `vector` container interface may become a hazard for the future development. Therefore we will select the solution which ensures that the highest amount of code is reused, and works according to our requirements. We desire that `cphstl::vector` is the only complete implementation of the interface specified in the C++ standard.

2.1 C macros

A predefined container class for the `vector` component framework can be viewed as an alias of the `vector` component framework given some template arguments in advance. The C++ programming language provides most language features as we know from the C programming language [20]. That includes the preprocessor directives that allow the programmer to let the preprocessor generate code at compile time. These directives are prefixed by `#`. One of these directives is `define` which is used to create an alias (also known as a macro).

The `define` directive takes two arguments, an identifier and a replacement text. What happens when the programmer writes the identifier is that the C preprocessor will substitute the identifier with the replacement text. The identifier can also have an associated parameter list which allows the replacement text to contain parameters. When the programmer supplies an identifier with arguments, the parameters in the replacement text will be substituted with the arguments. We can use the `define` directive to create an alias for the predefined container classes. An example where `fast_vector` is defined using macros is shown in Listing 2.

This solution comes with some major problems. The first obvious problem is that each alias needs a unique name. That is because overloading of aliases depending on their parameter count is not allowed in C-style macros. This means, if a predefined container class has a parameter list of length n with default values, we need n different macros with unique names. This fact makes this solution less attractive. Yet another problem is that macros are processed by the preprocessor, and in that state the compiler has no abstract syntax tree. This means that it has no knowledge of namespaces, so it is not possible to define a macro within the CPH STL namespace. This could cause conflicts if the user did define a macro with the same name.

Listing 2. Macro-based solution for `fast_vector`.

```
#define fast_vector_(V, A) vector<V, A, vector_framework<V, A,
    dynamic_array<V, A, direct_encapsulator<V, A>, true > >,
    rank_iterator< vector_framework<V, A, dynamic_array<V, A,
    direct_encapsulator<V, A >, true > >, false>, rank_iterator<
    vector_framework<V, A, dynamic_array<V, A, direct_encapsulator<V
    , A >, true > >, true> >
#define fast_vector(V) vector<V, std::allocator<V>, vector_framework
    <V, std::allocator<V>, dynamic_array<V, std::allocator<V>,
    direct_encapsulator<V, std::allocator<V> >, true > >,
    rank_iterator< vector_framework<V, std::allocator<V>,
    dynamic_array<V, std::allocator<V>, direct_encapsulator<V, std::
    allocator<V> >, true > >, false>, rank_iterator<
    vector_framework<V, std::allocator<V>, dynamic_array<V, std::
    allocator<V>, direct_encapsulator<V, std::allocator<V> >, true >
     >, true> >
```

Listing 3. Inheritance-based solution for `fast_vector`.

```
namespace cphstl {
  template <typename V,
            typename A = std::allocator<V> >
  class fast_vector : public vector<V, A, vector_framework<V, A,
      dynamic_array<V, A, direct_encapsulator<V, A>, true > >,
      rank_iterator< vector_framework<V, A, dynamic_array<V, A,
      direct_encapsulator<V, A >, true > >, false>, rank_iterator<
      vector_framework<V, A, dynamic_array<V, A, direct_encapsulator
      <V, A >, true > >, true> > {
  public:
    /* constructors and operator= */
  };
}
```

2.2 Inheritance and template programming

Inheritance and template-based programming can be mixed, such that we can define a class template which inherits from another class template [33]. This means that we can define `cphstl::fast_vector` as a class template which inherits from `cphstl::vector`. A skeleton of the implementation is shown in Listing 3. We can still provide the default arguments, such that the user can give just `V` and the default argument for `A` will be used. The user can also give both arguments without problems.

A problem with this solution is that we need to define all constructors in each derived class. For this particular example (`cphstl::fast_vector`) that fact does not cause any problems, since we are inheriting from the same class for any permutation of template arguments. A problem will only appear if the `vector` container gets more constructors, then all predefined container classes should be altered. It is not always the case that a class

Table 1. The differences between the constructors of the stack container and adaptor classes.

Container

```
template <
  typename V,
  typename A = std::allocator<V>,
  typename R = std::list<V, A>
>
class stack_container {
public:
  ...
  explicit stack_container(
    A const& = A());
  stack_container(
    stack_container<V, A, R> const&);
  ...
};
```

Adaptor

```
template <
  typename V,
  typename C = std::deque<V
      >
>
class stack_adaptor {
public:
  ...
  explicit stack_adaptor(
    const C& = C());
  ...
};
```

inherits from the same class for any permutation of the template arguments, for instance, the CPH STL implementation of `stack` does not.

Example 1. The `cphstl::stack` implementation is adaptive meaning that if the last template argument is an allocator, `cphstl::stack` inherits all members from the container variant of stack, otherwise it inherits all members from the adaptor variant of stack. The adaptor variant is described in the C++ standard and the container variant is a CPH STL extension. We provide a container variant of stack since the underlying container already provides iterators, and we observed that our users would prefer that iterators were available in some cases. □

The selection of which class `cphstl::stack` inherits from (as described in Example 1) can be implemented using C++ metaprogramming techniques. The problem is that the two classes (the stack adaptor and the stack container) do not provide the same constructors which is required in order to successfully implement `cphstl::stack` using inheritance, since the constructors are not inherited. The code relevant for this observation is shown in Table 1. We have not found any language features for solving this problem within the scope of the current C++ standard. To successfully solve this problem, we would need a language feature which allowed us to explicitly specify for a subclass that the constructors (in general, all members) should be inherited.

2.3 Inheriting constructors

The lack of language support for inheriting constructors in C++ turned out to be a well-known problem. The upcoming revision of the C++ standard, informally denoted C++0x, will include a language feature which allows inheritance of constructors. The proposal [24] states that if the **using** keyword, parameterized with the name of the base class is present in the declaration of a subclass, the constructors of that base class are inherited. Example 2 shows how the proposed syntax of the **using** keyword can be applied.

Example 2. Consider two classes `sub_class` and `base_class`. We desire that `sub_class` inherits all constructors from `base_class`. With the language feature described in [24] we can write the following C++ code to implement this scenario.

```
class base_class;

class sub_class : public base_class {
  using base_class::base_class;
public:
  ...
};
```

Notice, that the **using** keyword can be placed arbitrarily in the class declaration. □

The appearance of such a language feature will make the predefined container classes smaller, but most importantly this language feature provides us hope that it should be possible to implement the adaptive stack with the desired behaviour as described in Example 1. In the proposal of the inheriting constructors language feature, it is not clear whether it is allowed to inherit constructors from a class template or a template argument. For our implementation of the adaptive stack this is crucial. The most recent draft of the C++0x standard [15] confirms that it should be allowed to inherit constructors from a class template or a template argument. This means that the adaptive stack can be implemented with the desirable behaviour using inheritance.

2.4 Template aliases

Let us reconsider the implementation of the predefined container classes. Currently, we can only provide a valid solution using inheritance. In general we want to avoid inheritance because we have observed that bugs involved in such programs are hard to find since the polymorphic binding is performed at run time. We prefer compile-time polymorphic binding, since an error in such programs will usually result in an error message produced by the compiler, which is easier to find than run-time errors like segmentation faults. Example 3 partly justifies this claim; it shows one pitfall related to inheritance in C++, which most programmers may have encountered.

Example 3. Consider two classes `A` and `B`. The class `B` inherits all members from `A`. Both classes contain a member function `test`. The following code defines the classes and creates an object of `B`.

```
#include <iostream>

class A {
```

```
public:
  void test() {
    std::cout << "A" << std::endl;
  }
};

class B : public A {
public:
  void test() {
    std::cout << "B" << std::endl;
  }
};

int main() {
  A* x = new B();
  x->test();
}
```

When the call `x->test()` is issued, we expect that `B::test()` is called, but it turns out that `A::test()` is called. This happens since the member function in `A` is not defined to be **virtual**. □

An interesting language feature proposed for C++0x is called typedef templates (also known as template aliases). Before we will introduce this language feature, we will give some background on C++ generic programming. A *typedef* is short for type definition. A typedef is used to create an alias for a type. It is similar to the `define` directive, as we discussed earlier. Typedefs are fundamental in C++ generic programming and especially when designing STL containers. That is because class members can be types. We can perform type definitions for all types, also class templates. For example, we can create a type definition for an integer vector in the following way: `typedef std::vector<int, std::allocator<int> > int_vector`. Sometimes it can be useful to create an alias for a class template where the template arguments are not given in advance (see Example 4), as they were in the previous example.

Example 4. Boost [5] C++ libraries provide a so-called pool allocator (`boost::pool_allocator`). A pool allocator [3] maintains an object pool which is used to serve allocation requests. Furthermore when the allocator receives a deallocation request, the object is not deallocated, but it is stored in the object pool. Such an allocator may be a performance improvement for containers which are often updated (elements are erased and inserted). For convenience, we would desire an alias where we could write `pool_allocated_list<V>` to obtain an instance of `std::list<V, boost::pool_allocator<V> >`; `V` denotes the type of the value. □

We can implement the proposal in Example 4 using inheritance or using a typedef template. A *typedef template* is similar to a typedef but it is allowed to use template parameters in its declaration such that when the typedef template is used, the template arguments are given. This language feature

Listing 4. A typedef template for `cphstl::fast_vector`.

```
namespace cphstl {
  template <typename V, typename A>
  using fast_vector = vector<V, A, dynamic_array<direct_encapsulator
      <V, A>, false>, rank_iterator<dynamic_array<
      direct_encapsulator<V, A>, false>, false>, rank_iterator<
      dynamic_array<direct_encapsulator<V, A>, false>, true> >;
}
```

was proposed by Sutter [32], and the syntax is the following:

`template<`$\mathcal{L}$`> using` $\mathcal{A}$ `=` $\mathcal{D}$`;`

The elements in this syntax are the following: $\mathcal{L}$ the template parameter list, $\mathcal{A}$ the alias, and $\mathcal{D}$ the declaration of which the typedef template is an alias for. The same syntax appears in the draft of the upcoming C++ standard [15].

To clarify the use of this language feature, we can now specify the typedef template for `pool_allocated_list<V>` as described in Example 4. The declaration looks as follows:

```
template <typename V>
using pool_allocated_list = std::list<V, boost::pool_allocator<V> >;
```

An important observation is that this language feature becomes useful when implementing the predefined container classes. With inheritance we needed several lines of code, with typedef templates we need just a few lines. This is shown in Listing 4 where the full declaration of `cphstl::fast_vector` is given.

2.5 Conclusions

From the discussion in this section, we have learned that the language features proposed for C++0x ease and improve library development. More specifically we have learned that the adaptive stack can be implemented using inheritance with the extension of inheriting constructors. Currently it cannot be implemented in the desired way, which means that the current specification of the C++ programming language is not strong enough regarding this matter. Furthermore we learned that we can implement the predefined container classes with just a typedef template instead of a class declaration.

3. Template argument propagation

In this section we will study how to improve support for integrated use; the way of use where the user specifies the kernel, encapsulator, and iterator for realizing the `vector` container. We observed that the declarations for integrated use were long and hardly readable. That was because the user had to give each template argument several times in the worst case, the meaning

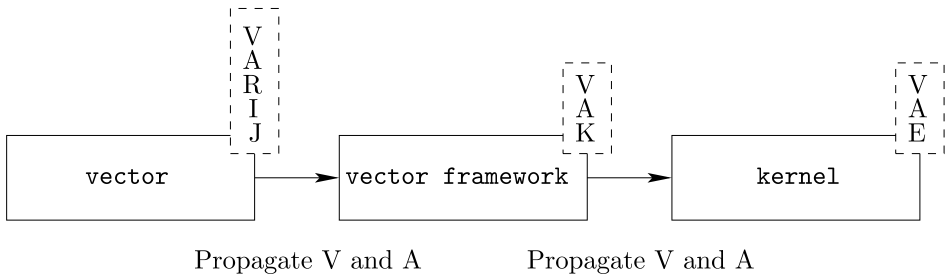

Figure 2. Template argument propagation.

of each template argument was not clear, and the default arguments were not sufficient. If we could just solve one of these problems, we would achieve a significant improvement regarding usability. We have found a technique, based on the regular `C++` language, which ensures that the user can give each template argument once; we can therefore solve one of our usability problems.

The idea in this technique is to propagate (or forward) each template argument, which is given to a class template `X`, to class templates which are instantiated by `X`. A simple algorithm for propagating template arguments is shown in Algorithm 1. The algorithm accepts sets of template arguments given to each class in a configuration. These sets are denoted $\mathcal{A}_x$ for a class x. In these sets template arguments are just given once, and the algorithm computes a full set of template arguments for set $\mathcal{A}_x$ for all x in the configuration. The algorithm has several limitations, including that all template arguments must be unique, i.e. a class template cannot be given twice with different permutations of types. This fact does not matter, since the purpose of the algorithm is to show how this mechanism works.

Algorithm 1 PROPAGATE($\mathcal{C}, \mathcal{A}, \mathcal{P}$)

Require: The class name $\mathcal{C}$. The set $\mathcal{A}_x$ containing pairs $\langle$parameter, argument$\rangle$ for a class x. The set $\mathcal{P}_x$ containing template parameters for a class x.

```
1: for $\langle p, a \rangle$ in $\mathcal{A}_\mathcal{C}$ do
2:     $\mathcal{A}_a \leftarrow \{\langle p', a' \rangle \mid \langle p', a' \rangle \in \mathcal{A}_\mathcal{C} \wedge p' \in \mathcal{P}_a\} \cup \mathcal{A}_a$
3:     PROPAGATE(a, $\mathcal{A}, \mathcal{P}$)
4: end for
```

A concrete example which shows how the template arguments are propagated when this technique is applied for our vector framework is shown below.

Example 5. Let us consider the `vector` and `vector_framework` classes. The propagating procedure works as follows (also shown in Figure 2):

– The user supplies `vector` with all template arguments required which

Table 2. A class containing inner classes.

```
class A {
public:
  /* classes */
  class B {
  public:
    void a_member_func() {
    }
  };
  class C {
  public:
    typedef int member;
    void another_member_func() {
    }
  };

  /* member function */
  void test() {
    (*this).d.test();
    (*this).b.a_member_func();
  }
```

continues in the right column

```
private:
  /* classes */
  class D {
  public:
    void test() {
    }
  }

  /* member data */
  B b;
  D d;
};

int main() {
  A a;
  A::C c;
  a.test();
  c.another_member_func();
}
```

are `V`, `A`, `R`, `I`, and `J`.

– The template arguments which are required by `vector_framework` and known by `vector` will automatically be given to `vector_framework`. These are `V` and `A`.
– The template argument which is not known by `vector` must be supplied by the user. This template argument is `K`.

This procedure can be repeated recursively such that `vector_framework` propagates template arguments to the kernel, and the kernel propagates template arguments to the encapsulator using the same principle. □

This technique can be widely applied when exercising C++ generic programming; we will see that the use of this technique has other applications as well. Since it is a technique which can be widely applied we have classified it as an idiom, and named it the *template argument propagation idiom*. We will now give some background on the C++ programming language required for understanding the implementation of this idiom.

3.1 Inner classes

The C++ programming language is very powerful for structuring elements in a large code base. These elements are functions, variables, and so on. The basic language features for structuring these elements are structs and classes.

Table 3. An inner class template.

```
template <typename P1>
class A {
public:
  template <typename P2>
  class B {
  public:
    void test() {
      P1 p1;
      P2 p2;
      ...
    }
  };
};

template <typename P1>
class C {
```

continues in the right column

```
public:
  void test() {
    b.test();
  }
private:
  typename P1::template B<P1> b;
};

int main() {
  A<int>::B<float> b;
  b.test();

  C<A<int> > c;
  c.test();
}
```

More advanced language features for structuring elements are namespaces, which become very useful in, for example, library development. Namespaces make it possible for the user to use several different libraries within the same code, for example, the user can use both elements from the CPH STL and the STL at the same time. A language feature, which is often overlooked, is the possibility of having *inner classes* [33] in a class, i.e. classes can contain other classes.

Let us consider a simple example shown in Table 2. Here, the class `A` has three inner classes. The classes `B` and `C` are declared public, and the class `D` is declared private. The use of inner classes makes it easy to do proper encapsulation. With these declarations we specified that the classes `B` and `C` can be instantiated outside `A` but `D` needs to be instantiated within `A` (if `A` contained **friend** declarations, the friends could also create instances of `D`). The classes contained in `A` can be accessed like any other member using the `::` infix operator, e.g. `C` can be accessed using the statement `A::C`. Accessing a member in `C` can be done using `A::C::member`.

Regarding this language feature, we are mostly concerned if it can be applied to class templates such that we can have inner class templates in class templates. This is possible, and in general, there is no difference between writing regular inner classes and inner class templates. An example of the use of inner class templates is shown in Table 3. In this example we have a class template `A` which consists of an inner class template `B`. As the reader can verify by examining this example, the only difference between the use of inner classes and inner class templates is that we provide template arguments for inner class templates. The use of inner class templates is similar to the construction given in Example 6.

Example 6. For most STL containers, in this example `vector`, we see the following recurring construction:

```
template <typename V, typename A = std::allocator<int> >
class vector {
public:
  ...
  template <typename I>
  void insert(I first, I second);
};

int main() {
  vector<int> v;
  int a[] = {1,2,3};
  v.insert(a, a+3);
  v.insert<int*>(a, a+3);
}
```

The user supplies vector with the template arguments `V` and `A`. The template argument `I` for the function template `insert` is deduced from the type of the iterator which is given as argument. The template argument can also be given explicitly as shown in the second call to `insert`. □

By using this language feature we can create even more complex constructions. Let us consider the class `C` from the example in Table 3. The class `C` accepts a class template containing an inner class template as template argument; `C` creates an object of the inner class template and calls a member function using this object. The declaration of creating the object is more complex than the declarations that we have already seen. Since the outer class template is given as template argument, we have to use the **template** keyword to access the inner class template (see line 22 in Table 3).

Observation 1. In Table 3, the class `C` provides the template arguments for the inner class template (in this example `B`) of the class template given as template argument (in this example `A`). This means that the user just specifies the template arguments for `A` and `C` supplies `B` with template arguments. □

3.2 The idiom

Observation 1 is the foundation for the template argument propagation idiom. The key aspect is that some class template can accept another class template by its template argument and provide template arguments for this class, using an inner class as a proxy for the real class. An example is shown in Listing 5 where we have two classes `X` and `Y`. The idea is that the user should only supply `X` with template arguments, and then `X` supplies `Y` with the template arguments needed. The class `Y` can also take template arguments. These template arguments are specified by the user. This is useful if `Y` takes template arguments which are not known by `X` as we described earlier.

Listing 5. General structure of the template argument propagation idiom.

```
class Y {
public:
  template <typename P1, typename P2, ...>
  class real_class {
    public:
    ...
  };

};

template <typename P1, typename P2, ..., typename PY = Y>
class X {
public:
  ...
  typename PY::template real_class<P1, P2, ...> y;
};

int main() {
  X<int, char, ..., Y> x;
  ..
}
```

This idiom could be directly applied to ensure that we will only specify template arguments once when using our frameworks. An example of how the idiom could be applied is shown in Listing 6. This example is almost equivalent to the example shown in Listing 5, where `vector` is `X` and `vector_framework` is `Y`. The difference is that `vector_framework` takes one template argument which is the kernel; this template argument is not known by `vector`. We assume in this example that the kernel is using this idiom such that the template arguments are propagated.

Every solution to a problem has its price, one may ask, what is the cost of this solution? In order to propagate all template arguments for every component in our layered architecture, this idiom should be applied to all components, which includes container classes, frameworks, kernels, iterators, and so on. Changing our entire code base would be a very time consuming task, and the code would become less readable. As earlier stated the use of this idiom does only solve one of our problems, namely that template arguments are given once. The two other problems still remain, namely that the meaning of each template argument was not clear, and overriding some template arguments meant that the default arguments could no longer be used. The fact that this solution comes with an expensive price tag and it does not solve all our problems makes it unattractive. We have found no way to solve all problems within the scope of the current C++ standard. In the next section we will consider a solution which solves all three problems, but this solution is beyond the current C++ standard. But first we will look into another application of this idiom.

Listing 6. A skeleton of `vector` and `vector_framework` with the template argument propagation idiom applied.

```
template <typename K>
class vector_framework {
public:
  template <typename V, typename A>
  class real_class {
  public:
    ...
  };
};

template <typename V, typename A, typename R, typename I, typename J
    >
class vector {
  ...
private:
  typename R::template real_class<V, A> r;
};

int main() {
  vector<int, std::allocator<int>, vector_framework<
      hashed_array_tree< ... > >, ... > v;
}
```

3.3 Cyclic template arguments

A recurring problem when exercising C++ generic programming is that a cyclic dependency of template arguments can appear. Given two classes `P` and `Q`, assume that both classes takes one template argument. Consider the scenario where `P` is given to `Q` and `Q` is given to `P` as template arguments. If the user tries to write a declaration for this scenario, he or she would end up with an infinite declaration: `P< Q< P< ... > > >`. Obviously, this declaration cannot be accepted by the compiler. We call the problem, caused by this scenario, the problem of *cyclic template arguments*. We observed that applying the template argument propagation idiom solves this problem; the solution is shown in Listing 7 for the current example. We can now rewrite the infinite declaration to the following finite declaration `P<Q>`. The example below describes the situation where we encountered this problem for the first time.

Example 7. The initial construction [28] for a dynamic array providing iterator validity and thereby partly referential integrity (for definition see [18]) in the CPH STL is shown in Figure 3. In this construction each encapsulator stores a pointer to the surrogate. This construction was later refactored and during this refactoring the surrogate pointer was moved to the iterator. The motivation of this change was to reduce the space consumption for each encapsulator by a pointer, i.e. the memory consumption for a vector storing

Listing 7. The template argument propagation idiom applied to two classes with a cyclic dependency by template arguments.

```
class Q {
public:
  template <typename Arg>
  class real_class {
  public:
    ...
  };
};

template <typename Arg>
class P {
public:
  ...
  typename Arg::template real_class< P< Arg > > y;
};

int main() {
  P<Q> p;
}
```

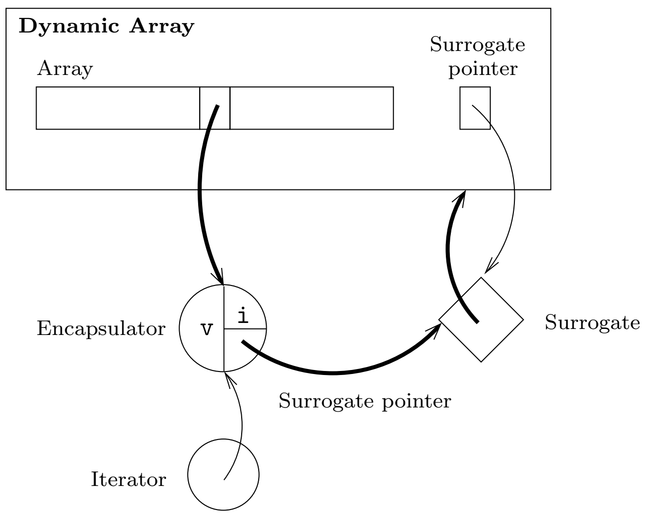

Figure 3. The initial construction of a dynamic array providing referential integrity.

n elements was reduced by n pointers. Yet another argument for moving the pointer to the surrogate was that the template arguments became cyclic within the initial construction: The surrogate class takes the realizator (dynamic array) as its template argument, the realizator takes an encapsulator as a template argument, and the encapsulator takes a surrogate as a template argument. Moving the surrogate pointer to the iterator removed this cyclic dependency. □

An interesting question one may ask: can we during the design phrase detect whether the problem of cyclic template arguments appear? The answer to this question is yes; the observation below states under which circumstances the problem will occur.

Observation 2. If the relationship between the components is interpreted as a directed graph (see Figure 3), the problem of cyclic template arguments exists if there exists a cycle in this graph (represented by the bold edges in the figure). □

We learned from Example 7 that the cyclic dependency can be removed by reorganizing the components. Such a reorganization may, in some cases, cause reduced flexibility, for example, the surrogate cannot be accepted as template argument in the framework, without applying the template argument propagation idiom. We found it acceptable that the surrogate was explicitly defined in the framework, but in some cases it might be unacceptable to explicitly define types. In such cases we have found no other options than applying the template argument propagation idiom. The problem has earlier been discussed in [7]. The solution proposed in [7] is denoted rebinding (this concept is also used in allocators) which is similar to the template argument propagation idiom.

4. The named template argument language extension

Several modern programming languages provide the feature *named arguments* (also known as *keyword arguments*). This feature provides a mechanism to call a method where the arguments are prefixed with the name of the parameter. These programming languages include JavaScript, Python, C#, and F#. We will study how this works in Python [25] by considering the code given in Listing 8. The function `fun` defined in lines 1–2 takes two arguments; both parameters have default arguments defined. This means that the function can be called with no arguments (line 4) and the default arguments will be used. The function can be called with one argument and the default argument for the last parameter will be used (line 5). Finally the function can be called with both arguments (line 6). These ways of use are identical to what is possible in C++.

What is beyond C++ in Listing 8 is that we can supply the arguments using the name of the parameters. The use of this language feature is shown in the fourth call (line 7) where the function is called with `ptwo = 2`. This means that `ptwo` in the function is set to 2 and the default argument for

Listing 8. The use of keyword arguments in Python.

```
1 def fun(pone = 'a', ptwo = 1):
2   print pone, ptwo
3
4 fun()                        # prints 'a 1'
5 fun('b')                     # prints 'b 1'
6 fun('b', 2)                  # prints 'b 2'
7 fun(ptwo = 2)                # prints 'a 2'
8 fun(pone = 'b')              # prints 'b 1'
9 fun(pone = 'b', ptwo = 2)    # prints 'b 2'
```

`pone` is used. The remaining calls (lines 8–9) are similar to the calls in lines 5–6. The reason why we study this language feature is that we found it relevant for C++ templates. When we studied the usability problems, which we identified earlier, we have just found a solution for one of the problems (template arguments were given several times). The two remaining problems which were that the meaning of the template arguments were not clear and the default arguments were not sufficient if one overrides the default argument for the last template parameter in the list.

Our hypothesis is that *named template arguments* will solve these problems. The idea is that template arguments can be given in a similar way as shown for function arguments in Python. The fact that the parameter name can be given as prefix for the argument should make the meaning of the template argument clear (if the name of the parameter is carefully chosen) and since the template argument is addressed by its parameter name the order is not important anymore. Therefore this language extension solves our problems. To realize such an extension we develop a preprocessor which takes C++ code mixed with the language extension code and produce C++ code which can be accepted by the compiler. We are not the first to propose named template arguments in C++ template programming. An earlier attempt [33] has been made to obtain this feature in C++; however this attempt relies on C++ metaprogramming techniques. Combined with the template argument propagation idiom this technique may solve all three problems, but it requires that the whole library is refactored.

Another solution could be to develop a domain-specific language (discussed in for example [23]) for specifying container declarations. A simple language is shown in Example 8. This language would partly solve our problems combined with the template argument propagation idiom. A preprocessor could translate the domain-specific language code embedded in C++ code into pure C++ code. The problem with this language is that the template arguments contained in $\mathcal{D}$, $\mathcal{R}$, $\mathcal{I}$ and $\mathcal{E}$ would not have any description associated such that the meaning of these parameters would be clear. Then we should add another feature to the language to provide proper readability. This fact makes this solution unattractive, since such a language would be hard to maintain. We have learned this lesson from earlier experiences [29].

Example 8. Consider a domain-specific language realized by the following regular expression:

$$\mathcal{L} = \mathcal{D}\ ((\texttt{using}\ \mathcal{R}) \cup \emptyset)\ ((,\texttt{encapsulator}\ \mathcal{E}) \cup \emptyset)\ ((,\texttt{iterator}\ \mathcal{I}) \cup \emptyset)$$

Where $\mathcal{D}$ is a C++ declaration for a container as defined in the C++ standard, $\mathcal{R}$ is the realizator, $\mathcal{E}$ is the encapsulator, and $\mathcal{I}$ the iterator class. □

We have not yet addressed the problem of propagating template arguments in context of the named template argument language extension. We learned that we should change our entire code base for applying the template argument propagation idiom. We want to avoid that. Fortunately, there is a smarter solution. If all template arguments given by the user are considered to be *global template arguments* we can automatically propagate the template arguments. For example, the user supplies `vector` with the type of the value `V`. The template argument `V` will now be global, so all classes (all classes in the configuration) contained in `vector` will know `V`. Another argument in favour for named template arguments, besides that it solves our usability problems, is that it is usable outside the CPH STL. That is why we classified it as a language extension. If such a language extension appeared in the C++ programming language, the language would become stronger for template-based programming. Not just because the code becomes more readable but mainly because the default template arguments would be usable for any template argument that is overridden.

4.1 The language extension

We will start explaining our language extension of named template arguments by an example. This example is shown in Listing 9. In this example, we declare a container type `C`. This container is a `set` which is an ordered container which stores unique elements. In this example the set is storing elements of type `int`, and it is realized using the binary search tree framework. The kernel is an AVL tree [1] and a space-efficient node is used. This example is equivalent to the first container declaration (Listing 1) we considered. So for this example, the preprocessor will translate the code given in Listing 9 to the code given in Listing 1.

The tokens `!<` and `!>` are used to enclose the declarations which are part of the language extension such that our preprocessor can easily recognize the declarations which it should process. The logic of our language extension is:

– Every argument enclosed by the tokens `!<` and `!>` must be a named argument, i.e. it is prefixed by its parameter. We denote a list of named arguments enclosed by the tokens `!<` and `!>` a *block*.
– An argument may contain another block of named template arguments (for example, see the argument for parameter `R`, line 2 in Listing 9).
– The argument for each parameter is memorized such that if a named argument is already given, that argument will be used. It is possible to

Listing 9. An example of a declaration using named template arguments.

```
typedef cphstl::set!<V=int,
                     R=cphstl::tree!<
                       N=cphstl::avl_tree_node!<packed=true!>,
                       B=cphstl::avl_tree_balancer
                     !>,
                     J=cphstl::node_iterator!<is_const=true!>,
                     I=cphstl::node_iterator!<is_const=false!>
                   !> C;
```

overwrite these arguments such that, if a named argument `X` is given, it can later be overwritten and the overwritten version will be used in the rest of the declaration.

– If a named argument is not given, and it is not memorized from an earlier declaration, the default arguments, as specified in the declaration of the class will be used (for example, in Listing 9 the named argument of `A`, the allocator, is omitted, here the default argument is used). If this is not specified either, an error is reported.

Already now we can see the advantages of our language extension: The meaning of the template arguments becomes clear. For example in the declaration of `node_iterator` it is now clear what the Boolean argument means, because the argument is prefixed by `is_const`. When `is_const` is true, an immutable iterator class is made, and when `is_const` is false, a mutable iterator class is made. In our paper on component frameworks [18, Listing 1] we had problems showing what the Boolean argument meant. We used an **enum** to show the meaning. Likewise for the node class. The Boolean argument determines if the node should be packed (space efficient) or not; According to the code example, that should be clear now.

Another advantage is that the declaration has been reduced in size, because each unique template argument is given once. Earlier we used 387 characters to write the declaration. Now we use 206 characters, which is a reduction of approximately 50%. Another aspect in this reduction is that only the significant template arguments for each class are shown. For example, the significant policies for the framework are the kernel and the encapsulator, which are the only template arguments given to the framework. The user should obtain a better overview of the code by this reduction. The order partly matters currently. In the example, given in Listing 9, `I` and `J` has been swapped according to the order of the template parameters for `set`, i.e. `I` is given before `J`. For these particular template arguments it does not matter, since `I` and `J` are not used by other classes given to `vector`. But if one desires to move `V` to the end of the template argument list, it is not possible in the current implementation.

We will now study how the preprocessor is implemented. It is implemented like a regular compiler with the phases of parsing, code emission, and so on [2]. However, some of the phases have been omitted, since we will,

$$\begin{array}{l}
\langle \text{nta\_statement} \rangle \rightarrow \textbf{id}\ \texttt{!<}\ \langle \text{nta\_list} \rangle\ \texttt{!>} \\
\langle \text{nta\_list} \rangle \rightarrow \langle \text{nta\_list}' \rangle\ \langle \text{nta\_entry} \rangle \\
\langle \text{nta\_list}' \rangle \rightarrow \langle \text{nta\_list} \rangle \\
\quad \mid \epsilon \\
\langle \text{argument\_list} \rangle \rightarrow \langle \text{argument\_list}' \rangle\ \textbf{id} \\
\langle \text{argument\_list}' \rangle \rightarrow \langle \text{argument\_list} \rangle \\
\quad \mid \epsilon \\
\langle \text{nta\_entry} \rangle \rightarrow \textbf{id} = \langle \text{nta\_statement} \rangle \\
\quad \mid \textbf{id} = \textbf{id}\ \texttt{<}\ \langle \text{argument\_list} \rangle\ \texttt{>} \\
\quad \mid \textbf{id} = \textbf{id}
\end{array}$$

Figure 4. BNF grammar for the language extension.

for example, not perform type checking since it is done by the underlying C++ compiler. We will now describe what happens in each phase.

Parsing of class templates: Class templates found in the included files are parsed and stored in a dictionary. This is needed since the preprocessor must have knowledge of the default arguments of each class template, in order to use the default arguments when a template argument is omitted.

Parsing of named template arguments: Named template argument expressions are identified and parsed using the grammar specified in Figure 4. The grammar is specified using the Barcus-Naur Form (BNF). The result of the parsing is abstract syntax trees of the named template argument expressions. These trees reflect the structure of the expressions. An example of such a tree is shown in Figure 5. What is not shown in the figure is that also namespaces are registered. That is needed since the same class name can be in several different namespaces. That is the case too, for the dictionary of class templates.

Code emission: Using the dictionary of class templates and the abstract syntax tree, the resulting C++ code is emitted. When traversing the abstract syntax tree and the list of class templates a dictionary of global template arguments is maintained. Such that when, for example, `V` is found its argument is registered in the global dictionary. The pseudo code for the emission procedure can be found in Listing 10.

4.2 Details

We will now give some more details regarding the implementation of the preprocessor. We will first consider the grammar given in Figure 4. The most interesting element in the grammar is ⟨nta_entry⟩. For a named template argument declaration: `C !< P = .. !>`, `P` can be the following:

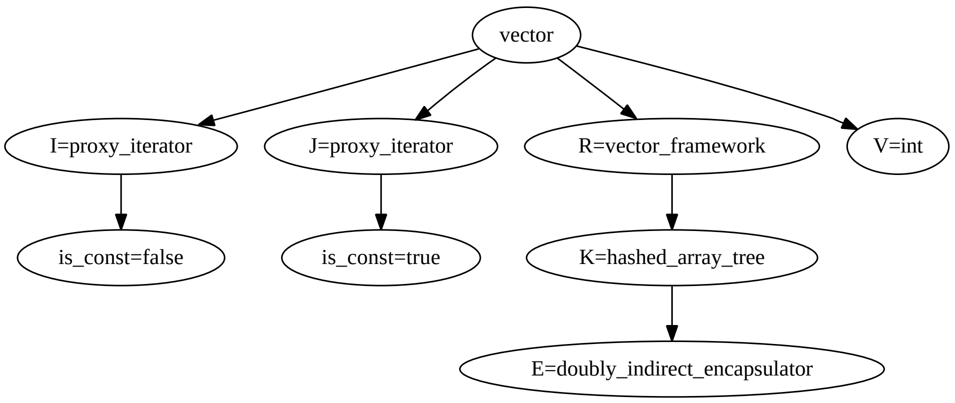

Figure 5. The abstract syntax tree for a simple vector declaration.

Listing 10. Python pseudo code for the code emission procedure.

```
# REQURE: 'ast' the abstract syntax tree, 'gd' the global dictionary
# containing the named arguments, 'ct' the dictionary of class
# templates.

def emit_code(class_name, parameter_dict):
  for (parameter, default_argument) in ct[class_name]:
    if parameter_dict.has_key(parameter):
      (class_name, arguments) = parameter_dict[parameter]

      if arguments != {}:
        # This is a nested statement, i.e. !< !>
        ret = emit_code(class_name, arguments)
        register_and_emit(ret)
      elif ct.has_key(class_name):
        # The user just gave the class name, we need to expand the
        # declaration by either ct or gd
        ret = emit_code(class_name, {})
        register_and_emit(ret)
      else:
        # simply use the class_name.
    elif gd.has_key(parameter):
      # use gd[parameter], since parameter is already given.
    else:
      # use the default arguments for this parameter.
      ret = emit_code(default_argument[0], default_argument[1])
      register_and_emit(ret)

emit_code(ast[0], ast[ast[0]])
```

– `P` can be a new named template argument declaration such that we can write `C !< P = C2 !< ... !> !>`.
– `P` can be an instantiation of a class template `C !< P = C3<...> !>`. The argument of parameter `P` is untouched.
– `P` can be a class name i.e. `C !< P = C4 !>`. This expression is similar to `C4 !< !>`, such that the global dictionary and the default arguments will be used to find appropriate template arguments for `C4`.

The process of emitting the code, shown in Listing 10, is so complex that it deserves some more explanation. Since the job of our preprocessor is to generate the full declaration of template arguments, we start by traversing the dictionary of class templates. Each entry in the dictionary contains a list of parameters and their default arguments. Within this traversal there are three different cases:

Case 1 is executed if the user supplied the argument for the current parameter (the user supplied arguments are kept in `parameter_dict`). This case has three different cases which are derived of $\langle$ nta_entry $\rangle$ as we described above.

Case 2 is executed if the user did not supply this argument and the argument is already known, i.e. it is already registered in the global dictionary. In this case we will use the argument obtained from the global dictionary.

Case 3 is executed if neither of the two first cases are executed. In this case we will use the default argument. If the default argument is non-existing an error is reported to the user.

4.3 More examples

We will now consider some more examples on the use of this language extension to better understand its novelty.

Example 9. Figure 5 shows a declaration of a `vector` storing elements of type `int`. The vector container is realized by the vector framework, where the kernel is a hashed array tree and the encapsulator stores elements doubly indirectly. The iterator used in this setting is the `proxy_iterator`. The declaration using named template arguments is the following:

```
cphstl::vector!<V=int,
                R=cphstl::vector_framework!<
                  K=cphstl::hashed_array_tree!<
                    E=cphstl::doubly_indirect_encapsulator
                  !>
                !>,
                I=cphstl::proxy_iterator!<is_const=false!>,
                J=cphstl::proxy_iterator!<is_const=true!>
              !> vec;
```

□

Example 10. The associative container `map` stores keys of type `K`. Each key is associated with a value of type `V`. In this example we will consider a `map` container realized by the binary search tree framework, where the kernel is an AA-tree. The iterator used is the generic iterator for data structures based on nodes `node_iterator`.

```
cphstl::map!<K=char,
             V=int,
             A=std::allocator<std::pair<char, int> >,
             R=cphstl::tree!<
               V=std::pair<char, int>,
               F=cphstl::unnamed::key_extractor,
               N=cphstl::aa_tree_node,
               B=cphstl::aa_tree_balancer
             !>,
             I=cphstl::node_iterator!<is_const=false!>,
             J=cphstl::node_iterator!<is_const=true!>
           !> mc;
```

Notice that `V` is overwritten, since the type of the value stored in the tree is the pair of $\langle \texttt{K}, \texttt{V} \rangle$. The keys are retrieved using the class template `key_extractor` which represents the function $F : \langle \texttt{K}, \texttt{V} \rangle \to \texttt{K}$. The `C++` standard requires that `K` and `V` are given separately to `map`. □

4.4 Reflection

This solution also comes with disadvantages. Usually the name of the template parameter has no meaning in the context of the interface. With this language extension this fact is no longer true. Now, the library developer needs to carefully select the names for each template parameter and he or she needs to consider which template arguments should be propagated. The library developer also needs to think of possible conflicts regarding template parameters, for example, maybe template parameter `V` has a meaning in one class and in another class it has a different meaning. This problem becomes obvious when using the `map` container and the search tree framework together, see Example 10.

Our language extension allows us to overwrite a template argument which is already given, but the feature should be used with caution since a type error can occur if an argument is overwritten and the later declaration expects the previous version of the argument. To get the code given in Example 10 working, it has been necessary to overwrite the argument `V`. It works since the argument `V` given to `map` is not used further on by classes given as arguments. Let us consider the following scenario: We add template parameter at the end of the template parameter list for `map`. The class, which is used as argument to this parameter, requires the argument `V` which is given to `map`. This scenario will cause a compile-time error since `V` was overwritten.

The obvious solution to this problem is to rename either the parameter `V` defined in `map` or the parameter `V` defined in the binary search tree framework. If the parameter `V` defined in the search-tree framework is changed, the node class also needs to be changed, in order to propagate the template argument. Since we have several node classes, the easiest solution would be to change the parameter defined in `map`, since no changes to other class templates are required.

Another solution is to introduce *local template arguments*, which are template arguments that are not propagated. In our current example, `V` should be a local template argument. Since the argument of `V` is not propagated we need to give the argument of `V` to all classes which take `V` as their template argument. In the scope of our example, we only need to give `V` to the node class, since the other classes obtain the type of the value from the node class.

A third solution is to introduce *sticky template arguments*, which are template arguments that are propagated but changes to them will not be propagated. This mechanism is similar to the call-by-value principle which we know from imperative programming, where a function is called by copies of the arguments such that any change to the arguments (which become local variables) is not propagated. In our current example, this mechanism would propagate `V` to the search-tree framework where it is overwritten but within the scope of `map` it is not overwritten. We will leave the decision of which solution would be the most appropriate as future work.

5. Framework configuration

Another disadvantage of component frameworks, which we have not discussed yet, is the poor error messages that will occur if the wrong components are given to the framework. During the development of the CPH STL architecture [17, 19], we decided to decouple the iterators from the containers. Later, we found this decoupling useful in the construction of the vector component framework. In the vector framework we have two different iterator classes which are the proxy iterator and the rank iterator. The rank iterator is used for a vector where the elements are stored directly in the array, and the proxy iterator is used for a vector where the elements are stored indirectly, i.e. in objects of which the array contains references to [18]. Whether the elements are stored directly depends of which encapsulator is given to the framework.

By decoupling the iterator we created the possibility of a component mismatch, since there is a relationship: When either `doubly_indirect_encapsulator` or `indirect_encapsulator` is given to the framework as encapsulator, `proxy_iterator` must also be given to the framework as iterator class. When `direct_encapsulator` is given to the framework as encapsulator, `rank_iterator` must also be given to the framework as iterator class. If the user tries to use the framework with an encapsulator which does not fit the iterator class, the user will get several screen lengths of error messages, and these error messages are not useful at all for finding the actual error. The compiler will typically write errors which relate to missing members or incorrect types. What we really need is a simple message which states that the iterator class does not fit the encapsulator class.

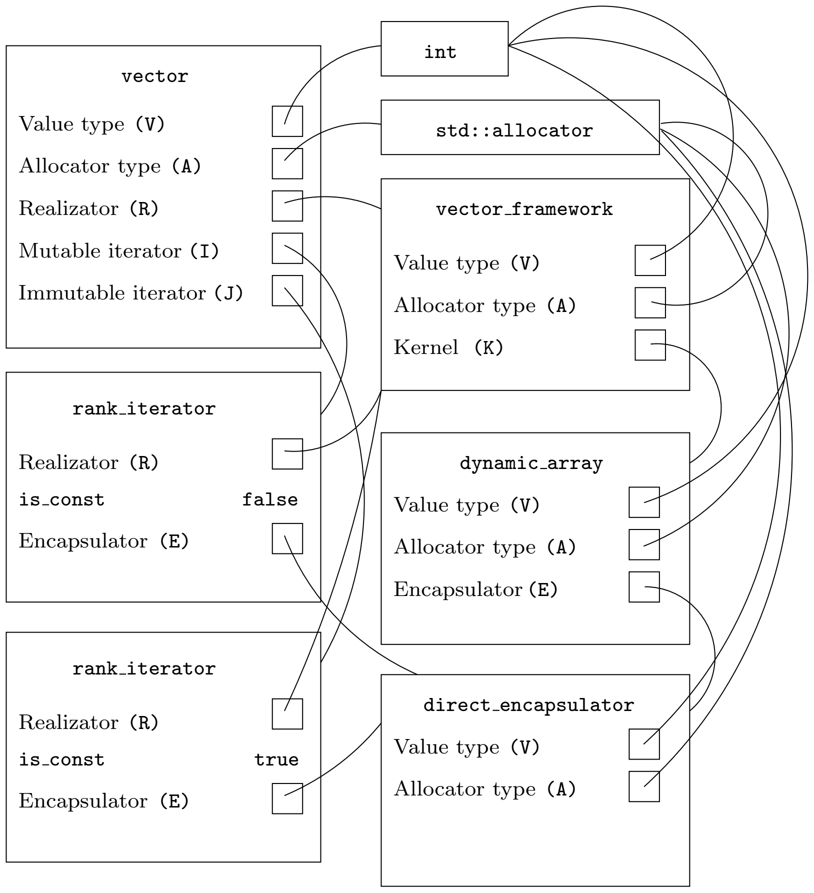

Figure 6. A configuration graph.

5.1 Formalization

A configuration of the framework results in a *configuration graph*; an example of such a graph is shown in Figure 6. A graph is formally defined by the tuple: $G := \langle V, E\rangle$, where V is the set of vertices and E is the set of edges. To avoid a component mismatch, we have extended the graph definition for a configuration graph with constraints. This graph is formalized with the following definition:

Definition 1. *A configuration graph is a directed graph defined by the following triplet* $\mathcal{G} := \langle \mathcal{V}, \mathcal{E}, \Gamma\rangle$, *where* $\mathcal{V}$ *is the set of vertices (classes involved in the configuration),* $\mathcal{E}$ *is the set of edges (the relationship between the classes by template arguments), and* Γ *the set of allowed edges in* $\mathcal{G}$.

The set of edges E in an ordinary graphs contains pairs of vertices $\langle v_s, v_e\rangle$, where v_s is the starting vertex and v_e is the ending vertex. This is not sufficient for us since a class can be given to a class template several times, but the template parameter will differ. Therefore each edge needs to have the template parameter associated as a label.

Definition 2. *Each edge $e \in \mathcal{E}$ is defined by a triplet $\langle v_s, v_e, p \rangle$ The element p is the name of the template parameter in the class v_s of which v_e is given to.*

Now, we have defined the configuration graph. But we have not defined how we can verify that a graph is correct. Trivially, this can be done as described in Proposition 1.

Proposition 1. *If Γ contains all allowable edges in $\mathcal{G}$ and $\mathcal{E} \subseteq \Gamma$, no component mismatch can occur.*

We want this invariant to be checked at compile time. If the invariant is not maintained, the compiler should provide a decent error message. We will in the following subsections consider different methods for verifying that the invariant described in Proposition 1 is maintained.

5.2 Concepts

The most significant contribution, proposed to C++0x, is C++ concepts. In the traditional form, polymorphism is obtained by creating a base class containing the desired interface. The interface consists of member functions which are defined as pure virtual member functions. A *pure virtual member function* is a member function, defined in a base class, which each subclass must implement. In the example below, we show how this language feature works.

Example 11. Consider the following C++ program:

```
#include <iostream>

class BaseClass {
public:
  virtual void a_member() = 0;

};

class SubClass : public BaseClass {
public:
  void another_member() {
    std::cout << "Test" << std::endl;
  }
};

int main() {
  SubClass s;
}
```

The member function `a_member` in `BaseClass` is declared to be a pure virtual member function. The subclass `SubClass` does not implement this member function which results in the following error message (generated using gcc 4.3.2):

```
test-virtual.c++: In function ’int main()’:
```

```
test-virtual.c++:17: error: cannot declare variable 's' to be of
    abstract type 'SubClass'
test-virtual.c++:9: note:   because the following virtual functions
    are pure within 'SubClass':
test-virtual.c++:5: note:       virtual void BaseClass::a_member()
```

These error messages are hardly understandable, but at least they give a hint of where to look for the error. □

Until now, we had no language features which allowed us to create a equivalent restriction for types in a template-based setting. More precisely, we cannot specify restrictions of the types that can be given to class templates as template arguments. C++ concepts provide such a language feature. With C++ concepts the programmer can specify concepts and concept maps. A *concept* is a set of constraints (members, axioms) one or more types must satisfy. After a concept is defined the programmer can use the concept by specifying that the argument of some template parameter, in a class or function template, should satisfy the concept. A *concept map* is used to make types which do not satisfy a concept, satisfy the concept by defining a mapping. Example 12 gives more details of how concepts are applied. The main reason for introducing concepts is to provide better error messages when an incompatible type is given as template argument to a class template. As we discussed earlier such a mismatch would produce several screen lengths of error messages by a contemporary compiler. With C++ concepts the compiler would simply write that the type given as template argument does not satisfy the required concept.

Example 12. Let us consider the function template `min`. Given two arguments of the same type, this function template returns the argument with the smallest value. This function template is part of the C++ standard library. Let us consider the scenario where `min` is called with a type which does not provide `operator<(...)`. This will cause an error, and the error messages produced by the compiler may not be helpful. With the declaration defined below using concepts, the compiler will simply write that the type `T` does not satisfy the concept `LessThanComparable`.

```
auto concept LessThanComparable<typename T> {
  bool operator<(T, T);
}

template <LessThanComparable T>
void min(T const& v1, T const& v2) {
  if(v1 < v2) {
    return v1;
  }
  return v2;
}
```

□

C++ concepts are more than just verifying that a type satisfies a specified interface. A new way of overloading, concept-based overloading, is possible, which is more elegant than, for example, tag dispatching. We briefly discussed this issue in the paper on component frameworks [18]. The idea in concept-based overloading is that several function templates can be defined with the same parameters and return type. The difference between these function templates is in the concepts which the input types should match. For example, we can have two versions of `min`, one using `LessThanComparable` and one using `GreaterThanComparable`. Concepts are described in depth in [14]. According to our formal definition of configuration graph verification, concept checking can be performed using the definition below:

Definition 3. *Given $\mathcal{V}$ and $\mathcal{E}$, we generate the set Γ using the following expression: $\Gamma := \{\langle v_s, v_e, p\rangle \mid \langle v_s, v_e, p\rangle \in \mathcal{E} \wedge \Phi(v_s, v_e, p) = \mathbf{true}\}$. Φ returns true if there exists a concept for parameter p in v_s and it is satisfied by v_e.*

In July 2009 the C++ standards committee decided to remove concepts from the C++0x specification [31]. This means that we need to wait for the next C++ standard to appear to get tools for solving the problems of component mismatches. Before this decision we had doubts [18] that concepts could solve our problems completely. We questioned whether concepts are strong enough to solve the problems encountered in library development. Consider two algorithms, encapsulated in functors, with the same interface. The behaviour of these algorithms is different, the first algorithm solves problem $\mathcal{X}$ and the second algorithm solves problem $\mathcal{Y}$. These functors are likely to be given to class templates as template argument. Let us consider the scenario where the incompatible functor is given to a class template. A run-time error may occur or even worse a semantic error, meaning that the program does not behave as desired. Such a semantic error is usually harder to find than a run-time error. Such a problem is shown in Example 13. With the current specification of concepts, we do not have an obvious way of performing a check at compile time which avoids this scenario. In context of our formal definition, this means that the set Γ may contain edges which are not allowed in a logical sense, since the set is constructed by the interfaces.

Example 13. Consider the algorithms `random_shuffle` and `sort` encapsulated in functors. These algorithms are defined in the C++ standard. The interface of these algorithms is the same:

```
template <RandomAccessIterator I>
void sort(I first, I last);
template <RandomAccessIterator I>
void random_shuffle(I first, I last);
```

Clearly the algorithms are designed to solve two different tasks, `sort` sorts a sequence enclosed by the given iterators and `random_shuffle` gives a random permutation of the sequence enclosed by the two iterators. Let us consider the scenario where `random_shuffle` is given to a class template `X` as template argument. This class template expects that `sort` is given as template argument. After the functor is invoked, the class template `X` performs binary search. Since the sequence is not sorted, no elements are likely to be found. This means that the program will probably not perform the right computation and the output produced by the program will be incorrect. Such an error can be hard to find in a complicated system with a large configuration graph. □

Techniques for contract programming could be used to avoid the scenario given in Example 13. For example, the D programming language proposes that a function should consist of three blocks `in`, `out`, and `body` [9]. All preconditions are put in the `in` block and all postconditions are put in the `out` block, and the functionality of the function is put in the `body` block. Such a contract for a function can be implemented without language features; one should simply put the preconditions at the beginning of the function and the postconditions in the end of the function. However, having the pre- and postconditions in the declaration of the function makes them clearer and idiomatic. Techniques for contract programming might solve some of our problems, but to determine whether the pre- and postconditions are true might give a performance overhead, since pre- and postconditions are evaluated at run time.

Example 14. A version of the class template `X` from Example 13, which verifies the post condition of the call to the functor, is shown below:

```
template <typename C, typename F>
class X {
public:
  bool member(C const& c, C::value_type* es, std::size_t
      number_of_es) {
    F f;
    f(c.begin(), c.end());
    /* check post condition of the call to f */
    assert(std::is_sorted(c.begin(), c.end());
    for(int i=0; i < number_of_es; ++i) {
      if(!std::binary_search(c.begin(), c.end(), es[i])) {
        return false;
      }
    }
    return true;
  }
};
```

Notice that we use $\Theta(n)$ worst-case time (for a container `c` storing n elements) to verify that the post condition is valid. □

Since concepts will not appear in the C++0x standard, we need to consider alternatives for specifying the relationship between our components (specifically which components can be accepted by our frameworks and containers), such that we can provide decent error messages. We know that the C++ standard is revised every fifth year, therefore it would take at least five years for concepts to appear in the C++ standard. We believe that component frameworks will not gain widespread acceptance unless we solve the problems related to their use.

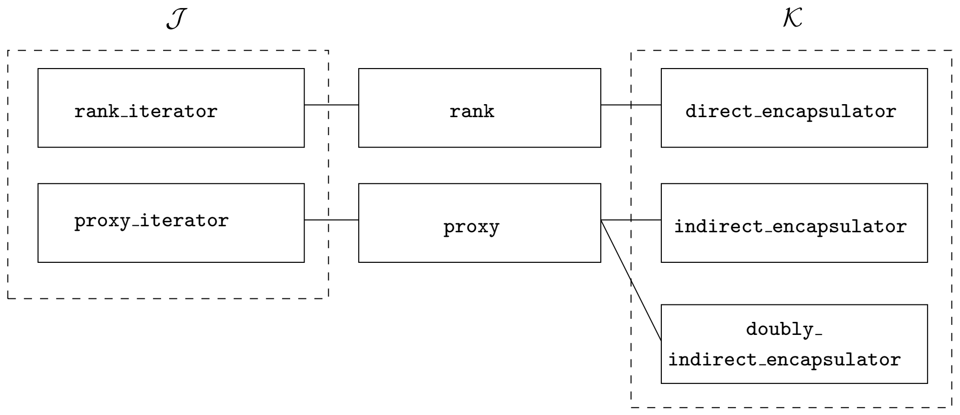

Figure 7. A family graph.

One alternative, which can be used as a substitute for C++ concepts, is the Boost concept checking library [4]. This library provides a functionality similar to C++ concepts, but it is implemented using C macros. The concepts are defined as regular classes (or structs), and in each class there are several assertions (by the macro `BOOST_CONCEPT_ASSERT`), which are similar to regular assertions (`assert` from the C standard library), just for concept classes. The macro `BOOST_CONCEPT_REQUIRES` is similar to the assertion macro, but it is used for function templates. The last significant macro is `BOOST_CONCEPT_USAGE` where it is possible to specify some desired properties of the types involved in the concepts (for example, copy construction and assignment).

On the Boost concept checking library homepage [4], several examples show that the error messages produced by the BOOST concept checking library are much better (and shorter) than the error messages which would be produced by the compiler (without any concept checking). However, the concept checking library still produces several lines of error messages, but what we desire is a simple one line error message which tells the user what the problem is. For example if `direct_encapsulator` is given to `proxy_iterator`, the compiler should just stop compilation with an error message, saying that the encapsulator provided does not fit the iterator. In other parts of the library the Boost concept checking mechanism may be useful, for example, to verify that the types, given to generic algorithms, satisfy some requirements.

5.3 Component families

A simple approach to perform the check whether an iterator fits an encapsulator is to define a component family. The idea of *component families* is that we have two finite sets of components; the first set (denoted $\mathcal{J}$) represents classes that accept one or more classes drawn from the second set (denoted

$\mathcal{K}$). The family is a relation of which classes in $\mathcal{J}$ can accept classes in $\mathcal{K}$. Given $j \in \mathcal{J}$ and $k \in \mathcal{K}$ we want to check if there is a connection between j and k.

We can perform such a check at compile time using C++ metaprogramming, but how can we report an error at compile time? C++0x provides *static assertions* (or *compile-time assertions*) [21, 15], which are similar to `assert` from the C standard library; the difference is that the static assertions are evaluated at compile time. The built-in function `static_assert` takes two arguments, the condition which should evaluate to true, and an error message. The condition must be written such that it can be evaluated at compile time. Fortunately, `static_assert` is available in gcc 4.3 and 4.4 [12] (by compiler option `-std=c++0x`), so we can already now test the code. It has been possible to create a mechanism similar to static assertions before (see, for example, [22, 16]), however there has been no way to provide a user-defined error message which is readable, i.e. a plain text string is printed. This is the significant improvement in C++0x static assertions.

Let us reconsider the problem of verifying that an encapsulator fits an iterator. In Figure 7, a component family graph is shown. As illustrated in the figure, we create two families which we call `rank` and `proxy`; these families have relations between iterators and encapsulators. The set of iterator classes is represented as the set $\mathcal{J}$ and the set of encapsulators is represented as the set $\mathcal{K}$. According to our formal definition of our configuration graph, we need to consider, how to construct $\Gamma_{\mathtt{f}} \subseteq \Gamma$.

Definition 4. *Given $\mathcal{V}$ and $\mathcal{E}$, we generate the set $\Gamma_{\mathtt{f}}$ using the following expression: $\Gamma_{\mathtt{f}} := \{\langle v_s, v_e, p\rangle \mid \langle v_s, v_e, p\rangle \in \mathcal{E} \wedge v_s \in \mathcal{J} \wedge v_e \in \mathcal{K} \wedge \Phi_{\mathtt{f}}(\langle v_s, v_e\rangle) = \mathbf{true}\}$. The function $\Phi_{\mathtt{f}}$ returns true if there exists a connection between j and k by the family relation `f`.*

The observant reader may question, why we need families. We could just implement a mechanism which specified and verified the set Γ (as defined in Proposition 1). If we did so, elements in $\mathcal{J}$ were directly connected to elements in $\mathcal{K}$. To justify the need of families, let us take a look at Figure 7. In the set $\mathcal{J}$ there is a one-to-one correspondence to a family. However, in the future there might be a many-to-one correspondence, since it is highly relevant that several iterator classes can occur for each family. It is also possible that other components (which are not iterators) can be a part of $\mathcal{J}$. Consider a kernel which only allows direct encapsulation.

For the approach without families, the developer of a kernel would need to declare the specific encapsulators that could be accepted by the kernel. If more encapsulators would come, the declaration would require changes. With the family approach, the new encapsulators should just be made member of the appropriate family. Since we allow our users to create there own components, including encapsulators, the family approach is preferable. To implement component families we would need six basic operations:

MAKE-FAMILY(f): Creates an empty family f with no connections.

ADD$_{\mathcal{K}}$(c): Adds a class c to the set $\mathcal{K}$.

$\textsc{add}_{\mathcal{J}}(c,p)$: Adds a class c to the set $\mathcal{J}$ and stores template parameter p.

$\textsc{connect}_{\mathcal{J}}(j,f)$: Connects $j \in \mathcal{J}$ to f.

$\textsc{connect}_{\mathcal{K}}(f,k)$: Connects f to $k \in \mathcal{K}$.

$\textsc{connection-exists}(f,j,k)$: Calls $\Phi_{\mathtt{f}}(\langle j,k\rangle)$.

We have implemented these operations in C++ for the family graph given in Figure 7. The implementation is not directly equivalent to the operations described above. But all together they make it possible to verify that a part of the configuration graph is correct.

We have decided to use C macros for the implementation, since the check should be performed at compile time, we have no other options since we desire a small and readable declaration for each operation. The macros are shown in Listing 11 (lines 1–16). We will now discuss the implementation of each macro.

`NEW_ENCAPSULATOR_FAMILY(f)`: implements the operation $\textsc{make-family}(f)$ by generating a general version of a class template named f`_classes`. This class template keeps a constant named `positive` which value is set to zero.

`JOIN_ENCAPSULATOR_FAMILY(f, e)`: implements the operations $\textsc{add}_{\mathcal{K}}(c)$ and $\textsc{connect}_{\mathcal{K}}(f,k)$. This is implemented by partial specializing the class template f`_classes` for the encapsulator `e` given as argument to the macro. In this specialization, the `positive` constant is set to one.

`IS_ENCAPSULATOR_IN_FAMILY(e, f)`: implements the operations $\textsc{add}_{\mathcal{J}}(c,p)$, $\textsc{connect}_{\mathcal{J}}(j,f)$, and $\textsc{connection-exists}(f,j,k)$. Because of simplicity we do not maintain the set $\mathcal{J}$. When this macro is used, the static assertion is generated. The constraint of this assertion is that the class template f`_classes`, given the encapsulator `e` as template argument, provides a constant `positive` of which value is one.

The declarations, which realize the families given in Figure 7, are shown in Listing 11 lines 18–31.

Some STL components are classified, for example, iterators have a tag such that the generic algorithms and other function templates can provide several different versions, typically one for random-access iterators and one for bidirectional iterators (see, for example, `advance`). The family approach is similar; we could inside each encapsulator define a type which stated the family of the encapsulator. A problem is likely to occur: if an encapsulator does not provide the required type, the compiler will give an error instead of printing the error message given by the static assertion. This problem could be solved by applying the SFINAE principle [33]. In general, we wanted to avoid this principle since the code becomes less readable. We used partial specializations to avoid a type error, since if an encapsulator is not in the required family, the general version will be used, and the static assertion will fail. Otherwise one of the specializations is used, and the invariant given by the assertion will be fulfilled.

Listing 11. Declarations of component families.

```
#define NEW_ENCAPSULATOR_FAMILY(f) template <typename V, typename A,
     typename E> \
  class f##_classes { \
  public: \
    enum {positive = 0}; \
  };

#define JOIN_ENCAPSULATOR_FAMILY(f, e) template <typename V,
    typename A> \
  class f##_classes< V, A, e <V, A> > { \
  public: \
    enum {positive = 1}; \
  };

#define IS_ENCAPSULATOR_IN_FAMILY(e, f) static_assert( \
  f##_classes<typename e::value_type, typename e::allocator_type, e
      >::positive == 1, \
  "Encapsulator "  #e " is not in family _" #f "_" \
);

namespace cphstl {
  NEW_ENCAPSULATOR_FAMILY(proxy)
  NEW_ENCAPSULATOR_FAMILY(rank)
  JOIN_ENCAPSULATOR_FAMILY(proxy, doubly_indirect_encapsulator)
  JOIN_ENCAPSULATOR_FAMILY(proxy, indirect_encapsulator)
  JOIN_ENCAPSULATOR_FAMILY(rank, direct_encapsulator)
}

template <typename R, typename is_const = false, typename E =
    typename R::encapsulator_type>
class proxy_iterator {
  IS_ENCAPSULATOR_IN_FAMILY(E, proxy)
public:
  ...
};
```

Listing 12. An example error message using component families.

```
/home/bo/CPHSTL/Source/Iterator/Code/proxy-iterator.h++: In
    instantiation of 'cphstl::proxy_iterator<cphstl::
    vector_framework<int, std::allocator<int>, cphstl::dynamic_array
    <int, std::allocator<int>, cphstl::direct_encapsulator<int, std
    ::allocator<int> >, false> >, false, cphstl::direct_encapsulator
    <int, std::allocator<int> > >':
use-test.c++:41:   instantiated from here
/home/bo/CPHSTL/Source/Iterator/Code/proxy-iterator.h++:32: error:
    static assertion failed: "Encapsulator E is not in family
    _proxy_"
```

Listing 13. A backward-compatible version of IS_ENCAPSULATOR_IN_FAMILY.

```
#ifdef __GXX_EXPERIMENTAL_CXX0X__
#define IS_ENCAPSULATOR_IN_FAMILY(e, f) static_assert( ... )
#else
#define IS_ENCAPSULATOR_IN_FAMILY(e, f)
#endif
```

We cannot assume that all our users use a version of gcc which provides static assertions. Since static assertions may not be available in older compilers, the user will get an error if he tries to use the vector component framework, since the `proxy_iterator` (as defined in Listing 11) uses a static assertion. One way of providing backward compatibility is to maintain a second iterator class which is insecure, in the meaning of no verification of the components is performed. Because of code-reuse considerations this solution is unattractive. Instead we can take advantage of the macro `__GXX_EXPERIMENTAL_CXX0X__`. If gcc is invoked with the support for C++0x extensions that macro is defined. We can now rewrite `IS_ENCAPSULATOR_IN_FAMILY` to be backward compatible, the code is shown in Listing 13. If gcc is not invoked with support for the `C++`0x extensions, we do no verification of whether the components fit together. Instead of doing no verification, we could use the macro-based static assertions as discussed in [22, 16].

We have now argued that the component family approach can be used to verify that the appropriate encapsulator is given to an iterator, i.e. we computed the set $\Gamma_{\mathtt{f}}$ for a family `f`. Let `F` denote the set of families involved in a configuration. The question is how can we compute $\Gamma := \bigcup_{\mathtt{f} \in \mathtt{F}} \Gamma_{\mathtt{f}}$? If we assume that all types are classified to belong in a certain family, we can do that. But can we define families for all types? The example below shows that it may be difficult to maintain a complete set of families for possible configurations of each framework.

Example 15. Consider an instance of `set` given the comparator `std::less`. We can define a family for the value types which can be accepted by `set` if the user defines his or her self-defined types to be members of this family. The comparator `std::less` requires that the types provide `operator<()`; hence a new family should be defined and the user-defined types should be included in this family. Likewise, for the comparator `std::greater` and other comparators. □

From this example, we can deduce that component families are not a good idea for external components, like the value type. However, for internal components in the framework, it seems like a good approach to avoid a component mismatch. For external components, concepts may be the best approach to detect a component mismatch, since the dependency between the components is defined by the interfaces, and does not rely on any predefined relationship.

6. Concluding remarks

In this work we studied the disadvantages related to component frameworks found in [18]. We believe that the result of our work is the following:

– We improved the usability of component frameworks with respect to integrated use. We found in [17] that just a few studies has been performed on the use of libraries therefore it would be interesting to find out (by an empirical study) whether the methods described in this work will improve usability in practice.
– We hope that this work will lead to acceptance of adaptable component frameworks in a template-based setting. Other library developers may have rejected a design, similar to the one given in [18], because they found the same disadvantages. The existence of the named template argument language extension makes such a design possible, not just in theory, but in practice.
– We formalized the component-mismatch problem by applying basic graph theory. We hope that this point of view can be useful for reasoning about, for example, `C++` concepts. Also, we hope that we emphasized that the existence of concepts in C++ is crucial for detecting a component mismatch for some kinds of components in the framework.

Software availability

The source code relevant for this study can be found in Appendix, including the full source code for the preprocessor.

Acknowledgements

I want to thank everybody who contributed to the CPH STL project. Their work gave us a starting point for designing the vector component framework; without the existence of the vector component framework, the problems studied would be unknown to us. Also I want to thank my supervisor Jyrki Katajainen for his collaboration on the syntax of the named template argument language extension and for his valuable feedback.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3**, 5 (1962), 1259–1263.
[2] A. W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press (1998).
[3] A. Aue, Improving performance with custom pool allocators for STL, *Dr. Dobb's Journal* (2005).
[4] Boost Community, The Boost concept check library, Website accessible at `http://www.boost.org/doc/libs/1_39_0/libs/concept_check/concept_check.htm` (2000–2007).

[5] Boost Community, Boost C++ libraries, Website accessible at `http://www.boost.org/` (2000–2009).
[6] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
[7] Computational Geometry Algorithms Library, CGAL User and Reference Manual, Worldwide Web Document (2009). Available at `http://www.cgal.org/Manual/last/doc_html/cgal_manual/contents.html`.
[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).
[9] Digital Mars, Contract programming, Worldwide Web Document. Available at `http://www.digitalmars.com/d/2.0/dbc.html`.
[10] A. Duret-Lutz, T. Géraud, and A. Demaille, Design patterns for generic programming in C++, *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems*, The USENIX Association (2001), 189–202.
[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Professional (1995).
[12] GNU, Status of experimental C++0x support in GCC 4.4, Worldwide Web Document. Available at `http://gcc.gnu.org/gcc-4.4/cxx0x_status.html`.
[13] GNU, *libstdc++*, Website accessible at `http://gcc.gnu.org/onlinedocs/libstdc++/` (1999-2008).
[14] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, *SIGPLAN Notices* **41**, 10 (2006), 291–310.
[15] ISO/NEC, Working draft, standard for programming language C++, Document number **N2914**, The C++ Standards Committee (2009).
[16] J. Katajainen, New CPH STL headers `<compile-time-assert>` and `<type>`, Worldwide Web Document (2001). Available at `http://www.cphstl.dk/Presentation/3rd-STL-workshop/New-headers/Jyrki-17.12.2001.pdf`.
[17] J. Katajainen and B. Simonsen, Applying design patterns to specify the architecture of a generic program library (2008).
[18] J. Katajainen and B. Simonsen, Adaptable component frameworks: Using `vector` from the C++ standard library as an example, *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, ACM (2009), 13–24.
[19] J. Katajainen and B. Simonsen, The design and description of a generic software library (2009, work in progress).
[20] B. W. Kerninghan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall Inc (1978).
[21] R. Klarer, J. Maddock, B. Dawes, and H. Hinnant, Proposal to add static assertions to the core language (rev. 3), Document number **N1720**, The C++ Standards Committee (2004).
[22] J. Maddock and S. Cleary, Boost.StaticAssert, Worldwide Web Document (2005). Available at `http://www.boost.org/doc/libs/1_39_0/doc/html/boost_staticassert.html`.
[23] M. Mernik, J. Heering, and A. M. Sloane, When and how to develop domain-specific languages, *ACM Computing Surveys* **37**, 4 (2005), 316–344.
[24] M. Michaud and M. Wong, Forwarding and inherited constructors (rev. 2), Document number **N1898**, The C++ Standards Committee (2005).
[25] Python Software Foundation, The official website of the Python programming language, Website accessible at `http://www.python.org/` (1990–2009).
[26] B. Simonsen, Foundations of an adaptable container library, M. Sc. Thesis, Department of Computer Science, University of Copenhagen (2009).
[27] B. Simonsen, A framework for implementing associative containers, CPH STL Report **2009-3**, Department of Computer Science, University of Copenhagen (2009).
[28] B. Simonsen, Towards stronger guarantees: Safer iterators, CPH STL Report **2009-1**, Department of Computer Science, University of Copenhagen (2009).
[29] B. Simonsen, View programming, Internal progress report (available on request),

Department of Computer Science, University of Copenhagen (2009).
[30] E. Sitarski, Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays, *Dr. Dobb's Journal* **21**, 11 (1996).
[31] B. Stroustrup, The C++0x "Remove Concepts" Decision, *Dr. Dobb's Journal* (2009).
[32] H. Sutter, Proposed addition to C++: Typedef templates, Document number **N1373**, The C++ Standards Committee (2002).
[33] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley (2003).

Appendix: Table of contents

Appendix A. Selective use

Appendix A.1 `stl-predefined-vectors.h++`

```
/*

  Predefined vectors for the vector component framework

  Author: Bo Simonsen, November 2009
*/

#include "stl-vector.h++"
#include "vector-framework.h++"
#include "dynamic-array.h++" // defines cphstl::dynamic_array_kernel
#include "hashed-array-tree.h++" // defines cphstl::
    dynamic_array_kernel
#include "indirect-encapsulator.h++" // defines cphstl::
    dynamic_array_kernel
#include "doubly-indirect-encapsulator.h++" // defines cphstl::
    dynamic_array_kernel
#include "proxy-iterator.h++" // defines cphstl::rank_iterator
#include "rank-iterator.h++" // defines cphstl::rank_iterator
#include <memory>

namespace cphstl {

  template <typename V,
            typename A = std::allocator<V> >
  class fast_vector : public vector<V, A,
                                    vector_framework<V, A,
                                        dynamic_array<V, A,
                                        direct_encapsulator<V, A>,
                                        true > >,
                                    rank_iterator< vector_framework<
                                        V, A, dynamic_array<V, A,
                                        direct_encapsulator<V, A >,
                                        true > >, false>,
                                    rank_iterator< vector_framework<
                                        V, A, dynamic_array<V, A,
                                        direct_encapsulator<V, A >,
                                        true > >, true> > {
  private:
    typedef vector<V, A, vector_framework<V, A, dynamic_array<V, A,
        direct_encapsulator<V, A>, true > >, rank_iterator<
        vector_framework<V, A, dynamic_array<V, A,
        direct_encapsulator<V, A >, true > >, false>, rank_iterator<
         vector_framework<V, A, dynamic_array<V, A,
        direct_encapsulator<V, A >, true > >, true> > superclass;
  public:
    explicit fast_vector(A const& a = A()) : superclass(a) {
    }
  explicit fast_vector(typename superclass::size_type s, V const&
      v = V(), A const& a = A()) : superclass(s, v, a) {
  }

  template <typename K>
  fast_vector(K f, K l, A const& a = A()) : superclass(f, l, a) {
  }

  fast_vector(fast_vector const& v) : superclass(v) {
  }
  fast_vector& operator=(fast_vector const& v) {
    superclass::operator=(v);
    return (*this);
  }
};

template <typename V,
          typename A = std::allocator<V> >
class safe_vector : public vector<V, A,
                                  vector_framework<V, A,
                                      dynamic_array<V, A,
                                      doubly_indirect_encapsulator
                                      <V, A>, false > >,
                                  proxy_iterator< vector_framework
                                      <V, A, dynamic_array<V, A,
                                      doubly_indirect_encapsulator
                                      <V, A >, false > >, false>,
                                  proxy_iterator< vector_framework
                                      <V, A, dynamic_array<V, A,
                                      doubly_indirect_encapsulator
                                      <V, A >, false > >, true> >
                                      {
private:
  typedef vector<V, A, vector_framework<V, A, dynamic_array<V, A,
      doubly_indirect_encapsulator<V, A>, false > >,
      proxy_iterator< vector_framework<V, A, dynamic_array<V, A,
      doubly_indirect_encapsulator<V, A >, false > >, false>,
      proxy_iterator< vector_framework<V, A, dynamic_array<V, A,
      doubly_indirect_encapsulator<V, A >, false > >, true> >
      superclass;


public:
  explicit safe_vector(A const& a = A()) : superclass(a) {
  }
  explicit safe_vector(typename superclass::size_type s, V const&
      v = V(), A const& a = A()) : superclass(s, v, a) {
  }

  template <typename K>
  safe_vector(K f, K l, A const& a = A()) : superclass(f, l, a) {
  }

  safe_vector(safe_vector const& v) : superclass(v) {
  }
  safe_vector& operator=(safe_vector const& v) {
    superclass::operator=(v);
    return (*this);
  }
};

template <typename V,
          typename A = std::allocator<V> >
class compact_vector : public vector<V, A,
                                     vector_framework<V, A,
                                         hashed_array_tree<V, A,
                                         direct_encapsulator<V, A>
                                          > >,
                                     rank_iterator<
                                         vector_framework<V, A,
                                         hashed_array_tree<V, A,
                                         direct_encapsulator<V, A
                                         > > >, false>,
                                     rank_iterator<
                                         vector_framework<V, A,
                                         hashed_array_tree<V, A,
                                         direct_encapsulator<V, A
                                         > > >, true> > {
private:
  typedef vector<V, A, vector_framework<V, A, hashed_array_tree<V,
       A, direct_encapsulator<V, A> > >, rank_iterator<
      vector_framework<V, A, hashed_array_tree<V, A,
      direct_encapsulator<V, A > > >, false>, rank_iterator<
      vector_framework<V, A, hashed_array_tree<V, A,
      direct_encapsulator<V, A > > >, true> > superclass;

public:
  explicit compact_vector(A const& a = A()) : superclass(a) {
  }
  explicit compact_vector(typename superclass::size_type s, V
      const& v = V(), A const& a = A()) : superclass(s, v, a) {
  }

  template <typename K>
  compact_vector(K f, K l, A const& a = A()) : superclass(f, l, a)
       {
  }

  compact_vector(compact_vector const& v) : superclass(v) {
  }
  compact_vector& operator=(compact_vector const& v) {
    superclass::operator=(v);
    return (*this);
  }
  };
}
```

Appendix A.2 `predefined-vectors-test.c++`

```
/*

   Test program for the predefined vector classes. This is not a
       full test
   since the vector implementation is already tested using the smoke
       -test.
   This test is just testing the constructors.

   Author: Bo Simonsen, November 2009

*/

#include <cassert>
#include <memory> // defines std::allocator
#include "stl-predefined-vectors.h++"

template <typename V>
void test_members(V& v) {
  v.push_back(5);
  v.pop_back();
  v.push_back(7);
  v.clear();
}

#define create_vectors(x, n) x<int> v##n; \
  x<int> vv##n(100);\
  x<int> vvv##n(arr, arr+3);\
  x<int> vvvv##n(v##n);

int main() {
  int arr[] = {1,2,3};
  create_vectors(cphstl::fast_vector, 1)
  create_vectors(cphstl::safe_vector, 2)
  create_vectors(cphstl::compact_vector, 3)
  test_members(v1);
  v1 = vv1;
  test_members(v2);
  v2 = vv2;
  test_members(v3);
  v3 = vv3;
}
```

Appendix B. The named template argument preprocessor

Appendix B.1 `nta.py`

```
#!/bin/python
```

```

# Preprocessor for named template arguments.
# The preprocessor should be invoked instead of g++.
# All arguments are given to g++.
# Made by Bo Simonsen <bo@geekworld.dk>, June 2009

import sys
import re
import os
import os.path
import time

include_dirs = []
included_files = []

p_include = re.compile('(#include (")([^ ]+)("))*')
p_namespace = re.compile(".*namespace(.*)$")
p_template = re.compile(".*template[ ]*<(.+)>[ ]*class ([^;:]+)")
p_parameter = re.compile('(typename|bool|int|char) ([^ ]+)[ ]*=?[
    ]*(.*)')

gd = {}

def parse_cpp_block(buf, l):
  def split_blocks(buf):
    blocks = []

    count = 0
    j = 0
    for i in xrange(0, len(buf)):
      if buf[i] == '<':
        count += 1
      if buf[i] == '>':
        count -= 1
      if buf[i] == ',' and count == 0:
        blocks.append(buf[j:i])
        j = i+1
      i += 1

    blocks.append(buf[j:])

    return blocks

  def transform_default_value((k,v)):
    i = v.find("<")
    j = v.rfind(">")

    if i == -1 and j == -1:
      return (k,(v, []))

    args = v[i+1:j]
    new_v = [x.strip() for x in split_blocks(args)]

    return (k,(v[:i], new_v))

  d = {}

  i = -1
  j = -1

  while 1:
    old_j = j+1

    i = buf.find("{", i+1)
    if i == -1:
      break

    count = 1

    j = i
    while count > 0:
      if buf[j+1] == '{':
        count += 1
      elif buf[j+1] == '}':
        count -= 1
      j+=1

    block = buf[i+1:j]
    newbuf = buf[old_j:i]

    if p_namespace.match(newbuf):
      tok = p_namespace.split(newbuf)[1].strip()
      l.append(tok)

    elif p_template.match(newbuf):
      arr = p_template.split(newbuf)
      class_name = arr[2].strip()
      blocks = [x.strip() for x in split_blocks(arr[1])]

      # We don't care about specialization!
      # i.e. class class_name<...>
      if class_name.find("<") == -1 and class_name.find(">") == -1:
        new_class_name = "::".join(l) + "::" + class_name
        new_blocks = [transform_default_value(tuple(p_parameter.
            split(x)[2:4])) for x in blocks]
        d[new_class_name] = new_blocks

    d.update(parse_cpp_block(block, l))

  return d

def parse_cpp_file(filename):

  def strip_nl_tab(buf):
    new_buf = ""
    for i in buf:
      if i != '\n' and i != '\t':
        new_buf += i
      else:
        new_buf += " "
    return new_buf

  fh = open(filename, "r")
  buf = fh.read()
  fh.close()

  buf = strip_nl_tab(buf)

  d = parse_cpp_block(buf, [])

  includes = p_include.split(buf)

  for i in xrange(0, len(includes)/5):
    fn = includes[(i*5)+3]
    if fn not in included_files:
      for l in include_dirs:
        try:
          d.update(parse_cpp_file(l + "/" + fn))
          included_files.append(fn)
        except:
          pass

  return d

def parse_nta_block(x):
  d = {}
  i = 0
  j = 0
  blocks = []

  while i < len(x):

    if x[i] == ',':
      blocks.append(x[j:i])
      j = i+1
    if (x[i] == '!' and x[i+1] == '<') or x[i] == '<':
      if x[i] == '<':
        tmp = i+1
      else:
        tmp = i+2

      count = 1
      while count > 0:
        if (x[tmp] == '!' and x[tmp+1] == '<') or x[tmp] == '<':
          count += 1
        if (x[tmp] == '!' and x[tmp+1] == '>') or x[tmp] == '>':
          count -= 1
        tmp = tmp+1

      if x[i] == '<':
        i = tmp-1
      else:
        i = tmp

    i+=1

  blocks.append(x[j:])

  for i in blocks:
    j = i.strip()
    arr = j.split('=', 1)

    key = arr[0].strip()
    val = arr[1].strip()

    q = val.find("!<")
    if q == -1:
      d[key] = (val, {})
    else:
      w = val.rfind("!>")
      d[key] = (val[:q], parse_nta_block(val[q+2:w]))
  return d



def generate_code(a, d, dd):

  result = a + "<"

  for (k, v) in dd[a]:
    if d.has_key(k):
      x = d[k]
      if x[1] != {}:
        y = generate_code(x[0], x[1], dd)
        result += y + ","
        gd[k] = y
      elif dd.has_key(x[0]):
        y = generate_code(x[0], {}, dd)
        result += y + ","
        gd[k] = y
      else:
        result += x[0] + ","
        gd[k] = x[0]
    elif gd.has_key(k):
      result += gd[k] + ","
    else:
      (a, b) = v
      if b != []:
```

```
      y = a + "<"
      for i in b:
        if gd.has_key(i):
          y += gd[i] + ","
      y += "> "

      result += y + ","
      gd[k] = y
    elif gd.has_key(a):
      result += gd[a] + ","
      gd[k] = gd[a]
    else:
      result += a + ","
      gd[k] = a

  result += "> \n"
  result = result.replace(",>", " >")

  return result

def parse_it(input_file):
  dd = parse_cpp_file(input_file)

  result_fn = "/tmp/nta-%s.c++" % os.path.basename(input_file )

  fhh = open(result_fn, "w")
  fh = open(input_file)
  while 1:
    buf = fh.readline()

    if buf == "":
      break

    c = buf.count("!<")
    cc = buf.count("!>")

    while cc != c:
      tmp_buf = fh.readline()
      if tmp_buf == "":
        print "Error!"
        sys.exit(1)

      c += tmp_buf.count("!<")
      cc += tmp_buf.count("!>")
      buf += tmp_buf

    if c != 0 and cc != 0:
      x1 = buf.find('!<')
      x2 = buf.rfind('!>')

      while buf[x1] != ' ':
        x1 -= 1

      x1 += 1
      x2 += 2

      gd.clear()

      d = parse_nta_block("x=" + buf[x1:x2])['x']

      print d

      new_buf = buf[:x1]
      new_buf += generate_code(d[0], d[1], dd)
      new_buf += buf[x2:]
      fhh.write(new_buf)
    else:
      fhh.write(buf)

  fhh.close()

  return result_fn

start = time.time()

i = 1
while i < len(sys.argv):
  if sys.argv[i].startswith('-I'):
    if len(sys.argv[i]) == 2:
      include_dirs.append(sys.argv[i+1])
      i += 1
    else:
      include_dirs.append(sys.argv[i][2:])

  i+=1

result_fn = parse_it(sys.argv[-1])
cmd = "g++ " + " ".join(sys.argv[1:-1]) + " " + result_fn
""" bla """
gcc_start = time.time()
if not os.system(cmd):
  print "Compilation succeded (nta time: %fs, gcc time: %fs)" % (
      gcc_start - start, time.time() - gcc_start)
  os.unlink(result_fn)
else:
  print "Compilation failed"
```

Appendix B.2 *search-tree-framework-test.c++*

```
/*

   Test program for the named template argument preprocessor.

   The preprocessor should be invoked by:

    python nta.py -I<includedir> -I ... search-tree-framework-test.
        c++

  Author: Bo Simonsen, June 2009

*/

#include "node-iterator.h++"
#include "stl_set.h++"
#include "stl_map.h++"
#include "stl_multiset.h++"
#include "stl_multimap.h++"
#include "tree.h++"
#include "red_black_tree_balance.h++"
#include "red_black_tree_node.h++"
#include "red_black_tree_bp_node.h++"
#include "splay_tree_balance.h++"
#include "splay_tree_node.h++"
#include "avl_tree_balance.h++"
#include "avl_tree_node.h++"
#include "avl_tree_bp_node.h++"
#include "aa_tree_node.h++"
#include "aa_tree_balance.h++"

int main() {
  typedef cphstl::set!<V=int,
                       R=cphstl::tree!<
                         N=cphstl::avl_tree_node!<se=true!>,
                         B=cphstl::avl_tree_balance_policy
                       !>,
                       I=cphstl::node_iterator!<is_const=false!>,
                       J=cphstl::node_iterator!<is_const=true!>
                     !> SC;
  typedef cphstl::map!<K=char,
                       V=int,
                       A=std::allocator<std::pair<char, int> >,
                       R=cphstl::tree!<
                         V=std::pair<char, int>,
                         F=cphstl::unnamed::key_functor,
                         N=cphstl::aa_tree_node,
                         B=cphstl::aa_tree_balance_policy
                       !>,
                       I=cphstl::node_iterator!<is_const=false!>,
                       J=cphstl::node_iterator!<is_const=true!>
                     !> MC;
  typedef cphstl::multiset!<V=int,
                       R=cphstl::tree!<
                         N=cphstl::red_black_tree_node,
                         B=cphstl::red_black_tree_balance_policy,
                         is_multiset=true
                       !>,
                      I=cphstl::node_iterator!<is_const=false!>,
                      J=cphstl::node_iterator!<is_const=true!>
                    !> MSC;
  typedef cphstl::multimap!<K=char,
                      V=int,
                      A=std::allocator<std::pair<char, int> >,
                      R=cphstl::tree!<
                        V=std::pair<char, int>,
                        F=cphstl::unnamed::key_functor,
                        N=cphstl::splay_tree_node,
                        B=cphstl::splay_tree_balance_policy,
                        is_multiset=true
                      !>,
                      I=cphstl::node_iterator!<is_const=false!>,
                      J=cphstl::node_iterator!<is_const=true!>
                    !> MMC;

  SC sc;
  MC mc;
  MSC msc;
  MMC mmc;

  for(int i=0; i < 10; ++i) {
    sc.insert(i);
    msc.insert(i);
    mc[(char) i + 65] = i;
    mmc.insert(std::pair<char, int>((char) i + 65, i));
  }
}
```

Appendix B.3 vector-framework-test.c++

```
/*

   Test program for the named template argument preprocessor.

   The preprocessor should be invoked by:

     python nta.py -I<includedir> -I ... search-tree-framework-test.c++

   Author: Bo Simonsen, June 2009

*/

#include <memory> // defines std::allocator
#include "stl-vector.h++" // defines cphstl::vector
#include "vector-framework.h++"
#include "dynamic-array.h++" // defines cphstl::dynamic_array_kernel
#include "hashed-array-tree.h++" // defines cphstl::dynamic_array_kernel
```

```
#include "indirect-encapsulator.h++" // defines cphstl::dynamic_array_kernel
#include "doubly-indirect-encapsulator.h++" // defines cphstl::dynamic_array_kernel
#include "proxy-iterator.h++" // defines cphstl::rank_iterator
#include "rank-iterator.h++" // defines cphstl::rank_iterator
#include <list>

int main() {
  typedef cphstl::vector!<V=int,
                          R=cphstl::vector_framework!<
                            K=cphstl::hashed_array_tree!<
                              E=cphstl::doubly_indirect_encapsulator
                            !>
                          !>,
                          I=cphstl::proxy_iterator!<is_const=false!>,
                          J=cphstl::proxy_iterator!<is_const=true!>
                        !> C;
  C v;
  for(int i=0; i < 10; i++) {
    v.insert(v.begin(), i);
  }

  C::iterator it = v.begin();
  int i = 9;
  while(it != v.end()) {
    assert(*it == i);
    ++it; --i;
  }
}
```

Appendix C. Component families

Appendix C.1 `families.h++`

```
#include "direct-encapsulator.h++"
#include "indirect-encapsulator.h++"
#include "doubly-indirect-encapsulator.h++"

#define NEW_ENCAPSULATOR_FAMILY(f) template <typename V, typename A, typename E> \
  class f##_classes { \
  public: \
    enum {positive = 0}; \
  };

#define JOIN_ENCAPSULATOR_FAMILY(f, c) template <typename V, typename A> \
  class f##_classes< V, A, c <V, A> > { \
  public: \
    enum {positive = 1}; \
  };

#ifdef __GXX_EXPERIMENTAL_CXX0X__
#define IS_ENCAPSULATOR_IN_FAMILY(e, f) static_assert( \
  f##_classes<typename e::value_type, typename e::allocator_type, e
      >::positive == 1, \
  "Encapsulator "  #e " is not in family _" #f "_" \
);
#else
#define IS_ENCAPSULATOR_IN_FAMILY(e, f)
#endif

namespace cphstl {
  NEW_ENCAPSULATOR_FAMILY(proxy)
  NEW_ENCAPSULATOR_FAMILY(rank)
  JOIN_ENCAPSULATOR_FAMILY(proxy, doubly_indirect_encapsulator)
  JOIN_ENCAPSULATOR_FAMILY(proxy, indirect_encapsulator)
  JOIN_ENCAPSULATOR_FAMILY(rank, direct_encapsulator)
}
```

Appendix C.2 *rank-iterator.h++*

```
/*
  A rank-based iterator is just a (pointer, index) pair where the
  pointer points to the data structure containing the cell referred
      to
  and the index is the rank of that cell in the sequence of cells
  storing the elements.

  The idea of combining iterators and const iterators into the same
  class is taken from [Matt Austern. Defining iterators and const
  iterators. C/C++ User's Journal 19,1 (2001), 74-79].

  Authors: Jyrki Katajainen, Bo Simonsen (C) 2008
*/

#ifndef __CPHSTL_RANK_ITERATOR__
#define __CPHSTL_RANK_ITERATOR__

#include <cstddef> // defines std::size_t and std::ptrdiff_t
#include <iterator> // defines std::random_access_iterator_tag
#include "type.h++" // defines cphstl::if_then_else
#include <utility> // defines std::pair

namespace cphstl {

  template <typename V, typename A, typename R, typename I, typename
      J>
  class vector;

  template <typename R, bool is_const = false, typename E = typename
      R::encapsulator_type>
```

```
class rank_iterator {
#ifdef IS_ENCAPSULATOR_IN_FAMILY
  IS_ENCAPSULATOR_IN_FAMILY(E, rank)
#endif

public:
  // types

  typedef std::random_access_iterator_tag iterator_category;
  typedef typename R::value_type value_type;
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;
  typedef typename if_then_else<is_const, value_type const*,
      value_type*>::type pointer;
  typedef typename if_then_else<is_const, typename R::
      const_reference, typename R::reference>::type reference;

  typedef E entry;
  typedef typename R::surrogate_type surrogate_type;

protected:

  // types

  typedef typename if_then_else<is_const, E const*, E*>::type
      node_pointer;

public:

  // friends

  friend class rank_iterator<R, !is_const, E>;

  template <typename V, typename A, typename S, typename I,
      typename J>
  friend class cphstl::vector;

  // structors

  rank_iterator();
  rank_iterator(rank_iterator<R, false, E> const&);
  rank_iterator(rank_iterator<R, true, E> const&);
  rank_iterator& operator=(rank_iterator const&);
  ~rank_iterator();

  // operators

  reference operator*() const;
  pointer operator->() const;
  rank_iterator& operator++();
  rank_iterator operator++(int);
  rank_iterator& operator--();
```

```
    rank_iterator operator--(int);
    rank_iterator& operator+=(difference_type);
    rank_iterator& operator-=(difference_type);
    rank_iterator operator+(difference_type) const;
    rank_iterator operator-(difference_type) const;
    difference_type operator-(rank_iterator const&) const;

    template <bool both>
    bool operator==(rank_iterator<R, both, E> const&) const;

    template <bool both>
    bool operator!=(rank_iterator<R, both, E> const&) const;

    template <bool both>
    bool operator<(rank_iterator<R, both, E> const&) const;

    template <bool both>
    bool operator>(rank_iterator<R, both, E> const&) const;

    template <bool both>
    bool operator<=(rank_iterator<R, both, E> const&) const;

    template <bool both>
    bool operator>=(rank_iterator<R, both, E> const&) const;

  protected:
    // converters to be used by the container friends
    rank_iterator(std::pair<size_type, surrogate_type*> const&);
    operator std::pair<size_type, surrogate_type*>() const;

    void advance(difference_type const& n);

    surrogate_type* surrogate;
    size_type position;
  };

  template<typename R, bool both, typename E>
  rank_iterator<R, both, E>
  operator+(typename R::difference_type, rank_iterator<R, both, E>
      const&);

}

#include "rank-iterator.i++" // implements cphstl::rank_iterator

#endif
```

Appendix C.3 ***rank-iterator.i++***

```
/*
  Implementation of cphstl::rank_iterator

  Authors: Jyrki Katajainen, Bo Simonsen (C) 2008
*/

#include <cassert> // defines assert macro
#include <iostream>

namespace cphstl {

  // default constructor

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::rank_iterator()
    : surrogate(0), position(size_type()) {
  }

  // copy constructor

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R,
      false, E> const& a)
    : surrogate(a.surrogate), position(a.position) {
  }

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::rank_iterator(rank_iterator<R, true
      , E> const& a)
    : surrogate(a.surrogate), position(a.position) {
  }

  // assignment

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>&
  rank_iterator<R, is_const, E>::operator=(rank_iterator<R, is_const
      , E> const& a) {
    (*this).surrogate = a.surrogate;
    (*this).position = a.position;
    return *this;
  }

  // destructor

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::~rank_iterator() {
  }

  // operator*

  template <typename R, bool is_const, typename E>
  typename rank_iterator<R, is_const, E>::reference
  rank_iterator<R, is_const, E>::operator*() const {
```

```
  return reference((*(*(*this).surrogate).subject()).access((*this
      ).position));
}

// operator->

template <typename R, bool is_const, typename E>
typename rank_iterator<R, is_const, E>::pointer
rank_iterator<R, is_const, E>::operator->() const {
  return pointer(&(*(*this).position).content());
}

template <typename R, bool is_const, typename E>
void
rank_iterator<R, is_const, E>::advance(difference_type const& n) {
  if (n == 0)
    return;
  (*this).position += n;
}

// operator++; pre-increment

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>&
rank_iterator<R, is_const, E>::operator++() {
  (*this).advance(1);
  return *this;
}

// operator++; post-increment

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>
rank_iterator<R, is_const, E>::operator++(int) {
  rank_iterator<R, is_const, E> temporary = *this;
  (*this).advance(1);
  return temporary;
}

// operator--; pre-decrement

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>&
rank_iterator<R, is_const, E>::operator--() {
  (*this).advance(-1);
  return *this;
}

// operator--; post-decrement

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>
```

```
rank_iterator<R, is_const, E>::operator--(int) {
  rank_iterator<R, is_const, E> temporary = *this;
  (*this).advance(-1);
  return temporary;
}

// operator+=

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>&
rank_iterator<R, is_const, E>::operator+=(difference_type n) {
  (*this).advance(n);
  return *this;
}

// operator-=

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>&
rank_iterator<R, is_const, E>::operator-=(difference_type n) {
  (*this).advance(-n);
  return *this;
}

// operator+

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>
rank_iterator<R, is_const, E>::operator+(difference_type n) const
    {
  rank_iterator<R, is_const, E> temporary = *this;
  temporary.advance(n);
  return temporary;
}

// operator-

template <typename R, bool is_const, typename E>
rank_iterator<R, is_const, E>
rank_iterator<R, is_const, E>::operator-(difference_type n) const
    {
  rank_iterator<R, is_const, E> temporary = *this;
  temporary.advance(-n);
  return temporary;
}

// iterator distance

template <typename R, bool is_const, typename E>
typename rank_iterator<R, is_const, E>::difference_type
rank_iterator<R, is_const, E>::operator-(rank_iterator<R, is_const
    , E> const& a) const {
```

```
  return (*this).position - a.position;
}

// operator==

template <typename R, bool is_const, typename E>
template <bool both>
bool
rank_iterator<R, is_const, E>::operator==(rank_iterator<R, both, E
    > const& a) const {
  return (*this).position == a.position;
}

// operator!=

template <typename R, bool is_const, typename E>
template <bool both>
bool
rank_iterator<R, is_const, E>::operator!=(rank_iterator<R, both, E
    > const& a) const {
  return !(*this == a);
}

// operator<

template <typename R, bool is_const, typename E>
template <bool both>
bool
rank_iterator<R, is_const, E>::operator<(rank_iterator<R, both, E>
     const& a) const {
  return ((*this) - a) < 0;
}

// operator>

template <typename R, bool is_const, typename E>
template <bool both>
bool
rank_iterator<R, is_const, E>::operator>(rank_iterator<R, both, E>
     const& a) const {
  return a < *this;
}

// operator<=

template <typename R, bool is_const, typename E>
template <bool both>
bool
rank_iterator<R, is_const, E>::operator<=(rank_iterator<R, both, E
    > const& a) const {
  return !(a < *this);
}
```

```

  // operator>=

  template <typename R, bool is_const, typename E>
  template <bool both>
  bool
  rank_iterator<R, is_const, E>::operator>=(rank_iterator<R, both, E
      > const& a) const {
    return !(*this < a);
  }

  // operator+(int, iterator)

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E> operator+(typename R::
      difference_type n, rank_iterator<R, is_const, E> const& a) {
    return a + n;
  }

  // parametrized constructor

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::rank_iterator(std::pair<size_type,
      surrogate_type*> const& p)
    : surrogate(p.second), position(p.first) {

  }

  // convertion operator

  template <typename R, bool is_const, typename E>
  rank_iterator<R, is_const, E>::operator std::pair<size_type,
      surrogate_type*>() const {
    return std::pair<size_type, surrogate_type*>((*this).position,
        (*this).surrogate);
  }

}
```