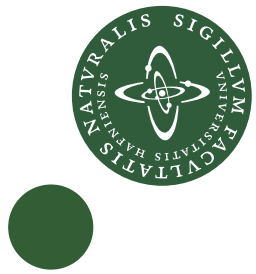


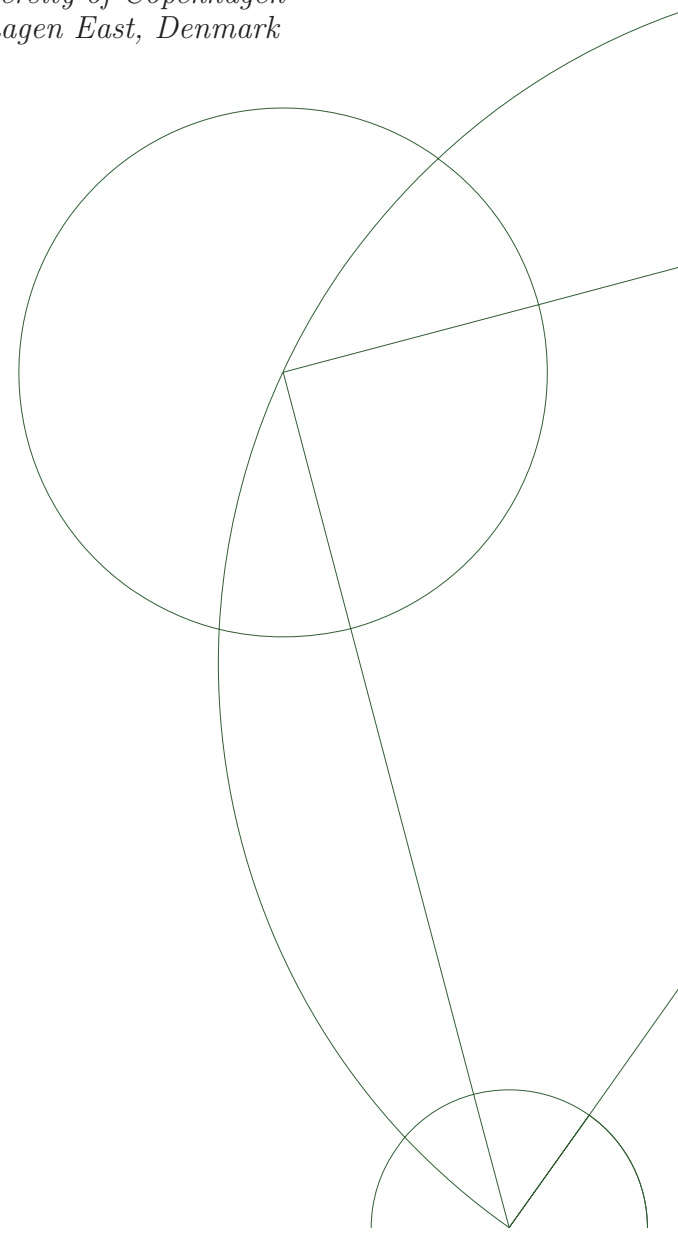
Thesis for the Master's degree in Computer Science
Speciale for cand.scient graden i datalogi

Foundations of an adaptable container library

Bo Simonsen

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

November 2009



To my family and friends

Abstract: The STL is a collection of generic containers and algorithms; the interfaces of those are specified in the C++ standard. The components in the STL are written in such a way that they can be used directly in application development. The CPH STL is an enhanced version of the STL; the significant enhancement in the CPH STL is that it provides containers with different trade-offs with respect to performance, space efficiency, and safety.

This thesis is a complete design specification of the entire library. This design specification consists of three parts: The first part is the specification of the architecture of the library. Along with this specification, we define the different concepts for individual components, but we provide no implementation details. The second part is a complete design for the `vector` container. We introduce component frameworks for STL containers, which gives a high degree of code reuse and flexibility from the user's point of view. In the third (and last) part we identify the problems related to the use of component frameworks, and provide solutions to these problems.

The key contribution of this thesis is that the architecture and the design described can be used as a starting point for designing future generic program libraries. Also, the problems related to the use of component frameworks are recurring in context of the C++ programming language. Therefore we expect that C++ programmers will find this part interesting as well.

Resumé: STL er en samling af generiske data strukturer og algoritmer. Disse generiske data strukturer og algoritmer er implementeret således at de kan benyttes uden modifikationer i applikationsudvikling. CPH STL er en forbedret udgave af STL. De nævneværdige forbedringer i CPH STL er at der tilbydes generiske data strukturer med forskellige karakteristika med hensyn til ydelse, plads effektivitet og sikkerhed.

Dette speciale er en komplet design specifikation for hele biblioteket. Denne specifikation består af tre dele: Den første del er en specifikation af arkitekturen for biblioteket. Sammen med denne specifikation definerer vi de forskellige koncepter for de individuelle komponenter; denne del indeholder ingen implementationsspecifikke detaljer. Den anden del er et komplet design for `vector`-containeren. Vi introducerer "component frameworks" for STL. Disse frameworks giver os en høj grad af kode genbrug, samt fleksibilitet for brugeren. I den tredie og sidste del identifiserer vi problemer der er relateret til brugen af component frameworks, og vi giver løsninger til disse problemer.

Specialets overordnede bidrag er at arkitekturen og designet kan bruges som udgangspunkt for at designe nye versioner af generiske programbiblioteker. Problemerne relateret til component frameworks fremkommer ofte i forbindelse med generisk programmering i C++. Derfor forventer vi at udviklere af generisk programmel i C++ vil finde denne del interessant.

Preface

This document is my Master's Thesis (Danish: *speciale*) in Computer Science written under the supervision of Jyrki Katajainen at Department of Computer Science at the University of Copenhagen. The main body of this thesis consists of three papers:

1. Jyrki Katajainen and Bo Simonsen. Applying Design Patterns to Specify the Architecture of a Generic Program Library (2008).[1]
2. Jyrki Katajainen and Bo 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.[2]
3. Bo Simonsen. Towards better usability of component frameworks. CPH STL report 2009-6, Department of Computer Science, University of Copenhagen (2009).

Acknowledgements

I want to thank everybody who contributed to the CPH STL project; their work gave me a starting point for developing the artifacts described in the thesis. I want to thank my friends and family for giving me support and encouragement to complete this work. Especially, I want to thank my mother for constantly supporting me and believing in me. Finally, but not least, I want to thank my supervisor and co-author Jyrki Katajainen for supporting me through my entire graduate study. Jyrki introduced me to many of the subjects described in this thesis, and taught me all about scientific writing. I really enjoyed the many hours we spent on writing the papers contained in the thesis.

[1] A revised version is in preparation with the title: "The Design and Description of a Generic Program Library".

[2] An extended and revised version is submitted to the special issue of Journal of Functional Programming on Generic Programming.

Table of contents

Introduction

The CPH STL [6] is an algorithmic library providing the fundamental data structures and algorithms described in most introductory textbooks on algorithms. The high-level goal of the CPH STL project is to create an enhanced version of the STL providing alternative versions of individual STL components. The components have different trade-offs with respect to performance, space-efficiency, and safety. In the beginning, the implementations of data structures and algorithms were structured in the same way as the implementations done by SGI [21] and GNU (`libstdc++`) [10].

We observed that much of the code for each individual container was identical, for example, iterators for most containers were copy-paste code. By applying design patterns and generic programming techniques we reduced the amount of code required for each container implementation significantly. We did also decouple each container from its implementation, which gave us the possibility of implementing several container interfaces for each underlying data structure. This made it possible to implement both LEDA interfaces and STL interfaces using the same underlying implementation of a data structure.

We applied the strategy and the template-method design pattern exhaustively such that our containers were no longer implementations of data structures but they became skeletons. We denote such skeleton a *component framework*. The idea is that the user gives template arguments which determine the resulting properties and behaviour of the container. These template arguments are *policies*, which are small classes with a well-defined interface and functionality. We provide several component frameworks, including a framework for `vector`. Here the user is allowed to give a kernel and an encapsulator to the framework by template arguments. The kernel determines the underlying data structure, e.g. a dynamic array or a hashed array tree, and the encapsulator determines whether the container should provide strong exception safety and referential integrity, just referential integrity, or none of the mentioned properties.

Allowing the user to configure the framework results in some undesirable usability issues. If the user does not desire the default configuration of the framework, the user has to give far too many template arguments. Even if the user just wants to change one template argument, he or she has to give all template arguments to the framework in the worst case. We experimented with C++ metaprogramming techniques to solve this problem, but several changes to the containers were required. Also, the readability of the code was reduced significantly. We wanted to maintain readability of the code, and keep the changes to the library minimal, therefore we found that the

only solution was to design a language extension. Due to this language extension the user only has to give the essential template arguments and readability of the declaration is maintained.

We found another problem when introducing frameworks. A component mismatch is likely to occur. That is because each component can only interact with a subset of all available components. If a component A is given to component B but B cannot accept A, the compiler will in the worst case fill a small computer screen with error messages. These error messages are not adequate at all, since they lack precision and readability. The upcoming C++ standard (C++0x) contained, until July 2009, C++ concepts which is a language feature that allow specification of requirements for types given as template arguments. However, C++ concepts were rejected by the C++ standards committee; this means that they will not appear in the upcoming C++ standard. Therefore we investigated other methods of verifying that our component frameworks are configured properly.

The result of the work, described in this section, is documented in the included papers.

1. Contributions

The significant contributions of this thesis are:

- We document the design and architecture of the CPH STL. That includes design problems and solutions relevant for all STL implementations and, in general, generic program libraries. It is known that such problems (and their solutions), in context of the STL, are not documented at all. The methods described in this thesis could be used as a starting point to build new generic program libraries or individual components in existing generic program libraries.
- We evaluate the language features provided by the C++ programming language for generic programming. We question whether C++ provides good enough language support for building generic program libraries. For example, we found that the tools provided by C++ are not sufficient to support adaptivity, meaning that the library would automatically select the best data structures and algorithms.
- We propose language features for C++ generic programming which solve some of the problems encountered while developing CPH STL. That includes a way of writing container declarations which improves the readability. We implemented this language extension using a preprocessor. Also, we propose that support for compile-time profiling should be built into the language.

2. Software availability

The code relevant for our study can be found at the following locations:

- The code relevant for the paper "Applying Design Patterns to Specify the Architecture of a Generic Program Library" can be found at the following URL: `http://www.cphstl.dk/Download/CPHSTL-MiniRelease-120109.tgz`.
- The code relevant for the paper "Adaptable Component Frameworks: Using `vector` from the C++ Standard Library as an Example" can be found in the CPH STL report 2009-4. The report is available at the following URL: `http://www.cphstl.dk/Source/Vector-framework/Report/doc.pdf`.
- The code relevant for the report "Towards better usability of component frameworks" can be found in the online version of the report. The report is available at the following URL: `http://www.cphstl.dk/Report/Better-Usability/better-usability.pdf`.

A compressed zip file, containing all the code, can be found at the following URL: `http://bo.geekworld.dk/files/code/thesis-files.zip`.

Background

In this chapter we will give some background on C++ generic programming, the STL, and software patterns which include idioms and design patterns. Many sources on C++ generic programming (for example [26, 1]) and the STL (for example [17, 15]) can be found. We will only need a subset of the terminology and techniques described in these sources, therefore we provide a distilled version of the terminology and techniques relevant for this study.

The well-known design patterns specified in [9] can also be applied in C++ generic programming. This is shown in [8], but only a subset of the patterns used in the CPH STL are described there. We will describe the generic structure of each design pattern relevant for our study. We will also describe several programming techniques which can be classified as idioms.

3. Polymorphism

In computer programming, a high degree of code reuse has always been a desirable property. Code reuse gives, among other things, good maintainability since copy-paste code is almost non-existing. A high degree of code reuse is usually obtained by packing individual peaces of code into small components. These components are denoted *classes* which have a collection of members (instances are denoted *objects*); a *member* in a class can be a type, a function, or data in form of a variable.

In a typical design, several classes may have a subset of members in common. Since one wants to, by all means, avoid copy-paste code most languages provide a feature called *inheritance*. Inheritance means that a class can inherit all members from one or more classes[3]. Let us study how inheritance works by using an example. We want to design a system which can store a collection of publications. These publications can be books, articles, theses, and so on. We could design classes for each kind of publication containing the members: author, title, year, and so on. Instead, since we want to maximize the amount of reused code, we use inheritance such that we get a base class `publication` which contains the common members. The code of such a system is shown below:

```
1 #include <string>
2 #include <iostream>
3 using namespace std;
4
```

[3] Not all widespread programming languages have the feature of "multiple inheritance", for example, Java does not.

```
5  typedef enum {MSc = 0, PhD = 1, Lic = 2, Dr = 3} thesis_type;
6
7  class publication {
8  public:
9    publication(string _author, string _title, short _year) : author(
         _author), title(_title), year(_year) { }
10   virtual void print() {
11     cout << "Author:␣" << (*this).author << endl;
12     cout << "Title:␣" << (*this).title << endl;
13   }
14 private:
15   string author;
16   string title;
17   short year;
18 };
19
20 class book : public publication {
21 public:
22   book(string _author, string _title, short _year, string _publisher
         ) : publication(_author, _title, _year), publisher(_publisher)
          { }
23   void print() {
24     cout << "Book" << endl;
25     publication::print();
26     cout << "Publisher:␣" << (*this).publisher << endl;
27   }
28 private:
29   string publisher;
30 };
31
32 class thesis : public publication {
33 public:
34   thesis(string _author, string _title, short _year, thesis_type
         _type) : publication(_author, _title, _year), type(_type) { }
35
36   void print() {
37     if(type == MSc) {
38       cout << "Master's␣Thesis" << endl;
39     }
40     else if(type == PhD) {
41       cout << "Ph.D.␣Thesis" << endl;
42     }
43     publication::print();
44   }
45 private:
46   thesis_type type;
47
48 };
49
50 void print_publications(publication** p, int n) {
51   for(int i = 0; i < n; i++) {
52     p[i]->print();
```

```
53      std::cout << "*-*-*-*-*-*-*-*-*-*-*-**-*-*-*-*-**-*-*-*-*" <<
            std::endl;
54    }
55  }
56
57  int main() {
58    book* b = new book("David␣Vandevoorde␣and␣Nicolai␣M.␣Josuttis", "C
          ++␣templates", 2003, "Pearson␣Education␣Inc.");
59    thesis* t = new thesis("Bo␣Simonsen", "Foundations␣of␣an␣adaptable
          ␣container␣library", 2009, MSc);
60    publication* publications[2];
61    publications[0] = dynamic_cast<publication*>(b);
62    publications[1] = dynamic_cast<publication*>(t);
63
64    print_publications(publications, 2);
65  }
```

What happens in this example is that we create two objects; one of the type `book` and one of the type `thesis`. These two objects are stored in an array, which type is the superclass `publication`. Since the two classes are inherited from the same base class, such a cast is possible. The function `print_publication` does simply traverse the array and calls the `print` member function of the objects. It may seem like that the `print` member function of the `publication` class is called, but what really happens is that the member functions of the subclasses, `book` and `thesis`, are called depending on the actual type of the objects.

The concept which we use, when one type can be used like another type, is *polymorphism*. When inheritance is used to achieve polymorphism we call it *dynamic polymorphism* or *run-time polymorphism* since it requires that elements are allocated at the heap which can only take place at run time. In this thesis, we will study a different kind of polymorphism which is *static polymorphism* or *compile-time polymorphism*. This kind of polymorphism is realized in C++ using template-based programming which we will study in the following section.

4. C++ generic programming

The principle of *generic programming* means to create programs that are adaptable and general. Achieving the goal of adaptability and generality requires, in most cases, that the programs do not rely on explicit types. Instead, type parameters are used which are substituted with real types by the compiler during the compilation. In the C++ programming language these type parameters are denoted *template parameters*; the instances of the template parameters are denoted *template arguments* [26]. All functions and self-defined types (`struct` and `class`) may be declared with template parameters. In other programming languages these type parameters have different names, for example, in Haskell the type parameters are denoted *type variables* [25].

Table 1. A function template and a class template.

A function template

```
template <typename T>
void fun(T const& t) {
  ...
}

int main() {
  int a = 5;
  char b = 'a';
  fun(a);
  fun(b);
}
```

A class template

```
template <typename T>
class my_class {
public:
  my_class(T const& _t) {
    (*this).t = _t;
  }
  ...
private:
  T t;
};

int main() {
  my_class<int> a(5);
  my_class<char> b('a');
}
```

4.1 Function and class templates

A *class template* is a class that is declared with one or more template parameters and respectively a *function template* is a function which is declared with one or more template parameters. An example of a function template and a class template is shown in Table 1.

The function template `fun` shown in the left part of Table 1 has one template parameter `T`. Upon compilation of the program, two different versions of `fun` are created by the compiler: `fun<int>` and `fun<char>`, i.e. `T` is substituted with the corresponding types. Notice that the template argument is not given explicitly for `fun` but it is deduced from the type of the argument, e.g. `fun` is called with variable `a`, the compiler knows that `a` is of type `int`, hence `int` is given as template argument to `fun`. It is allowed to explicitly give the template arguments, for example, `fun<int>(a)`.

The class template `my_class` shown in the right part of Table 1 has one member function, namely a parameterized constructor. Objects of this class template can be created using this constructor. The class template has one template parameter. In the example the class template is instantiated two times with different template arguments which are `int` and `char`. Since the template arguments are not the same, two different versions of the class template will be generated by the compiler: `my_class<int>` and `my_class<char>`. Notice that for class templates the template arguments must be given explicitly (unlike the function templates where the template arguments are deduced from the types of the arguments).

Both built-in types (`int`, `char`, and so on) and self-defined types can be given as template arguments. In addition to types, a template argument can also be a constant. These constants should be of a countable type which

exclude the types `float`, `double`, and self-defined types. The template arguments are immutable, i.e. after they are given they cannot be changed.

A class template can take *default template arguments* which are template arguments that are used, if the user did not explicitly give them. We will demonstrate in the example below how this concept is applied. Within the scope of the current C++ standard, function templates cannot take default arguments but the upcoming revision of the C++ standard will allow function templates to take default arguments.

Example 1. All STL containers take the following template arguments: the type of the value (`V`) and the type of the allocator (`A`). However, the user is only required to give the type of the value. The user can give the type of the allocator but it is not required. This behaviour results in the following C++ code, where the allocator has a default argument.

```
1 template<typename V,
2          typename A = std::allocator<V> >
3 class container {
4   ...
5 };
```

□

4.2 Specializations

An individual class template can exist in several versions, we may have one version of the class template for one permutation of types (template parameters may be intermixed), and another version of the class template for another permutation of types. We call such a version a *specialization*. We have, in general, two kinds of specializations:

Partial specialization: A specialization where template parameters and explicit types are intermixed. This means that a subset of the template parameters can be locked to specific template arguments, and the specialization is used if this subset of template arguments is given to the class template. The remaining template parameters will be instantiated to the respective arguments given by the user. An example of partial specializations is shown in the right part of Table 2.

Full specialization: A specialization which consists exclusively of types. This means that such a specialization is only used if the specified types are given as template arguments. This kind of specialization is also known as *explicit specialization*. An example of full specializations is shown in the left part of Table 2.

If a permutation of types does not match a specialization the regular class template is used. If no such class template exists, the compilation does not succeed.

The concept of specialization makes the C++ programming language strong with respect to generic programming. This concept can be used to perform various optimizations, for example, generic containers and algorithms can be specialized to achieve optimizations for some specific types

Table 2. A full specialization of a class template and a partial specialization of a class template.

Full specializations

```
template <typename A, typename B>
class my_class {
  ...
};

template <>
class my_class<int, bool> {
  ...
};

template <>
class my_class<float, bool> {
  ...
};
```

Partial specializations

```
template <typename B>
class my_class <int, B> {
  ...
};

template <typename B>
class my_class <float, B> {
  ...
};
```

(`std::vector<bool>` is a classic example, providing a bit vector instead of a regular dynamic array). Another application of specializations is *traits classes* which are classes that are used to deduce properties of types at compile time. In Table 3, two examples of traits classes are shown; these class templates are widely used in the development of CPH STL.

The first example, shown in the left column of Table 3, is a class template called `if_then_else`, which consists of two partial specializations. This class template is widely used in our library and its purpose is to perform an if-then-else statement at compile time. The class template takes three template arguments, a Boolean argument, and two types. If the Boolean argument is `true`, the member `type` will be set to the first type argument, and if the Boolean argument is `false` the member `type` will be set to the second type argument. This behaviour is realized by the two specializations. Often, we take advantage of that the Boolean argument can be the result of an evaluation of an expression which takes place at compile time.

Example 2. Consider two template arguments `X` and `Y`. We select `X` if its size is smaller than the size of `Y`, otherwise we select `Y`. The following example realizes the desired behaviour:

```
typedef cphstl::if_then_else<sizeof(X) < sizeof(Y), X, Y>::type
selected_type;
```

□

The second example, shown in the right column of Table 3, is a regular class template and a partial specialization. This class template is also highly relevant for library development, since it checks whether two types are the same at compile time. The specialization is used when the two types given as template arguments are the same; it defines the member `are_same` to the

Table 3. The traits classes (a) `cphstl::if_then_else` and (b) `cphstl::types`.

(a)

```
template <bool, typename T,
    typename U>
class if_then_else;

template <typename T, typename U>
class if_then_else<true, T, U> {
public:
  typedef T type;
};

template <typename T, typename U>
class if_then_else<false, T, U> {
public:
  typedef U type;
};
```

(b)

```
template <typename X, typename
     Y>
class types {
public:
  enum { are_same = 0 };
};

template <typename X>
class types<X, X> {
public:
  enum { are_same = 1 };
};
```

constant one. If they are not the same, the regular class template will be used, which defines the member `are_same` to the constant zero.

Example 3. The function template `mystery` takes two template arguments. If the two types equals the first template argument is the return type, otherwise the return type is a pair of the two types.

```
template <typename X, typename Y>
cphstl::if_then_else<cphstl::types<X, Y>::are_same,
                     X,
                     std::pair<X, Y> >
mystery(X const& x, Y const& y) {
  ...
}
```

□

4.3 Operator overloading

In the set of programming languages considered imperative and widespread it is quite rare to find a programming language which provides a language feature for altering the meaning of operators. C++ is such a language, almost every operator can be overloaded. The most common operators to override are: `+`, `-`, `++`, `--`, `*`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `()`. An operator can be overridden in C++ by implementing a member function called `operator` postfixed by the operand. Operator overloading can also be made using function templates.

Example 4. Reconsider the example given in Table 1. Here we define the class `my_class`. We desire that we can write:

```
my_class<int> a(5);
my_class<int> b(7);
```

```
3 my_class<int> c = a + b;
```

We can implement an operator overloading for `my_class`, such that addition is supported, in the following way.

```
1  template <typename T>
2  class my_class {
3  public:
4    my_class(T const&);
5    my_class operator+(my_class const& o) {
6      my_class tmp((*this).t);
7      tmp.t += o.t;
8      return tmp;
9    }
10   ...
11 };
```

The operator overloading could also be implemented as the function template:

```
1 template <typename T>
2 my_class<T> operator+(my_class<T> const& x, my_class<T> const& y) {
3   ...
4 }
```

□

Operator overloading supports generic programming in C++, since it can provide desired semantics for class types. For example, consider a function template which computes the sum of all objects stored in an array. The type stored in array is given to this function template as template argument. With the presence of the addition operator (`operator+`) in `my_class`, we can now give that function an array of `my_class` objects, and it will compute the sum successfully. An array of a built-in type (`int`, `char`) can be accepted by the same function template.

Several concepts in C++ programming rely on the possibility of operator overloading. For example, a *functor* is a class which contains one operator, `operator()`. An object of this class can be invoked like a regular function. Consider an object `x`, the `operator()` is invoked by `x(`$\mathcal{A}$`)`, where $\mathcal{A}$ is the list of arguments which `operator()` expects. Also, STL iterators are a result of operator overloading, which we will see later.

A special operator is the *conversion operator*. We say that it is a special operator, since it is not like the other operators. A usual operator is defined for operands, but a conversion operator is defined for a type. When a typecast from `A` to `B` is requested, and `A` provides a conversion operator for type `B`, this conversion operator is invoked. This conversion operator should, of course, return an object of type `B`. The example below shows how the conversion operator can be implemented and used.

Example 5. Reconsider `my_class` one more time. We can define a conversion operator, such that if an object of `my_class` is casted to `T`, the member `t` is returned.

```
1  template <typename T>
2  class my_class {
3  public:
4    my_class(T const&);
5    my_class operator+(my_class const& o);
6    operator T() {
7      return (*this).t;
8    }
9    ...
10 private:
11   T t;
12 };
13
14 int main() {
15   my_class<int> a(5);
16   std::cout << static_cast<int>(a) << std::endl;
17 }
```

□

4.4 Template metaprogramming

The principle of *metaprogramming* is to write programs that write or modify programs. A general interpretation of the C++ programming language is that it is two languages: The standard language which is described in many textbooks which includes loops, functions, classes, and other language features found in most imperative programming languages. This language is defined to be everything which is evaluated at run time. The second language is the template metaprogramming language, which consists of code that is evaluated at compile time. Additionally, the compiler directives (for example, `#define`) belong to this language since they are processed at compile time.

We have already seen examples of the use of this language. Reconsider the example given in Example 2. Here, we are using some information given by the types to select the appropriate type. This code is executed at compile time, hence this kind of code belongs to the template metaprogramming language.

The existence of the template metaprogramming language allows us to perform computations at compile time. We can also perform optimization techniques like loop unrolling (see [26]) using this programming language. In fact, the programming language is proven to be Turing complete [27, 5], such that every computation which can be performed using a Turing machine [13] can also be performed using the template metaprogramming language.

We will clarify how such computations can be performed at compile time using the template metaprogramming language with the example shown below. The syntax for writing these metaprograms is quite complicated, as the reader can verify by examining the example. We believe that the D programming language [7] has succeeded in creating a better way of writing metaprogramming code, since that code is almost similar to the code one

would write for run-time computations (see Example 7). It would be a benefit for C++ if the same behaviour existed [28].

Example 6. The nth Fibonacci number F_n for $n \geq 0$ is defined by the following recursive formula:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

The following C++ program will compute the 20th Fibonacci number at compile time:

```
1  template <int N>
2  class fib {
3  public:
4    enum { result = fib<N-1>::result + fib<N-2>::result };
5  };
6
7  template <>
8  class fib<0> {
9  public:
10   enum { result = 0};
11 };
12
13 template <>
14 class fib<1> {
15 public:
16   enum { result = 1};
17 };
18
19 int main() {
20   int a = fib<20>::result; // a contains 6765
21 }
```

In this example we have a class template which is used when $n \geq 2$ and two full specializations which are used when $n = 0$ or $n = 1$. What happens is that `fib<20>` will generate code for the classes `fib<19>` and `fib<18>`; `fib<18>` will generate code for `fib<17>` and `fib<16>`. The code generation continues in the same way until the specializations `fib<0>` and `fib<1>` are reached. Notice, that the computation of each Fibonacci number is performed once therefore the worst-case number of template instantiations is $\Theta(n)$ for the nth Fibonacci number (unlike the naive recursive algorithm which has exponential running time). □

Example 7. In this example we will show how the D metaprogramming language can be used for compile-time computation. The metaprogram given in Example 6 can be implemented in the following way using the D metaprogramming language:

```
1  template Fib(ulong n)
2  {
```

```
3   static if(n == 0 || n == 1) {
4     const Fib = n;
5   }
6   else {
7     const Fib = Fib!(n - 1) + Fib!(n - 2);
8   }
9 }
10
11 void main() {
12     const a = Fib!(20); // a contains 6765
13 }
```

Writing the same code in a regular imperative programming language, such that the computation is performed at run time, will result in similar code. □

5. The STL

The STL is a collection of generic algorithms and containers. The algorithms and containers are written in such a way that they can be used for any type that is given by the user. The iterators are the glue between the containers and algorithms such that the algorithms can access the data stored in the container. This is illustrated in Figure 1.

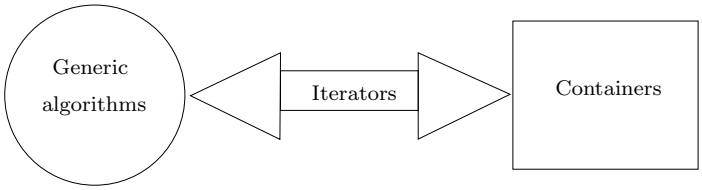


Figure 1. The algorithms, iterators, and containers.

From the programmer's point of view, the STL is a toolbox with reusable algorithmic components which are ready to be used in application development with no modifications. Let us consider an example of a program where the components are used. The example below shows a generic algorithm called `random_vector` which purpose is to generate a sequence of distinct elements which are randomly selected.

```
1 #include <algorithm>
2 #include <vector>
3 #include <iostream>
4 #include <iterator>
5
6 template <typename T>
7 class random_number_generator {
8 public:
9   random_number_generator(T _max) : max(_max) {
```

```
10      srand(time(NULL));
11    }
12    T operator()() const {
13      T tmp = (rand() if(tmp == 0) {
14        return 1;
15      }
16      return tmp;
17    }
18  private:
19    T max;
20  };
21
22  template <typename I>
23  void random_vector(I start, I stop, std::size_t n) {
24    typedef typename std::iterator_traits<I>::value_type V;
25
26    random_number_generator<V> f(30);
27
28    std::generate_n(start, n, f);
29    std::sort(start, stop);
30
31    I end = std::unique(start, stop);
32
33    while(end != stop) {
34      int random_number = f();
35
36      I it = std::lower_bound(start, end, random_number);
37
38      if(*it != random_number) {
39        I new_it = end;
40        new_it++;
41        std::copy_backward(it, end, new_it);
42        end++;
43        *it = random_number;
44      }
45    }
46  }
47
48  int main() {
49    std::vector<int> v(10, 0);
50    random_vector(v.begin(), v.end(), 10);
51    ...
52  }
```

In this setting we have one container object of type `vector` storing objects of type `int`. The container `vector` is a dynamic array; the difference between a `vector` and an ordinary array is that memory management is done automatically and it is transparent to the user. The `vector` container is constructed using a parameterized constructor which initializes the `vector` to contain 10 elements with the value zero.

The `random_vector` generic algorithm is called using the iterators given

by the `vector` container (but other containers can be used as well). An iterator pointing to the beginning of the sequence is obtained by the call to `begin()` and an iterator pointing to the end element is obtained by the call to `end()`. This element is located one element past the last element stored in the container. We will now study the implementation of `random_vector`. This generic algorithm uses five other generic algorithms to implement the desired behaviour; these algorithms are:

`generate_n` calls the functor given as the third argument (`f`), `n` times (`n` is given as the second argument) and stores the results in the container where the iterator given as the first argument is the starting position. The behaviour of the algorithm is illustrated in Figure 2(a).

`sort` rearranges the elements in the sequence enclosed by the iterators given as arguments such that the elements appear in sorted order.

`unique` removes duplicates appearing consecutively in the sequence enclosed by the iterators given as arguments. An iterator pointing to the last element in the sequence is returned. The behaviour of the algorithm is illustrated in Figure 2(b).

`lower_bound` searches the sequence enclosed by the iterators, given as the first and second argument, for an element which is less than or equal to the element given as the third argument. An iterator pointing to the element found is returned.

`copy_backward` performs backward copying of the elements stored in the sequence enclosed by the iterators, given as the first and second argument. The iterator given as third argument determines the starting position of the resulting sequence. The concept of backward copying means that the elements are copied in reverse order, such that the last element is copied first. The behaviour of the algorithm is illustrated in Figure 2(c).

We can now understand what happens in the `random_vector` generic algorithm. We generate a sequence consisting of random numbers (by `generate_n` and the functor `random_number_generator` which gives a random number every time it is invoked), sort the sequence (by `sort`), and remove the duplicate elements (by `unique`). After these actions the size of the sequence may be less than `n`; we need exactly `n` elements, therefore we may be forced to generate additional elements. This action happens in the while loop. The while loop runs as long as the number of distinct elements is less than `n`. Every time the loop runs we generate a new element, if the element is not present in the sequence, the element is inserted into the sequence; the check is performed using `lower_bound`. The iterator given by `lower_bound` points at the position where the new element should be inserted. The insertion is done by copying all elements (by `copy_backward`), starting from the iterator given by `lower_bound`, one slot forward. After the copying procedure, the generated element is copied to the slot which the iterator given by `lower_bound` points to.

In the following subsections we will study the concepts of containers and generic algorithms a bit closer.

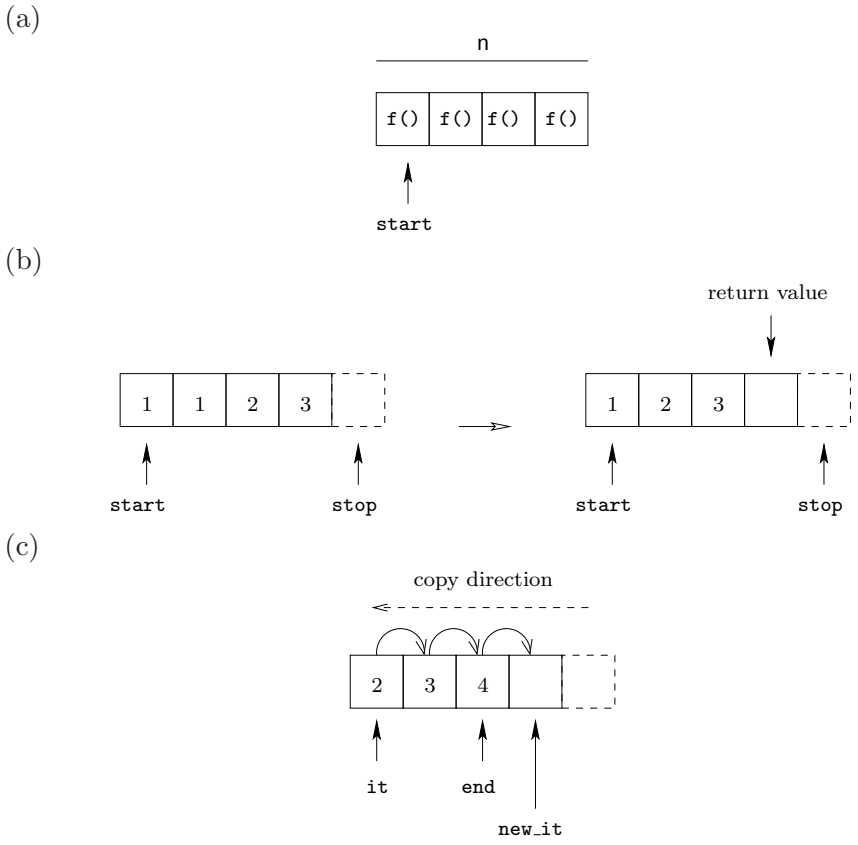


Figure 2. Three algorithms: (a) `generate_n`, (b) `unique`, and (c) `copy_backward`.

5.1 Containers

A *container* is a class template, which stores a collection of objects which can be updated dynamically using the member functions of the class template. All containers in the STL provide, among other things, the following member functions:

`I insert(I pos, V const& v)`: Inserts an element `v` before the element in the sequence which is pointed to by the iterator `pos`. We denote this member function *insert by locator*.

`void erase(I pos)`: Erases the element pointed to by the iterator `pos`. We denote this member function *erase by locator*.

`I begin()`: Returns a mutable iterator[4] pointing to the first element stored in the container.

[4] For ordered containers, this iterator can be used to violate the invariants of the data structure. This seems like a mistake in the C++ standard.

`I end()`: Returns a mutable iterator pointing to the element which is located one position past the end.

`size_type size()`: Returns the number of elements stored in the container.

`size_type max_size()`: Returns the maximum number of elements the container can store.

Additionally, all containers provide a parameterized constructor, a copy constructor, a destructor, and a `=` operator. Each container provides additional member functions which may differ from the other containers. The STL provides the following containers (the template parameters `C` and `A` are the comparator and allocator respectively; we will describe these later):

`list<V, A>` is a doubly-linked list storing elements of type `V`; `list` allows all modifying operations (insert and erase by locator, insert and erase in the front or the end) to be done in $O(1)$ worst-case time. Iteration can only be done sequentially, by starting in either the beginning or the end; advancing the iterator one step takes $O(1)$ worst-case time. A typical implementation of `list` is illustrated in Figure 3(a) .

`vector<V, A>` is a dynamic array storing elements of type `V`; `vector` provides $O(1)$ amortized time for the operations insert and erase in the end, $O(n)$ worst-case time for insert and erase in the middle and the beginning, and random access in $O(1)$ worst-case time (n denoting the number of elements stored). The `vector` class provides the operator `[]` that works as ordinary array indexing. The data stored in the container must be in a contiguous memory segment, which excludes all other implementation alternatives. The doubling array [4] (where the capacity is doubled when an expansion is needed) is the most typical implementation; such a data structure is illustrated in Figure 3(b) where four elements are inserted into the container.

`set<K, C, A>` is an ordered collection of unique elements of type `K`. All modifying operations, with a few exceptions, take $O(\lg n)$ worst-case time; the exceptions include insert by locator and erase by locator which take $O(1)$ amortized time. Because the elements are ordered `set` provides additional member functions including `lower_bound`, `upper_bound`, `find`, `insert(V const&)`, and `erase(V const&)`; these operations take $O(\lg n)$ worst-case time. The container provides sequential iteration like `list`, however advancing the iterator takes $O(1)$ amortized time. A typical implementation is a binary search tree, like the one illustrated in Figure 3(c). Because of the amortized time bounds for insert by locator and erase by locator the only possible implementation is a red-black tree [12].

`multiset<K, C, A>` is similar to `set`, except multiple elements of type `K` with the same value are allowed.

`map<K, V, C, A>` stores a collection of unique keys of type `K`. Each key has a value associated of type `V`. The `map` container is similar to `set`, but the actual value stored is the pair $\langle$`K , V`$\rangle$.

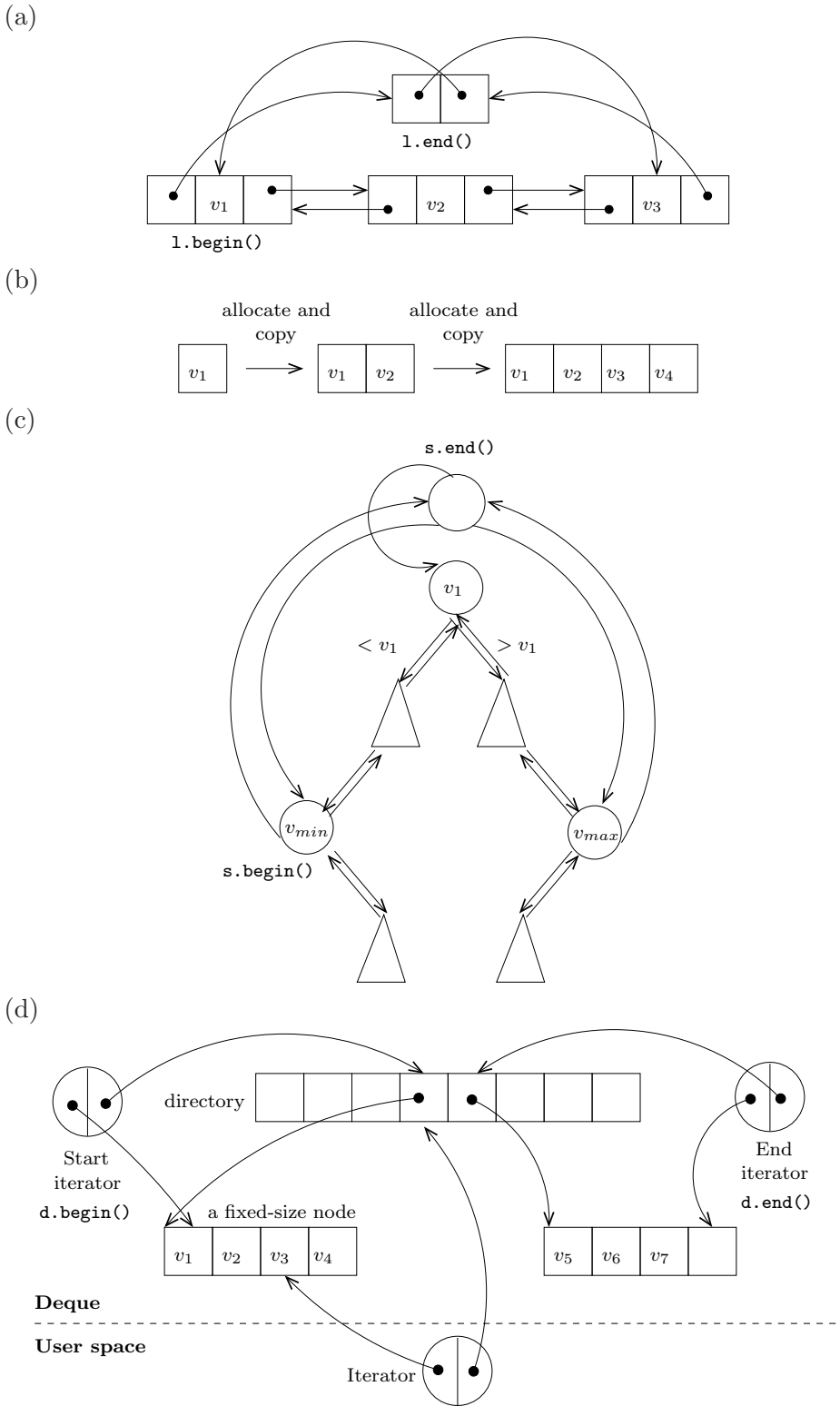


Figure 3. Typical implementations of (a) `list`, (b) `vector`, (c) `set`, and (d) `deque`.

`multimap<K, V, C, A>` is similar to `map` but allows several elements with the same key.

`deque<V, A>` is a double-ended queue. This data structure offers the same guarantees as `vector` (including random access), but it also offers $O(1)$ amortized cost for insertion and deletion in the beginning of the sequence. A typical implementation of `deque` (described, for example, in [16]) is illustrated in Figure 3(d).

5.2 Iterators

An *iterator* is a class template providing pointer semantics, meaning that it can be used in the same way as a pointer by the operators `++`, `*`, and so on. An iterator is a generalization of a pointer, meaning that it points to an element, and it provides member functions for altering and retrieving that element. Also, it provides member functions for advancing and comparing positions.

As we learned earlier, by studying the containers, the iterators are created by the containers. We saw that each container can create a *mutable iterator* (with `begin()` and `end()`), which is an iterator which allows modifications of the element pointed to by the iterator. Each container can also create an *immutable iterator* which is an iterator that prohibit any modification of the element pointed to by the iterator. Each container can also create a *reverse iterator* which is a special kind of iterator, where each advance operation is negated such that `++` means `--` and `--` means `++`. Also `rbegin()` and `rend()` have the opposite meanings, `rbegin()` gives `--end()` and `rend()` gives `--begin()`.

Most containers provide *bidirectional iterators* which means that the iterators can move one step forward or one step backward with one advance operation. The containers `vector` and `deque` provide *random-access iterators* which allow the iterator to be advanced an arbitrary number of positions. In the example below, we will show how iterators are used.

Example 8. Several programming languages (for example, Haskell and Python) provide a function called `zip` which works as follows: `zip` is given two sequences; it takes one element from the first sequence and one element from the second sequence and creates a pair of these elements. It continues until the end of one of the sequences is reached.

We want to implement `zip` in C++; the example below shows the use of `zip` in Haskell:

```
1 Prelude> zip [1,2,3,4] ['a', 'b', 'c']
2 [(1,'a'),(2,'b'),(3,'c')]
```

Below is an implementation of `zip` in C++ (we return a `vector` of the zipped elements):

```
1 template <typename I, typename J>
2 std::vector<std::pair<typename I::value_type, typename J::value_type
    >
```

```
zip(I begin_c1, I end_c1, J begin_c2, J end_c2) {
  typedef std::pair<typename I::value_type, typename J::value_type>
      pair_type;
  typedef std::vector<pair_type> result_type;
  result_type result;
  I i = begin_c1;
  J j = begin_c2;
  while(i != end_c1 && j != end_c2) {
    result.push_back(pair_type(*i, *j));
    ++i;
    ++j;
  }
  return result;
}
```

As the reader can verify, the way iterators are used is identical to the way regular pointers are used. □

5.3 Algorithms

A *generic algorithm* is a function template which performs operations on a container or a sequence (or an array). In general there are two different types of algorithms: mutating algorithms which may make modifications to the data stored and non-mutating algorithms which read the data stored and make no modifications. Each algorithm is specified in the C++ standard. This specification usually consists of preconditions, invariants, and postconditions. Included in this specification is also the time-complexity requirements of each algorithm. The library developers have the freedom to implement an arbitrary algorithm, as long as it satisfies the time-complexity requirements given and, of course, satisfies the invariants. The same applies to the implementations of the containers.

To clarify how the algorithms work, we will study the most simple of all generic algorithms included in the STL.

Generic algorithm 1. The code given below is a full implementation of the `swap` algorithm. The purpose of this algorithm is to swap the two values given as arguments; the arguments are references to the values:

```
template <typename T>
void swap(T& a, T& b) {
  T c = a;
  a = b;
  b = c;
}
```

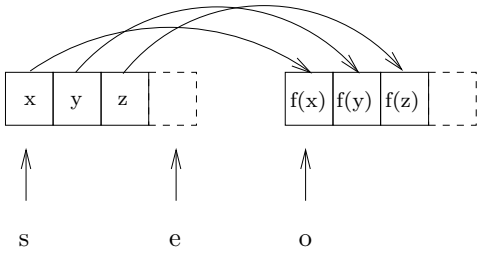
Since the arguments are references, the changes will be propagated to the scope of the calling function. □

The collection of algorithms in the STL is large; below is a description of some selected algorithms (several versions of each individual algorithm may be available):

Generic algorithm 2.
`template <typename I, typename O, typename F>`
`O transform(I s, I e, O o, F f);`
For each element `x` in the sequence enclosed by the iterators `s` and `e`, the functor `f` is invoked with `x` as argument. The result is stored in the sequence starting from `o`. Figure 4(a) shows how `transform` works. This algorithm is similar to the `map` higher-order function found in many functional programming languages. □

Generic algorithm 3.
`template <typename I, typename O, typename F>`
`O remove_if_copy(I s, I e, O o, F f);`
For each element `x` in the sequence enclosed by the iterators `s` and `e`, the functor `f` is invoked with `x` as argument. If `f(x)` is true, `x` is copied to the other sequence starting from `o`. Figure 4(b) shows how `remove_if_copy` works. This algorithm is similar to the `filter` higher-order function found in many functional programming languages. □

(a)



(b)

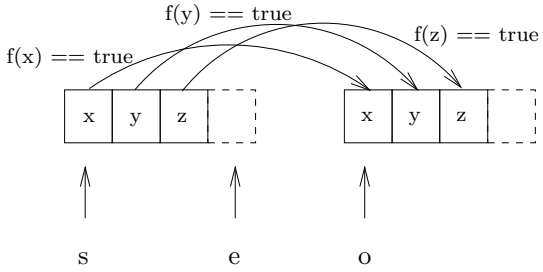


Figure 4. The algorithms (a) `transform` and (b) `remove_if_copy`.

The next algorithm is not classified as an algorithm in the C++ standard (it is part of the numerics library), but according to our definition of an algorithm it is also a generic algorithm.

Generic algorithm 4.
`template <typename I, typename T, typename F>`
`T accumulate(I s, I e, T init, F f);`
Let `a` denote an accumulator, initialized to `init`. For each element `x` in

the sequence, enclosed by the iterators `s` and `e`, `a = f(a, x)` is computed. Below is an example which shows how the factorial function can be computed using `accumulate`.

```
int array[] = {2, 3, 4, 5, 6};
std::cout << std::accumulate(array, array+5, 1, std::multiplies<int
    >());
```

This code is equivalent to the following C++ code:

```
std::multiplies<int> f;
int a = 1;
for(int i = 0; i < 5; ++i) {
  a = f(a, array[i]);
}
std::cout << a;
```

This algorithm is similar to the `foldl` higher-order function found in for example Haskell, and `reduce` found in Python. □

5.4 Other components

The memory management and ordering of elements are decoupled from the containers and algorithms in order to obtain the highest degree of genericity and flexibility. This decoupling introduced two new concepts: an allocator and a comparator. An *allocator* is a class template containing member functions for memory allocation and deallocation. A *comparator* is a class template containing one member function, which is invoked every time a container or an algorithm performs a comparison between two elements. The allocator is given as template argument to a container. Likewise for the comparator but some containers do not accept a comparator (e.g. `list`, however some member functions in `list` require a comparator, for example, `sort`) because the elements are not ordered.

Typically `std::allocator` is used as an allocator, but several other alternatives exist. For example, Boost [2] provides a pool allocator, where an object pool is used to serve allocation requests, which can improve performance when many deallocations and allocations take place. Also a block allocator has been implemented [18], which allocates a large amount of memory which is used to serve allocation requests. This can speed up allocations of small memory portions. The user can design his or her own allocators by implementing a class with the members required by the C++ standard.

Regarding the comparator, `std::less` is typically used. If `std::less` is invoked with two elements `x` and `y`, it will return `x < y`. The comparator `std::greater` will perform `x > y`. The fact that comparison is decoupled from the containers is especially relevant for ordered containers (the associative containers `set`, `map`, `multiset`, and `multimap`), and for generic algorithms which rely on ordering (`sort`, `binary_search`, `lower_bound`, `upper_bound`, and so on). That is because the user can specify an arbitrary ordering.

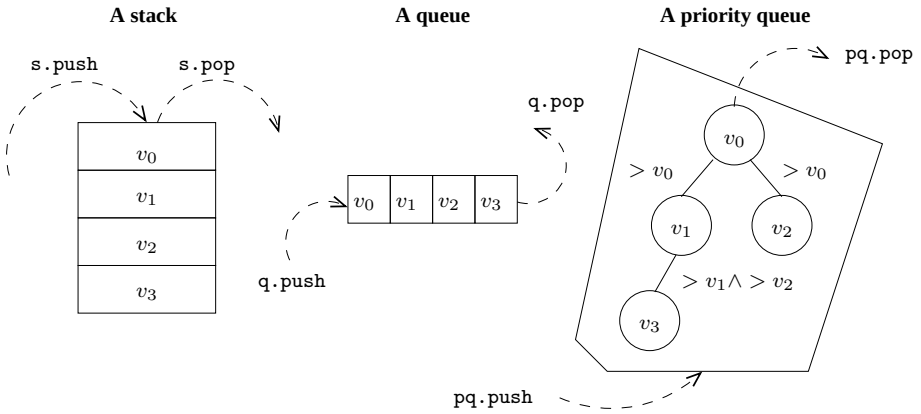


Figure 5. The three adaptors in the STL.

We will see several examples on the use of comparators and allocators in the papers included in this thesis.

5.5 Adaptors

In addition to containers, the STL also provides a number of adaptors. An *adaptor* is realized by a container, but an adaptor provides a different interface and it provides a restricted subset of the member functions found in the container. The STL provides three adaptors `stack`, `queue`, and `priority_queue`; these concepts are known from the introductory textbooks on data structures, see Figure 5 for illustration. We will now describe each adaptor with respect to their use:

`stack<V, R>` is a stack storing elements of type `V`. The stack is realized using the container of type `R`. A stack can be realized by `list`, `deque`, or `vector`. The class template `stack` provides, among other things, three member functions which allow the user to modify it: `top` which gives the top element, `pop` which removes the top element, and `push` which inserts a new element at the top.

`queue<V, R>` is a queue storing elements of type `V`. Like `stack`, `queue` gets the realizing container as template argument `R`. Similar to stack, `queue` provides `push` and `pop`, but `push` appends an element to the end, and `pop` erases the element stored at the beginning. The class template `queue` allows the user to get the element at the beginning by the member function `front` and the element at the end by the member function `back`.

`priority_queue<V, R, C>` implements a priority queue storing elements of type `V`, and realized by the container of type `R`. The comparator of type `C` is used to compare the elements. This adaptor provides `push`, `pop` and `top`. These member functions can be realized by the generic

algorithms `push_heap` and `pop_heap`; the implementation of which are given, for example, in [4].

6. Software patterns

In algorithm theory, many hard computational problems exist. These problems are hard since they cannot be solved (computationally) in polynomial time; hence they are classified as NP-hard problems [4]. Sometimes it is enough to just provide approximate solutions to these computational problems, therefore several approximation algorithms and heuristics for several problems have been found. An approximation algorithm finds a solution within a provable error bound in polynomial time, and a heuristic has no such error bound.

Also, in software design several hard problems exist. That is especially the case when the software designer seeks the highest degree of reusability and flexibility regarding the resulting design. A software pattern identifies a recurring problem within a domain, and provides a skeleton of a solution. One can say that software patterns are similar to approximation algorithms (or heuristics), since they give approximate solutions to certain design problems. In general, software patterns are classified in three different categories [20]: architectural patterns, design patterns and idioms.

A software *architectural pattern* [3] provides solutions to a high-level design problem. These patterns define an organization of software components consisting of several classes. When specifying the software architecture for a system, some properties might be more desirable than others. The catalogue of architectural patterns gives software designers an overview of the weaknesses and strengths of using each pattern, so that the designer can choose the one which fits the requirements of the system.

Example 9. In network design, the OSI model [24] is widely used. The OSI model divides the process of communication on a network into several layers (top down): the application, presentation, session, transport, network, data link, and physical layers. When the user wants to transmit a packet, he or she communicates with the application layer, which propagates the information to the layers below, until the packet is transmitted by the physical layer. When a packet arrives the process is reversed.

An example of the use of this model is the TCP/IP protocol[5]. The clear advantage of using such a layered architecture in this respect is that the layers are interchangeable. This means that any transmitting media can be used with this protocol (Ethernet, Wireless Ethernet, and so on); it just requires that the bottom layers are changed. □

A *design pattern* provides solutions to a design problem, within the scope of one or a few software components, consisting of a few classes. The difference between a design pattern and an architectural pattern is that the

[5] A general interpretation is that the TCP/IP protocol consists of four layers: link, internet, transport and application layers.

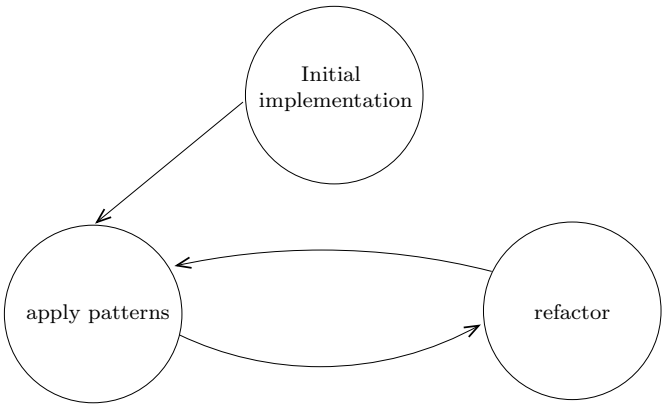


Figure 6. The CPH STL development process model.

architectural patterns are solutions for system-wide problems, whereas the design patterns are more low level, since only a few classes are involved in the solutions.

Example 10. We have already seen one design pattern, namely the iterator design pattern. The *iterator design pattern* is defined by its intention "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation". This pattern has been applied in the STL in order to obtain a unified way of traversing the elements stored in a container. As we have seen in the earlier section on the STL, the implementations of the containers are very different, so such mechanism is needed to provide transparency for the user and the generic algorithms with respect to traversal. This transparency means that the generic algorithms can operate on any container disregarding the underlying representation. □

An *idiom* is a solution to a low-level design problem. An idiom is a programming technique for creating constructions, which are not build into the language. A classic example of an idiom is swapping two values. A construction for performing this action is actually build into the C++ programming language by the generic `swap` algorithm, which we saw earlier. However, in some languages it may not be built in.

We have widely adopted idioms and design patterns into the development of the library components in the CPH STL. Since these are very important we will describe some relevant idioms and design patterns in the following sections. The process model, of how the development of library components was performed, is illustrated in Figure 6.

6.1 The pimpl idiom

The first idiom which we address is informally known as the *the pimpl idiom*, which is short for "pointer to implementation" [23]. We found it relevant to apply this idiom when implementing *container decorators* (or *views*), which

are similar to the adaptors, as we described in the section on the STL. The difference is that the container decorators provide the same interface as the container. Each decorator keeps an instance of the container, and their purpose is to provide different representation or a subset of the data stored in the container.

One problem with these decorators is that the container may be copy constructed inside the decorator. This means that the elements are stored twice. We wanted to avoid this; therefore we applied the pimpl idiom so that the decorator keeps a pointer to the container. The skeleton of a container decorator is shown below:

```
1  template <typename C>
2  class decorator {
3  public:
4    typedef typename C::size_type size_type;
5    ...
6    decorator(C const& _c) {
7      (*this).c = &_c;
8    }
9
10   size_type size() {
11     return (*(*this).c).size();
12   }
13 private:
14   C* c;
15 };
```

The original purpose of the pimpl idiom [23] was to avoid recompilation when implementing programs (especially libraries) which relied on dynamic polymorphism. However in the context of generic program libraries, avoiding copy construction is more applicable.

6.2 The CRTP idiom

Most programmers may have encountered problems when using inheritance in C++ programming. A well-known problem occurs in the following scenario: Consider a base class `Base`, and several classes which inherit from this class. Let us denote one of these `Derived`. The class `Base` should store a member of `Derived`. Since we have several derived classes, we cannot use an explicit type, neither the type of the base class without using type casting.

The *Curiously Recurring Template Pattern* (CRTP) [26], which we denote the CRTP idiom, describes how to implement the scenario mentioned above in an elegant way that does not involve any type casting. The idea of this idiom is to let the derived class give the type of itself to the base class such that the base class can use the type of the derived class. A skeleton of this kind of use of the CRTP idiom is shown below:

```
1  template <typename D>
2  class Base {
```

```
public:
  ...
private:
  D* derived_instance;
};
class Derived : public Base<Derived> {
public:
  ...
};
```

We found this idiom relevant when constructing node classes, for example, for binary search trees. A node for any kind of binary search tree needs to have `left`, `right`, and `parent` pointers. To design a base class containing these pointers becomes hard since the type of the pointers should be the derived class e.g. `avl_tree_node`, `red_black_tree_node`. We use the CRTP idiom to solve this problem. The relevant lines of our node base and sub classes look like this (more details can be found in [22]):

```
template <typename V, typename N>
class tree_node {
protected:
  typedef N node_type;
  node_type* _left;
  node_type* _right;
  node_type* _parent;
};

template <typename V>
class avl_tree_node : public tree_node<V, avl_tree_node<V> > {
public:
  ...
};
```

6.3 The overriding idiom

So far we have seen some examples of how to obtain polymorphism by template-based programming. We have seen how class templates can accept other class templates. A recurring construction in the CPH STL is that some class template `A` accepts another class template `B` and calls to `A` may be propagated to `B`. This is shown in the example below:

```
class B {
public:
  void member1() { ... }
  void member2() { ... }
};
template <typename C = B>
class A {
public:
  void member1() { c.member1(); }
  void member2() { ... }
```

```
11    void member3() { ... }
12  private:
13    C c;
14  }
```

This construction is in some way similar to inheritance. In some cases it is relevant for us to provide a default member function in `A` if `B` does not provide one. In our current example, this means that `A` should have a default member function for `member1` if `B` does not provide `member1`.

The overriding idiom allows us to do such check. We use the SFINAE (substitution failure is not an error) principle, which allows us to perform check of class members without causing a compiler error. The code given below is a skeleton for making such checks:

```
1   template <typename U>
2   class has_member4 {
3   public:
4     typedef char NotFound;
5     struct Found { char x[2]; };
6     template <void (U::*)()>
7     struct TestIt ;
8
9     template <class T>
10    static Found Test(TestIt<&T::member4>*);
11    template <class T>
12    static NotFound Test(...);
13
14    enum { positive = (sizeof(Test<U>(0)) == sizeof(Found)) };
15  };
16  template <typename C = B>
17  class A {
18  public:
19    ...
20  private:
21    void member4_dispatch(cphstl::int2type<0>) {
22      /* default implementation; C has no member4 */
23    }
24    void member4_dispatch(cphstl::int2type<1>) {
25      /* C has member4, let us call that */
26      c.member4();
27    }
28  public:
29    void member4() {
30      member4_dispatch(cphstl::int2type<has_member4<C>::positive>());
31    }
32  private:
33    C c;
34  }
```

The `has_member4` class template checks if the class given as template argument has the member function `void member4()`. We use a technique called

tag dispatching[6] to call the appropriate member function.

We use this idiom to realize the concept of *flexible interfaces*, which means if the underlying data structure does not provide an implementation, a default is used[7]. The current C++ programming language does not provide a more elegant solution to implement flexible interfaces, however a solution based on C++ concepts is more elegant.

7. Design patterns

The structure of the design patterns discussed in [9] is realized by dynamic polymorphism. Most patterns introduce flexibility such that the relationship between the classes is not explicitly defined. The implicit relationship means that one class in the setting can accept another class drawn from a set of possible classes, denoted $\mathcal{S}$. In [9], this implicit relationship is implemented using inheritance, such that all classes in $\mathcal{S}$ inherit from a base class, which we denote `Base`; the constructor of the accepting class requires an instance of `Base`.

Our desire is to apply the design patterns, but in a generic setting. This means that a class drawn from the set $\mathcal{S}$ is given to the accepting class as template argument. We have only found one paper [8] which describes how this can be implemented for a subset of the design patterns given in [9]. We think that most of the constructions given in [8] are too complicated. Since we prefer simpler constructions, we will in this section, describe the structure of the design patterns in a generic setting.

The generic versions of the design patterns are not directly equivalent to the conventional versions. In the conventional form, the design patterns provide type safety, meaning that it is verified at compile time whether the relationship between the classes match, simply because of the use of base classes. We have no such mechanism in a generic setting; C++ concepts [11] provide such a mechanism, but since it will not be included in the language yet, the generic versions of the design patterns may be unattractive in some domains.

7.1 Generic bridge

We have already discussed the elements of the STL. In addition to the STL there are several other algorithmic libraries. LEDA [14] is in the algorithm community a highly-respected library that provides generic implementations of data structures and algorithms in a wide area, ranging from computational geometry to classic data structures and algorithms.

The main difference between the STL and LEDA is that in LEDA the algorithms and data structures are not decoupled from each other. This

[6] This technique is explained in our paper on the CPH STL architecture which is included in this thesis.

[7] This concept is explained in our paper on component frameworks which is also included in this thesis.

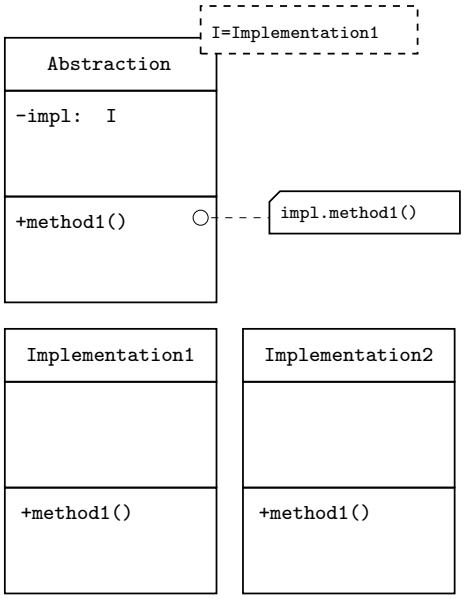


Figure 7. The generic version of the bridge design pattern.

means that, for example, the graph algorithms are located in the graph container. Another difference is that some LEDA containers take an *implementation parameter*, which gives the user the opportunity to specify which data structure that will be used for realizing a container.

This means that LEDA provides several containers and each container can be realized by several possible data structures. The bridge design pattern allows such constructions to be implemented without code duplication. The *bridge design pattern* is defined by its intention which is as follows "Decouple an abstraction from its implementation so that the two can vary independently".

The generic version of the bridge design pattern is illustrated in Figure 7. The class `Abstraction` accepts the type of the implementation as parameter. The default implementation used is `Implementation1`, however `Implementation2` can also be used by explicitly passing this class as template argument `I`. The abstraction class will forward the calls of member functions to the corresponding member functions in the implementation class.

This pattern is widely used in the CPH STL. We use it exhaustively to decouple containers (abstract data types) from their implementations (data structures). The CPH STL is similar to LEDA in this respect. We will discuss this use in depth later.

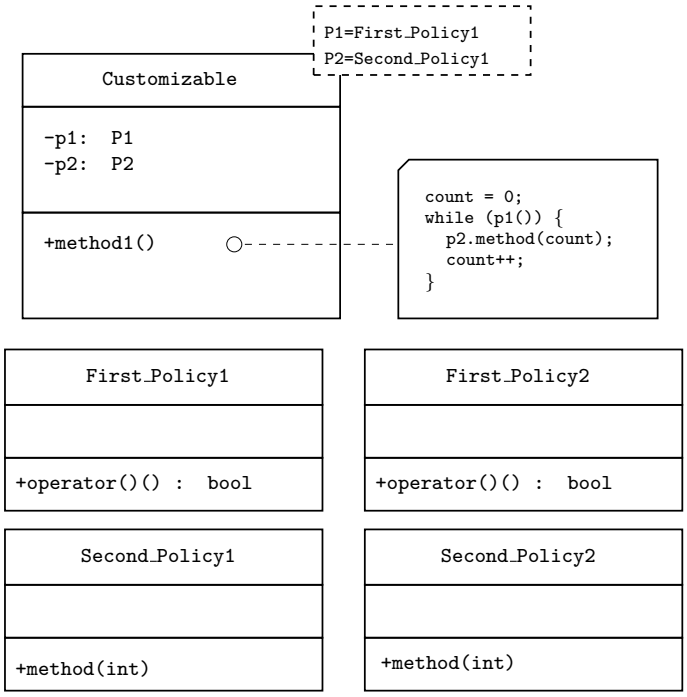


Figure 8. The generic version of the strategy design pattern.

7.2 Generic strategy

We have already seen one application of the strategy design pattern, namely the comparator in the STL. Recall that the comparator is used for containers and algorithms which perform operations that are based on an ordering. For example, the `sort` algorithm accepts a comparator such that if `std::less<V>` is given the elements are sorted in increasing order, and if `std::greater<V>` is given the elements are sorted in decreasing order; `V` denotes the type of the values stored in the sequence. According to pattern terminology each comparator is a strategy; we define the *strategy design pattern* by the following intention "Define a family of strategies, encapsulate each one, and make them interchangeable".

The generic version of the strategy pattern is denoted a *policy* [26]. The difference between a strategy and a policy is that a strategy can be changed at run time and a policy can be changed at compile time. The generic version of the strategy design pattern is illustrated in Figure 8. The class `Customizable` accepts two classes given as template arguments. The template parameter `P1` accepts the classes `First_policy1` or `First_policy2`; the template parameter `P2` accepts the classes `Second_policy1` or `Second_policy2`. The classes given by template arguments are used in the member function denoted `method1` to perform some computation.

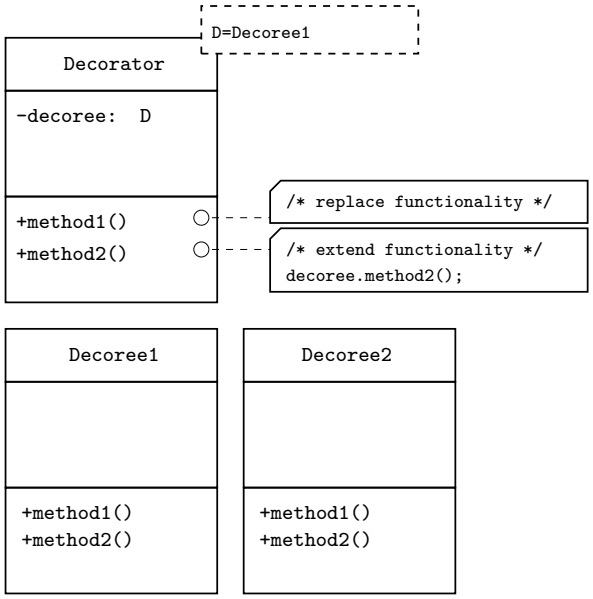


Figure 9. The generic version of the decorator design pattern.

This pattern has been widely applied to the components in the CPH STL. Most of our containers take customization parameters, for example, a storage policy which states how the data should be stored. We will discuss the use of policies in depth in the papers included in this thesis.

7.3 Generic decorator

Sometimes it may be necessary to extend the functionality or alter the behaviour of existing classes. The straight forward solution to implement such an extension would be to take a copy of the existing code and modify it. But because of code reusability considerations we desire a more refined solution. This design problem is specified by the *decorator design pattern*, and it is defined by its intention "Define additional responsibilities to a set of objects or replace functionalities of a set of objects".

The structure (illustrated in Figure 9) is similar to the bridge pattern. The main difference is that a decorator extends or replaces functionality. This means that some implementation can be done in the scope of the decorator class. The alternative structure (illustrated in Figure 10) is based on inheritance; this structure is given in [8]. This approach might be good for large classes, in order to obtain a high degree of code reusability. However, inheritance can be problematic, for example, the current C++ standard does not allow inheritance of constructors and `operator=`.

In the CPH STL, we have designed several container and iterator decorators (denoted *custom iterators* and *container views*). These are used to mod-

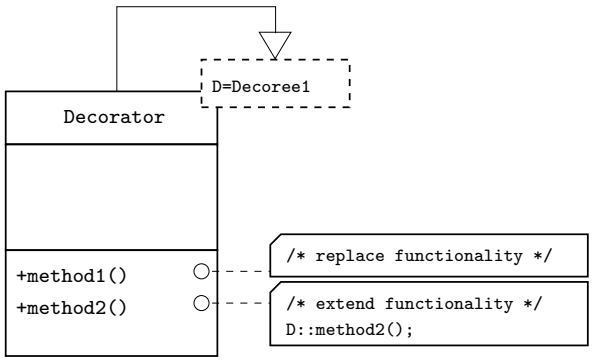


Figure 10. An alternative generic version of the decorator design pattern. The class `Decoree1` is illustrated in Figure 9.

ify the representation of data in several different ways, without modifying the data stored in the underlying container. For example, `transform_view` implements the generic algorithm `transform`, but the elements are not copied or modified. A view provides the same semantics as a regular container, but with no modifying operations. This means that iterators can be obtained by `begin()` and `end()`. When these member functions are invoked a smart iterator is returned instead of a regular iterator. This iterator returns the transformed value of the element of which the iterator points to when the `*` operator is invoked. The transformed value is generated using the functor given to `transform_view` (like `transform` also accepts a functor).

7.4 Generic adaptor

During our study of the STL we learned that the STL provides several adaptors. These adaptors are using existing data structures to realize an abstract data type. Adaptors are not just relevant for the STL, it is common in most libraries, for example, LEDA is designed in such a way that the user uses pointers to elements to locate the elements stored in the containers. However LEDA provides STL adaptors such that the user can get STL compatible iterators and in that way use the STL generic algorithms on LEDA containers. The observation that adaptors are recurring has turned it into a design pattern, denoted the *adaptor design pattern*, which is defined by the following intention "Convert the interface of a class into another interface expected by specific clients. Adaptor lets classes work together that could not otherwise be possible because of incompatible interfaces".

The idea of an adaptor is to design an interface which uses an existing interface to obtain the functionality required by the desired interface. The generic version of the adaptor design pattern is shown in Figure 11. The class `Adaptor` accepts a class as template argument. This class contains the interface which is used to realize the adaptor. We denote such a class an *adaptee*. In the setting, given in Figure 11, the adaptee is denoted `Adaptee1`.

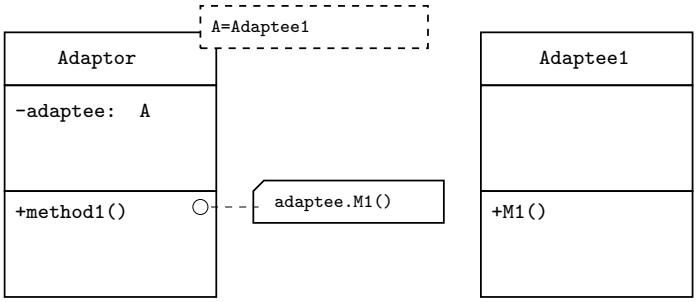


Figure 11. The generic version of the adaptor design pattern.

The calls to the member functions in `Adaptor` will be propagated to the corresponding member functions in `Adaptee1`.

We use adaptors widely in the CPH STL. After we decoupled the containers and the realizators (data structures) we noticed that we could provide support for LEDA interfaces, using our existing realizators [19]. All these interfaces are nothing but adaptors for the existing realizators.

7.5 Generic abstract factory

When designing flexible software components with loose coupling, it is desirable that the relationship between the components becomes implicit. Consider a cross-platform software system containing classes for GUI (Graphical User Interface), printing, I/O, and so on. We do not want to check every time which operating system we are using in order to select the right classes. The *abstract factory design pattern* deals with this kind of problem, and it is defined by its intention "Provide an interface for creating families of related or dependent objects without specifying their concrete classes".

The generic version of the abstract factory design pattern is illustrated in Figure 12. In this setting, we define a `concrete_factory` class for each family of types. In this particular example we have two different families namely A and B, containing two classes each. If `factory` is given `concrete_factoryA`, classes of family A are created, symmetrically for family B.

The abstract factory design pattern is applied in the `vector` component framework. In the CPH STL, a `vector` stores a collection of segments. These segments can either store the value directly in the array or store pointers to objects containing the value. The encapsulator determines which kind of storage mechanism that is used. A kernel organizes the segments; because of code reuse we want just one kernel class for every data structure. We applied the abstract factory design pattern to allow that objects can be created for the different kinds of storage mechanisms. Below is the relevant code for the `direct_encapsulator` and `indirect_encapsulator` classes.

```
1 namespace cphstl {
```

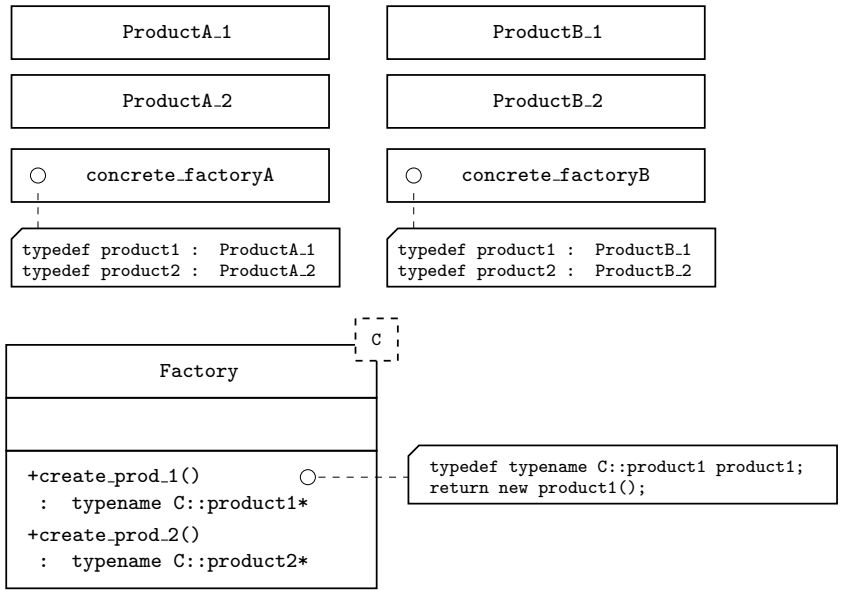


Figure 12. The generic version of the abstract factory design pattern.

```
2   template <typename V, typename A>
3   class direct_encapsulator {
4   private:
5     typedef direct_encapsulator<V, A> this_type;
6   public:
7     typedef this_type slot_type;
8     typedef this_type* segment_type;
9     ...
10  };
11
12  template <typename V, typename A>
13  class indirect_encapsulator {
14  private:
15    typedef indirect_encapsulator<V, A> this_type;
16  public:
17    typedef this_type* slot_type;
18    typedef this_type** segment_type;
19    ...
20  };
21 }
```

The kernel uses the type `segment_type` to determine the type of the segments and the type of the directory containing segments. The type `slot_type` is used by the kernel to determine the type of each element stored in the `vector`.

7.6 Generic proxy

Several programming languages provide a language feature called *lazy evaluation*. This concept means that computation is performed when it is needed. Haskell provides lazy evaluation which makes it possible to create infinite lists. For example, the Haskell program below provides the function `fib_list` which returns an infinite list of Fibonacci numbers:

```
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
5
6 fib_list :: [Int]
7 fib_list = map fib [1..]
```

If the user tries to print the value of `fib_list`, the program will never terminate successfully. However, if we just take the first n elements from the list (by `take`), it will terminate successfully.

Most imperative programming languages (for example, C++ and Java) have no direct support for lazy evaluation, but with some constructions we can obtain a similar behaviour. Consider a program operating on a collection of files. These files are represented by classes, which are opened and read when the program starts. This can result in an undesired delay at program start. This problem can be solved by introducing a proxy class for each file, such that every file is opened and read at the point of time when that file is used. Solutions to similar problems are generalized to be a pattern, namely the *proxy design pattern*, which is defined by the following intention "Provide a surrogate or placeholder for another object to control access to it".

In the CPH STL, we have applied this pattern to solve a problem regarding strong exception safety. Each container must have a variant which provides the strong exception safety guarantee, meaning that a container operation either completes or throws an exception and the container remains in the same state as before the exception was thrown. According to the specification given in the C++ standard, `swap` must not throw an exception. Since the allocators and comparators are swapped when the containers are swapped, an exception may be thrown.

A solution to the problem described above is to introduce an allocator proxy and a comparator proxy, which provide the same member functions as their corresponding real subjects, but they store instances of the real subjects as pointers. An elementary operation, like swapping a pointer, cannot fail. The code for the comparator proxy is partly shown below (the allocator proxy is similar):

```
1 namespace cphstl {
2   template <typename C>
3   class comparator_proxy {
4   public:
```

```
5       typedef typename C::first_argument_type first_argument_type;
6       typedef typename C::second_argument_type second_argument_type;
7       comparator_proxy(C const& c = C()) {
8         (*this).c = new C(c);
9       }
10      comparator_proxy(comparator_proxy&);
11      comparator_proxy operator=(comparator_proxy const&);
12      comparator_proxy operator=(C const& c);
13      ~comparator_proxy() {
14        delete (*this).c;
15      }
16      bool operator()(first_argument_type const& t1,
17                      second_argument_type const& t2) const {
18        return (*(*this).c).operator()(t1, t2);
19      }
20      C subject() const {
21        return *((*this).c);
22      }
23    private:
24      C* c;
25    };
26  }
```

7.7 Generic template method

We discussed the allocator concept earlier. Each container accepts an allocator which contains member functions for memory management (for example, allocation and deallocation). Some may argue that the allocator is just a policy or a strategy. We believe the allocator is more than that, since the code of the container is written without code for memory management. That code is moved to the allocator, which provides simple functions, like `allocate` and `deallocate`. The container uses these member functions without knowledge of how memory is allocated. This design is an application of the *template-method design pattern*, which intention is the following "Define the skeleton of an algorithm in an operation, deferring some steps to policies. Template method lets policies redefine certain steps of an algorithm without changing the algorithm's structure".

One may say that this pattern extends the strategy design pattern, by allowing a family of operations to be encapsulated into classes. The generic version of the template-method design pattern is illustrated in Figure 13. The `Skeleton` class accepts a policy class, given as template argument. The policy class contains a collection of operations; this collection of operations defines the variable functionality within the context of the `Skeleton` class; hence the `Skeleton` class contains the code which is independent of the possible policy classes. The operations within the `Skeleton` class should now be composed by the operations defined in the policy classes. An alternative structure based on template-based inheritance can be found in [8].

The kernel concept in our paper on component frameworks is the result of

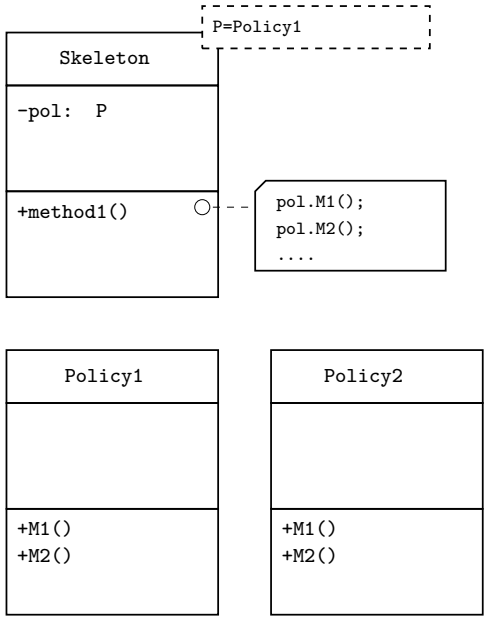


Figure 13. The generic version of the template-method design pattern.

our use of the template method design pattern. We will study this concept by using the following example. The insert by locator member function of the framework is defined by the following code (`k` denotes the kernel object, and `f` denotes the factory[8] object):

```
1  iterator insert(iterator pos, value_type const& v) {
2    (*this).k.grow(1);
3
4    (*this).f.create(v, (*this).k.access((*this).k.size()), (*this).k.
         size());
5
6    if (pos.first != (*this).k.size()) {
7      (*this).block_copy_backward(pos.first, (*this).k.size(), 1);
8    }
9
10   (*this).k.size((*this).k.size() + 1);
11   return concrete_iterator(pos.first, (*this).s);
12 }
```

What happens in this member function is the following. First, we call `grow` in the kernel to request capacity for one additional element. The contract between the kernel and the framework is that either is the call successful and one extra slot is available or an exception is thrown. Afterwards, the element is constructed using the `access` member function which maps a logical index

[8] The factory should not be confused with the abstract factory design pattern, see the paper on component frameworks for details.

to a physical address. Finally the elements are copied by backward copying and the size is adjusted. As the observant reader may notice, we make no assumptions of how the underlying data structure is organized.

References

[1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).

[2] Boost Community, *Boost C++ libraries*, Website accessible at `http://www.boost.org` (1999–2008).

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons Ltd. (1996).

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).

[5] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley (2000).

[6] Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2009).

[7] Digital Mars, D programming language, Website accessible at `http://www.digitalmars.com/d/` (1999–2009).

[8] 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*, USENIX Association (2001), 189–202.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[10] GNU, *libstdc++*, Website accessible at `http://gcc.gnu.org/onlinedocs/libstdc++/` (1999-2008).

[11] 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.

[12] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.

[13] H. R. Lewis and C. H. Papadimitriou, *Elements of the theory of computation*, 2nd Edition, Prentice-Hall International, Inc. (1998).

[14] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press (2000).

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

[16] B. B. Mortensen, The deque class in the Copenhagen STL: First attempt, CPH STL report **2001-4**, Department of Computer Science, University of Copenhagen (2001).

[17] D. R. Musser, G. J. Derge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001).

[18] J. M. L. Muñoz, *A Custom Block Allocator for Speeding Up VC++ STL*, Webpage accessible at `http://www.codeproject.com/KB/stl/blockallocator.aspx` (2006).

[19] M. Neidhardt, Extending the CPH STL with LEDA APIs, Technical report, Department of Computer Science, University of Copenhagen (2009).

[20] G. Rode, Evaluating software design patterns, Master's Thesis, Department of Computer Science, University of Copenhagen (2008).

[21] Silicon Graphics, Inc., *Standard template library programmer's guide*, Website accessible at `http://www.sgi.com/tech/stl/` (1993–2004).

[22] B. Simonsen, A framework for implementing associative containers, CPH STL report **2009-3**, Department of Computer Science, University of Copenhagen (2009).

[23] H. Sutter, Pimpls — Beauty marks you can depend on, *C++ report* **10**, 5 (1998). (Available at http://www.gotw.ca/publications/mill04.htm.)

[24] A. S. Tanenbaum, *Computer Networks*, 4th Edition, Prentice Hall PTR (2002).

[25] S. Thomson, *Haskell: The Craft of Functional Programming*, 2nd Edition, Addison-Wesley (1999).

[26] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-

Wesley (2003).
[27] T. L. Veldhuizen, C++ templates are Turing complete, Technical report, Indiana University, Computer Science (2003).
[28] M. Zalewski, Private communication (2009).

Applying Design Patterns to Specify the Architecture of a Generic Program Library[1]

Jyrki Katajainen and Bo Simonsen

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

Abstract. The standard template library (STL), now part of the C++ standard library, ships with every standard-compliant C++ compiler. In this paper, we apply design patterns to specify the structure of the CPH STL, which is a special edition of the STL developed at the University of Copenhagen. Three design patterns have had significant influence on the structure of the library: bridge, strategy, and iterator. With the use of design patterns we obtain a high-quality design with many desirable characteristics including simplicity, ease of maintenance, loose coupling, extensibility, and reusability. Because of a high degree of parameterization, the usability of components becomes a central issue and we propose a solution that may be of independent interest for developers of other template libraries.

Keywords. Generic programming, design patterns, program libraries

[1] Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

Table of contents

1. Introduction

In 1985, in his influential paper [43], Naur advocated that one should see programming as theory building. On a software project, developers build up in their minds a *theory*, a certain kind of insight of the system under construction, and at the same time bind this theory to their code. Naur adapted this notion of theory from Ryle [48]. In modern software-development literature, the theory is called a *software architecture* [15], a *conceptual model* [24], a *common vision* [32], and a *domain model* [23]. In spite of these different names, it is commonly accepted that the same theory should underlie design, implementation, and team communication. Often it is also important to communicate the theory, or parts of it, to the user of the system developed.

In this paper, we specify the theory behind a generic program library, namely the Copenhagen standard template library (CPH STL) [19]. This library is an enhanced edition of the STL [51], which has become a part of the C++ standard library [14]. The STL is organized around three fundamental concepts: containers, algorithms, and iterators. A *container* is implemented as a class template, i.e. as a class parameterized with a type, an integral constant, a function, or a mixture of these. A container represents a dynamic collection of objects and supports a certain set of operations for the manipulation of these objects. The container classes specified in the current C++ standard include `list`, `vector`, `deque`, `set`, `multiset`, `map`, and `multimap`. In the STL, an *algorithm* is implemented as a function template, i.e. as a function parameterized like a class template. An *iterator* is implemented as a class template and is used as an interface between algorithms and containers.

Instead of just providing one implementation for each individual component, the CPH STL provides several alternative implementations with different performance and safety characteristics. Also, the library provides some extended functionality in the form of new components. The development of the CPH STL has been used as a reality exercise when training software developers. The first prototypes of the components, available when we started this work, were independent packages programmed by groups of one to three people. Often the groups worked independently of each others and there was a small amount of shared code across the packages. To eliminate redundant code and to increase code reuse, refactoring of the whole library was inevitable. In accordance with Naur's writings, we found it necessary to specify the theory behind the library in order to facilitate the development of the next versions of the components.

According to Stepanov [42, Foreword], the principal designer and the original implementor of the STL, the cornerstones of the design of the STL were:

- generic programming,
- efficiency,
- von Neumann computational model, and

– value semantics.

Most of these fundamentals are still valid for the CPH STL, so let us look at them one at the time.

In the context of the C++ programming language [14], generic programming means programming with function and class templates. In other words, individual components can take type arguments as well as value arguments; type arguments are expanded at compile time and value arguments at run time. Stepanov advocated [42, Foreword] that generic programming is more than just programming with templates by saying that the objective of generic programming "is to develop a taxonomy of algorithms, data structures, memory allocation mechanisms, and other software artifacts in a way that allows the highest level of reuse, modularity, and usability". The STL is the most well-known application of generic programming in this extended meaning.

The development of generic library components is challenging since a component should work efficiently for arbitrarily many types that can be provided as arguments. In practice, a generic component can be made as efficient as its non-generic counterpart by letting the compiler generate a separate instance of the generic component for every permutation of type arguments. A generic component can be faster than a corresponding hand-coded component since generic code can permit some new form of inlining that would not be possible otherwise. A generic component can also be slower since the compiler can miss some optimization opportunities. In case of generic source code, a good optimizing compiler is a prerequisite for the generation of efficient target code.

The classical von-Neumann architecture consists of a processor and a one-dimensional memory that can be accessed in unit time. In other words, the architecture defines both a model of computation and a model of cost. Much of our code is designed with this architecture in mind although we have also investigated other computer architectures. These include LRU-cache model [12], ideal-cache model [47], two-level memory model [44], superscalar processor model [41], and multi-core processor model [22]. The main conclusion one can draw from these investigations is that many of the components in the STL have to be rewritten if one wants to take full advantage of the facilities provided by these architectures.

In the STL, containers operate on *value semantics* (also called *copy semantics*). Every time an element is inserted into a container, a copy of that element is taken and this copy is hereafter owned by the container. If a user requires *pointer semantics* (also called *reference semantics*), handles to elements are to be stored instead. For efficiency reasons, sometimes it may be relevant to rely on *move semantics* [30] where containers take over the ownership of elements. In most cases value semantics works fine, but we have found that it is a hindrance to the reuse of components. A typical problem occurs when a container is implemented using two or more components and when an element is moved from one of these components to another. Because of such a move, references to elements stored in the con-

tainer can be kept valid only if the components store handles to elements, and if elements are not moved physically.

In the CPH STL, kernels for node-based containers rely on move semantics which will allow a high degree of reuse without loss of efficiency. In other words, the interfaces of the kernels are similar to those used in textbooks on algorithms and data structures (compare [17]), where memory management is moved almost completely outside the kernels.

Example 1. Let `N` denote the type of nodes. A node-based kernel built on move semantics supports, among other things, the operations:

```
N* begin() const;
void insert(N*);
void erase(N*);
```

Let `r` be a node-based kernel `r.begin` returns a pointer to the node storing the first element of `r`, `r.insert` takes a pointer to a node and inserts that node into `r`, and `r.erase` takes a pointer to a node and removes that node from `r`. A node can be in one kernel at a time, and after an insert the kernel owns the given node. An erase removes the given node from the internal structure of the kernel and releases the ownership; thereafter that node can be freed or moved to another kernel of the same type. □

Already in the original design of the STL, design patterns were used although this connection has often been left implicit in earlier writings. The seminal book on design patterns [26] was written at the same time as the STL was developed. That is, they are both based on the same heritage. In the traditional approach, design patterns rely on dynamic binding. In the STL, however, design patterns rely on static binding. Generic design patterns are discussed, for example, in [21, 53]. We decided to make the use of design patterns explicit since they enable us to describe the architectural concepts of the CPH STL in a clean way. Also, they provide a natural way to explain how components can be customized and reused. This description should make the genericity and flexibility of the library obvious for the reader.

We expect our readers to be developers, testers, maintainers, and users of the C++ standard library and other program libraries. We assume that the reader is familiar with generic programming as it is exercised in C++ (see, for example [18, Chapter 6]), and design patterns as described in [26]. Brief descriptions of the design patterns relevant for this paper are given in Table 1.

We have organized the main body of the text in accord with design patterns: bridge (Section 2), strategy (Section 3), and iterator (Section 4). We reflect on our design in Section 5, discuss missing language features that would have simplified the development in Section 6, and compare the design to some related work in Section 7. Finally, we offer a few concluding remarks in Section 8.

Table 1. Design patterns relevant for this study. The descriptions are taken from [26, 21] and modified so that they apply both to inheritance-based dynamic binding and template-based static binding.

Pattern	Description
Adapter	Convert the interface of a class into another interface expected by specific clients. Adapter lets classes work together that could not otherwise be possible because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Decorator	Define additional responsibilities to a set of objects or replace functionalities of a set of objects.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Iterator	Provide a way to access the elements of an aggregate object without exposing its underlying representation.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Strategy	Define a family of strategies, encapsulate each one, and make them interchangeable. This lets the strategy vary independently from clients that use it.

2. Bridges and realizations

One mission of the CPH STL project is to design alternative versions of individual STL components. This means that there can be several kernels, which implement the same container. As an example, `set` and `multiset` can both be implemented using an AVL tree [4] or a red-black tree [29]. Also, there can be several interfaces using the same kernel. For example, the containers provided by LEDA [39] can be implemented using CPH STL kernels.

The interface of each container is implemented as a *bridge class*, the design of which is based on the bridge design pattern [26]. The intention of this design pattern is to decouple an abstraction from its implementation, which in our context means that the kernel, which realizes the container, is decoupled from the interface. We will hereafter denote the kernel as a *realization*. The addition of the bridge class provides flexibility such that it is easy to change the realization of a container. Our bridges are not just wrappers that delegate the work to realizations, but we allow new member functions to be implemented within the bridge class so that realization classes can be kept minimal.

Example 2. Many bridge classes provide the member function `clear` which empties a container. Often it can be implemented at the bridge by deleting

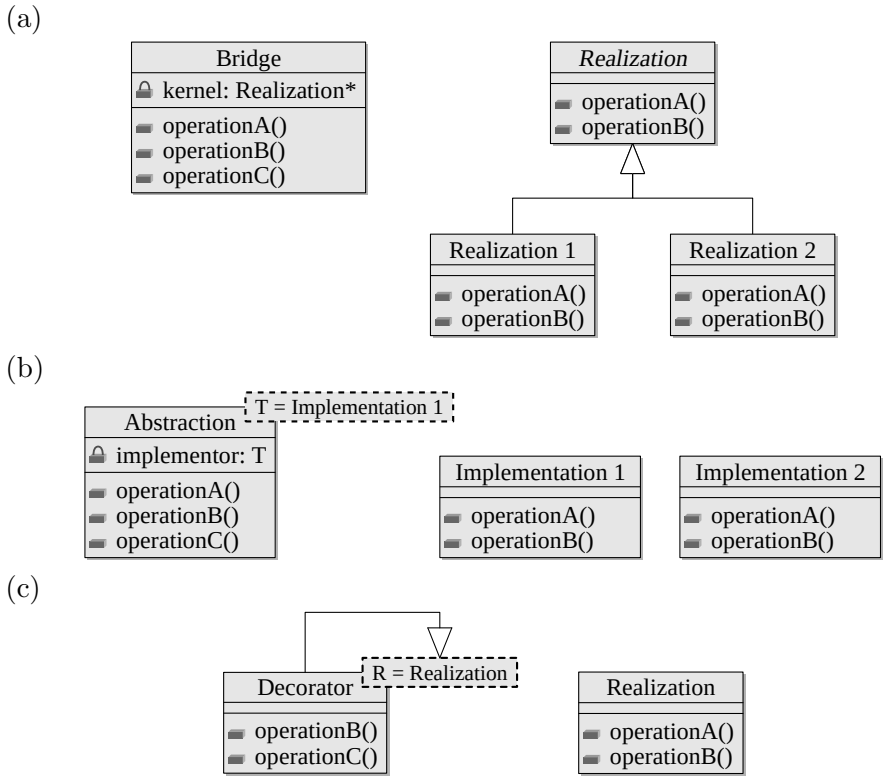


Figure 1. (a) The conventional bridge pattern, (b) the generic bridge pattern, and (c) the generic decorator pattern.

all the elements one by one, i.e. by calling `erase(begin(), end())`. This is just the required behaviour since the member functions `begin` and `end` return iterators to the first and past-the-end elements of the container, and `erase` deletes all the elements within the half-open range specified by the given iterators. □

As illustrated in Figure 1(a), in its conventional form the bridge pattern [26] uses inheritance to obtain a unified type for the realizations. This unified type is an abstract class which is a base class for each realization class. A bridge stores a pointer to an object of this unified type. Because of dynamic binding, the responsibility of creating such an object can be moved outside the bridge.

Our design uses the generic bridge pattern as illustrated in Figure 1(b). In our design the realization is given as a type argument, so we do not have a unified type. Compared to the solution based on inheritance, the member functions common to the bridge and realization have to be provided in both; at the bridge the work is just delegated to the realization. This means more code, but it makes the interface explicit.

An alternative design is based on the generic decorator pattern [21] illustrated in Figure 1(c). Given two classes `X` and `Y`, `Y` inherits all the members of `X`, `Y` can add new functionality by providing new members and extend old functionality by overriding inherited members. In the generic decorator pattern, `X` is given to `Y` as a type argument. In our context, `X` would be the realization class and `Y` the container class. This design was rejected because the realization can provide member functions which will become visible in the container, but the container should provide a specific interface, which the realization is not allowed to extend. This is a real problem, since a realization can be used to implement several interfaces, e.g. both LEDA and STL interfaces. The problem can be solved using private inheritance, but then the decorator will become identical to our bridge, where some functionality is in the bridge and some is delegated to the realization.

In addition to supporting several data structures, our desire is to provide three types of realizations for each container: one that is optimized for speed, one that is space efficient, and one that is safe [35]. In Figure 2, the `set` and `vector` bridge classes, and some of their realization classes and customization classes are shown. A realization can be parameterized with a customization parameter (or parameters) in order to obtain some desirable properties. For example, a red-black tree is parameterized with the type of nodes to achieve different space efficiency: six, five, or four words per element, excluding the space reserved by the element itself. In a similar way, a dynamic array is parameterized with the type of array entries, i.e. instead of using the type of elements directly this type is encapsulated in a class.

Example 3. In the CPH STL, the type parameter list for `set` is as follows:

```
template <
  typename V,
  typename C = std::less<V>,
  typename A = std::allocator<V>,
  typename R = std::set<V, C, A>,
  typename I = typename R::iterator,
  typename J = typename R::const_iterator
>
class set;
```

Here `V` is the type of the elements stored, `C` the type of the comparator used in element comparisons, `A` the type of the allocator used for reserving and freeing memory, `R` the type of the realization, `I` the type of the mutable iterator, and `J` the type of the immutable iterator. Because of the default type parameters, it is possible to use `cphstl::set` in the same way as `std::set`. □

Example 4. Assume that `V` denotes the type of the elements and let `u` be an object of type `V`. A user of the CPH STL can employ the `set` bridge class with the AVL tree as the underlying realization in the following way:

```
typedef std::less<V> C;
typedef std::allocator<V> A;
typedef cphstl::avl_tree_node<V> N;
```

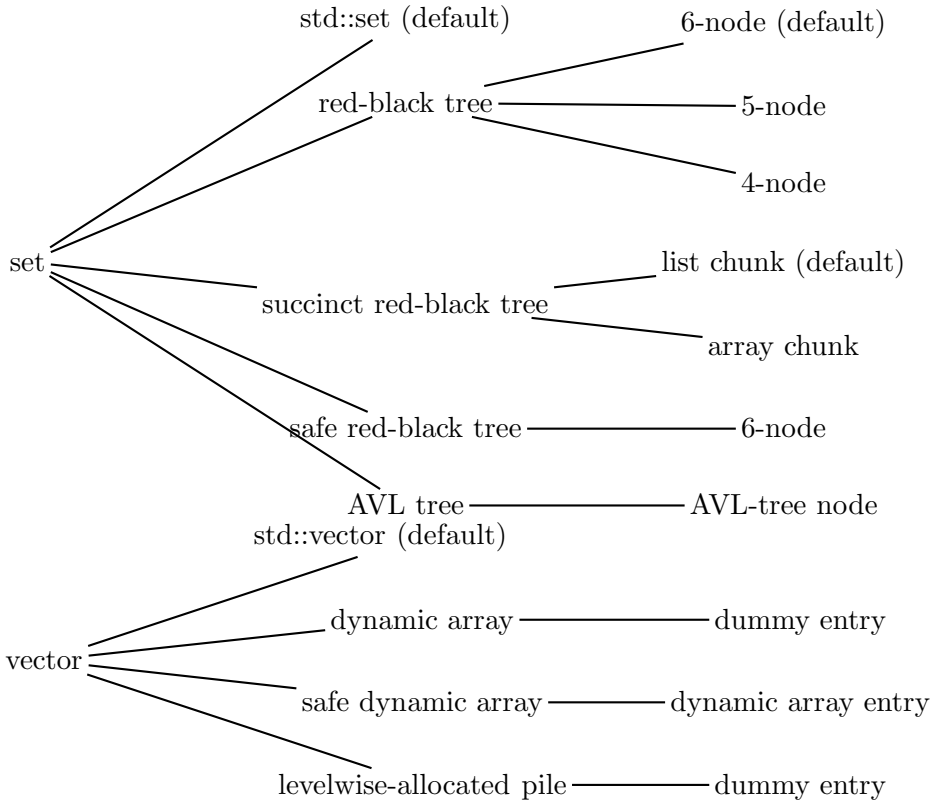


Figure 2. Two bridge classes, some of their realization classes and customization classes.

```
typedef cphstl::identity_functor<V> F;
typedef cphstl::avl_tree<V, V, F, C, A, N, false> R;
typedef cphstl::node_iterator<N, false> I;
typedef cphstl::node_iterator<N, true> J;
typedef cphstl::set<V, C, A, R, I, J> B;
B s;
s.insert(u);
```

□

In the original design of the STL the adapter design pattern was used; `stack`, `queue`, and `priority_queue` are adaptors to existing containers. We upgraded these adaptors to full-fledged containers. After this upgrade, these containers are used in the same way as the corresponding adaptors only when default type parameters are in use. On the other hand, these containers provide increased flexibility and extended functionality, like full iterator support, and the use of these components is unified with the use of the other containers.

Example 5. Assume that `V` denotes the type of the elements and `A` the type of the allocator. Furthermore, assume that `u` and `v` are objects of type `V`. The following code extract shows the difference between the use of `cphstl::stack` and `std::stack`.

```
typedef cphstl::list_based_stack<V, A> R;
typedef cphstl::stack<V, A, R> S;
typedef S::const_iterator J;

S s;
s.push(u);
s.push(v);
for(J p = s.begin(); p != s.end(); ++p) {
  std::cout << *p << std::endl;
}

std::stack<V, std::list<V, A> > r;
r.push(u);
r.push(v);
std::stack<V, std::list<V, A> > t(r);
while(!t.empty()) {
  std::cout << t.top() << std::endl;
  t.pop();
}
```

With `cphstl::stack` it is possible to traverse the elements without modifying the container, whereas with `std::stack` providing no iterator support a copy of the container has to be taken. □

Even though we abolished the original container adaptors from the CPH STL, the container adaptor—or better, container decorator—concept can still be useful for the users of our library. In the literature, this issue is discussed under the names *container adaptors* and *views* (see, for example, [25, 45]). Such decorators provide much of the same functionality as views in relational databases. The idea is to store the data only once and provide transformed views of the same data that the users can operate on. This way programs may become more readable and easier to maintain. Within the library we have found use for container decorators when providing different traversal mechanisms for dynamically changing containers much like in [25].

3. Strategies and policies

The strategy pattern was already used in the original design of the STL. In the literature strategies are sometimes referred to as policies [53, 6]. Specifically, a *policy* is a strategy that is selected at compile time. The purpose of a policy class is to customize the behavior of another class. In the STL, allocator and comparator classes given to container classes can be seen as policies.

Example 6. For `cphstl::meldable_priority_queue` class template, which is a CPH STL extension, the second type parameter specified is the type of the comparator used in element comparisons. If the elements stored are of type `V` and `std::less<V>` is the type of the given comparator, the elements will be organized in max-heap order. On the other hand, if `std::greater<V>` is the type of the given comparator, the elements will be

organized in min-heap order. Instead of using these predefined functors, the user can also rely on his or her own comparator. □

In our design, policy classes are widely used as customization parameters. In addition to the element type, bridge and realization classes take policy classes as type arguments. One can even say that the realization is a policy itself. The iterator class is given to the bridge class as a policy to make it possible to visit the elements in a container in a specific order. Furthermore, many realization classes accept a storage policy which specifies how the data is stored and manipulated. The reason for customizing the realization class with a storage policy is to provide different time-space trade-offs. Currently, our red-black tree implementation accepts several storage policies, which can be used to reduce space requirements from six words per element to four words per element.

Example 7. Let `V` denote the type of the elements stored. In the CPH STL, the storage policy defining AVL-tree nodes has the following interface:

```
template <typename V>
class avl_tree_node {
public:
  typedef V value_type;
  typedef avl_tree_node<V> node_type;
private:
  node_type* _parent;
  node_type* _left;
  node_type* _right;
  node_type* _prev;
  node_type* _next;
  short _balance;
  value_type _data;
public:
  node_type*& parent();
  node_type*& left();
  node_type*& right();
  short& balance();
  node_type* predecessor() const;
  node_type*& predecessor();
  node_type* successor() const;
  node_type*& successor();
  value_type const& content() const;
  value_type& content();
};
```

The member functions `predecessor`, `successor`, and `content` come in two versions. This is the normal idiom for writing get and set methods in C++. The immutable version is used for reading (get) and the mutable version for writing (set). The two versions are needed to guarantee `const` correctness when these functions are called (by the iterator operations). If only one version is provided, as for `parent`, `left`, `right`, `balance`, the same member function is used for both reading and writing. □

In the literature, even more aggressive use of policies is encouraged (see, for example, [5, 8]). In [8] a method is described to untangle the searching and balancing of binary search trees and to encapsulate them into policy classes. A balancing policy defines how the search tree is restructured after a modifying operation, and a search policy defines how searching is done, for example, whether to search for an element with a specific key or a specific rank. In this way the same realization can implement both a red-black tree [29] and a splay tree [50] by changing these two policies and the underlying storage policy.

Policy-based design has several advantages including avoidance of redundant code, simplification of maintenance, larger flexibility for the library user, and smaller parts of the code to test. There are of course also disadvantages including lack of overview concerning the code, configuration difficulties because of possible component mismatch. There can be some efficiency problems as well, because it can be difficult for the compiler to optimize policy-based code. Despite of these disadvantages, we prefer policy-based design because of the flexibility provided.

Example 8. Policies together with simple template metaprogramming can also be used for performance tuning as discussed, for example, in [18, Section 10.12]. We have observed that update operations for `vector` based on a dynamic array are slow when elements being manipulated are expensive to copy, which is often the case for user-defined types. This inefficiency is due to relocations of elements, involving elements constructions and destructions. A faster behaviour can be obtained by letting the array store pointers to elements instead. Depending on the type of elements one of these realizations can be selected at compile time.

```
template <typename V, typename A>
class dynamic_array_selector {
public:
  typedef cphstl::dynamic_array_entry<V> D;
  typedef cphstl::safe_dynamic_array<V, A, D> Q;
  typedef cphstl::dummy_entry<V> E;
  typedef cphstl::dynamic_array<V, A, E> R;
  typedef typename cphstl::if_then_else<cphstl::type<V>::is_class, Q ↩
      , R>::type realization;
};

template <
  typename V,
  typename A = std::allocator<V>,
  typename R = typename dynamic_array_selector<V, A>::realization,
  ...
>
class vector {
  ...
};
```

□

4. Abstract and concrete iterators

In the STL, iterators are the glue between containers and generic algorithms. An *iterator* is an object that points to another object, and provides the same semantics as a regular pointer. An iterator can be used to locate an element in a container, for example, when deleting an element. It can also be used to traverse the elements in a container or part of a container. According to the iterator design pattern [26], the iterator should provide this functionality without exposing the underlying representation of the container to the user.

An iterator is called a *bidirectional iterator* if it can move one step forward and one step backward with one iterator operation (`++`, `--`), and a *random-access iterator* if it is possible to move the iterator an arbitrary number of positions in either directions, no matter where the iterator is pointing to. Of the STL container classes, `list`, `map`, `multimap`, `set`, and `multiset` must support bidirectional iterators, and `deque` and `vector` random-access iterators.

Our design for iterators is layered: we have concrete iterators at the implementation level and abstract iterators at the interface level. A *concrete iterator* can iterate over the elements stored in a realization. An *abstract iterator* encapsulates the concrete iterator and provides the member functions which are relevant for the user in order to access the elements in the underlying container.

There are four kinds of classes involved in our overall design: bridge, iterator, realization, and storage classes. The bridge is responsible for creating abstract iterators from concrete iterators, and for converting concrete iterators to abstract iterators. It has member types `iterator` and `const_iterator`, and member functions `begin` and `end`, whose work is delegated to the realization. The iterator class defines an interface for accessing and traversing elements; the interface is specified in the C++ standard [14, Chapter 24]. The storage class determines the behaviour of the iterator by providing the member functions which the iterator uses to do the actual work.

To make the conversion between concrete and abstract iterators possible, each abstract iterator class must provide a conversion operator which converts an abstract iterator to a concrete iterator, and a parameterized constructor which creates an abstract iterator from a concrete iterator. These two member functions are declared as private, but the bridge classes need access to these member functions. To allow this, all bridge classes must be friends of an abstract iterator. The friend declarations make the interconnections between components explicit.

This construction makes the conversion between concrete iterators and abstract iterators completely transparent, because when a conversion from an abstract iterator to a concrete iterator is requested, e.g. when the bridge passes an iterator to the realization, the conversion operator is invoked automatically. Symmetrically, when a concrete iterator is to be converted to an abstract iterator, e.g. when the realization passes an iterator to the bridge,

Table 2. The member functions which each node class should provide.

Function prototype	Description
`node_type* successor() const`	Return a pointer to the next node
`node_type* predecessor() const`	Return a pointer the previous node
`value_type const& content() const`	Return a `const` reference to the element stored at the node
`value_type& content()`	Return a reference to the element stored at the node

the parameterized constructor of the abstract iterator is invoked from the bridge.

In order to separate the friend declarations from the iterator, the proxy design pattern could be applied by inserting a proxy class between the bridge class and the iterator class; in context of the proxy pattern, to control access to the private members of the iterator. This means that friends should be defined inside the proxy class, and the conversion operator and parameterized constructor should be moved to the proxy class. Now the proxy should be a friend of the iterator. We avoided this construction since it complicates the design.

We have observed that two kinds of iterators are sufficient to implement iterators for all STL containers. A *node iterator* provides a bidirectional iteration mechanism and an *entry iterator* a random-access iteration mechanism. These iterator classes are predefined in the CPH STL, but the user can design his or her own iterator classes (and the corresponding storage policies), and instantiate any of the bridge classes with these class templates.

The node iterator is suitable for the realization of containers which rely on nodes, i.e. class templates `list`, `set`, `multiset`, `map`, and `multimap`. The `node_iterator` class takes two type arguments: the type of the node `N` and a Boolean constant `is_const`. The Boolean constant specifies whether the iterator to be implemented is `const_iterator` or `iterator` [7]. The node iterator uses the member functions defined in the node class (see Table 2) to implement the increment (`operator++`), decrement (`operator--`), and dereferencing (`operator*` and `operator->`) operators.

Example 9. The following friend template inside the `node_iterator` class template gives `set` access to the private members of `node_iterator`:

```
template <typename V, typename C, typename A, typename R>
class set;
...
template <typename N, bool is_const = false>
class node_iterator {
  template <typename V, typename C, typename A, typename R>
  friend class cphstl::set;
  ...
```

Table 3. The member functions which each entry class should provide.

Function prototype	Description
`entry_type* advance(difference_type n) const`	Return a pointer to the entry located `n` positions from the current position
`difference_type distance(entry_type* p) const`	Return the distance between `p` and the current position
`value_type const& content() const`	Return a `const` reference to the element stored at the entry
`value_type& content()`	Return a reference to the element stored at the entry

};

Because the bridge classes are forward declared, the node-iterator file does not need to include the files where the bridge classes are defined. □

The entry iterator is suitable for realizing iterators for the container classes `vector` and `deque`. An *entry* is a location in an array together with the operations provided, which makes it a storage policy. Some of the operations are iterator related (see Table 3). The entry iterator takes two type arguments: the type of the entry `E` and a Boolean constant `is_const`. The entry iterator is based on the same principles as the node iterator, but the member functions which the iterator (and thereby the entry) provides are different because of the random-access property.

Every standard-conforming container should come with immutable and mutable iterators providing forward and backward iteration mechanisms, and also reverse versions of these iterators traversing the container in the opposite direction. Applications may, however, require more advanced ways of iterating over the elements. To do this, iterators coming under the names *custom iterators* [10], *smart iterators* [11], and *iterator adaptors* [1] have been proposed in the literature. Even an attempt is made to standardize these components [2]. Our parameterization allows any of these parameterized iterators to be made the native iterator of a container. On the other hand, iterators of any of our containers can be given as an argument for these parameterized iterators, so the same container can provide several iteration orderings at the same time. That is, these constructions complement each others and are an interesting addition to the toolbox of the users of our library.

5. Reflections

In this section we reflect on our design. An overview of selected components related to `set`, using an AVL tree as its realization, is shown in Figure 3. For other containers the big picture is similar.

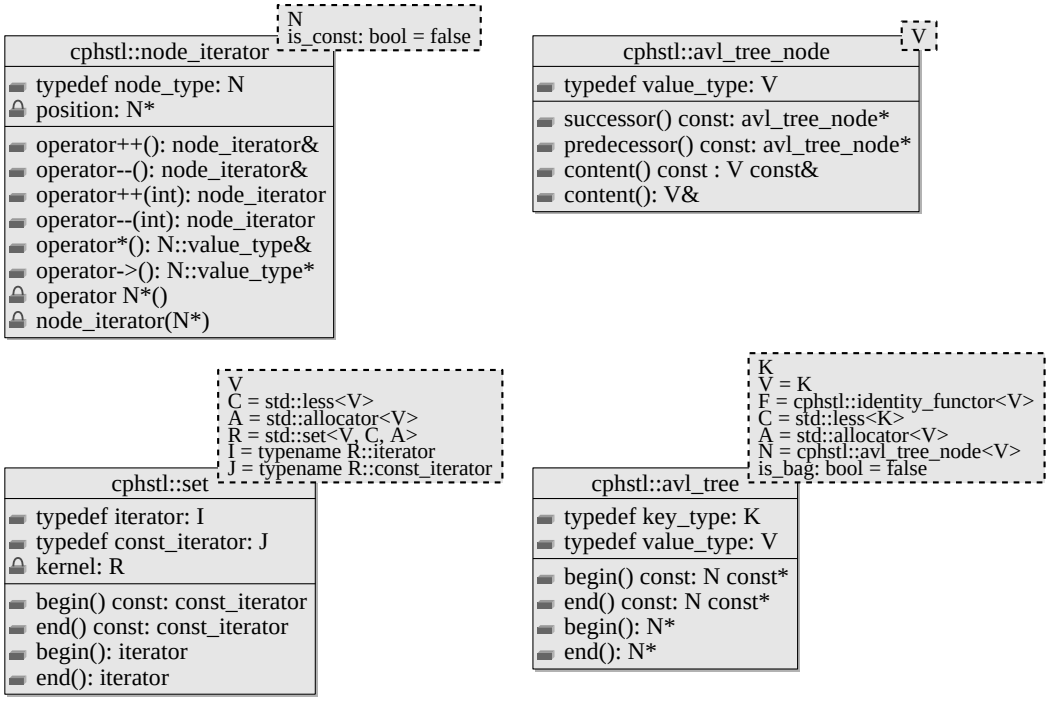


Figure 3. An overview of selected components related to `set`.

To evaluate the quality of our design we considered the desirable characteristics discussed in [37] and [38, Chapter 5]. We are doing well in the following areas:

Simplicity: Whenever we had to make a design decision, we aimed at achieving simplicity. We have consciously avoided the use of inheritance and advanced template metaprogramming. Furthermore, we avoided solutions which would result in extra classes with no functionality but only describing structure. For example, we rejected the usage of the proxy pattern in our iterator design.

Ease of maintenance: Much of the functionality provided by containers can be implemented at bridge classes. This means that we avoid redundant code in realization classes, and these can be kept minimal. Also, we have a high degree of code reuse in the scope of iterators. Since most container classes use the same kind of storage policy, we can implement the iterator classes by using a small well-defined interface in the storage policy.

Loose coupling: Due to the addition of bridge classes, we made the coupling between the interfaces and their implementations looser. We also decoupled the data structure from the traversal mechanism, since the iterators are given to the bridges as policies.

Extensibility: It is relatively easy for the user to change the realization of

a container. Furthermore, the user can customize the data structures by implementing new storage policies. It is also possible for the user to design new iterator classes either by extending predefined iterators or by implementing new iterator classes.

Reusability: We can produce several different interfaces which use the same realizations. For example, we can use the realizations to implement both STL and LEDA interfaces.

Efficiency: Already the first implementation of the STL, written by Stepanov and his collaborators [51], provided components with good performance; some even outdid most of their hand-crafted competition. This was a deciding factor for integrating the STL into the C++ standard library [52]. All our data structures require linear space, linear on the number of elements currently stored, and versions optimized for speed rely on data structures known to provide the best practical performance and to fulfil the strict complexity requirements specified in the C++ standard.

Robustness: For safe versions of our data structures, iterators and references to elements stored inside a data structure are required to stay valid at all times, and all operations are required to provide the strong guarantee of exception safety. In other words, if an exception is thrown, the data structure being manipulated remains in the state in which it was before the operation started. In spite of these strong guarantees, the complexity requirements specified in the C++ standard can still be fulfilled [34].

On the other hand, our infrastructure has obvious disadvantages.

Usability: Because of the increased flexibility from the library user's point of view a lot of type arguments has to be given to bridge classes in order to use the containers, if the user is not satisfied with default realizations. For example, the user needs to provide the same type arguments for both the bridge and realization classes.

Stratification: To do any customization, the user needs to be aware of both the interface and implementation levels. So it is not enough to work at one level of abstraction at a time.

When constructing software, compromises are to be made on contradicting objectives. In our case several contradicting objectives were unavoidable: extensibility versus usability, extensibility versus stratification, reusability versus simplicity, and reusability versus usability. Within our design we focused on maximizing extensibility and reusability, but the price we had to pay was decreased usability and less stringent stratification.

These drawbacks are serious so we experimented with possible solutions. A considerable improvement in usability is achieved by allowing named type arguments in the declarations of bridges, letting the user supply the arguments in arbitrary order, and deducing the arguments not specified from the defaults. In order to implement this idea, we considered using advanced metaprogramming techniques like in [3], but due to language limitations we

found this approach unsatisfactory. We came to the conclusion that it would be better to rely on a preprocessor that can process named type arguments and substitute the remaining type arguments by their defaults. This idea is further elaborated in the following example.

Example 10. In declarations we desire the following syntax where the use of `!` is borrowed from the D programming language [20]:

```
cphstl::set!<V = int, N = my_node, R = cphstl::avl_tree> s;
cphstl::set!<V = char, A = my_allocator<V> > t;
```

A preprocessor should translate this code to the following:

```
cphstl::set<int, std::less<int>, std::allocator<int>, cphstl:: ↩
    avl_tree<int, int, cphstl::identity_functor<int>, std::less<int ↩
    >, std::allocator<int>, my_node<int>, false>, cphstl:: ↩
    node_iterator<my_node, false>, cphstl::node_iterator<my_node, ↩
    true> > s;
cphstl::set<char, std::less<char>, my_allocator<char>, std::set<char ↩
    , std::less<char>, my_allocator<char> >, std::set<char, std:: ↩
    less<char>, my_allocator<char> >::iterator, std::set<char, std:: ↩
    less<char>, my_allocator<char> >::const_iterator> t;
```

The preprocessor uses the named type arguments given between the symbols `!<` and `>`. Naturally, the symbol pairs can be nested. The preprocessor substitutes each type argument by either the one given or the default, which is defined in a table provided as input for the preprocessor or, alternatively, generated automatically but scanning through all include files. When the type arguments are given explicitly between the symbols `<` and `>`, no substitution takes place. □

6. Language limitations

We discuss language features that would have made the development of the library easier. Many of the features are proposed to be included in the next C++ standard, known as C++0x [31].

In our current implementation, appropriate bridge classes are friends of every abstract iterator class. That is, the `friend` declarations are needed, but their number is limited. It would be desirable to be able to remove the list of friend declarations and define the class given as a type argument as `friend`. For example, by adding an extra type parameter `B`, denoting the bridge, the code for `node_iterator` would look as

```
template <typename B, typename N, bool is_const = false>
class node_iterator {
   ...
   friend class B; // ERROR: not allowed
}
```

Unfortunately, this construction is not allowed by the current C++ standard [14], but it is proposed to be included in the future version of C++ [40].

A minor problem with the iterator construction was found, for exam-

ple, in the `insert` member function of `set`. In the realization class the return type of this member function is `std::pair<N*, bool>` and in the bridge class the corresponding return type is `std::pair<I, bool>`. When the realization returns a pair of type `std::pair<N*, bool>` to the bridge, it will try to construct a pair of type `std::pair<I, bool>` using the parameterized constructor of `std::pair`. The parameterized constructor of `node_iterator` will now be invoked in the constructor of `std::pair`, but the construction should take place in the bridge (recall the bridge is a friend of the iterator). This gives a compilation error. An attempt to solve this problem was to split the pair inside the bridge, but it requires knowledge of type `N*`, and the bridge does not have that knowledge. It only knows the realization type, and it cannot assume that the realization has member type `N`, since it is not part of the C++ standard. The best solution right now is to let `std::pair` be a friend of the iterator class, until the new C++ standard is accepted which provides the `auto` keyword [33]. This keyword acts as a placeholder for a type until it is deduced, i.e. it gives access to types which are not known.

When the bridge pattern is used in conjunction with inheritance, there is a natural restriction on which classes can be accepted as realizations, since they all have a common base class. When templates are used, any types will be accepted as type arguments and it is not possible to add any requirements specifying acceptable types. In case of a component mismatch, the user of the library may get a cryptic error message from the compiler. Here static assertions [36] and concepts [28], planned additions to the existing C++ standard, will come to our rescue and let us specify such requirements explicitly.

The implementation of all STL container classes `set`, `multiset`, `map`, and `multimap` can base on the same underlying data structure. One difference is that a collection of type `set` and `multiset` store elements, and a collection of type `map` and `multimap` (key, satellite data) pairs. Another difference is that a collection of type `multiset` and `multimap` can contain duplicates, whereas in a collection of type `set` and `map` elements are unique. These differences can be communicated to the realization using type parameters: a functor `F` which translates elements into keys, and a Boolean constant `is_bag` which tells whether duplicates are allowed. For some member functions the behaviour varies depending on the value of `is_bag`. We need to make a partial specialization of these member functions, but this is not allowed by the current C++ standard; the standard requires that the whole class is specialized. A way of circumventing this rule is to rely on a member function overloading; in the literature the technique is called *tag dispatching*. The idea is to construct a type from a Boolean constant and use this type as a tag for overloaded member functions [6, Section 2.4]. We prefer that a Boolean constant could be used directly to specialize a member function instead of this workaround.

Example 11. This example shows how tag dispatching is used for specializing the `insert` member function in the AVL tree:

```
template <
  typename K,
  typename V = K,
  typename F = cphstl::identity_functor<V>,
  typename C = std::less<K>,
  typename A = std::allocator<V>,
  typename N = cphstl::avl_tree_node<V>,
  bool is_bag = false
>
class avl_tree {
protected:
  std::pair<N*, bool> insert_dispatch(N*, cphstl::bool2type<false>);
  N* insert_dispatch(N*, cphstl::bool2type<true>);

public:
  typedef typename cphstl::if_then_else<is_bag, N*, std::pair<N*,  ↩
      bool> >::type return_type;

  return_type insert(N* x) {
    return insert_dispatch(x, cphstl::bool2type<is_bag>());
  }
};
```

□

7. Related work

In this section, we briefly compare our design to the designs provided in other libraries. In our survey, we only focus on facilities related to the `vector` and `set` container classes. We look inside GNU `libstdc++`, SGI STL, Boost, and LEDA.

The GNU standard C++ library [27] is included in GNU C/C++ compiler. The version of `libstdc++` used for this evaluation was release 4.2.3. Iterators for `vector` are built using the class template `normal_iterator`, which wraps a pointer and has the same operators as a regular pointer. The iterator object is constructed from a pointer, the type of which is given as a type argument. Each operator of the iterator simply applies the same operator to its pointer. This construction is only usable for pointers to arrays. The `set` container class is designed as a bridge class, but the realization is hard-coded to be a red-black tree. The iterators (immutable and mutable iterators) for the red-black tree are defined explicitly as `struct`s, which implies that the encapsulation of the data member is not handled properly. Since iterators are defined explicitly as class members inside each container class, the amount of reuse is low.

The SGI STL [49] is included in the SGI C++ compiler environment. The `vector` class template does not provide an iterator, but it defines the iterator to be a pointer to an element of the type provided as a type argument. This yields several problems with respect to iterator validity, and it is not possible to extend the iterator without writing an entirely new iterator. The implementation of `set` is fixed to be a red-black tree. The iterators for `set`

are implemented as one class, which takes two type arguments such that immutable and mutable iterators are implemented within the same class [7]. The iterator is implemented as a `struct` so every data member is public, and encapsulation is not handled properly.

The Boost library [13] provides a mechanism for constructing iterators with less code than creating entire iterators from scratch. Since this library provides generic add-ons, not a full implementation of the STL, the iterator construction provided cannot rely on the underlying container like our construction. The Boost iterator construction is divided into three parts: the iterator facade, iterator adaptor, and specifier. The *iterator facade* provides the interface defined for iterators in the C++ standard. The operators delegate their work to the member functions defined in subclasses. This is similar to the node iterator which uses the node class to do the actual work. The *iterator adaptor* defines the interface which each specifier should implement. The adaptor class inherits the iterator operators from the facade class. The *specifier* is the class which should be implemented in order to make it possible to iterate over the data stored in a container. According to our terminology, the specifier class inherits abstract iterator operations from the iterator adaptor class and provides concrete iterator operations that are used in the implementation of the abstract iterator operations. The concrete iterator operations are defined to be private, since they are only used by the abstract iterator operations. The abstract iterator operations are defined to be public, because they should be accessible to the user. The CPH STL iterator construction and the Boost iterator construction have a lot in common, but to reduce redundant code, the CPH STL construction is bound to containers. However, we found it confusing that in the specifier class, abstract iterator operations and concrete iterator operations are not segregated from each other.

In the algorithm research community, LEDA [39] is a highly-respected library of data structures and algorithms, which provides tools for solving geometric and graph-theoretic problems. LEDA has containers like the STL. The user operates with so-called *items* to access the elements stored in a container. The item class is defined for each container, e.g. for a dictionary `d`, the `d.lookup` member function returns an object of type `dic_item`. The user can get the key (satellite data) stored inside the item by invoking the `d.key` (`d.inf`) member function with the item as its argument. In our terminology, items are similar to concrete iterators. Unlike the STL, not all containers provide a traversal mechanism; some do like the list container. For a list `l` one can use `l.succ(item)` to get the item to the next element. Notice that the `succ` member function is located in the container, not in the item as in the STL. Containers, which do not provide the `succ` member function, provide an iteration mechanism `forall(p, d)`, which performs a complete traversal of `d` such that `p` refers to the current item. LEDA has some bridge classes, for which it is possible to give the implementation as a type argument. A dictionary using an AVL tree can be created using the statement: `dictionary<string, int, avl_tree> d`. Unfortunately,

LEDA has some limitations. It is not possible for the user to define their own items, i.e. storage policies, and the separation between algorithms and containers has not taken place, but many general-purpose algorithms are defined as member functions inside the containers.

8. Concluding remarks

In generic programming, the vision is to create software artifacts in a way that allows the highest level of reuse, modularity, and usability. The design patterns used to specify the architecture of the library are themselves reusable artifacts. Even though other developers may not be able to use exactly the same patterns, a broad spectrum of design patterns is available, and can be used for designing and building complex systems like generic program libraries. The components provided by our library are created to be reusable. On the other hand, our inspection of other STL implementations revealed that at the level of source code reuse was not the best possible. We are not the first to make this observation [9], but still this is surprising since these implementations are widely used and available on almost every computer with a C++ compiler. As to reuse, with our design we moved one step forward by removing some redundant code in the implementation of iterators. The same class template is used for realizing both immutable and mutable iterators, and only two iterator classes are needed for sustaining all bridge classes, not one per bridge class.

The design patterns applied by us can be considered standard; we consciously avoided using exotic patterns and complicated techniques so that it would be easier for our developers and users to understand the overall design. The bridge pattern has been used in earlier implementations of the STL, but full advantage of the pattern has not be taken. In several places we tried to attain loose coupling between components, for example, between bridges and iterators and between realizations and their customizations. These good design principles are reusable and applicable in the development of other program libraries.

As to modularity, the bridge design pattern creates a solid foundation for a layered software architecture [15, 23]. At the moment, four conceptual layers exist: application layer, decorator layer (container, algorithm, and iterator decorators), interface layer (bridges, algorithms, and iterators), and implementation layer (realizations and policies). One of our problems is that all these layers are visible for a general user, but the library is extendible at each layer. Contrary to earlier implementations of the STL, our iterator design supports stringent encapsulation as required by the iterator pattern. The cost of getting the design right is a bit more complicated code. However, when C++0x becomes reality, it will be easier to realize our design. This is because of the possibility to declare a type argument as a friend and to manipulate objects whose types are not known.

Because of heavy reliance on templates, usability is an issue that has to be

addressed. Due to language limitations and inadequate compiler support, the development of components relying on policy-based design is tedious. Moreover, the use of such components can be difficult because of a potential mismatch between type parameters and type arguments. With the concepts to be provided by C++0x, we will obtain the possibility of specifying the types which are accepted as arguments. This enables the compiler to detect component mismatches, which simplifies both the development and use of components—and improves the usability in general.

It is well-known that in C++ template metaprogramming has its shortcomings; sometimes it can be cumbersome to write generic code and the resulting code can have poor readability. This problem triggered the fundamental idea of creating a C++ preprocessor which supports named type arguments in declarations. Named type arguments would not solve our problem as such; a more advanced deduction mechanism is required, too. We leave the further development of this idea for the programming-language research community. One of the challenges is to integrate the use of concepts with our preprocessing framework.

We aimed at an orthogonal and a simple design. Our evaluation of the design was analytical and heuristic, and we concentrated on a few key characteristics. It would be interesting to try to verify the quality of the design empirically. We plan to conduct a study on the suitability of our design for forthcoming refactoring work. In such a study, it would relevant to investigate quantitatively how high the code reuse is, and qualitatively how easy it is to maintain and extend the library. As far as we know, only a few studies [46, 16] have been conducted on the usability of program libraries.

Software availability

The programs discussed in this study are accessible via the home page of the CPH STL project [19].

Acknowledgements

We thank the members of the CPH STL development team; this paper was built on their shoulders. We have directly benefited from the work of Johannes Serup (queue and stack); Tina A. G. Andersen, Ulrik Schou Jørgensen, Mads D. Kristensen, and Claus Ullerlund (dynamic array); Filip Bruman, Daniel P. Larsen, and Christian Wolfgang (levelwise-allocated pile); Stephan Lynge (AVL tree); Hervé Brönnimann (red-black tree); Claus René Jensen and Finn Krog (succinct red-black tree); Jørgen T. Haahr and Frej Soya (safe red-black tree); Dirk Hasselbalch and Sune Sloth Simonsen (binomial heap). Also, we thank Christian Heide Damm, Kenny Erleben, Klaus Byskov Hoffmann, and Claus Jensen for commenting a preliminary version of this paper; and Kasper Hornbæk for fruitful discussions.

References

[1] D. Abrahams and J. Siek, Policy adaptors and the Boost iterator adaptor library, *Proceedings of the 2nd Workshop on C++ Template Programming* (2001).

[2] D. Abrahams, J. Siek, and T. Witt, Iterator facade and adaptor, Document number **N1641**, The C++ Standards Committee (2004).

[3] D. Abrahams and D. Wallin, *The Boost Parameter Library*, Web document available at `http://www.boost.org/doc` (2005).

[4] G. M. Adel'son-Vel'skiĭ and E. M. Landis, An algorithm for the organization of information, *Soviet Mathematics* **3**, 5 (1962), 1259–1263.

[5] A. Alexandrescu, *Generic<Programming>: A Policy-Based `basic_string` Implementation*, C++ Expert Forum, Available at `http://www.cuj.com/experts/1096/toc.htm` (2001).

[6] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).

[7] M. Austern, Defining iterators and const iterators, *C/C++ Users Journal* **19**, 1 (2001), 74–79.

[8] M. H. Austern, B. Stroustrup, M. Thorup, and J. Wilkinson, Untangling the balancing and searching of balanced binary search trees, *Software—Practice and Experience* **33**, 13 (2003), 1273–1298.

[9] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, Beyond templates: A study of clones in the STL and some general implications, *Proceedings of the 27th International Conference on Software Engineering*, ACM Press (2005), 451–459.

[10] C. Baus and T. Becker, Custom iterators for the STL, *Proceedings of the 1st Workshop on C++ Template Programming* (2000).

[11] T. Becker, Smart iterators and STL, *C/C++ Users Journal* **16**, 9 (1998), 39–45.

[12] J. Bojesen, Managing memory hierarchies, Master's thesis, Department of Computer Science, University of Copenhagen (2000).

[13] Boost Community, *Boost C++ libraries*, Website accessible at `http://www.boost.org` (1999–2008).

[14] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1, BS ISO/IEC 14882:2003*, John Wiley and Sons, Ltd. (2003).

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons Ltd. (1996).

[16] S. Clarke and C. Becker, Using the cognitive dimensions framework to measure the usability of a class library, *Proceedings of the 15th Workshop of the Psychology of Programming Interest Group*, Keele University (2003).

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, The MIT Press (2001).

[18] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley (2000).

[19] Department of Computer Science, University of Copenhagen, *The CPH STL*, Website accessible at `http://cphstl.dk/` (2000–2008).

[20] Digital Mars, *D programming language*, Website accessible at `http://www.digitalmars.com/d/` (1999–2008).

[21] 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.

[22] K. Egdø, A software transactional memory library for C++, Master's thesis, Department of Computer Science, University of Copenhagen (2008).

[23] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley (2004).

[24] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley (1997).

[25] E. Gamess, D. R. Musser, and A. J. Sánchez-Ruíz, Complete traversals and their implementation using the standard template library, *CLEI Electronic Journal* **1**, 2

(1998).
[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).
[27] GNU, *libstdc++*, Website accessible at `http://gcc.gnu.org/onlinedocs/libstdc++/` (1999-2008).
[28] 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.
[29] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts, A new representation for linear lists, *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing*, ACM (1977), 49–60.
[30] H. E. Hinnant, P. Dimov, and D. Abrahams, A proposal to add move semantics support to the C++ language, Document number **N1377**, The C++ Standards Committee (2002).
[31] ISO/IEC, Working draft: Standard for programming language C++, Document number **N2798**, The C++ Standards Committee (2008).
[32] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley (1999).
[33] J. Järvi, B. Stroustrup, and G. D. Reis, Deducing the type of variable from its initializer expression (rev. 4), Document number **N1984**, The C++ Standards Committee (2006).
[34] J. Katajainen, Making operations on standard-library containers strongly exception safe, *Proceedings of the 3rd DIKU-IST Joint Workshop on Foundations of Software. Report 07/07*, Department of Computer Science, University of Copenhagen (2007), 158–169.
[35] J. Katajainen, Stronger guarantees for standard-library containers, *Algorithm Engineering. Oberwolfach report 25/2007*, Mathematisches Forchungsinstitut Oberwolfach (2007), 31–35.
[36] 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).
[37] T. Korson and J. McGregor, Technical criteria for the specification and evaluation of object-oriented libraries, *Software Engineering Journal* **7**, 2 (1992), 85–94.
[38] S. McConnell, *Code Complete*, 2nd Edition, Microsoft Press (2004).
[39] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press (2000).
[40] W. M. Miller, Extended `friend` declarations (rev. 3), Document number **N1791**, The C++ Standards Committee (2005).
[41] S. Mortensen, Refining the pure-C cost model, Master's thesis, Department of Computer Science, University of Copenhagen (2001).
[42] D. R. Musser, G. J. Derge, and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Edition, Addison-Wesley (2001).
[43] P. Naur, Programming as theory building, *Microprocessing and Microprogramming* **15**, 5 (1985), 253–261.
[44] J. H. Olsen and S. C. Skov, Cache-oblivious algorithms in practice, Master's thesis, Department of Computer Science, University of Copenhagen (2002).
[45] G. Powell and M. Weiser, Container adaptors, Technical report **SC 99-41**, Konrad-Zuse-Zentrum für Informationstechnik Berlin (1999).
[46] K. Rodden and A. Blackwell, Class libraries: A challenge for programming usability research, *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*, Brunel University (2002), 186–195.
[47] F. Rønn, Cache-oblivious searching and sorting, Master's thesis, Department of Computer Science, University of Copenhagen (2003).
[48] G. Ryle, *The Concept of Mind*, The University of Chicago Press (1949).
[49] Silicon Graphics, Inc., *Standard template library programmer's guide*, Website accessible at `http://www.sgi.com/tech/stl/` (1993–2004).

[50] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *Journal of the ACM* **32** (1985), 652–686.
[51] A. Stepanov and M. Lee, The standard template library, Technical report **95-11(R.1)**, Hewlett-Packard Laboratories (1995).
[52] B. Stroustrup, Private communication (2006).
[53] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley (2003).

Adaptable Component Frameworks: Using `vector` from the C++ Standard Library as an Example [1]

Jyrki Katajainen and Bo Simonsen

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

Abstract. The CPH STL is a special edition of the STL, the containers and algorithms part of the C++ standard library. The specification of the generic components of the STL is given in the C++ standard. Any implementation of the STL, e.g. the one that ships with your standard-compliant C++ compiler, should provide at least one realization for each container that has the specified characteristics with respect to performance and safety. In the CPH STL project, our goal is to provide several alternative realizations for each STL container. For example, for associative containers we can provide almost any kind of balanced search tree. Also, we do provide safe and compact versions of each container. To ease the maintenance of this large collection of implementations, we have developed component frameworks for the STL containers. In this paper, we describe the design and implementation of a component framework for `vector`, which is undoubtedly the most used container of the C++ standard library. In particular, we specify the details of a `vector` implementation that is safe with respect to referential integrity and strong exception safety. Additionally, we report the experiences and lessons learnt from the development of component frameworks which we hope to be of benefit to persons engaged in the design and implementation of generic software libraries.

Keywords Generic Programming, C++ Standard Library, STL, Robustness, Efficiency

Reprinted from WGP'09,, Proceedings of the ACM SIGPLAN Workshop on Generic Programming, August 30, 2009, Edinburgh, Scotland, UK., pp. 13–24.

[1] Partially supported by the Danish Natural Science Research Council under contract 272-05-0272 (project "Generic programming—algorithms and tools").

Table of contents

1. Introduction

The design and implementation of the standard-library `vector` has a great pedagogical value when illustrating the use of various programming language facilities and programming techniques. For example, in his recent textbook [39], Stroustrup devotes three of the 27 chapters (115 pages or about 9% of the whole book) to a `vector` implementation that is roughly equivalent to the standard-library `vector`. However, textbooks have seldom enough space to describe a complete `vector` implementation. The book on the standard template library (STL) by Plauger et al. [29] is an interesting exception; their complete `vector` implementation consists of 365 logical lines of code (LOC), excluding the partial specialization for Boolean elements, which is even longer than the primary class template. (Observe that in our use of the LOC metric we ignore comment lines and lines with a single parenthesis, and we calculate long statements as single lines.) For other complete implementations, we refer to the source code shipped with your C++ compiler and the documentation of the Silicon Graphics Inc. implementation of the STL [35].

This work is part of the Copenhagen STL (CPH STL) project initiated in 2000 [13]. The goal in this project is to

- provide an enhanced edition of the STL, i.e. the containers and algorithms part of the C++ standard library [9, 19];
- study and analyse existing specifications for and implementations of the STL to determine the best approaches to optimization;
- place the programs developed in the public domain and make them freely available on the Internet;
- provide benchmark results to give library users better grounds for assessing the quality of different STL components; and
- carry out experimental algorithmic research.

The architecture of the CPH STL is described in [22]. Two important tools used when describing the foundations of the library are C++ concepts [15] and design patterns [14]. In this paper we use these tools in an informal way; for a pathway to a more formal treatment, we refer to the above-mentioned papers and the references mentioned therein.

The STL is organized around three fundamental concepts: containers, iterators, and algorithms. Containers are class templates that provide iterators, and algorithms are function templates that work for various kinds of iterators. It is this decoupling of algorithms and containers, and type parameterization in general, that makes the components of the STL so flexible. In the modern literature on C++ design (see, for example, the book by Alexandrescu [2]), it is advocated that even a greater degree of flexibility is achieved by parameterizing generic components with *policies* which are classes or class templates describing configurable behaviour. The paradigm is referred to as policy-based design. According to our terminology, 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.

In this paper, we describe the design and implementation of a component framework for the `vector` container, we report the experiences and lessons learnt from its development, and we evaluate the efficiency of the existing realizations. In total, 15+ developers have been involved in the development of `vector` in the CPH STL. Some of the progress reports have been published on the project website [21, 24].

1.1 Standard-compliant `vector` and relevant extensions

A `vector` stores a sequence of elements such that elements can be accessed by their indices and also by their iterators at constant cost. Compared to an `array`, whose size is fixed (at compile time or at run-time), the size of a `vector` can vary and memory management is handled automatically. In the computing literature, this data structure has been discussed under many names, including dynamic array [16, 33], dynamic table [11], extendible array [32], extensible array [8], flexible array (term used in Algol 68), growable array [31], resizable array [10], and variable-length array [6, 37]. As to the `vector` class in C++, its full specification together with all associated operations can be found in the C++ standard [9, 19]. The `vector` class has two template parameters that allow the user to specify the type of the elements stored and the type of the allocator used for allocating and deallocating memory. We have extended the interface with additional template parameters, which allow the user to specify the type of the data structure used for storing the elements, the type of mutable iterators and immutable iterators (colloquially **const** iterators) used when traversing over the sequence. Because of the default values provided, these extra template parameters do not affect the normal use of the container.

There are several aspects in the specification of `vector` [9, 19] that may not be satisfactory for all users.

Referential integrity: In some applications a `vector` may be used to maintain references to other objects, and these objects may again keep references back to the array. Many programmers have been bitten by the bug that, because of the reallocation of the underlying array, the references back are no more valid. This is an error that is difficult to find. Simply, the rules specified in the C++ standard, when and under what circumstances iterators and references to elements are kept valid, are difficult to remember. Hence, the memory burden on working programmers could be reduced if references and iterators were kept valid by all operations, except when an element is erased.

Strong exception safety: A container operation is *strongly exception safe* [1] if it completes successfully, or throws an exception and makes no changes to the manipulated container and leaks no resources. The rules specified in the C++ standard, which operations guarantee strongly exception safety and under what circumstances, are difficult to remember. Hence, there is a need for a `vector` for which all operations guarantee the strong form of exception safety.

Unspecified behaviour: In the C++ standard, the behaviour of `front`, `back`, and `pop_back` member functions is not specified if the underlying container is empty. Clearly, there is a need for a `vector` for which the behaviour of these member functions is specified. Also, the behaviour of `operator[]` is unspecified when the array index is out of bounds. Often this comes as a big surprise for novice programmers. Even though range checking is done by `at` member function, this function is seldom used. Hence, there is a need for a `vector` for which range checking can be switched on and off when desired.

Contiguous storage: The C++ standard requires that the elements of a `vector` are stored contiguously in memory. However, in the literature many interesting implementations have been proposed which do not keep the elements in a contiguous memory segment (see, for example, [10, 16, 21, 37]). Naturally, this requirement is important in some low-level applications relying on address-of operations, but there should also be space for `vector` implementations that do not fulfil this requirement.

Space utilization: In the C++ standard, no space bounds are specified for the container classes. Because of performance considerations, standard `vector` implementations do not release the allocated memory even if the number of elements gets smaller. As pointed out in [8], in some applications, like long-running programs in servers, such a behaviour can be unacceptable. Many such programs running simultaneously can fill the whole memory although only a small portion of the memory is in actual use. A natural requirement is that no container should use more than linear extra space, linear in the number of elements stored. However, in some applications even this amount can be unacceptable, since elements may be large and the space usage is measured in elements (not in bytes or words). More space-economical `vector` implementations are known: If `n` denotes the number of elements stored, the bound $O(\sqrt{\texttt{n}})$ on the amount of extra space, i.e. the amount of space used in addition to the elements themselves, is known to be achievable [10, 21, 37].

Amortized time bounds: Many member functions of `vector` are required to have $O(1)$ cost in the amortized sense. In this point the C++ standard is unclear since the sequence of operations over which the amortization is performed is never specified. Due to reallocations, the worst-case cost of a single operation like `push_back` can be linear, as is the case for the most common implementations. This can have fatal consequences for other data structures that use a `vector`. For example, a binary heap is expected to support its operations at the logarithmic worst-case cost, but if a `vector` is used in its implementation, this worst-case behaviour does not hold any more [8]. It is known that all `vector` operations can be supported at $O(1)$ worst-case cost, except that insertions and erasures have $O(\sqrt{\texttt{n}})$ worst-case cost if only $O(\sqrt{\texttt{n}})$ extra space is available [21] and, for an arbitrary small but fixed $\varepsilon > 0$, $O(\texttt{n}^{\varepsilon})$ worst-case cost if $O(\texttt{n}^{1-\varepsilon})$ extra space is available [33]. Clearly, it is relevant to provide `vector` implementations that

guarantee good worst-case performance for all operations.

In a normal implementation of the STL, one realization is provided for each container. In the CPH STL, we want to provide at least three predefined realizations for each container: one that is fast, one that is safe, and one that is space efficient. As to `vector`, the user can select between `cphstl::fast_vector`, `cphstl::safe_vector`, and `cphstl::compact_vector`. Moreover, `cphstl::vector` is guaranteed to be standard compliant.

The fast version is implemented by expanding the array by a constant fraction and never contracting the array. The safe version is based on the same expansion strategy, but it also applies a similar contraction strategy (compare [8]). The safe implementation provides referential integrity and strong exception safety. The point is that the safety guarantees are provided without relaxing the performance requirements specified in the C++ standard. This is in a stark contrast with the earlier work (see, e.g. [1]), where the technique of making a complete copy is offered as an option to achieve the strong guarantee of exception safety. However, it took a long time for us to get this version correct. For example, the solution sketched in an earlier working paper [20] was not fully correct, but a bug was found during the implementation phase. The compact version is implemented using a hashed array tree [37] as the underlying data structure.

By examining the specification in the C++ standard carefully, an observant reader can see that the requirements are produced by reverse engineering one particular implementation, one that is similar to `cphstl::fast_vector` storing elements contiguously. Hence, it should not come as a surprise that other implementations are not fully standard compliant. In particular, our safe and compact versions do not store the elements in a contiguous memory segment. As a consequence of this the elements are not addressable. Additionally, we have to rely on different kinds of proxy objects so some operations, like `operator[]` and `operator*` for iterators, return an implementation-defined proxy object, instead of a reference or **const** reference to an element as required by the standard. For the very same reason `vector<bool>` is sometimes said to be an almost container with an almost random-access iterator since it does not fulfil all the requirements specified for the container and random-access iterator concepts.

1.2 Outline of the present paper

Instead of just providing some predefined behaviours, we develop a component framework which allows us (and others) to extend the library with new facilities. Using the terms of Oppermann and Simm [30], the CPH STL is both *adaptive*, i.e. its components are able to change their behaviour based on the type arguments given by the user, and *adaptable*, i.e. the components can be changed and extended by the user who can provide new implementations for the template arguments accepted by the component framework. Our framework can be used for realizing most of the known `vector` imple-

mentations. The component framework for `vector` is described in Sections 2, 3, and 4. When developing this framework, we took inspiration from a similar framework introduced for binary search trees by Austern et al. [5]; a component framework for associative containers is also available at the CPH STL [36].

We had several reasons for introducing component frameworks into our library. We wanted a high level of code reuse, ease of maintenance, and fair benchmarking. Now it is possible to provide a new `vector` kernel by writing a few member functions, whereas a complete `vector` implementation [9] must provide 40 member functions and seven operators. Also, to a high degree we have been able to avoid copy-paste code which eases the maintenance of the library considerably. Furthermore, we can do benchmarking by changing the kernels and policies, and keeping the other parts of the code unchanged. This really shows the effect of a particular change. Hence, hopefully, our benchmarks report differences in the performance of data structures, not the cleverness of the programmers. We make some additional remarks on reusability in Section 6.

Naturally, it is interesting when a container library can automatically adapt itself to different usage scenarios, and perform optimizations and other tasks without user intervention. In the literature, the topic has been discussed under the name active libraries [12]. For a long time, generic programming has known to be a promising approach for generating customized software components. However, in this point we are more pragmatic than earlier authors. In our opinion, in C++, the facilities provided for compile-time reflection and metaprogramming are still too primitive to be of great practical value. We discuss adaptivity from our point of view in Section 7.

Our generic component frameworks are open and adaptable. In the literature many different words are used to describe adaptation activities, including customization, configuration, modification, extension, personalization, and tailoring. In different contexts the meaning of these words can vary. When we talk about adaptability, we mean that the library offers several levels of usage (similar thoughts appear in a more general context, for example, in [17, 27]):

Normal generic use: A generic class template defines a family of classes. As part of a normal instantiation process a user can select a class from this family by specifying the types to be used for substituting the template parameters. The user can use the components of the CPH STL in the same way as the components of the C++ standard library.

Selective use: The user can choose between alternative predefined behaviours, like between the fast, safe, and compact container implementations. A type of use, where parameters impact the performance of components, is common in generic software libraries. For example, in LEDA [26] some container classes accept additional implementation arguments.

Integrated use: The user can compose existing—internal or external—components. For example, in the Boost graph library [34] the per-

formance of many graph algorithms can be tuned by non-functional parameters.

Extended use: The user can extend the library by writing new components. Already the users of the C++ standard library can provide their own allocators and comparators, but we go even further. We allow our users to design and implement their own iterators, policies, and container kernels.

To facilitate extending use, it is necessary that the source code of the library is made available for the users. We close our discussion on adaptability in Section 8.

Our contribution can be summarized as follows.

- We show how a component (`vector`) from the `C++` standard library can be extended to a component framework still providing the same functionality as required by the C++ standard. Our description can serve as a starting point for future work when building similar component frameworks.
- We show that a framework-based implementation of a component (`vector`) has an acceptable performance overhead. (See Section 5.)
- We show that a component (`vector`) can be made to guarantee the strong form of exception safety for all container operations and fulfil the same theoretical performance requirements as the corresponding unsafe variant. The programming techniques used have already shown to be useful when implementing other safe components.
- We show that the cost of safety can be high in terms of actual running time. This is mainly due to the loss of spatial locality in memory references and the overhead caused by additional memory management. (See Section 5.)

We hope that the ideas presented in this paper will be of benefit to other persons engaged in the design and implementation of generic software libraries, or in the tools used in their development.

2. Decomposition

In this section we give an overview of our adaptable component framework for `vector`, and in the following two sections we describe some of the architectural elements in greater detail. The design of a component framework can be seen as an application of the template-method design pattern [14]. However, since we rely on C++ templates, not on inheritance, the implementation-specific details are specified by the template arguments given for the component framework. Hence, the design is also related to the strategy design pattern.

During the years the CPH STL has become a multi-interface library which supports the C++ standard library [9], LEDA [26], and its own application-programming interfaces (APIs) for several data structures. All APIs are

decoupled from their implementations using the bridge design pattern. Because of this design choice, we can conveniently support several APIs. Many of the member functions provided by these APIs are actually convenience functions, so to start with we extracted a core of all the member functions. We call this core a *realizator*. The iterators are also decoupled from the realizators in order to provide a common means of realizing items for the LEDA APIs and iterators for the STL APIs.

After this initial phase, the realizator interface for `vector` has to provide 20 member functions. The realizator class specifies a skeleton that must be filled in by the user with policies. A policy is the generic variant of a strategy used in the strategy design pattern [2]. A policy can be used to customize the class it is given to. In the original design of the STL, allocators and comparators can be seen as policies that are given to the container classes.

In our source of inspiration [5], a component framework for binary search trees was introduced. It is natural that the main variability between the different variants of balanced search trees is the balancing mechanism, which can be placed in a policy class. Other variabilities are the searching mechanism used for searching specific elements and the encapsulation mechanism used for storing the information (nodes can store a colour or some other balancing information). To create a similar component framework for `vector`, we had to do a variability and commonality analysis of the data structures proposed in the literature. Such an analysis revealed that most implementations are built on the following concepts:

Slot: This is a memory location which stores a single element, or a pointer to a proxy that stores the element or knows where the element is stored.

Segment: All `vector` implementations maintain a collection of memory segments, each consisting of a sequence of slots.

Directory: There is a directory that keeps track of the memory segments reserved. A directory can be of varying complexity; if there is only one memory segment in all, the directory is trivial, but more complicated alternatives are also possible.

We decided to package the management of memory segments and the directory inside a small *kernel* which is given as a template argument to the framework. There is a clear contract between the framework and the selected kernel: the framework takes in the elements and moves them around, and the kernel takes care of memory management. As a concept a kernel is defined by the minimal interface which any implementation must comply with. Let $\mathbb{N}$ denote the type of sizes and `P` the type of a proxy functioning as a substitute for a reference to an element. In addition to a constructor and destructor, a `vector` kernel should provide the following operations:

$\mathbb{N}$ **`size() const`:** Get the number of elements stored.

void `size`($\mathbb{N}$ `n`): Set the number of elements stored to `n`.

$\mathbb{N}$ **`max_size() const`:** Get the maximum number of elements that can be stored.

$\mathbb{N}$ **`capacity() const`:** Get the current capacity of the kernel.

`P access(`$\mathbb{N}$ `i)`**:** Convert a logical index `i` to a reference to the data stored at the corresponding slot.

void `grow(`$\mathbb{N}$ δ`)`**:** Increase the number of elements stored by $\delta \in \mathbb{N}$.

void `shrink()`**:** Fit the capacity to the number of elements stored.

In addition to the kernel, the framework accepts the types of elements and an allocator as template arguments. The full conceptual specification of the `ValueType` and allocator concepts can be found in the C++ standard. A container can access the elements via iterators. The conceptual specification of iterator concepts can also be found in the C++ standard. To implement an iterator, one should somehow specify the slot, segment, and directory in which the element pointed to lies. When this information is available, it is possible to locate the element and to advance an iterator forward and backward arbitrary many steps.

The purpose of the proxy design pattern is to provide a means of controlling access to an object. We have found it necessary to employ several different proxies in order to achieve many of the desirable safety properties. The proxies used appear in two varieties. A *surrogate* is used as a substitute for some real subject; to implement this proxy, we maintain a single pointer to the real subject and access the real subject via this pointer. An *encapsulator* [28] is used as a replacement for a real subject such that the proxy and the real subject are functionally identical. In particular, we need a surrogate for a kernel and we have to encapsulate an element, a reference to an element, and a pointer to an element. The purpose of proxies will become clearer when we give more details on the safe variants of `vector`.

In Figure 1, we use the CRC cards [7] to summarize the most important concepts involved.

The user can assemble a realizator by specifying the policies required by the framework. For example, to get a `vector` that stores the elements directly inside the slots and maintains the elements in a single contiguous memory segment, the user could write the code given in Listing 1. Observe that, as proposed in [4], we have combined the implementations of mutable iterators and immutable iterators into the same class; the selection of a proper iterator is done by a Boolean value.

3. Kernels

In this section we will briefly describe the kernels which are available in our `vector` framework at the moment. The kernels are dynamic array [8], hashed array tree [37], and levelwise-allocated pile [21]. The selection of these kernels was based on the results of previous benchmarks performed in our research group and the desirable properties of the data structures. A dynamic array can be used to realize a `vector` that stores the elements contiguously, hashed array tree is space efficient, and levelwise-allocated pile offers good worst-case running times. Throughout this section we assume that the elements are stored directly at the slots; in the next section we will

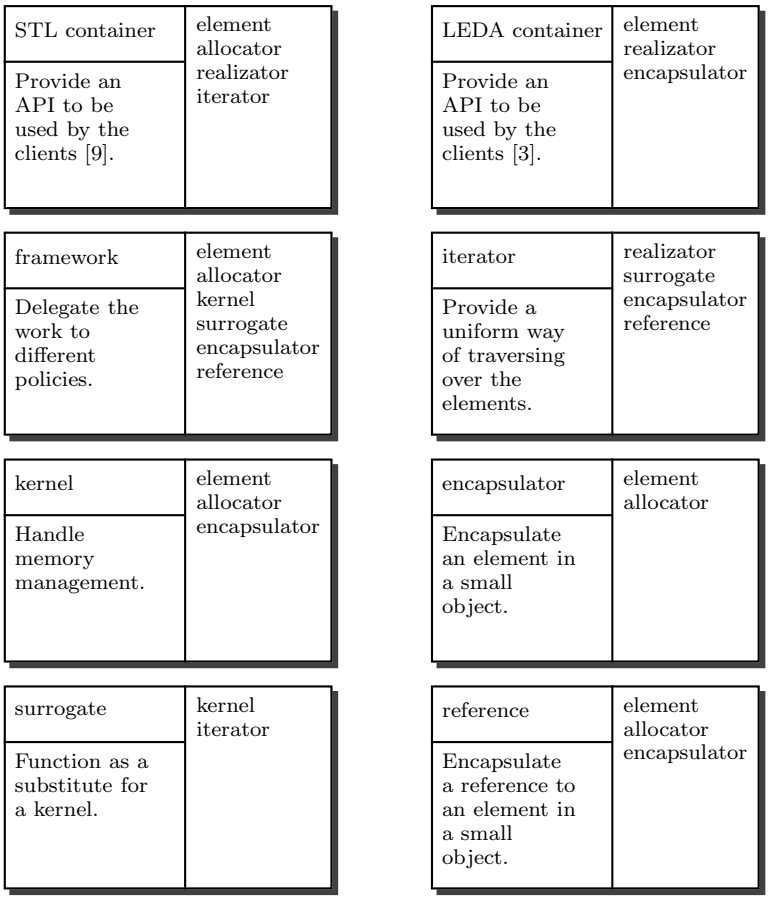


Figure 1. The big picture. In each CRC card, the class/concept name is listed in the upper-left corner, the responsibilities appear on the left below the name and the collaborators on the right.

consider other options to encapsulate elements.

Each kernel has a *size* which denotes the number of elements stored, and a *capacity* which denotes the actual number of slots allocated for storing the elements. If `n` denotes the size and `N` the capacity of a kernel, we use $\lambda \stackrel{\text{def}}{=} \mathtt{n}/\mathtt{N}$ to denote the current *load factor*. When $\lambda = 1$ and we want to increase the size of the kernel, an expansion is necessary and the *expansion factor* α determines the capacity just after the expansion such that $\mathtt{N} = \alpha\mathtt{n}$ and $\lambda = 1/\alpha$. When the load factor becomes too small, a contraction may take place; the *contraction threshold* β specifies the minimum acceptable load factor.

The worst-case space consumption of the data structures is summarized in Table 1 for some typical values of α and β.

Listing 1. An example of the use of the framework.

```
#include "direct-encapsulator.h++"
#include "dynamic-array.h++"
#include <memory> // defines std::allocator
#include "rank-iterator.h++"
#include "stl-vector.h++" // defines cphstl::vector
#include "vector-framework.h++"

int main() {
  enum {immutable = true};

  typedef int V;
  typedef std::allocator<V> A;
  typedef cphstl::direct_encapsulator<V, A> E;
  typedef cphstl::dynamic_array<V, A, E> K;
  typedef cphstl::vector_framework<V, A, K> R;
  typedef cphstl::rank_iterator<R> I;
  typedef cphstl::rank_iterator<R, immutable> J;
  typedef cphstl::vector<V, A, R, I, J> C;

  C v;
}
```

Table 1. Worst-case space consumption of our `vector` kernels when elements are stored directly at the slots. Here `n` denotes the number of elements stored.

Kernel	Space consumption
Dynamic array ($\alpha = 2$; $\beta = 1/4$)	$6\mathtt{n} + O(1)$
Hashed array tree ($\alpha = 4$; $\beta = 1/8$)	$\mathtt{n} + O(\sqrt{\mathtt{n}})$
Levelwise-allocated pile ($\alpha = 2$; $\beta = 1/2$)	$2\mathtt{n} + O(\lg \mathtt{n})$

3.1 Dynamic array

A *dynamic array* is an array, the capacity of which varies as a function of its size. The elements are kept in a contiguous memory segment, and when this segment has no empty slots or too many empty slots, the elements are reallocated to another array. Actually, a dynamic array is a family of data structures depending on the expansion factor and the contraction threshold used. By peeking at the source code of `std::vector` that comes with our compiler (`gcc` version 4.2.4), we saw that it used expansion factor $\alpha = 2$ and contraction threshold $\beta = 0$ (no contraction done). In our current implementation, $\alpha = 2$ and $\beta = 1/4$, but the `shrink` operation can be switched to do nothing if wanted.

The reorganization of the array is done as follows: Allocate a new array of load factor $1/\alpha$, copy the elements from the old array into the new array, deallocate the old array, and adjust the pointer which gives the start address of the array. The reason why we have slack between 1 and $1/\alpha$, and $1/\alpha$

and β is that the reorganization is rather expensive because of memory allocations and copy operations. If we did not have this extra slack, a sequence of intermixed insertion and erasure operations could make this data structure very expensive and unattractive.

Since the elements are stored in an array, the efficiency of all array operations is the same as for a fixed-sized array, except the cost associated with the reorganizations. According to the standard amortized analysis (see, for example, [11, Section 17.4]), the additional cost incurred by reorganizations is only $O(1)$ per modifying operation. For our implementation, the worst-case space consumption of a dynamic array storing `n` elements can be as high as $6\mathtt{n} + O(1)$. The worst case occurs when the old array uses only 1/4 of its capacity, which means that $4\mathtt{n}$ slots are in use, and the new array uses double the current size, which means $2\mathtt{n}$ slots.

3.2 Hashed array tree

The hashed array tree consists of two parts: a directory of size `m` and $\Theta(\mathtt{m})$ segments of size `m`. The directory stores pointers to the beginning of respective segments. We denote `m` as the *segment size* and we ensure that it is a power of two at all times. We only allocate space for segments which store elements, and we maintain the invariant that at most $O(1)$ segments are non-full. When the maximum capacity for the current segment size is used, a reorganization is performed. In such reorganization the new segment size is determined and all elements are relocated to a new data structure using this new segment size. Also, when the load factor gets below 1/8, a similar reorganization is carried out. A lookup of an element with index `i` is done by accessing the slot $\mathtt{d}[\lfloor \mathtt{i}/\mathtt{m} \rfloor][\mathtt{i} \bmod \mathtt{m}]$ in the directory `d`. In the current implementation this computation is done fast using a shift and a bitwise-and operation. In Figure 2, an example of the data structure storing the integers $\langle 0, 1, \ldots, 15 \rangle$ is shown.

The hashed array tree is our preferable data structure for the compact variant of `cphstl::vector` since the memory overhead can be bounded by $O(\sqrt{\mathtt{n}})$, `n` being the current size. To achieve this space bound, the reorganization has to be done such that a memory segment in the old data structure is immediately released after all its elements have been moved into the new data structure. Since in both data structures the sizes of the directories and the sizes of the non-full segments are proportional to $\sqrt{\mathtt{n}}$, the amount of extra space used, even during reorganization, is only $O(\sqrt{\mathtt{n}})$. Observe that for the safe version this optimization is not possible since the copy constructor for elements is provided by the user and it can fail by throwing an exception. Therefore, an element copy is not necessarily reversible, and the old segments can first be released after all copies have been taken. Otherwise, some data may be lost.

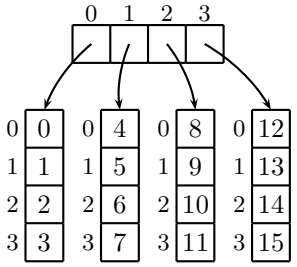


Figure 2. The organization of data in a hashed array tree.

3.3 Levelwise-allocated pile

A levelwise-allocated pile is similar to a hashed array tree. However, its directory is a small `vector` (whose initial capacity is set to 32) and memory segments are arrays of size $2^{\mathtt{k}}$ where `k` is a parameter stored at the kernel. The data structure is expanded by increasing `k` by one and allocating a new segment of size $2^{\mathtt{k}}$, and contracted by decreasing `k` by one and deallocating the last empty segment provided that the second last non-empty segment has lost more than, say, 8 elements. A lookup of an element with index `i` is performed by accessing the slot $\mathtt{d}[\lfloor \lg(\mathtt{i}+1) \rfloor][\mathtt{i}-2^{\lfloor \lg(\mathtt{i}+1) \rfloor}+1]$ in the directory `d`. An example of a levelwise-allocated pile storing integers $\langle 0, 1, \ldots, 14 \rangle$ is shown in Figure 3.

This data structure is attractive since elements are never moved because of an expansion or a contraction. Due to the dynamization strategy the amount of space allocated is never more than $2\mathtt{n} + O(\lg \mathtt{n})$ if there are `n` elements in total. However, since the memory segments are of varying size, some space may be lost due to memory fragmentation. Also, the lookup formula can be problematic since it requires the calculation of the whole-number logarithm. (This is a primitive operation in all Intel processors.) Compared to a hashed array tree, the computation of the whole-number logarithm is more expensive than performing a shift and a bitwise-and operation.

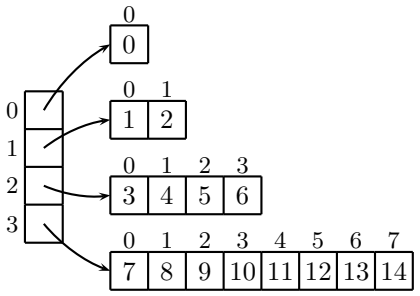


Figure 3. The organization of data in a levelwise-allocated pile.

4. Proxies

Up to now we have assumed that the elements are stored directly at the slots. There are two major problems with direct encapsulation. First, modifying operations may invalidate iterators and references to elements held within the data structure (referential integrity). Second, it may not be possible to revert to the former state of the data structure if the copy constructor of the element throws an exception (strong exception safety). In this section we will present the key ideas how to avoid both of these problems.

4.1 Referential integrity

The reason why lists and associative containers can guarantee referential integrity is that they store the elements in separate allocated objects. The same indirect-encapsulation mechanism can be used for `vector`; this way we can achieve referential integrity and partially strong exception safety. We denote the allocated object an *element encapsulator* since its purpose is to encapsulate an element in an appropriate way. After this modification, a kernel maintains pointers to encapsulators and the iterators also point to encapsulators. To maximize genericity, our equivalent version to `std::vector` stores an array of encapsulators. For this version, every encapsulator is a class containing the element along with member functions for accessing that element.

Keeping just one pointer, from a memory segment maintained by the kernel to an encapsulator, is not enough for guaranteeing referential integrity. When an iterator is advanced `k` slots, the iterator needs a pointer from the encapsulator to the corresponding slot in the kernel, so it can get the pointer to the encapsulator which lies `k` slots from the current slot. The backpointer from the encapsulator to the kernel slot does not point to the memory segment explicitly since a memory segment may be reallocated every time the size of the kernel changes. Instead, each encapsulator stores an index of the corresponding slot. Additionally, each iterator has to keep a pointer to the encapsulator and to the kernel. Now the iterator can execute an advance operation by retrieving the index from the current encapsulator and then using the `access` member function of the kernel to get the pointer to the desired encapsulator. During insertions and erasures we need to update the indices in the encapsulators, but since the pointers from the kernel to the encapsulators are either copied or moved, this additional work does not result in any increase in the asymptotic time complexity. Indirect encapsulation is illustrated in Figure 4.

By letting the iterators contain pointers to kernels may still cause inconsistency. Namely, when two containers are swapped, iterators get invalidated. This problem can be solved by introducing a *kernel surrogate* which is a small object containing just one pointer to the kernel. The idea is that iterators should, instead of a pointer to the kernel, hold a pointer to the surrogate. The surrogate is allocated by an allocator and a backpointer to

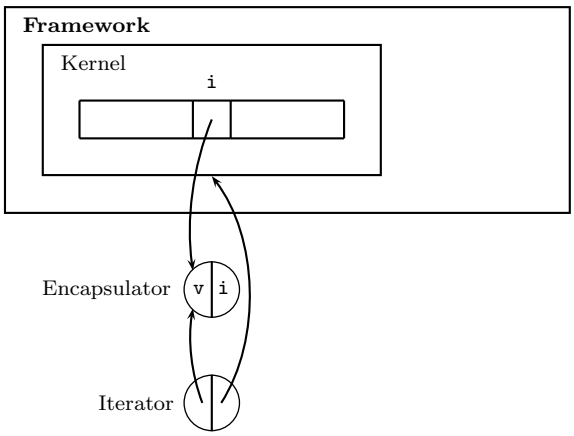


Figure 4. The encapsulator mechanism.

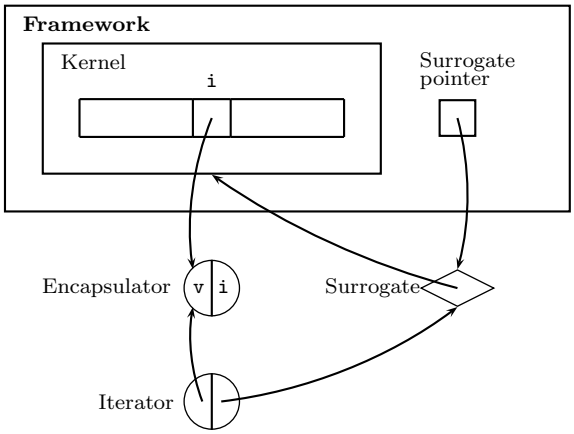


Figure 5. The surrogate mechanism.

the surrogate is maintained in the framework instance. Swapping two containers is now done as follows: First the pointers stored in the surrogates are swapped, and then the backpointers to the surrogates in the framework instances are swapped. The surrogate mechanism is illustrated in Figure 5.

4.2 Strong exception safety

General programming techniques for crafting exception-safe programs are discussed in [38], and specifics for creating a strongly exception-safe `vector` in [20]. We will not repeat the material that can be found from the earlier sources, but concentrate on a single issue that we have found problematic: How to make `operator[]` strongly exception safe?

Let us look into the scenario shown in Listing 2. In this program, `vector`

Listing 2. An error scenario for `operator[]`

```
#include <stdexcept> // defines std::domain_error
#include <stl-vector.h++> // defines cphstl::vector

class my_class {
public:

  my_class(int const& a) {
  }

  my_class const& operator=(my_class const&) {
    throw std::domain_error("...");
  }
};

int main() {
  cphstl::vector<my_class> v;
  v.insert(v.begin(), my_class(5));
  v[0] = my_class(6); // my_class::operator= fails
}
```

`v` that consists of objects of type `my_class` is created, an element is inserted into `v`, and the created value is modified. During the last operation an exception is thrown, and the container is now in an inconsistent state. This means that our `vector` does not provide the strong form of exception safety. One may argue that the exception was not thrown in the scope of the container, so it is the user's responsibility to handle possible exceptions. We disagree, since the user cannot necessarily recover from this error.

To provide a safe mechanism for performing this operation, we will ensure that this exception is handled within the scope of the library. According to the C++ standard, `operator[]` should return a reference to the type of the value, which we cannot control. Instead, we will return a *reference proxy*, which we can control. The behaviour is almost the same as if a reference was returned. The reference proxy has `operator=` as its member function which will perform the assignment within a **try-catch** block. We need to make some changes to the underlying data structure for it to work, since if an exception occurs, we cannot necessarily undo this action because an exception can be thrown in the copy constructor, too. Moving the element outside the encapsulator, and allocating the space for it with an allocator, solves our problem. Now `operator=` allocates a new element and explicitly invokes the assignment operator; if the operation fails, we deallocate the element whose allocation failed and the container will still be in the same state as before the exception was thrown. If an exception is not thrown, the new element is attached to the encapsulator and the old element is deallocated. Referential integrity is still maintained since the iterators point to encapsulators. In this situation, we say that the elements are encapsulated

doubly indirectly. An overview of the different ways to encapsulate elements is given in Figure 6.

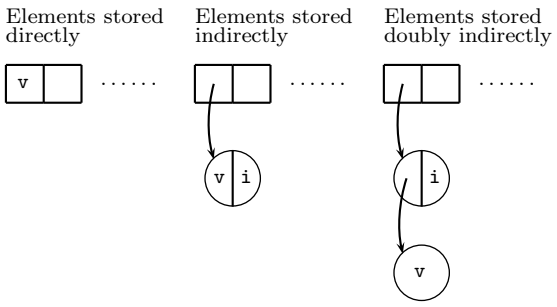


Figure 6. The three different encapsulation strategies.

To maximize genericity the creation of encapsulator objects takes place in another class, in a so-called *factory*. This class is needed since an object of an encapsulator class is not necessarily created in the same way. For example, for the encapsulators which encapsulate elements indirectly, the object needs to be allocated and afterwards constructed; for the encapsulator which encapsulates elements directly, the encapsulator is just constructed. To provide the two alternative behaviours, we used partial specialization when implementing the factory class.

4.3 Iterators

As to iterators, we have predefined two different class templates: one supporting direct encapsulation (*rank iterator*) and another supporting indirect encapsulation (*proxy iterator*). The rank iterator keeps an index, which corresponds to the current slot, and a pointer to the surrogate object. The proxy iterator keeps a pointer to the encapsulator object, which corresponds to the current slot, and a pointer to the surrogate object.

To make the framework work for both kinds of iterators, the member functions cannot accept iterators as input arguments or as return values. Inside the framework, indices are used instead. For the communication between the framework and container, the iterator class provides a conversion mechanism to convert an iterator to an index, and vice versa. This conversion between iterators and indices is completely transparent; it is done by a parameterized constructor and a conversion operator. Both of these member functions are protected so that they can only be used by the friends; in particular, the `vector` container must be a friend of the iterator classes. If this was not the case, the iterator encapsulation would break down. A sequence diagram illustrating the conversion mechanism is shown in Figure 7.

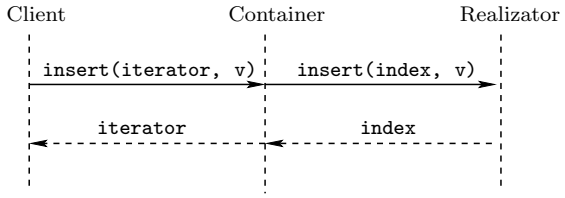


Figure 7. A sequence diagram showing what happens in an `insert` operation.

5. Benchmarks

There are two questions related to our framework which could be interesting to answer:

1. Does the use of the framework result in any performance loss?
2. What is the extra cost associated with safety?

To answer these questions we performed some experiments using the framework. In this section we describe the experiments ran and report the results obtained.

The overall picture of the experimental results was very consistent across the computers where we ran the benchmarks. The results reported here were carried out on a PC with the following configuration:
CPU: Intel Core 2 Duo at 2.4 GHz
Memory size: 2 GB
Cache size: 2 MB
Operating system: Ubuntu 8.04.2, kernel 2.6.24
Compiler: `gcc` 4.2.4 with optimization flag `-O3`.
The experiments were run on a dedicated machine by closing down all unnecessary system processes. Each individual experiment was repeated 10 times to be sure that the clock precision would not cause big inaccuracies in the results.

In our experiments, the elements stored were integers. We considered the three kernels combined with different encapsulation policies (but we only report the results for the dynamic array with doubly-indirect encapsulation). For the sake of comparison, we also report the results obtained for `std::vector`. Let `v` and `w` be two integer `vector`s. We performed five experiments for different values of `n`:

`push_back`: For $\texttt{i} \in [0, \texttt{n})$: `v.push_back(i)`.
`pop_back`: For $\texttt{i} \in [0, \texttt{n})$: `v.pop_back()`.
`operator[]` sequential access: For $\texttt{i} \in [0, \texttt{n})$: `v[i] = 0`.
`operator[]` random access: For $\texttt{i} \in [0, \texttt{n})$: `v[w[i]] = 0`. Before this, the elements in `w` were randomly shuffled.
`insert`: For $\texttt{k} \in [0, 100)$: `v.insert(v.begin() + n/2, k)`.
In our graphs we report the execution times per operation. The time needed for all initializations is excluded in the numbers reported.

The results obtained are shown in Figures 8, 9, 10, 11, and 12. In general, `std::vector` is much faster than the CPH STL implementations. However, the dynamic array with direct encapsulation, which is a similar to `std::vector`, is not much slower. In an earlier study [36] we have shown that it is possible to implement a component framework with an acceptable loss in performance. This also seems to be true for our `vector` framework. Even if our safe variants maintain the desired asymptotic complexity, the constant factors introduced are high. Each level of indirection increases the execution time by a significant additive term. Cache misses and memory allocations are expensive in contemporary computers!

A thorough inspection of the figures gives rise to two additional remarks. All our kernels ensure that the amount of space used is linear in the number of elements stored (provided that the `reserve` member function is not called). From Figure 9 we can see that this makes `pop_back` much slower than that available in the standard implementation. However, the cost of `pop_back` is comparable to that of `push_back` which should be acceptable for most applications. From Figure 12 we can see that `insert` is extremely slow for a levelwise-allocated pile. The execution time of the direct version is about the same as that of the indirect version. This means that the operation is CPU bound, indicating that the computation of the whole-number logarithm is expensive. The problem is that the framework calls the `access` member of the kernel when copying the elements, and this is done for each element. If copying was implemented in the kernel, most of these computations could be avoided. This example shows that a framework-based approach can incur extra overhead.

6. Reusability

It is well-known that LOC is a questionable software metric. In spite of this we carried out a brief analysis on our code base. So far, we have implemented three different `vector` kernels and each implementation comes with three variants: fast, safe that provides iterator validity, and extra safe that also provides the strong form of exception safety. We wanted to avoid the situation where these nine variants would require nine times as much code as a single complete implementation. We have succeeded in this.

There are different ways of organizing template source code. We try to provide a declaration of a component in a separate header file (`.h++` files) and a definition of the member functions in another implementation file (`.i++` files) if we expect that the component will be used by external users. In components that are small or are only meant for internal use, the member functions are implemented inline, and no separate implementation file is provided. When interpreting the results of LOC calculations, the code duplication due to separate declarations can be problematic. Since the declarations could be generated automatically, we ignore the overhead caused by them.

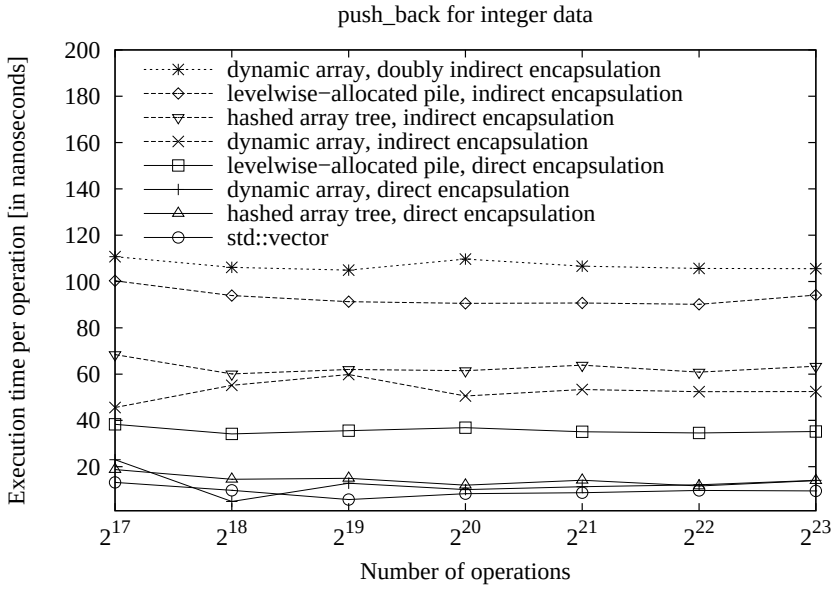


Figure 8. Experiment with `push_back`.

All source code related to the existing implementations is published in an electronic appendix associated with this paper [23]. Table 2 summarizes the (logical) LOC used by each file.

By looking at these numbers and the actual code, we can still identify some code duplication; the three encapsulator classes and the three partial specializations of the factory class for each type of encapsulator are very similar. Probably some additional language support would be needed to be able to handle encapsulators in a cleaner way. (For Smalltalk, an extension of the run-time system has been proposed for this purpose [28].) One can see that the kernels are relatively small. Each kernel has to provide nine member functions and normally we use about 100 LOC, or less, for the implementation. It is the kernel that crystallizes the essence of a data structure. We expect to see these kernels in textbooks on algorithms and data structures.

As we wrote in the introduction, a complete implementation of `vector` described in [29] took 365 LOC. In their implementation, iterators were realized as pointers to elements so no separate classes were needed for them. Also, no separate declarations for any of the classes were provided. In our case, a dynamic-array kernel with direct encapsulation would correspond to their implementation. Hence, if we ignore the declarations, we use 249 (`stl-vector.i++`) + 137 (`vector-framework.i++`) + 12 (`surrogate.h++`) + 20 (`direct-encapsulator.h++`) + 95 (`reference-proxy.h++`) + 121 (`rank-iterator.h++`) + 33 (`factory.h++`) + 67 (`dynamic-array.h++`) + 22 (`slot-swap.i++`) + 25 (`uninitialized-copy.i++`) = 781 LOC to obtain

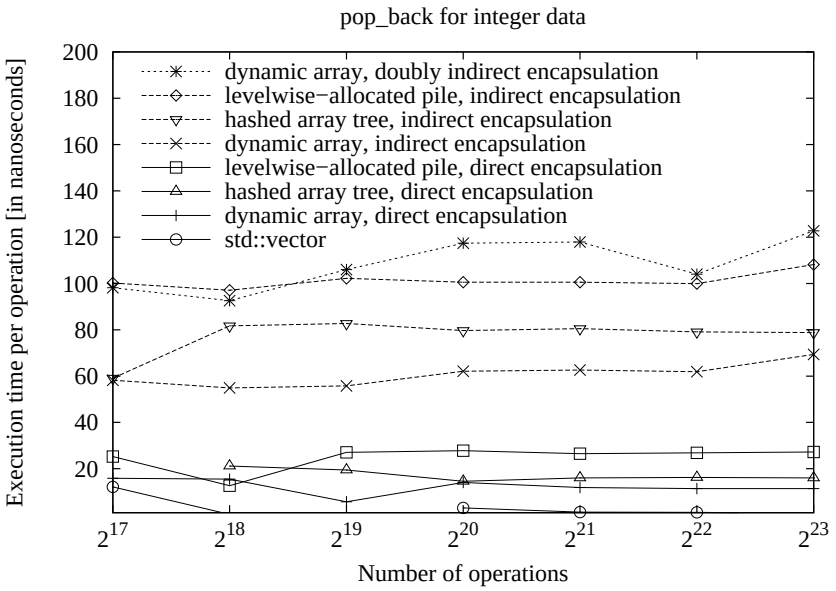


Figure 9. Experiment with `pop_back`.

Table 2. LOC counts for our files.

File	LOC
`stl-vector.h++`	102
`stl-vector.i++`	249
`vector-framework.h++`	62
`vector-framework.i++`	137
`surrogate.h++`	12
`direct-encapsulator.h++`	20
`indirect-encapsulator.h++`	39
`doubly-indirect-encapsulator.h++`	72
`reference-proxy.h++`	95
`rank-iterator.h++`	73
`rank-iterator.i++`	121
`proxy-iterator.h++`	65
`proxy-iterator.i++`	140
`factory.h++`	33
`dynamic-array.h++`	67
`hashed-array-tree.h++`	101
`levelwise-allocated-pile.h++`	59
`slot-swap.i++`	22
`uninitialized-copy.i++`	25

about the same functionality. Because of generality, we have more than doubled the amount of code needed. We will leave it for the reader to decide

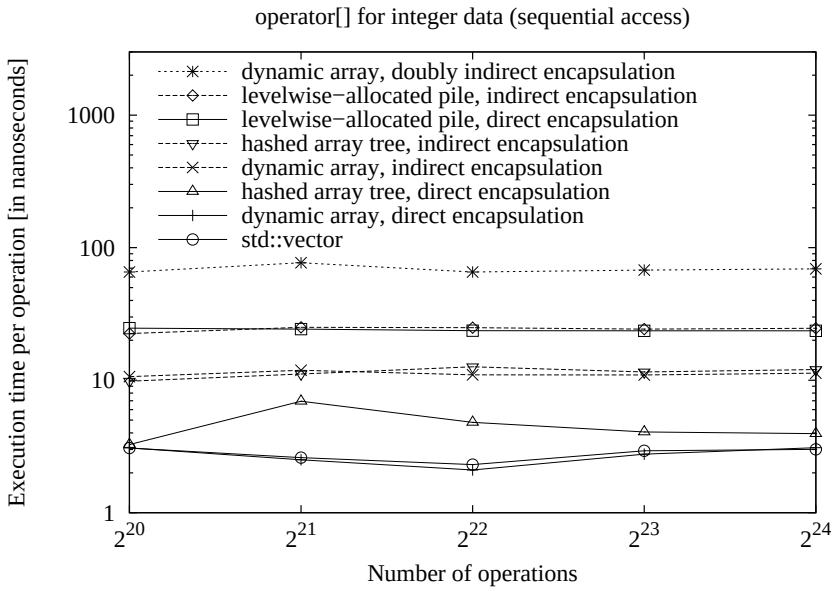


Figure 10. Experiment with `operator[]`. Each element is visited once in sequential order.

whether it is worth paying this price in the increase on the complexity and the amount of code. The increased complexity is in particular apparent in code that is common for both the safe and unsafe components. The common pieces must be carefully crafted to be sure that the safety of the safe implementations is not lost.

7. Adaptivity

In this section we describe in which ways our current implementation of the component framework for `vector` could be made adaptive. For benchmarking purposes, in the actual realizations we still have full control over the instantiation of template parameters. We also give a list of the language facilities in C++ that could be improved to make the development of active libraries easier.

7.1 Overriding default implementations

A naive implementation of `insert` moves the elements between the given position and the end of the `vector` forward and copies the given element(s) into the hole created. According to the contract made between the framework and each kernel, the framework is responsible for `insert`. However, sometimes the framework does not have enough information to do the movement of elements efficiently. For example, our benchmarks showed that `insert` was unnecessarily slow for levelwise-allocated piles. To recover from this

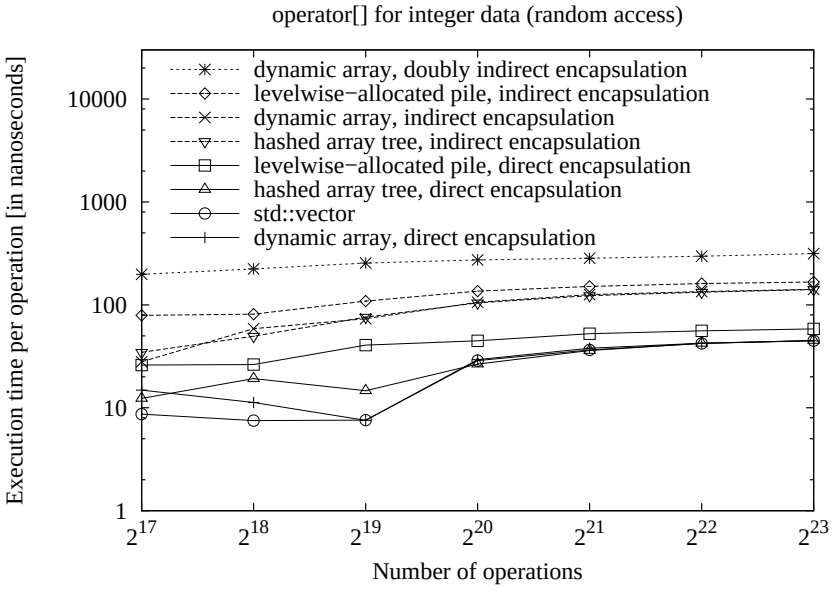


Figure 11. Experiment with `operator[]`. Each element is visited once, and these visits are done in random order.

inefficiency, we can let the kernel implement `insert` as well. After this the framework can invoke the function provided by the kernel. This leads to a general optimization strategy that resembles member-function overriding achieved via inheritance.

Optimization 1. If a policy provides an implementation of a member function, for which a framework provides a default implementation, override the default implementation by invoking the function in the policy.

Our prototype implementation of this optimization relies on the substitution-failure-is-not-an-error principle [40, Section 8.3]. We wrote a macro `HAS_SINGLE_ELEMENT_INSERT` that tests whether the kernel has an `insert` member function that takes an index and a reference to an immutable element as parameters and returns nothing. This macro is then used as a compile-time function that returns a Boolean value. In the framework the actual implementation of `insert` invokes a private member function that comes in two versions, one that invokes the member function in the kernel and another that provides the default implementation. The selection of the correct version of that private member function is done by converting the Boolean value returned by the macro to a type and by relying on function overloading. The programming technique used here is called tag dispatching, and it has been used in many places in earlier implementations of the STL.

A more elegant implementation could be obtained by relying on concept-based overloading. First, a concept `HasSingleElementInsert` is defined to

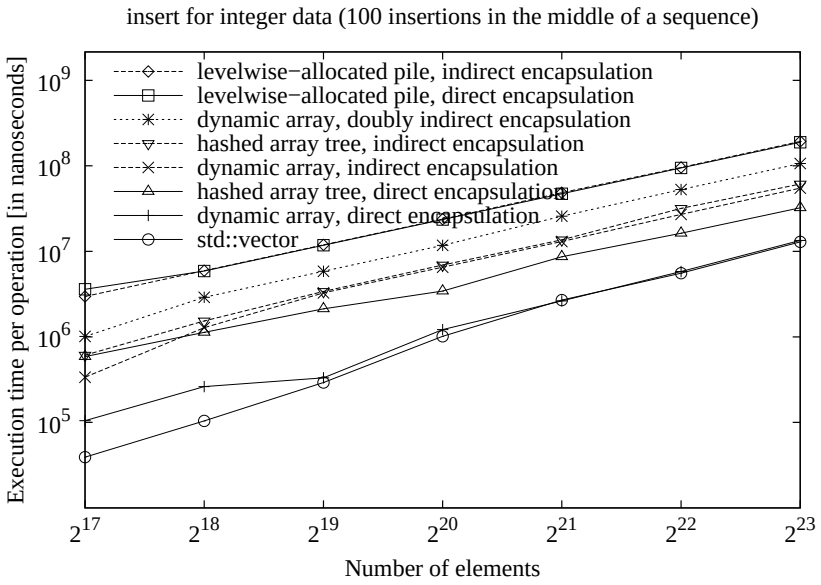


Figure 12. Experiment with `insert`. Repeatedly insert new elements in the middle of the sequence.

specify that the given type must have a member function with the signature `void insert(size_type, value_type const&)`. Second, this concept is used to define two overloaded versions of `insert` in the framework. The first version requires that the kernel, which is one of the template parameters, fulfils the requirement specified by the concept and the second version requires that the kernel does not fulfil this requirement. As above, the first version employs the member function in the kernel and the second version provides the default implementation. Since we did not have a compiler available that could handle concepts, we were not able to try this approach in practice.

We hope that the reader can recognize the significance of this idea: it leads to extremely flexible interfaces and makes the development of efficient component frameworks easier. Possibly even direct language support should be provided for this facility.

7.2 Selecting the fastest copy algorithm

In our `vector` framework, copying of elements from one memory segment to another is an often-recurring operation. To speed up copying, a standard optimization described, for example, in [25] is to utilize an efficient bitwise copying method if such copying will have a correct outcome. This is true, for example, for all plain-old-data (POD) types.

Optimization 2. If both in the source and the target the elements are stored in a contiguous memory segment, if the elements are POD types, and

if the sizes of the elements in both arrays are the same, copy the elements using the fast `memcpy` function, which is available at the standard C library.

One way of implementing this optimization is to use the type traits available at the standard library together with tag dispatching. However, according to the technical report on C++ library extensions [18], it is unspecified under what circumstances `std::tr1::is_pod<V>::value` is true. Hence, it is unspecified when the optimization is in use, if it is in use at all. Clearly, under these premises it is difficult to build a portable active library. In general, the facilities for compile-time reflection, i.e. the ability of a program to inspect its own high-level properties at compile time, could be improved in C++.

7.3 Selecting the best-suited encapsulation policy

We observed that for `vector` implementations based on direct encapsulation are slow when elements being manipulated are expensive to copy. This inefficiency is due to relocations of elements, involving element constructions and destructions. A faster behaviour can be obtained by letting the array store pointers to elements.

Optimization 3. If indirect encapsulation is more profitable than direct encapsulation, store elements indirectly; otherwise store them directly.

To implement this kind of optimization, we would need a compile-time operator `costof` that evaluates the cost of a given expression at compile time. The idea that a compiler does profiling during compilation is interesting. Since there is no operator `costof` available, we are only able to approximate this optimization. For example, `sizeof` can provide a good estimation whether a copy of an element will be more expensive than a copy of a pointer, but this is not necessarily the case. For example, think of a socket that is a small object but it can be costly to copy. Also, the expression for `costof` should be chosen carefully to take into account the cost of indirection and the cost of cache misses. We admit that profiling can slow down compilation too much so it might be wiser to rely on an external configuration tool.

As to the selection of a suitable encapsulation policy, a similar situation appears when instantiating a kernel that guarantees strong exception safety and referential integrity. Depending on whether the copy constructor for the elements can throw an exception or not, the simplest possible encapsulation policy can be selected without loosing the strong form of exception safety.

Optimization 4. If the copy constructor for the elements cannot throw an exception, store elements indirectly; otherwise store them doubly indirectly.

To implement this optimization, the `has_nothrow_copy` type trait from the standard library could be used. However, again the technical report on C++ library extensions [18] does not specify under what circumstances, if any, `std::tr1::has_nothrow_copy<V>::value` evaluates to true.

8. Adaptability

.

For years, the CPH STL has been an interesting teaching tool when educating software developers at our university. We have been convinced that the library might also be used at other universities for teaching purposes. However, up to now this has not happened. After introducing component frameworks into the library we expect that the deployment at other sites will actually happen.

The development of component frameworks is demanding. First an attempt of trying to extend an existing component framework with new features reveals the weaknesses of earlier design decisions. To understand a complete component framework and to extend it requires good developer skills. We claim that the CPH STL is a good platform for training these skills.

The development of component frameworks, and generic programming in general, requires extreme discipline. Even if the user or the developer of a component framework makes a trivial mistake, the error message produced by the compiler can be extensive. This is simply because the types involved are so complicated; the description of a type based on a component framework with all the instantiated policies can easily fill a small computer screen. The developer community has hoped that C++ concepts (see, for example, [15, 19]) could solve the problem with poor error messages, but we doubt that. We question whether it is a good idea to encode complicated adaptations into types. Even though adaptability of component frameworks is a nice feature, with current tools the development of frameworks is tedious.

The components of the CPH STL are extensible. We have already now a collection of programming exercises for our students. You could test your developer skills by solving any of the following exercises.

Exercise 1. Implement a new `vector` kernel for the CPH STL. Highly relevant candidates to consider include tiered vectors described in [16] and blockwise-allocated piles described in [21].

Exercise 2. In our current implementation of a levelwise-allocated pile the directory is a fixed-sized array. To make the data structure fully dynamic and to provide the best possible worst-case performance bounds, we would need a `vector` implementation that realizes `push_back` and `pop_back` at $O(1)$ worst-case cost. Develop a `vector` kernel that gives these performance guarantees.

Exercise 3. Extend the framework such that the user can specify both the encapsulation policy (direct, indirect, and doubly indirect encapsulation) and the ownership policy (client owns, container owns, and realizator owns) for the elements stored in a `vector`.

Exercise 4. Components obtained by instantiating component frameworks are often built on several layers of abstraction. This would make the work of compilers harder, and sometimes performance penalties are introduced. Investigate the assembler code produced by your compiler to see what are

the causes for the performance penalties in our `vector` implementations. Can you tell your compiler vendor how these could be avoided? Can you tell us how we could have avoided them?

The CPH STL is like any other software; it will never become complete. By releasing these extensible component frameworks, we do not even aim at producing a complete—ultimate—release of the library. The whole point is to use the library in education, and let coming software developers extend the library. It is a fascinating idea that the users will continue the design and development of the library by extending frameworks and writing new components.

9. Conclusions

We conclude the paper with brief messages to different stakeholders in the software-library community.

Users of generic software libraries: The CPH STL provides fast, safe, and compact variants for many of the existing standard-library containers. Similar facilities could be provided, and are already provided, by other container libraries. Hopefully, we have made it clear that safety comes with a price tag. However, in applications, where safety has a higher priority than performance, it is natural to use the safe variants. This way one can avoid many hard-to-find bugs. The safe components may be particularly useful for educational purposes.

Designers of programming-language facilities: An important proclamation made in this paper was that in C++ the facilities provided for compile-time reflection and metaprogramming are far too primitive to be of practical value. Also, a stronger support for writing generic encapsulators would be desirable. We hope that better programming-language support for generic programming will be available in the near future.

Developers of generic software libraries: When developing the `vector` framework, we encountered a problem which we had not thought of before and for which we could not provide any general solution: How to avoid or detect gracefully a mismatch between the template parameters given? It would be easy to check that a given class `K` conceptually fulfils all kernel requirements and another given class `E` all encapsulator requirements, but what if `E` was not designed to work with `K` at all. The poor user will waste his or her valuable development time to find out this sad fact. We leave the problem of designing mismatch-free component libraries as a challenge for other library developers.

Teachers of software developers: We have used the CPH STL in the exercises (weekly assignments and mini-projects) of our courses (generic programming and software construction) to teach both design and programming. The student feedback from these courses has been overly

positive. Students have found the assignments interesting and challenging. But, yes, we have also received complaints about a heavy workload. Our recommendation is that projects are not made longer than three weeks before the students have enough practical experience in generic programming. Due to the lack of adequate (open-source) tools, weak students would waste their time if the project periods were longer. But still, as put by one of our students, it is better to over-challenge than to simplify the assignments.

Software availability

The programs discussed in this study are available via the home page of the CPH STL project [13] in the form of a PDF document [23] and a `tar` file.

Acknowledgements

We thank the following people who have been directly involved in the development of `vector` in the CPH STL project; much of our work is built on their work: Tina A. G. Andersen, Filip Bruman, Marc Framvig-Antonsen, Ulrik Schou Jørgensen, Mads D. Kristensen, Wojciech Sikora-Kobylinski, Daniel P. Larsen, Bjarke Buur Mortensen, Jan Presz, Jens Peter Svensson, Mikkel Thomsen, Ole Hyldahl Tolshave, Claus Ullerlund, Bue Vedel-Larsen, and Christian Wolfgang. Also, we thank the anonymous referees for their insightful comments that sharpened our understanding of subject matter.

References

[1] David Abrahams. Exception-safety in generic components: Lessons learned from specifying exception-safety for the C++ standard library. In *Selected Papers from the International Seminar on Generic Programming*, Lecture Notes in Computer Science **1766**. Springer-Verlag, 2000, 69–79.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] Algorithmic Solutions. *The LEDA User Manual*, Version 6.2. Web document available at `http://www.algorithmic-solutions.info/leda_manual`, 2008.

[4] Matt Austern. Defining iterators and const iterators. *C/C++ User's Journal* **19**(1), 2001, 74–79.

[5] Matthew H. Austern, Bjarne Stroustrup, Mikkel Thorup, and John Wilkinson. Untangling the balancing and searching of balanced binary search trees. *Software—Practice and Experience* **33**(13), 2003, 1273-1298.

[6] Phil Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. LAMP Report **2002-003**. School of Computer and Communication Sciences, Swiss Federal Institute of Technology Lausanne, 2002.

[7] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. *SIGPLAN Notices* **24**(10), 1989, 1–6.

[8] John Boyer. Algorithm alley: Resizable data structures. *Dr. Dobb's Journal* **23**(1), 1998, 115–116, 118, 129.

[9] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition. John Wiley and Sons, Ltd., 2003.

[10] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **1663**. Springer-Verlag, 1999, 37–48

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press, 2001.

[12] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, Lecture Notes in Computer Science **1766**. Springer-Verlag, 2000, 25–39.

[13] Department of Computer Science, University of Copenhagen. The CPH STL. Website accessible at `http://cphstl.dk`, 2000-2009.

[14] Erich Gamma, Richard Helm, Ralph Johnson, John and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. *SIGPLAN Notices* **41**(10), 2006, 291–310.

[16] Michael T. Goodrich and John G. Kloss II. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **1663**. Springer-Verlag, 1999, 205–216.

[17] Austin Henderson and Morten Kyng. There's no place like home: Continuing design in use. In *Design at Work: Cooperative Design of Computer Systems.* Lawrence Erlbaum Associates, 1991, 219–240.

[18] ISO/IEC. *Draft Technical Report on C++ Library Extensions*. Document Number **N1836**. The C++ Standards Committee, 2005.

[19] ISO/IEC. *Working Draft: Standard for Programming Language C++*. Document Number **N2857**. The C++ Standards Committee, 2009.

[20] Jyrki Katajainen. Making operations on standard-library containers strongly exception safe. In *Proceedings of the 3rd DIKU-IST Joint Workshop on Foundations of Software*. Report **07/07**. Department of Computer Science, University of Copenhagen, 2007, 158–169.

[21] Jyrki Katajainen and Bjarke Buur Mortensen. Experiences with the design and implementation of space-efficient deques. In *Proceedings of the 5th Workshop on Algorithm Engineering*, Lecture Notes in Computer Science **2141**. Springer-Verlag, 2001, 39–50.

[22] Jyrki Katajainen and Bo Simonsen. The design and description of a generic software library. Work in progress, 2009.

[23] Jyrki Katajainen and Bo Simonsen. Vector framework: Electronic appendix. CPH STL Report **2009-4**. Department of Computer Science, University of Copenhagen, 2009.

[24] Mads D. Kristensen. Vector implementation for the CPH STL. CPH STL Report **2004-2**. Department of Computer Science, University of Copenhagen, 2004.

[25] John Maddock and Steve Cleary. C++ type traits. *Dr. Dobb's Journal* **25**(10), 2000, 38–44.

[26] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[27] Anders Mørch. Three levels of end-user tailoring: Customization, integration, and extension. In *Computers and Design in Context*. The MIT Press, 1997, 51–76.

[28] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. *SIGPLAN Notices* **21**(11), 1986, 341–346.

[29] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice Hall PTR, 2001.

[30] R. Oppermann and H. Simm. Adaptability: User-initiated individualization. In *Adaptive User Support—Ergonomic Design of Manually and Automatically Adaptable Software*. Lawrence Erlbaum Associates, 1994, 14–66.

[31] Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*. ACM-SIAM, 2008, 37–48.

[32] Rajeev Raman and S. Srinivasa Rao. Succinct dynamic dictionaries and trees. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science **2719**. Springer-Verlag, 2003, 357–368.

[33] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science **2125**. Springer-Verlag, 2001, 426–437.

[34] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 2002.

[35] Silicon Graphics, Inc. Standard template library programmer's guide. Website accessible at `http://www.sgi.com/tech/stl`, 1993–2009.

[36] Bo Simonsen. A framework for implementing associative containers. CPH STL Report **2009-3**. Department of Computer Science, University of Copenhagen, 2009.

[37] Edward Sitarski. Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays *Dr. Dobb's Journal* **21**(11), 1996.

[38] Bjarne Stroustrup. Appendix E: Standard-library exception safety. *The C++ Programming Language*, Special Edition. Addison-Wesley, 2000.

[39] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Pearson Education, Inc., 2009.

[40] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, Inc., 2003.

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.

Table of contents

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 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.

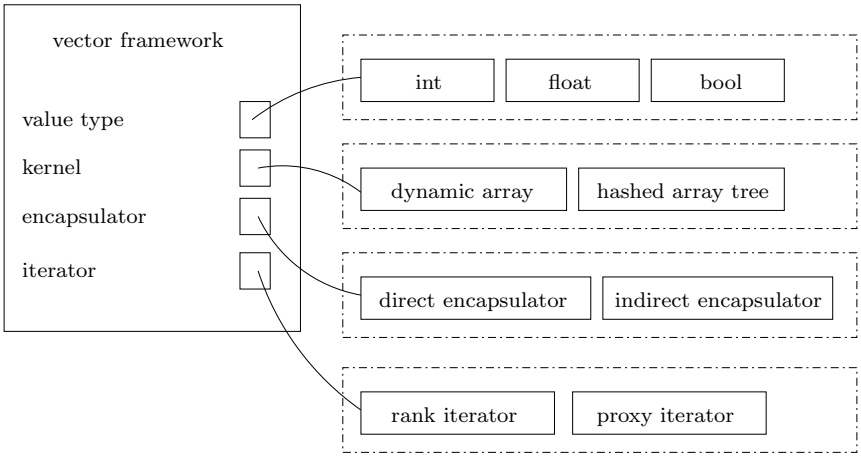


Figure 1. The conceptual view of the `vector` framework.

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

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

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 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

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;
```

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 the 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 to 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.

[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`.

`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 $O(\sqrt{n})$.

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` container 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

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= */
  };
}
```

default arguments, 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.

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

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());
  ...
};
```

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 inherits from the same class for any permutation of 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.

```
1 class base_class;
2
3 class sub_class : public base_class {
4   using base_class::base_class;
5 public:
6   ...
7 };
```

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`.

```
1  #include <iostream>
2
3  class A {
4  public:
5    void test() {
6      std::cout << "A" << std::endl;
7    }
8  };
9
10 class B : public A {
11 public:
12   void test() {
13     std::cout << "B" << std::endl;
14   }
15 };
16
17 int main() {
18   A* x = new B();
19   x->test();
20 }
```

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 C++ libraries [5] provide a so-called pool allocator (`boost::pool_allocator`). A pool allocator [3] maintains an object pool (or a free-list) 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 (elem-

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

```
1 namespace cphstl {
2   template <typename V, typename A>
3   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> >;
4 }
```

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

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

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:

```
1 template <typename V>
2 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 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: $\quad\mathcal{A}_a \leftarrow \{\langle p', a'\rangle \mid \langle p', a'\rangle \in \mathcal{A}_\mathcal{C} \wedge p' \in \mathcal{P}_a\} \cup \mathcal{A}_a$
3: $\quad$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 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`.

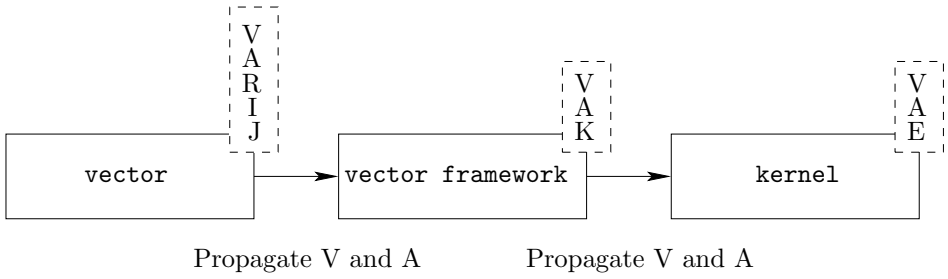


Figure 2. Template argument propagation.

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. 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

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();
  }
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();
}
```

continues in the right column

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 {
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();
}
```

continues in the right column

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:

```
1  template <typename V, typename A = std::allocator<int> >
2  class vector {
3  public:
4    ...
5    template <typename I>
6    void insert(I first, I second);
7  };
8
9  int main() {
10   vector<int> v;
11   int a[] = {1,2,3};
12   v.insert(a, a+3);
13   v.insert<int*>(a, a+3);
14 }
```

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

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;
  ..
}
```

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.

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 in our `vector` component framework 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

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;
}
```

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.

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

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

```
1  class Q {
2  public:
3    template <typename Arg>
4    class real_class {
5    public:
6      ...
7    };
8  };
9
10 template <typename Arg>
11 class P {
12 public:
13   ...
14   typename Arg::template real_class< P< Arg > > y;
15 };
16
17 int main() {
18   P<Q> p;
19 }
```

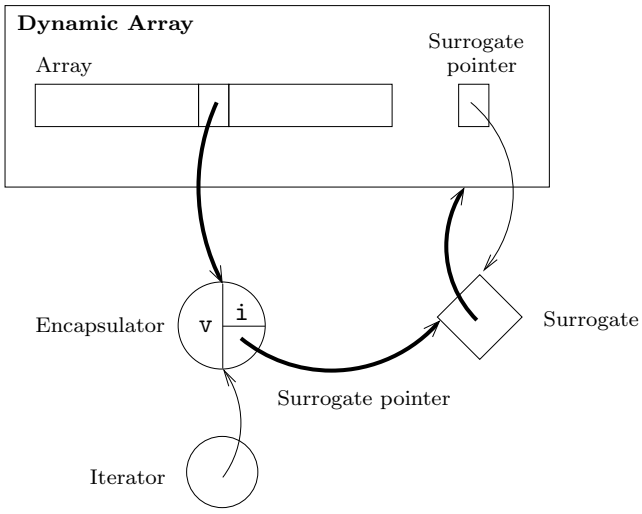


Figure 3. The initial construction of a dynamic array providing referential integrity.

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 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

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'
```

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 `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 prob-

lems 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:

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

```
1 typedef cphstl::set!<V=int,
2                       R=cphstl::tree!<
3                         N=cphstl::avl_tree_node!<packed=true!>,
4                         B=cphstl::avl_tree_balancer
5                       !>,
6                       J=cphstl::node_iterator!<is_const=true!>,
7                       I=cphstl::node_iterator!<is_const=false!>
8                     !> C;
```

- 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 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

⟨nta_statement ⟩ → **id** `!<` ⟨nta_list ⟩ `!>`

⟨nta_list ⟩ → ⟨nta_list′ ⟩ ⟨nta_entry ⟩

⟨nta_list′ ⟩ → ⟨nta_list ⟩
| ϵ

⟨argument_list ⟩ → ⟨argument_list′ ⟩ **id**

⟨argument_list′ ⟩ → ⟨argument_list ⟩
| ϵ

⟨nta_entry ⟩ → **id** = ⟨nta_statement ⟩
| **id** = **id** `<` ⟨argument_list ⟩ `>`
| **id** = **id**

Figure 4. BNF grammar for the language extension.

`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, 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.

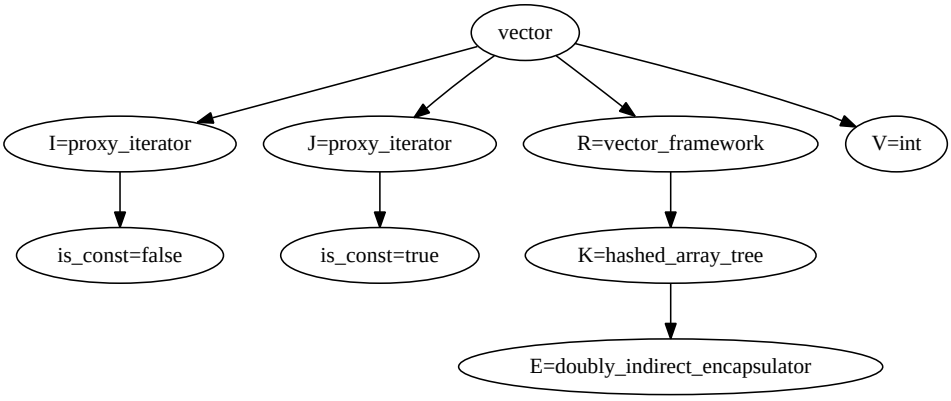


Figure 5. The abstract syntax tree for a simple `vector` declaration.

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]])
```

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:

- `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 ⟨ nta_entry ⟩ 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 nonexisting 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:

```
1 cphstl::vector!<V=int,
2                 R=cphstl::vector_framework!<
3                   K=cphstl::hashed_array_tree!<
4                     E=cphstl::doubly_indirect_encapsulator
5                   !>
6                 !>,
```

```
7                 I=cphstl::proxy_iterator!<is_const=false!>,
8                 J=cphstl::proxy_iterator!<is_const=true!>
9               !> 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`.

```
1  cphstl::map!<K=char,
2               V=int,
3               A=std::allocator<std::pair<char, int> >,
4               R=cphstl::tree!<
5                 V=std::pair<char, int>,
6                 F=cphstl::unnamed::key_extractor,
7                 N=cphstl::aa_tree_node,
8                 B=cphstl::aa_tree_balancer
9               !>,
10              I=cphstl::node_iterator!<is_const=false!>,
11              J=cphstl::node_iterator!<is_const=true!>
12            !> mc;
```

Notice that `V` is overwritten, since the type of the value stored in the tree is the pair of $\langle \mathtt{K}, \mathtt{V} \rangle$. The keys are retrieved using the class template `key_extractor` which represents the function $F : \langle \mathtt{K}, \mathtt{V} \rangle \rightarrow \mathtt{K}$. 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 frame-

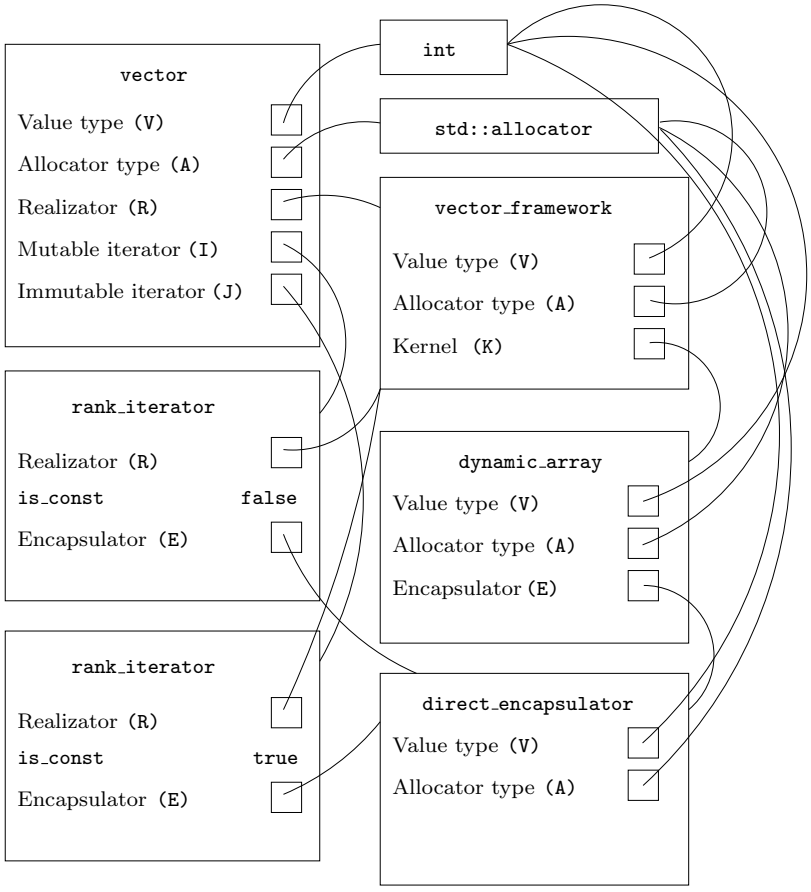


Figure 6. A configuration graph.

work 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.

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 *is 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:

```
1 #include <iostream>
2
3 class BaseClass {
4 public:
5   virtual void a_member() = 0;
6
7 };
8
9 class SubClass : public BaseClass {
10 public:
11   void another_member() {
12     std::cout << "Test" << std::endl;
13   }
14 };
15
```

```
16 int main() {
17   SubClass s;
18 }
```

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):

```
1 test-virtual.c++: In function 'int main()':
2 test-virtual.c++:17: error: cannot declare variable 's' to be of
    abstract type 'SubClass'
3 test-virtual.c++:9: note:   because the following virtual functions
    are pure within 'SubClass':
4 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`.

```
1 auto concept LessThanComparable<typename T> {
2   bool operator<(T, T);
3 }
4
5 template <LessThanComparable T>
6 void min(T const& v1, T const& v2) {
```

```
7   if(v1 < v2) {
8     return v1;
9   }
10  return v2;
11 }
```

□

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:* $\Gamma := \{\langle v_s, v_e, p\rangle \mid \langle v_s, v_e, p\rangle \in \mathcal{E} \wedge \Phi(v_s, v_e, p) = \textbf{true}\}$. *$\Phi$ 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:

```
1 template <RandomAccessIterator I>
2 void sort(I first, I last);
3 template <RandomAccessIterator I>
```

```
4 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:

```
1  template <typename C, typename F>
2  class X {
3  public:
4    bool member(C const& c, C::value_type* es, std::size_t
         number_of_es) {
5      F f;
6      f(c.begin(), c.end());
7      /* check post condition of the call to f */
8      assert(std::is_sorted(c.begin(), c.end());
9      for(int i=0; i < number_of_es; ++i) {
10       if(!std::binary_search(c.begin(), c.end(), es[i])) {
11         return false;
12       }
13     }
14     return true;
15   }
16 };
```

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.

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`

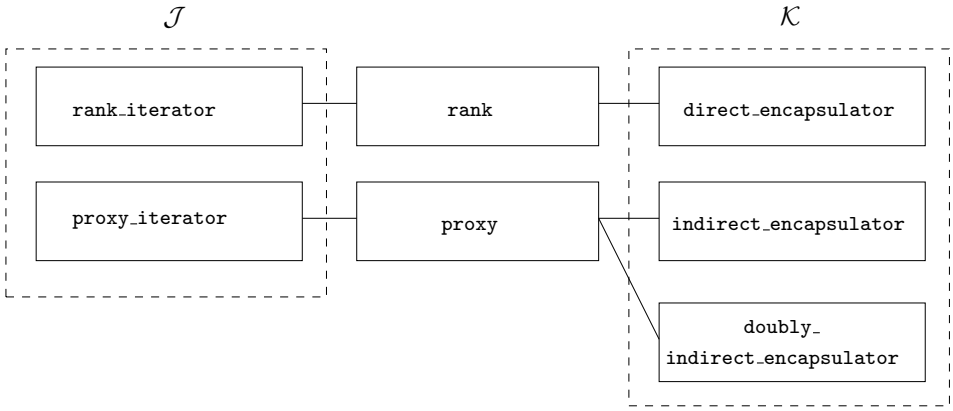


Figure 7. A family graph.

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:* $\Gamma_{\mathtt{f}} := \{\langle v_s, v_e, p\rangle \mid \langle v_s, v_e, p\rangle \in \mathcal{E} \wedge v_s \in \mathcal{J} \wedge v_e \in \mathcal{K} \wedge \Phi_{\mathtt{f}}(\langle v_s, v_e\rangle) = \textbf{true}\}$. *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 $\mathcal{K}$.
ADD$_{\mathcal{J}}$(c, p): Adds a class c to the set $\mathcal{J}$ and stores template parameter p.
CONNECT$_{\mathcal{J}}$(j, f): Connects $j \in \mathcal{J}$ to f.
CONNECT$_{\mathcal{K}}$(f, k): Connects f to $k \in \mathcal{K}$.
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 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 ADD$_{\mathcal{K}}$(c) and 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 ADD$_{\mathcal{J}}$(c, p), CONNECT$_{\mathcal{J}}$(j, f), and 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

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
```

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.

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 $\Gamma_{\texttt{f}}$ for a family `f`. Let `F` denote the set of families involved in a configuration. The question is how can we compute $\Gamma := \bigcup_{\texttt{f}\in\texttt{F}} \Gamma_{\texttt{f}}$? 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.

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, Master's 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).