Generative container implementation in C++

Andreas P. Priesnitz
Institute for Numerical and Applied Mathematics
University of Göttingen
priesnit@math.uni-goettingen.de
http://www.num.math.uni-goettingen.de/priesnit/

Keywords: Generative Programming, Generic Programming, STL, Container
Classification: 1 year's work in 3rd year of Ph.D. studies

Abstract. The STL offers a broad selection of efficient container types. But their applicability for, e.g., graph type definitions suffers from several shortcomings. Extensions to the container interface are suggested to introduce appropriate solutions. Finally, a sketch of a mixin layer hierarchy is presented that facilitates generative creation of container types which offer the new features.

1 Introduction

Generic programming facilitates the implementation of a combinatorially growing number of types at linear programming effort. Additionally, it provides static type information that allows an optimizing compiler to generate code comparable to that of a hand–crafted version.

These advantages allow to develop widely applicable and highly efficient generic software like the STL presented in [SL95], which is in the first place a library of powerful and efficient container types.

But the implementation of a graph by an adjacency list (as in the high performant generic graph library BGL, former GGCL, see [Lee99]) uncovers several insufficiencies within the STL:

- The common interface of the different containers is too narrow, basically limited to size information and the creation of iterator objects.
  To maintain a persistent reference to some element (as necessary, e.g., in an adjacency list), container–dependent workarounds are required (using either an integral number or a pointer or some `key_type`), as iterators may become invalid by modifications to the container.

- In general, the structure of a container cannot be replicated. To associate additional data with the elements stored in a `list`, one is either forced to define the container a–priori to hold element/data–pairs, or to use hash

tables if the lifetime of the data is shorter than that of the container (like for any temporary data created and used by some algorithm). An alternative (yet unsafe and inefficient) solution to the latter case is to define the container to store a `void*` with each element, which may be set to point to temporary data associated with this element.

- Generic programming is frequently restricted to the decoupling of algorithm and data structure definitions by the introduction of a common interface. This leads — e.g., in the popular STL versions — to "monolithic" implementations of data structures, usually forcing the programmer to rewrite the complete implementation for a new data structure.

- While the type of a container's elements can be specified as a template parameter, most other properties are invariantly chosen by the mere choice of the container implementation. This complicates the automatic selection of a container type according to given requirements. Furthermore, various design details are left to the implementation.

In the following, a design of efficient container types is presented which provides solutions to these drawbacks and whose scope is (almost) a superset of that of the STL `vector` and `list`.

2 Specifying a position in a container

Even if the restriction of persistency were imposed, the iterator concept is too "fat" to be the optimal choice for serving as a permanent reference to a container element. An iterator holds positional information and knowledge of how to step to the next (or previous) position and how to retrieve data stored at that position. Only the first of these features is relevant to describe, e.g., the point of insertion of new elements into a container.

Thus, the container interface will be enriched by the simpler concept of a `Key` to denote the position of some element in the container. A `Key` object is guaranteed to remain valid if changes are applied to the container (unless the referred element is erased). STL associative containers offer such a `key_type`.

While `Key` may be an integral type for a `vector` (if insertion and removal of elements is performed only at its end), for a `list` it has to be a pointer. It is important to note that the type of the list node pointed to does not matter, as this information is only relevant for element access or stepping to the next node. For the latter functionalities, a cast can be applied by some other object to perform the action according to a given `Key`. As a consequence, the `Key` type is independent of the `list` element type, which offers a clean solution to the problem of defining a `list` holding references to its own elements (coming up, e.g., in graph implementations), which can not be realized satisfactorily using the STL.

All container functions that require positional information will take a `Key` instead of an iterator as parameter. Furthermore, all containers (even a `list`)

will offer an access function taking a `Key` object and returning a (C++) reference to the according element. It is suggested to use `operator()` instead of `operator[]` for this function. First reason: this writing comes closer to the view of a container being a map from a domain of `Key` objects to a range of elements (generally speaking, a functor whose result type is a reference). Second: This way an additional `operator[]` taking integral arguments for random–access containers may exist, but confusion is avoided.

There should also be a `KeyIterator` traversing the set of `Key` objects in the same order as a conventional iterator does for the according elements.

3 Sharing structure among containers

The next challenge is to provide dynamic *sharing* of structure information among various containers. Given such a sharable container, it shall be possible to define *dependent* containers, in the sense that changes to the structure of the former (like insertions and removals) are automatically applied to the latter ones (which will not permit any individual modifications themselves).

Elements of a dependent container can therefore be regarded as being permanently pairwise associated to the elements of the original container by the common order of traversal.

To implement this functionality, a sharable container (if modifiable) needs to maintain a list of references to its dependants. As these may have different types (according to the type of the elements they store), they inherit from a common base class with `virtual` declarations for the modifying functions. When a shared container performs a modification, it will internally apply the same modification by a virtual function call to the dependants — basically. A more detailed examination discovers several possibilities to vary behaviour and efficiency. Anyhow, it must be pointed out that the use of virtual function calls is inevitably imposed by the request for dynamic data association.

To prohibit the direct invocation of modifying operations on a dependent container, they are declared `private`.

To provide simultaneous access to associated elements, a dependent container defines a function that maps an object of the sharable container's `Key` type to the object of its own `Key` type which addresses the associated element.

The mapping will rely on relative addressing. While this is no problem for random–access containers, a `list`–like container has to store the list nodes in an internal `vector`–like structure (similar to using a `vector` as an *allocator* for a `list`) and to reflect the internal container's `Key` type.

The duality of relative and absolute addressing may be respected for the connectivity representation. Either the pointer–to–node or `Key` type can model a node's links to the next/previous node — though the latter hardly seems to be beneficial in the common case.

Note that the links have to be stored, essentially, only in the shared container, from where dependent containers may retrieve this information instead of storing it individually. Therefore, if speed is not the primary issue, this rep-

resentation is space–optimal. Otherwise, connectivity information may be held in the dependants as well.

4 Layered container implementation

The generative approach to the design of a type family is to name a set of (functionally) orthogonal features and to implement them by individual layers, using the *mixin* technique of parameterizing a class by its base class (see [SB98]). The conditional composition of the layers is accomplished through *template metaprogramming* techniques (see [Vel95],[CE00]).

The set of a type's significant orthogonal features is not necessarily reflected in its interface, as some properties combine several more elementary ones. Therefore, the design hierarchy will consist of base layers which implement the core functionality (usually in form of `protected` features) and of application layers on top of them which establish the `public` interface.

To implement container types with the above mentioned features, base layers that model a random–access container are appropriate:

1. *Memory* covers the static or dynamic allocation of a consecutive area of memory by an array or an allocator. The effective type of a memory unit (*node*) is propagated upward from the final layer (see [CE00]). A (constant) pointer to the first node of the buffer is made available. For dynamic allolcation, reallocation may be allowed.

2. *Addressing* defines the `Key` type to be a pointer or integral type. Absolute addressing is valid only if no reallocation may be performed. The `Key` of the first node and the access operator are introduced, as well as an `Accessor` type that models the functionality of data retrieval and serves as a common base class for the later definition of efficient iterators.

3. *Capacity* implements the container's size, the `Key` of and the pointer to the node "behind" the last node. This information is equivalent, hence any of the features may be stored explicitly and serve to derive the other two. Therefore, this layer consists of three sub–layers, each implementing one feature, alternatively storing or deriving it.

4. *Layout* introduces the traversal functionality through *stepper* types that model the knowledge on how to set `Key` objects and node pointers to the previous/next position. The container also offers these operations itself.

Independent of the fact that the base layers represent a random–access container, the following optional layers may add further functionalities:

5. *Permutability* allows to change the order of traversal. This is accomplished by refining the nodes to explicitly store links to the predecessor and the successor node, as usual in linked lists. The stepper types are equivalently redefined, using the `Accessor` type.

6. *Modifiability* allows to insert and remove nodes at the end of the container. Virtual capacity information is used to implement a sub–container view whose end position can be increased and decreased. If the capacity of the base container is exhausted, reallocation is performed.
   Given the permutability feature, insertion and removal at any position of the container can be implemented.

Now, various application layers introduce the usual interface in a very straightforward manner. Iterators, e.g., are defined by wrapping a stepper object and a pointer to the actual node.

7. *Sharability* optionally introduces the functionality described above. A generator for the creation of dependent container types takes the dependant's element type as its sole argument. It generates the dependent container type by applying the complete layer hierarchy to a base class which declares modifying functions to be `virtual`. Modifications can then be applied to the dependent containers via pointers of base class type stored in a list. The dependent container type is finally wrapped by a layer that declares all modifying functions to be `private`.

5 Outlook

Using this flexible container implementation, a generative graph implementation is planned that provides efficient association of data to vertices and edges. Common design patterns will have to be identified and extracted.

For optimal performance, sub–container views and slicing need to be added.

The extension of the design to individual node allocation and to associative or other containers seems feasible, though no attempts are planned.

References

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications.* Addison Wesley, Massachusetts, 2000.

[Lee99] Lie-Quan Lee. The high performance generic graph component library. Master's thesis, University of Notre Dame, 1999.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP'98*, pages 550–570, Brussels, Belgium, 1998.

[SL95] Alex Stepanov and Meng Lee. *The Standard Template Library.* Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, February 1995.

[Vel95] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.