16

Library Organization and Containers

It was new. It was singular.
It was simple. It must succeed!
– H. Nelson

Design criteria for the standard library — library organization — standard headers — language support — container design — iterators — based containers — STL containers — ***vector*** — iterators — element access — constructors — modifiers — list operations — size and capacity — ***vector<bool>***— advice — exercises.

16.1 Standard Library Design [org.intro]

What ought to be in the standard C++ library? One ideal is for a programmer to be able to find every interesting, significant, and reasonably general class, function, template, etc., in a library. However, the question here is not, ''What ought to be in *some* library?'' but ''What ought to be in the *standard* library?'' The answer ''Everything!'' is a reasonable first approximation to an answer to the former question but not to the latter. A standard library is something that every implementer must supply so that every programmer can rely on it.

The C++ standard library:

[1] Provides support for language features, such as memory management (§6.2.6) and run-time type information (§15.4).

[2] Supplies information about implementation-defined aspects of the language, such as the largest ***float*** value (§22.2).

[3] Supplies functions that cannot be implemented optimally in the language itself for every system, such as ***sqrt***( ) (§22.3) and ***memmove***( ) (§19.4.6).

[4] Supplies nonprimitive facilities that a programmer can rely on for portability, such as lists (§17.2.2), maps (§17.4.1), sort functions (§18.7.1), and I/O streams (Chapter 21).

[5] Provides a framework for extending the facilities it provides, such as conventions and

support facilities that allow a user to provide I/O of a user-defined type in the style of I/O for built-in types.

[6] Provides the common foundation for other libraries.

In addition, a few facilities – such as random-number generators (§22.7) – are provided by the standard library simply because it is conventional and useful to do so.

The design of the library is primarily determined by the last three roles. These roles are closely related. For example, portability is commonly an important design criterion for a specialized library, and common container types such as lists and maps are essential for convenient communication between separately developed libraries.

The last role is especially important from a design perspective because it helps limit the scope of the standard library and places constraints on its facilities. For example, string and list facilities are provided in the standard library. If they were not, separately developed libraries could communicate only by using built-in types. However, pattern matching and graphics facilities are not provided. Such facilities are obviously widely useful, but they are rarely directly involved in communication between separately developed libraries.

Unless a facility is somehow needed to support these roles, it can be left to some library outside the standard. For good and bad, leaving something out of the standard library opens the opportunity for different libraries to offer competing realizations of an idea.

16.1.1 Design Constraints [org.constraints]

The roles of a standard library impose several constraints on its design. The facilities offered by the C++ standard library are designed to be:

[1] Invaluable and affordable to essentially every student and professional programmer, including the builders of other libraries.

[2] Used directly or indirectly by every programmer for everything within the scope of the library.

[3] Efficient enough to provide genuine alternatives to hand-coded functions, classes, and templates in the implementation of further libraries.

[4] Either policy-free or give the user the option to supply policies as arguments.

[5] Primitive in the mathematical sense. That is, a component that serves two weakly related roles will almost certainly suffer overheads compared to individual components designed to perform only a single role.

[6] Convenient, efficient, and reasonably safe for common uses.

[7] Complete at what they do. The standard library may leave major functions to other libraries, but if it takes on a task, it must provide enough functionality so that individual users or implementers need not replace it to get the basic job done.

[8] Blend well with and augment built-in types and operations.

[9] Type safe by default.

[10] Supportive of commonly accepted programming styles.

[11] Extensible to deal with user-defined types in ways similar to the way built-in types and standard-library types are handled.

For example, building the comparison criteria into a sort function is unacceptable because the same data can be sorted according to different criteria. This is why the C standard library *qsort*( ) takes

a comparison function as an argument rather than relying on something fixed, say, the < operator (§7.7). On the other hand, the overhead imposed by a function call for each comparison compromises *qsort*( ) as a building block for further library building. For almost every data type, it is easy to do a comparison without imposing the overhead of a function call.

Is that overhead serious? In most cases, probably not. However, the function call overhead can dominate the execution time for some algorithms and cause users to seek alternatives. The technique of supplying comparison criteria through a template argument described in §13.4 solves that problem. The example illustrates the tension between efficiency and generality. A standard library is not just required to perform its tasks. It must also perform them efficiently enough not to tempt users to supply their own mechanisms. Otherwise, implementers of more advanced features are forced to bypass the standard library in order to remain competitive. This would add a burden to the library developer and seriously complicate the lives of users wanting to stay platform-independent or to use several separately developed libraries.

The requirements of ''primitiveness'' and ''convenience of common uses'' appear to conflict. The former requirement precludes exclusively optimizing the standard library for common cases. However, components serving common, but nonprimitive, needs can be included in the standard library in addition to the primitive facilities, rather than as replacements. The cult of orthogonality must not prevent us from making life convenient for the novice and the casual user. Nor should it cause us to leave the default behavior of a component obscure or dangerous.

16.1.2 Standard Library Organization [org.org]

The facilities of the standard library are defined in the ***std*** namespace and presented as a set of headers. The headers identify the major parts of the library. Thus, listing them gives an overview of the library and provides a guide to the description of the library in this and subsequent chapters.

The rest of this subsection is a list of headers grouped by function, accompanied by brief explanations and annotated by references to where they are discussed. The grouping is chosen to match the organization of the standard. A reference to the standard (such as §s.18.1) means that the facility is not discussed here.

A standard header with a name starting with the letter ***c*** is equivalent to a header in the C standard library. For every header <***cX***> defining names in the ***std*** namespace, there is a header <***X.h***> defining the same names in the global namespace (see §9.2.2).

Containers		
<***vector***>	*one-dimensional array of* ***T***	§16.3
<***list***>	*doubly-linked list of* ***T***	§17.2.2
<***deque***>	*double-ended queue of* ***T***	§17.2.3
<***queue***>	*queue of* ***T***	§17.3.2
<***stack***>	*stack of* ***T***	§17.3.1
<***map***>	*associative array of* ***T***	§17.4.1
<***set***>	*set of* ***T***	§17.4.3
<***bitset***>	*array of booleans*	§17.5.3

The associative containers ***multimap*** and ***multiset*** can be found in <***map***> and <***set***>, respectively. The ***priority_queue*** is declared in <***queue***>.

General Utilities		
<utility>	*operators and pairs*	§17.1.4, §17.4.1.2
<functional>	*function objects*	§18.4
<memory>	*allocators for containers*	§19.4.4
<ctime>	*C-style date and time*	§s.20.5

The *<memory>* header also contains the ***auto_ptr*** template that is primarily used to smooth the interaction between pointers and exceptions (§14.4.2).

Iterators		
<iterator>	*iterators and iterator support*	Chapter 19

Iterators provide the mechanism to make standard algorithms generic over the standard containers and similar types (§2.7.2, §19.2.1).

Algorithms		
<algorithm>	*general algorithms*	Chapter 18
<cstdlib>	***bsearch*( ) *qsort*( )**	§18.11

A typical general algorithm can be applied to any sequence (§3.8, §18.3) of any type of elements. The C standard library functions ***bsearch*( )** and ***qsort*( )** apply to built-in arrays with elements of types without user-defined copy constructors and destructors only (§7.7).

Diagnostics		
<exception>	*exception class*	§14.10
<stdexcept>	*standard exceptions*	§14.10
<cassert>	*assert macro*	§24.3.7.2
<cerrno>	*C-style error handling*	§20.4.1

Assertions relying on exceptions are described in §24.3.7.1.

Strings		
<string>	*string of* ***T***	Chapter 20
<cctype>	*character classification*	§20.4.2
<cwtype>	*wide-character classification*	§20.4.2
<cstring>	*C-style string functions*	§20.4.1
<cwchar>	*C-style wide-character string functions*	§20.4
<cstdlib>	*C-style string functions*	§20.4.1

The *<cstring>* header declares the ***strlen*( )**, ***strcpy*( )**, etc., family of functions. The *<cstdlib>* declares ***atof*( )** and ***atoi*( )** that convert C-style strings to numeric values.

Input/Output		
<*iosfwd*>	*forward declarations of I/O facilities*	§21.1
<*iostream*>	*standard iostream objects and operations*	§21.2.1
<*ios*>	*iostream bases*	§21.2.1
<*streambuf*>	*stream buffers*	§21.6
<*istream*>	*input stream template*	§21.3.1
<*ostream*>	*output stream template*	§21.2.1
<*iomanip*>	*manipulators*	§21.4.6.2
<*sstream*>	*streams to/from strings*	§21.5.3
<*cstdlib*>	*character classification functions*	§20.4.2
<*fstream*>	*streams to/from files*	§21.5.1
<*cstdio*>	***printf***( ) *family of I/O*	§21.8
<*cwchar*>	***printf***( )*-style I/O of wide characters*	§21.8

Manipulators are objects used to manipulate the state of a stream (e.g., changing the format of floating-point output) by applying them to the stream (§21.4.6).

Localization		
<*locale*>	*represent cultural differences*	§21.7
<*clocale*>	*represent cultural differences C-style*	§21.7

A ***locale*** localizes differences such as the output format for dates, the symbol used to represent currency, and string collation criteria that vary among different natural languages and cultures.

Language Support		
<*limits*>	*numeric limits*	§22.2
<*climits*>	*C-style numeric scalar-limit macros*	§22.2.1
<*cfloat*>	*C-style numeric floating-point limit macros*	§22.2.1
<*new*>	*dynamic memory management*	§16.1.3
<*typeinfo*>	*run-time type identification support*	§15.4.1
<*exception*>	*exception-handling support*	§14.10
<*cstddef*>	*C library language support*	§6.2.1
<*cstdarg*>	*variable-length function argument lists*	§7.6
<*csetjmp*>	*C-style stack unwinding*	§s.18.7
<*cstdlib*>	*program termination*	§9.4.1.1
<*ctime*>	*system clock*	§s.18.7
<*csignal*>	*C-style signal handling*	§s.18.7

The <***cstddef***> header defines the type of values returned by ***sizeof***( ), ***size_t***, the type of the result of pointer subtraction, ***ptrdiff_t*** (§6.2.1), and the infamous ***NULL*** macro (§5.1.1).

Numerics		
<*complex*>	*complex numbers and operations*	§22.5
<*valarray*>	*numeric vectors and operations*	§22.4
<*numeric*>	*generalized numeric operations*	§22.6
<*cmath*>	*standard mathematical functions*	§22.3
<*cstdlib*>	*C-style random numbers*	§22.7

For historical reasons, ***abs***( ), ***fabs***( ), and ***div***( ) are found in <*cstdlib*> rather than in <*cmath*> with the rest of the mathematical functions (§22.3).

A user or a library implementer is not allowed to add or subtract declarations from the standard headers. Nor is it acceptable to try to change the contents of headers by defining macros before they are included or to try to change the meaning of the declarations in the headers by declarations in their context (§9.2.3). Any program or implementation that plays such games does not conform to the standard, and programs that rely on such tricks are not portable. Even if they work today, the next release of any part of an implementation may break them. Avoid such trickery.

For a standard library facility to be used its header must be included. Writing out the relevant declarations yourself is *not* a standards-conforming alternative. The reason is that some implementations optimize compilation based on standard header inclusion and others provide optimized implementations of standard library facilities triggered by the headers. In general, implementers use standard headers in ways programmers cannot predict and shouldn't have to know about.

A programmer can, however, specialize utility templates, such as ***swap***( ) (§16.3.9), for nonstandard-library, user-defined types.

16.1.3 Language Support [org.lang]

A small part of the standard library is language support; that is, facilities that must be present for a program to run because language features depend on them.

The library functions supporting operators ***new*** and ***delete*** are discussed in §6.2.6, §10.4.11, §14.4.4, and §15.6; they are presented in <*new*>.

Run-time type identification relies on class ***type_info***, which is described in §15.4.4 and presented in <*typeinfo*>.

The standard exception classes are discussed in §14.10 and presented in <*new*>, <*typeinfo*>, <*ios*>, <*exception*>, and <*stdexcept*>.

Program start and termination are discussed in §3.2, §9.4, and §10.4.9.

16.2 Container Design [org.cont]

A container is an object that holds other objects. Examples are lists, vectors, and associative arrays. In general, you can add objects to a container and remove objects from it.

Naturally, this idea can be presented to users in many different ways. The C++ standard library containers were designed to meet two criteria: to provide the maximum freedom in the design of an individual container, while at the same time allowing containers to present a common interface to users. This allows optimal efficiency in the implementation of containers and enables users to write code that is independent of the particular container used.

Container designs typically meet just one or the other of these two design criteria. The container and algorithms part of the standard library (often called the STL) can be seen as a solution to the problem of simultaneously providing generality and efficiency. The following sections present the strengths and weaknesses of two traditional styles of containers as a way of approaching the design of the standard containers.

16.2.1 Specialized Containers and Iterators [org.specialized]

The obvious approach to providing a vector and a list is to define each in the way that makes the most sense for its intended use:

```
template<class T> class Vector {      // optimal
public:
    explicit Vector(size_t n); // initialize to hold n objects with value T()

    T& operator[](size_t);     // subscripting
    // ...
};

template<class T> class List {   // optimal
public:
    class Link { /* ... */ };

    List();           // initially empty
    void put(T*);     // put before current element
    T* get();         // get current element

    // ...
};
```

Each class provides operations that are close to ideal for their use, and for each class we can choose a suitable representation without worrying about other kinds of containers. This allows the implementations of operations to be close to optimal. In particular, the most common operations such as *put*() for a *List* and *operator*[]() for a *Vector* are small and easily inlined.

A common use of most kinds of containers is to iterate through the container looking at the elements one after the other. This is typically done by defining an iterator class appropriate to the kind of container (see §11.5 and §11.14[7]).

However, a user iterating over a container often doesn't care whether data is stored in a *List* or a *Vector*. In that case, the code iterating should not depend on whether a *List* or a *Vector* was used. Ideally, the same piece of code should work in both cases.

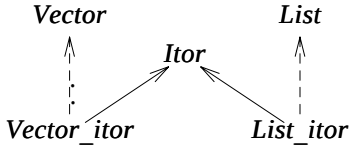
A solution is to define an iterator class that provides a get-next-element operation that can be implemented for any container. For example:

```
template<class T> class Itor {   // common interface (abstract class §2.5.4, §12.3)
public:
    // return 0 to indicate no-more-elements

    virtual T* first() = 0;      // pointer to first element
    virtual T* next() = 0;       // pointer to next element
};
```

We can now provide implementations for ***Vector***s and ***List***s:

```
template<class T> class Vector_itor : public Itor<T> {      // Vector implementation
        Vector<T>& v;
        size_t index;     // index of current element
public:
        Vector_itor(Vector<T>& vv) :v(vv), index(0) { }
        T* first() { return (v.size()) ? &v[index=0] : 0; }
        T* next() { return (++index<v.size()) ? &v[index] : 0; }
};

template<class T> class List_itor : public Itor<T> {        // List implementation
        List<T>& lst;
        List<T>::Link p;      // points to current element
public:
        List_itor(List<T>&);
        T* first();
        T* next();
};
```

Or graphically, using dashed lines to represent ''implemented using:''



The internal structure of the two iterators is quite different, but that doesn't matter to users. We can now write code that iterates over anything for which we can implement an ***Itor***. For example:

```
int count(Itor<char>& ii, char term)
{
        int c = 0;
        for (char* p = ii.first(); p; p=ii.next()) if (*p==term) c++;
        return c;
}
```

There is a snag, however. The operations on an ***Itor*** iterator are simple, yet they incur the overhead of a (virtual) function call. In many situations, this overhead is minor compared to what else is being done. However, iterating through a simple container is the critical operation in many high-performance systems and a function call is many times more expensive than the integer addition or pointer dereferencing that implements ***next***( ) for a ***vector*** and a ***list***. Consequently, this model is unsuitable, or at least not ideal, for a standard library.

However, this container-and-iterator model has been successfully used in many systems. For years, it was my favorite for most applications. Its strengths and weaknesses can be summarized like this:

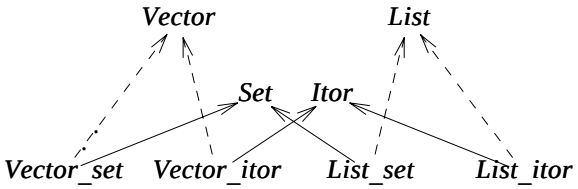
+ Individual containers are simple and efficient.
+ Little commonality is required of containers. Iterators and wrapper classes (§25.7.1) can be used to fit independently developed containers into a common framework.

+ Commonality of use is provided through iterators (rather than through a general container type; §16.2.2).
+ Different iterators can be defined to serve different needs for the same container.
+ Containers are by default type safe and homogeneous (that is, all elements in a container are of the same type). A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
+ The containers are non-intrusive (that is, an object need not have a special base class or link field to be a member of a container). Non-intrusive containers work well with built-in types and with ***struct***s with externally-imposed layouts.
– Each iterator access incurs the overhead of a virtual function call. The time overhead can be serious compared to simple inlined access functions.
– A hierarchy of iterator classes tends to get complicated.
– There is nothing in common for every container and nothing in common for every object in every container. This complicates the provision of universal services such as persistence and object I/O.

A + indicates an advantage and a - indicates a disadvantage.

I consider the flexibility provided by iterators especially important. A common interface, such as ***Itor***, can be provided long after the design and implementation of containers (here, ***Vector*** and ***List***). When we design, we typically first invent something fairly concrete. For example, we design an array and invent a list. Only later do we discover an abstraction that covers both arrays and lists in a given context.

As a matter of fact, we can do this ''late abstraction'' several times. Suppose we want to represent a set. A set is a very different abstraction from ***Itor***, yet we can provide a ***Set*** interface to ***Vector*** and ***List*** in much the same way that I provided ***Itor*** as an interface to ***Vector*** and ***List***:



Thus, late abstraction using abstract classes allows us to provide different implementations of a concept even when there is no significant similarity between the implementations. For example, lists and vectors have some obvious commonality, but we could easily implement an ***Itor*** for an ***istream***.

Logically, the last two points on the list are the main weaknesses of the approach. That is, even if the function call overhead for iterators and similar interfaces to containers were eliminated (as is possible in some contexts), this approach would not be ideal for a standard library.

Non-intrusive containers incur a small overhead in time and space for some containers compared with intrusive containers. I have not found this a problem. Should it become a problem, an iterator such as ***Itor*** can be provided for an intrusive container (§16.5[11]).

16.2.2 Based Containers [org.based]

One can define an intrusive container without relying on templates or any other way of parameterizing a type declaration. For example:

```
struct Link {
        Link* pre;
        Link* suc;
        // ...
};

class List {
        Link* head;
        Link* curr;             // current element
public:
        Link* get();            // remove and return current element
        void put(Link*);        // insert before current element
        // ...
};
```

A ***List*** is now a list of ***Link***s, and it can hold objects of any type derived from ***Link***. For example:

```
class Ship : public Link { /* ... */ };

void f(List* lst)
{
        while (Link* po = lst->get()) {
                if (Ship* ps = dynamic_cast<Ship*>(po)) {       // Ship must be polymorphic (§15.4.1)
                        // use ship
                }
                else {
                        // Oops, do something else
                }
        }
}
```

Simula defined its standard containers in this style, so this approach can be considered the original for languages supporting object-oriented programming. These days, a common class for all objects is usually called ***Object*** or something similar. An ***Object*** class typically provides other common services in addition to serving as a link for containers.

Often, but not necessarily, this approach is extended to provide a common container type:

```
class Container : public Object {
public:
        virtual Object* get();                  // remove and return current element
        virtual void put(Object*);              // insert before current element
        virtual Object*& operator[](size_t);    // subscripting
        // ...
};
```

Note that the operations provided by ***Container*** are virtual so that individual containers can override them appropriately:

```
class List : public Container {
public:
    Object* get();
    void put(Object*);
    // ...
};

class Vector : public Container {
public:
    Object*& operator[](size_t);
    // ...
};
```

One problem arises immediately. What operations do we want ***Container*** to provide? We could provide only the operations that every container can support. However, the intersection of the sets of operations on all containers is a ridiculously narrow interface. In fact, in many interesting cases that intersection is empty. So, realistically, we must provide the union of essential operations on the variety of containers we intend to support. Such a union of interfaces to a set of concepts is called a *fat interface* (§24.4.3).

We can either provide default implementations of the functions in the fat interface or force every derived class to implement every function by making them pure virtual functions. In either case, we end up with a lot of functions that simply report a run-time error. For example:

```
class Container : public Object {
public:
    struct Bad_op { // exception class
        const char* p;
        Bad_op(const char* pp) :p(pp) { }
    };

    virtual void put(Object*) { throw Bad_op("put"); }
    virtual Object* get() { throw Bad_op("get"); }
    virtual Object*& operator[](int) { throw Bad_op("[]"); }
    // ...
};
```

If we want to protect against the possibility of a container that does not support ***get()***, we must catch ***Container::Bad_op*** somewhere. We could now write the ***Ship*** example like this:

```
class Ship : public Object { /* ... */ };

void f1(Container* pc)
{
    try {
        while (Object* po = pc->get()) {
            if (Ship* ps = dynamic_cast<Ship*>(po)) {
                // use ship
            }
```

```
                else {
                    // Oops, do something else
                }
            }
        }
        catch (Container::Bad_op& bad) {
                // Oops, do something else
        }
    }
```

This is tedious, so the checking for ***Bad_op*** will typically be elsewhere. By relying on exceptions caught elsewhere, we can reduce the example to:

```
void f2(Container* pc)
{
    while (Object* po = pc->get()) {
        Ship& s = dynamic_cast<Ship&>(*po);
        // use ship
    }
}
```

However, I find unnecessary reliance on run-time checking distasteful and inefficient. In this kind of case, I prefer the statically-checked alternative:

```
void f3(Itor<Ship>* i)
{
    while (Ship* ps = i->next()) {
        // use ship
    }
}
```

The strengths and weakness of the ''based object'' approach to container design can be summarized like this (see also §16.5[10]):

- – Operations on individual containers incur virtual function overhead.
- – All containers must be derived from ***Container***. This implies the use of fat interfaces, requires a large degree of foresight, and relies on run-time type checking. Fitting an independently developed container into the common framework is awkward at best (see §16.5[12]).
- \+ The common base ***Container*** makes it easy to use containers that supply similar sets of operations interchangeably.
- – Containers are heterogeneous and not type safe by default (all we can rely on is that elements are of type ***Object****). When desired, type-safe and homogeneous containers can be defined using templates.
- – The containers are intrusive (that is, every element must be of a type derived from ***Object***). Objects of built-in types and structs with externally imposed layouts cannot be placed directly in containers.
- – An element retrieved from a container must be given a proper type using explicit type conversion before it can be used.
- \+ Class ***Container*** and class ***Object*** are handles for implementing services for every object or

every container. This greatly eases the provision of universal services such as persistence and object I/O.

As before (§16.2.1), + indicates an advantage and - indicates a disadvantage.

Compared to the approach using unrelated containers and iterators, the based-object approach unnecessarily pushes complexity onto the user, imposes significant run-time overheads, and restricts the kinds of objects that can be placed in a container. In addition, for many classes, to derive from ***Object*** is to expose an implementation detail. Thus, this approach is far from ideal for a standard library.

However, the generality and flexibility of this approach should not be underestimated. Like its alternatives, it has been used successfully in many applications. Its strengths lie in areas in which efficiency is less important than the simplicity afforded by a single ***Container*** interface and services such as object I/O.

16.2.3 STL Containers [org.stl]

The standard library containers and iterators (often called the STL framework, §3.10) can be understood as an approach to gain the best of the two traditional models described previously. That wasn't the way the STL was designed, though. The STL was the result of a single-minded search for uncompromisingly efficient and generic algorithms.

The aim of efficiency rules out hard-to-inline virtual functions for small, frequently-used access functions. Therefore, we cannot present a standard interface to containers or a standard iterator interface as an abstract class. Instead, each kind of container supports a standard set of basic operations. To avoid the problems of fat interfaces (§16.2.2, §24.4.3), operations that cannot be efficiently implemented for all containers are not included in the set of common operations. For example, subscripting is provided for ***vector*** but not for ***list***. In addition, each kind of container provides its own iterators that support a standard set of iterator operations.

The standard containers are not derived from a common base. Instead, every container implements all of the standard container interface. Similarly, there is no common iterator base class. No explicit or implicit run-time type checking is involved in using the standard containers and iterators.

The important and difficult issue of providing common services for all containers is handled through ''allocators'' passed as template arguments (§19.4.3) rather than through a common base.

Before I go into details and code examples, the strengths and weaknesses of the STL approach can be summarized:

+ Individual containers are simple and efficient (not quite as simple as truly independent containers can be, but just as efficient).
+ Each container provides a set of standard operations with standard names and semantics. Additional operations are provided for a particular container type as needed. Furthermore, wrapper classes (§25.7.1) can be used to fit independently developed containers into a common framework (§16.5[14]).
+ Additional commonality of use is provided through standard iterators. Each container provides iterators that support a set of standard operations with standard names and semantics. An iterator type is defined for each particular container type so that these iterators are as simple and efficient as possible.

+ To serve different needs for containers, different iterators and other generalized interfaces can be defined in addition to the standard iterators.
+ Containers are by default type-safe and homogeneous (that is, all elements in a container are of the same type). A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
+ The containers are non-intrusive (that is, an object need not have a special base class or link field to be a member of a container). Non-intrusive containers work well with built-in types and with ***struct***s with externally imposed layouts.
+ Intrusive containers can be fitted into the general framework. Naturally, an intrusive container will impose constraints on its element types.
+ Each container takes an argument, called an ***allocator***, which can be used as a handle for implementing services for every container. This greatly eases the provision of universal services such as persistence and object I/O (§19.4.3).
– There is no standard run-time representation of containers or iterators that can be passed as a function argument (although it is easy to define such representations for the standard containers and iterators where needed for a particular application; §19.3).

As before (§16.2.1), + indicates an advantage and - indicates a disadvantage.

In other words, containers and iterators do not have fixed standard representations. Instead, each container provides a standard interface in the form of a set of operations so that containers can be used interchangeably. Iterators are handled similarly. This implies minimal overheads in time and space while allowing users to exploit commonality both at the level of containers (as with the based-object approach) and at the level of iterators (as with the specialized container approach).

The STL approach relies heavily on templates. To avoid excessive code replication, partial specialization to provide shared implementations for containers of pointers is usually required (§13.5).

16.3 Vector [org.vector]

Here, ***vector*** is described as an example of a complete standard container. Unless otherwise stated, what is said about ***vector*** holds for every standard container. Chapter 17 describes features peculiar to ***list***s, ***set***s, ***map***s, etc. The facilities offered by ***vector*** – and similar containers – are described in some detail. The aim is to give an understanding both of the possible uses of ***vector*** and of its role in the overall design of the standard library.

An overview of the standard containers and the facilities they offer can be found in §17.1. Below, ***vector*** is introduced in stages: member types, iterators, element access, constructors, stack operations, list operations, size and capacity, helper functions, and ***vector<bool>***.

16.3.1 Types [org.types]

The standard ***vector*** is a template defined in namespace ***std*** and presented in ***<vector>***. It first defines a set of standard names of types:

```
template<class T, class A = allocator<T> > class std::vector {
public:
    // types:
```

```
    typedef T value_type;                                   // type of element
    typedef A allocator_type;                               // type of memory manager
    typedef typename A::size_type size_type;
    typedef typename A::difference_type difference_type;

    typedef implementation_dependent1 iterator;             // T*
    typedef implementation_dependent2 const_iterator;       // const T*
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    typedef typename A::pointer pointer;                    // pointer to element
    typedef typename A::const_pointer const_pointer;
    typedef typename A::reference reference;                // reference to element
    typedef typename A::const_reference const_reference;

    // ...
};
```

Every standard container defines these typenames as members. Each defines them in the way most appropriate to its implementation.

The type of the container's elements is passed as the first template argument and is known as its ***value_type***. The ***allocator_type***, which is optionally supplied as the second template argument, defines how the ***value_type*** interacts with various memory management mechanisms. In particular, an allocator supplies the functions that a container uses to allocate and deallocate memory for its elements. Allocators are discussed in §19.4. In general, ***size_type*** specifies the type used for indexing into the container, and ***difference_type*** is the type of the result of subtracting two iterators for a container. For most containers, they correspond to ***size_t*** and ***ptrdiff_t*** (§6.2.1).

Iterators were introduced in §2.7.2 and are described in detail in Chapter 19. They can be thought of as pointers to elements of the container. Every container provides a type called ***iterator*** for pointing to elements. It also provides a ***const_iterator*** type for use when elements don't need to be modified. As with pointers, we use the safer ***const*** version unless there is a reason to do otherwise. The actual types of ***vector***'s iterators are implementation-defined. The obvious definitions for a conventionally-defined ***vector*** would be ***T**** and ***const T****, respectively.

The reverse iterator types for ***vector*** are constructed from the standard ***reverse_iterator*** templates (§19.2.5). They present a sequence in the reverse order.

As shown in §3.8.1, these member typenames allow a user to write code using a container without having to know about the actual types involved. In particular, they allow a user to write code that will work for any standard container. For example:

```
template<class C> typename C::value_type sum(const C& c)
{
    typename C::value_type s = 0;
    typename C::const_iterator p = c.begin();       // start at the beginning
    while (p!=c.end()) {                            // continue until the end
        s += *p;        // get value of element
        ++p;            // make p point to next element
    }
    return s;
}
```

Having to add *typename* before the names of member types of a template parameter is a nuisance. However, the compiler isn't psychic. There is no general way for it to know whether a member of a template argument type is a typename (§C.13.5).

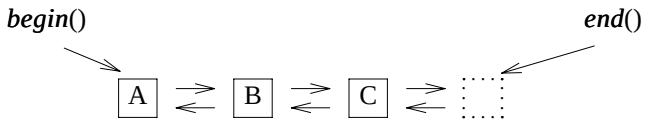
As for pointers, prefix * means dereference the iterator (§2.7.2, §19.2.1) and ++ means increment the iterator.

16.3.2 Iterators [org.begin]

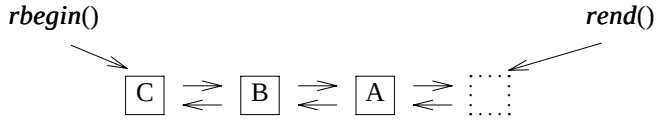
As shown in the previous subsection, iterators can be used to navigate containers without the programmers having to know the actual type used to identify elements. A few key member functions allow the programmer to get hold of the ends of the sequence of elements:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // iterators:

    iterator begin();                       // points to first element
    const_iterator begin() const;
    iterator end();                         // points to one-past-last element
    const_iterator end() const;

    reverse_iterator rbegin();              // points to first element of reverse sequence
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();                // points to one-past-last element of reverse sequence
    const_reverse_iterator rend() const;

    // ...
};
```

The *begin()/end()* pair gives the elements of the container in the ordinary element order. That is, element *0* is followed by element *1*, element *2*, etc. The *rbegin()/rend()* pair gives the elements in the reverse order. That is, element *n-1* is followed by element *n-2*, element *n-3*, etc. For example, a sequence seen like this using an *iterator*:



can be viewed like this using a *reverse_iterator* (§19.2.5):



This allows us to use algorithms in a way that views a sequence in the reverse order. For example:

```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
    return find_first(c.rbegin(),c.rend(),v).base();
}
```

The *base*( ) function returns an *iterator* corresponding to the *reverse_iterator* (§19.2.5). Without reverse iterators, we could have had to write something like:

```
template<class C>
typename C::iterator find_last(const C& c, typename C::value_type v)
{
    typename C::iterator p = c.end();  //search backwards from end
    while (p!=c.begin()) {
        --p;
        if (*p==v) return p;
    }
    return p;
}
```

A reverse iterator is a perfectly ordinary iterator, so we could have written:

```
template<class C>
typename C::reverse_iterator find_last(const C& c, typename C::value_type v)
{
    typename C::reverse_iterator p = c.rbegin();  // view sequence in reverse order
    while (p!=c.rend()) {
        if (*p==v) return p;
        ++p;            // note: not decrement (--)
    }
    return p;
}
```

16.3.3 Element Access [org.element]

One important aspect of a *vector* compared with other containers is that one can easily and efficiently access individual elements in any order:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // element access:

    reference operator[](size_type n);          // unchecked access
    const_reference operator[](size_type n) const;

    reference at(size_type n);                  // checked access
    const_reference at(size_type n) const;
```

```
        reference front();                          // first element
        const_reference front() const;
        reference back();                           // last element
        const_reference back() const;

        // ...
};
```

Indexing is done by *operator*[]() and *at*(); *operator*[]() provides unchecked access, whereas *at*() does a range check and throws *out_of_range* if an index is out of range. For example:

```
void f(vector<int>& v, int i1, int i2)
try {
        for(int i = 0; i < v.size(); i++) {
                // range already checked: use unchecked v[i] here
        }

        v.at(i1) = v.at(i2); // check range on access

        // ...
}
catch(out_of_range) {
        // oops: out-of-range error
}
```

This illustrates one idea for use. That is, if the range has already been checked, the unchecked subscripting operator can be used safely; otherwise, it is wise to use the range-checked *at*() function. This distinction is important when efficiency is at a premium. When that is not the case or when it is not perfectly obvious whether a range has been correctly checked, it is safer to use a vector with a checked [] operator (such as *Vec* from §3.7.1) or a checked iterator (§19.3).

The default access is unchecked to match arrays. Also, you can build a safe (checked) facility on top of a fast one but not a faster facility on top of a slower one.

The access operations return values of type *reference* or *const_reference* depending on whether or not they are applied to a *const* object. A reference is some suitable type for accessing elements. For the simple and obvious implementation of *vector<X>*, *reference* is simply *X*& and *const_reference* is simply *const X*&. The effect of trying to create an out-of-range reference is undefined. For example:

```
void f(vector<double>& v)
{
        double d = v[v.size()];   // undefined: bad index

        list<char> lst;
        char c = lst.front();     // undefined: list is empty
}
```

Of the standard sequences, only *vector* and *deque* (§17.2.3) support subscripting. The reason is the desire not to confuse users by providing fundamentally inefficient operations. For example, subscripting could have been provided for *list* (§17.2.2), but doing that would have been dangerously inefficient (that is, $O(n)$).

The members *front*() and *back*() return references to the first and last element, respectively.

They are most useful where these elements are known to exist and in code where these elements are of particular interest. A ***vector*** used as a ***stack*** (§16.3.5) is an obvious example. Note that ***front***( ) returns a reference to the element to which ***begin***( ) returns an iterator. I often think of ***front***( ) as the first element and ***begin***( ) as a pointer to the first element. The correspondence between ***back***( ) and ***end***( ) is less simple: ***back***( ) is the last element and ***end***( ) points to the last-plus-one element position.

16.3.4 Constructors [org.ctor]

Naturally, ***vector*** provides a complete set (§11.7) of constructors, destructor, and copy operations:

```
template <class T, class A = allocator<T> > class vector {
public:
        // ...
        // constructors, etc.:

        explicit vector(const A& = A( ));
        explicit vector(size_type n, const T& val = T( ), const A& = A( ));    // n copies of val
        template <class In>                                  // In must be an input iterator (§19.2.1)
                vector(In first, In last, const A& = A( )); // copy from [first:last[
        vector(const vector& x);

        ~vector( );

        vector& operator=(const vector& x);

        template <class In>                          // In must be an input iterator (§19.2.1)
                void assign(In first, In last);      // copy from [first:last[
        void assign(size_type n, const T& val);      // n copies of val

        // ...
};
```

A ***vector*** provides fast access to arbitrary elements, but changing its size is relatively expensive. Consequently, we typically give an initial size when we create a ***vector***. For example:

```
vector<Record> vr(10000);

void f(int s1, int s2)
{
        vector<int> vi(s1);

        vector<double>* p = new vector<double>(s2);
}
```

Elements of a vector allocated this way are initialized by the default constructor for the element type. That is, each of ***vr***'s ***10000*** elements is initialized by ***Record***( ) and each of ***vi***'s ***s1*** elements is initialized by ***int***( ). Note that the default constructor for a built-in type performs initialization to ***0*** of the appropriate type (§4.9.5, §10.4.2).

If a type does not have a default constructor, it is not possible to create a vector with elements of that type without explicitly providing the value of each element. For example:

```
class Num {     // infinite precision
public:
    Num(long);
    // no default constructor
    // ...
};

vector<Num> v1(1000);               // error: no default Num
vector<Num> v2(1000,Num(0));        // ok
```

Since a **vector** cannot have a negative number of elements, its size must be non-negative. This is reflected in the requirement that **vector**'s **size_type** must be an **unsigned** type. This allows a greater range of vector sizes on some architectures. However, it can also lead to surprises:

```
void f(int i)
{
    vector<char> vc0(-1);     // fairly easy for compiler to warn against
    vector<char> vc1(i);
}

void g()
{
    f(-1);     // trick f() into accepting a large positive number!
}
```

In the call `f(-1)`, `-1` is converted into a (rather large) positive integer (§C.6.3). If we are lucky, the compiler will find a way of complaining.

The size of a **vector** can also be provided implicitly by giving the initial set of elements. This is done by supplying the constructor with a sequence of values from which to construct the **vector**. For example:

```
void f(const list<X>& lst)
{
    vector<X> v1(lst.begin(),lst.end());     // copy elements from list

    char p[] = "despair";
    vector<char> v2(p,&p[sizeof(p)-1]);      // copy characters from C-style string
}
```

In each case, the **vector** constructor adjusts the size of the **vector** as it copies elements from its input sequence.

The **vector** constructors that can be invoked with a single argument are declared **explicit** to prevent accidental conversions (§11.7.1). For example:

```
vector<int> v1(10);                 // ok: vector of 10 ints
vector<int> v2 = vector<int>(10);   // ok: vector of 10 ints
vector<int> v3 = v2;                // ok: v3 is a copy of v2
vector<int> v4 = 10;                // error: attempted implicit conversion of 10 to vector<int>
```

The copy constructor and the copy-assignment operators copy the elements of a **vector**. For a **vector** with many elements, that can be an expensive operation, so **vector**s are typically passed by reference. For example:

```
void f1(vector<int>&);          // common style
void f2(const vector<int>&);    // common style
void f3(vector<int>);           // rare style

void h()
{
    vector<int> v(10000);

    // ...

    f1(v);      // pass a reference
    f2(v);      // pass a reference
    f3(v);      // copy the 10000 elements into a new vector for f3() to use
}
```

The ***assign*** functions exist to provide counterparts to the multi-argument constructors. They are needed because = takes a single right-hand operand, so ***assign()*** is used where a default argument value or a range of values is needed. For example:

```
class Book {
    // ...
};

void f(vector<Num>& vn, vector<char>& vc, vector<Book>& vb, list<Book>& lb)
{
    vn.assign(10,Num(0));               // assign vector of 10 copies of Num(0) to vn

    char s[] = "literal";
    vc.assign(s,&s[sizeof(s)-1]);       // assign "literal" to vc

    vb.assign(lb.begin(),lb.end());     // assign list elements

    // ...
}
```

Thus, we can initialize a ***vector*** with any sequence of its element type and similarly assign any such sequence. Importantly, this is done without explicitly introducing a multitude of constructors and conversion functions. Note that assignment completely changes the elements of a vector. Conceptually, all old elements are erased and the new ones are inserted. After assignment, the size of a ***vector*** is the number of elements assigned. For example:

```
void f()
{
    vector<char> v(10,´x´);         // v.size()==10, each element has the value 'x'
    v.assign(5,´a´);                // v.size()==5, each element has the value 'a'
    // ...
}
```

Naturally, what ***assign()*** does could be done indirectly by first creating a suitable ***vector*** and then assigning that. For example:

```
void f2(vector<Book>& vh, list<Book>& lb)
{
    vector<Book> vt(lb.begin(),lb.end());
    vh = vt;
    // ...
}
```

However, this can be both ugly and inefficient.

Constructing a *vector* with two arguments of the same type can lead to an apparent ambiguity:

```
vector<int> v(10,50);    // vector(size,value) or vector(iterator1,iterator2)? vector(size,value)!
```

However, an *int* isn't an iterator and the implementation must ensure that this actually invokes

```
vector(vector<int>::size_type, const int&, const vector<int>::allocator_type&);
```

rather than

```
vector(vector<int>::iterator, vector<int>::iterator, const vector<int>::allocator_type&);
```

The library achieves this by suitable overloading of the constructors and handles the equivalent ambiguities for *assign*() and *insert*() (§16.3.6) similarly.

16.3.5 Stack Operations [org.stack]

Most often, we think of a *vector* as a compact data structure that we can index to access elements. However, we can ignore this concrete notion and view *vector* as an example of the more abstract notion of a sequence. Looking at a *vector* this way, and observing common uses of arrays and *vector*s, it becomes obvious that stack operations make sense for a *vector*:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // stack operations:

    void push_back(const T& x);    // add to end
    void pop_back();               // remove last element
    // ...
};
```

These functions treat a *vector* as a stack by manipulating its end. For example:

```
void f(vector<char>& s)
{
    s.push_back('a');
    s.push_back('b');
    s.push_back('c');
    s.pop_back();
    if (s[s.size()-1] != 'b') error("impossible!");
    s.pop_back();
    if (s.back() != 'a') error("should never happen!");
}
```

Each time ***push_back()*** is called, the ***vector s*** grows by one element and that element is added at the end. So ***s[s.size()-1]***, also known as ***s.back()*** (§16.3.3), is the element most recently pushed onto the ***vector***.

Except for the word ***vector*** instead of ***stack***, there is nothing unusual in this. The suffix ***_back*** is used to emphasize that elements are added to the end of the ***vector*** rather than to the beginning. Adding an element to the end of a ***vector*** could be an expensive operation because extra memory needs to be allocated to hold it. However, an implementation must ensure that repeated stack operations incur growth-related overhead only infrequently.

Note that ***pop_back()*** does not return a value. It just pops, and if we want to know what was on the top of the stack before the pop, we must look. This happens not to be my favorite style of stack (§2.5.3, §2.5.4), but it's arguably more efficient and it's the standard.

Why would one do stack-like operations on a ***vector***? An obvious reason is to implement a ***stack*** (§17.3.1), but a more common reason is to construct a ***vector*** incrementally. For example, we might want to read a ***vector*** of points from input. However, we don't know how many points will be read, so we can't allocate a vector of the right size and then read into it. Instead, we might write:

```
vector<Point> cities;

void add_points(Point sentinel)
{
    Point buf;

    while (cin >> buf) {
        if (buf == sentinel) return;
        // check new point
        cities.push_back(buf);
    }
}
```

This ensures that the ***vector*** expands as needed. If all we needed to do with a new point were to put it into the ***vector***, we might have initialized ***cities*** directly from input in a constructor (§16.3.4). However, it is common to do a bit of processing on input and expand a data structure gradually as a program progresses; ***push_back()*** supports that.

In C programs, this is one of the most common uses of the C standard library function ***realloc()***. Thus, ***vector*** – and, in general, any standard container – provides a more general, more elegant, and no less efficient alternative to ***realloc()***.

The ***size()*** of a ***vector*** is implicitly increased by ***push_back()*** so the ***vector*** cannot overflow (as long as there is memory available to acquire; see §19.4.1). However, a ***vector*** can underflow:

```
void f()
{
    vector<int> v;
    v.pop_back();       // undefined effect: the state of v becomes undefined
    v.push_back(7);     // undefined effect (the state of v is undefined), probably bad
}
```

The effect of underflow is undefined, but the obvious implementation of ***pop_back()*** causes memory not owned by the ***vector*** to be overwritten. Like overflow, underflow must be avoided.

16.3.6 List Operations [org.list]

The *push_back*( ), *pop_back*( ), and *back*( ) operations (§16.3.5) allow a *vector* to be used effectively as a stack. However, it is sometimes also useful to add elements in the middle of a *vector* and to remove elements from a *vector*:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // list operations:

    iterator insert(iterator pos, const T& x);              // add x before 'pos'
    void insert(iterator pos, size_type n, const T& x);
    template <class In>                    // In must be an input iterator (§19.2.1)
        void insert(iterator pos, In first, In last);       // insert elements from sequence

    iterator erase(iterator pos);                           // remove element at pos
    iterator erase(iterator first, iterator last);          // erase sequence
    void clear();                                           // erase all elements

    // ...
};
```

To see how these operations work, let's do some (nonsensical) manipulation of a *vector* of names of fruit. First, we define the *vector* and populate it with some names:

```
vector<string> fruit;

fruit.push_back("peach");
fruit.push_back("apple");
fruit.push_back("kiwifruit");
fruit.push_back("pear");
fruit.push_back("starfruit");
fruit.push_back("grape");
```

If I take a dislike to fruits whose names start with the letter *p*, I can remove those names like this:

```
sort(fruit.begin(),fruit.end());
vector<string>::iterator p1 = find_if(fruit.begin(),fruit.end(),initial('p'));
vector<string>::iterator p2 = find_if(p1,fruit.end(),initial_not('p'));
fruit.erase(p1,p2);
```

In other words, sort the *vector*, find the first and the last fruit with a name that starts with the letter *p*, and erase those elements from *fruit*. How to write predicate functions such as *initial*(*x*) (is the initial letter *x*?) and *initial_not*( ) (is the initial letter different from *p*?) is explained in §18.4.2.

The *erase*(*p1*,*p2*) operation removes elements starting from *p1* up to and not including *p2*. This can be illustrated graphically:

```
fruit[]:
                                p1          p2
                                |           |
                                v           v
    apple   grape   kiwifruit  peach  pear  starfruit
```

The *erase(p1,p2)* removes *peach* and *pear*, yielding:

```
fruit[]:
    apple    grape    kiwifruit    starfruit
```

As usual, the sequence specified by the user is from the beginning to one-past-the-end of the sequence affected by the operation.

It would be tempting to write:

```
vector<string>::iterator  p1 = find_if(fruit.begin(),fruit.end(),initial('p'));
vector<string>::reverse_iterator  p2 = find_if(fruit.rbegin(),fruit.rend(),initial('p'));
fruit.erase(p1,p2+1);      // oops!: type error
```

However, *vector<fruit>::iterator* and *vector<fruit>::reverse_iterator* need not be the same type, so we couldn't rely on the call of *erase()* to compile. To be used with an *iterator*, a *reverse_iterator* must be explicitly converted:

```
fruit.erase(p1,p2.base());      // extract iterator from reverse_iterator (§19.2.5)
```

Erasing an element from a *vector* changes the size of the *vector*, and the elements after the erased elements are copied into the freed positions. In this example, *fruit.size()* becomes *4* and the *starfruit* that used to be *fruit[5]* is now *fruit[3]*.

Naturally, it is also possible to *erase()* a single element. In that case, only an iterator for that element is needed (rather than a pair of iterators). For example,

```
fruit.erase(find(fruit.begin(),fruit.end(),"starfruit"));
fruit.erase(fruit.begin()+1);
```

gets rid of the *starfruit* and the *grape*, thus leaving *fruit* with two elements:

```
fruit[]:
    apple    kiwifruit
```

It is also possible to insert elements into a vector. For example:

```
fruit.insert(fruit.begin()+1,"cherry");
fruit.insert(fruit.end(),"cranberry");
```

The new element is inserted before the position mentioned, and the elements from there to the end are moved to make space. We get:

```
fruit[]:
    apple    cherry    kiwifruit    cranberry
```

Note that *f.insert(f.end(),x)* is equivalent to *f.push_back(x)*.

We can also insert whole sequences:

```
fruit.insert(fruit.begin()+2,citrus.begin(),citrus.end());
```

If *citrus* is a container

citrus[]:

lemon grapefruit orange lime

we get:

fruit[]:

apple cherry lemon grapefruit orange lime kiwifruit cranberry

The elements of *citrus* are copied into *fruit* by *insert*(). The value of *citrus* is unchanged.

Clearly, *insert*() and *erase*() are more general than are operations that affect only the tail end of a *vector* (§16.3.5). They can also be more expensive. For example, to make room for a new element, *insert*() may have to reallocate every element to a new part of memory. If insertions into and deletions from a container are common, maybe that container should be a *list* rather than a *vector*. A *list* is optimized for *insert*() and *erase*() rather than for subscripting (§16.3.3).

Insertion into and erasure from a *vector* (but not a *list* or an associative container such as *map*) potentially move elements around. Consequently, an iterator pointing to an element of a *vector* may after an *insert*() or *erase*() point to another element or to no element at all. Never access an element through an invalid iterator; the effect is undefined and quite likely disastrous. In particular, beware of using the iterator that was used to indicate where an insertion took place; *insert*() makes its first argument invalid. For example:

```
void duplicate_elements(vector<string>& f)
{
    for(vector<string>::iterator p = f.begin(); p!=f.end(); ++p) f.insert(p,*p); // No!
}
```

Just think of it (§16.5[15]). A *vector* implementation would move all elements – or at least all elements after *p* – to make room for the new element.

The operation *clear*() erases all elements of a container. Thus, *c.clear*() is a shorthand for *c.erase*(*c.begin*(),*c.end*()). After *c.clear*(), *c.size*() is *0*.

16.3.7 Addressing Elements [org.addressing]

Most often, the target of an *erase*() or *insert*() is a well-known place (such as *begin*() or *end*()), the result of a search operation (such as *find*()), or a location found during an iteration. In such cases, we have an iterator pointing to the relevant element. However, we often refer to elements of a *vector* by subscripting. How do we get an iterator suitable as an argument for *erase*() or *insert*() for the element with index *7* of a container *c*? Since that element is the 7th element after the beginning, *c.begin*()+*7* is a good answer. Other alternatives that may seem plausible by analogy to arrays should be avoided. Consider:

```
template<class C> void f(C& c)
{
    c.erase(c.begin()+7);          // ok
    c.erase(&c[7]);                // not general
```

```
    c.erase(c+7);                       // error: adding 7 to a container makes no sense
    c.erase(c.back());                  // error: c.back() is a reference, not an iterator
    c.erase(c.end()-2);                 // ok (second to last element)
    c.erase(c.rbegin()+2);              // error: vector::reverse_iterator and vector::iterator
                                        // are different types
    c.erase((c.rbegin()+2).base());     // obscure, but ok (see §19.2.5)
}
```

The most tempting alternative, `&c[7]`, actually happens to work with the obvious implementation of *vector*, where `c[7]` refers directly to the element and its address is a valid iterator. However, this is not true for other containers. For example, a *list* or *map* iterator is almost certainly not a simple pointer to an element. Consequently, their iterators do not support `[]`. Therefore, `&c[7]` would be an error that the compiler catches.

The alternatives `c+7` and `c.back()` are simple type errors. A container is not a numeric variable to which we can add `7`, and `c.back()` is an element with a value like `"pear"` that does not identify the pear's location in the container `c`.

16.3.8 Size and Capacity [org.size]

So far, *vector* has been described with minimal reference to memory management. A *vector* grows as needed. Usually, that is all that matters. However, it is possible to ask directly about the way a *vector* uses memory, and occasionally it is worthwhile to affect it directly. The operations are:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...
    // capacity:

    size_type size() const;                     // number of elements
    bool empty() const { return size()==0; }
    size_type max_size() const;                 // size of the largest possible vector
    void resize(size_type sz, T val = T());     // added elements initialized by val

    size_type capacity() const;     // size of the memory (in number of elements) allocated
    void reserve(size_type n);      // make room for a total of n elements; don't initialize
                                    // throw a length_error if n>max_size()
    // ...
};
```

At any given time, a *vector* holds a number of elements. This number can be obtained by calling `size()` and can be changed using `resize()`. Thus, a user can determine the size of a vector and change it if it seems insufficient or excessive. For example:

```
class Histogram {
    vector<int> count;
public:
    Histogram(int h) : count(max(h,8)) {}
    void record(int i);
    // ...
};
```

```
void Histogram::record(int i)
{
    if (i<0) i = 0;
    if (count.size()<=i) count.resize(i+i);    // make lots of room
    count[i]++;
}
```

Using *resize()* on a *vector* is very similar to using the C standard library function *realloc()* on a C array allocated on the free store.

When a *vector* is resized to accommodate more (or fewer) elements, all of its elements may be moved to new locations. Consequently, it is a bad idea to keep pointers to elements in a *vector* that might be resized; after *resize()*, such pointers could point to deallocated memory. Instead, we can keep indices. Note that *push_back()*, *insert()*, and *erase()* implicitly resize a *vector*.

In addition to the elements held, an application may keep some space for potential expansion. A programmer who knows that expansion is likely can tell the *vector* implementation to *reserve()* space for future expansion. For example:

```
struct Link {
    Link* next;
    Link(Link* n =0) : next(n) {}
    // ...
};

vector<Link> v;

void chain(size_t n) // fill v with n Links so that each Link points to its predecessor
{
    v.reserve(n);
    v.push_back(Link(0));
    for (int i = 1; i<n; i++) v.push_back(Link(&v[i-1]));
    // ...
}
```

A call *v.reserve(n)* ensures that no allocation will be needed when the size of *v* is increased until *v.size()* exceeds *n*.

Reserving space in advance has two advantages. First, even a simple-minded implementation can then allocate sufficient space in one operation rather than slowly acquiring enough memory along the way. However, in many cases there is a logical advantage that outweighs the potential efficiency gain. The elements of a container are potentially relocated when a *vector* grows. Thus, the links built between the elements of *v* in the previous example are guaranteed only because the call of *reserve()* ensures that there are no allocations while the vector is being built. That is, in some cases *reserve()* provides a guarantee of correctness in addition to whatever efficiency advantages it gives.

That same guarantee can be used to ensure that potential memory exhaustion and potentially expensive reallocation of elements take place at predictable times. For programs with stringent real-time constraints, this can be of great importance.

Note that *reserve()* doesn't change the size of a *vector*. Thus, it does not have to initialize any new elements. In both respects, it differs from *resize()*.

In the same way as *size*() gives the current number of elements, *capacity*() gives the current number of reserved memory slots; *c*.*capacity*()-*c*.*size*() is the number of elements that can be inserted without causing reallocation.

Decreasing the size of a *vector* doesn't decrease its capacity. It simply leaves room for the *vector* to grow into later. If you want to give memory back to the system, assign a new value to the *vector*. For example:

```
v = vector<int>(4,99);
```

A *vector* gets the memory it needs for its elements by calling member functions of its allocator (supplied as a template parameter). The default allocator, called *allocator* (§19.4.1), uses *new* to obtain storage so that it will throw *bad_alloc* if no more storage is obtainable. Other allocators can use different strategies (see §19.4.2).

The *reserve*() and *capacity*() functions are unique to *vector* and similar compact containers. Containers such as *list* do not provide equivalents.

16.3.9 Other Member Functions [org.etc]

Many algorithms – including important sort algorithms – involve swapping elements. The obvious way of swapping (§13.5.2) simply copies elements. However, a *vector* is typically implemented with a structure that acts as a handle (§13.5, §17.1.3) to the elements. Thus, two *vector*s can be swapped much more efficiently by interchanging the handles; *vector*::*swap*() does that. The time difference between this and the default *swap*() is orders of magnitude in important cases:

```
template <class T, class A = allocator<T> > class vector {
public:
    // ...

    void swap(vector&);

    allocator_type get_allocator() const;
};
```

The *get_allocator*() function gives the programmer a chance to get hold of a *vector*'s allocator (§16.3.1, §16.3.4). Typically, the reason for this is to ensure that data from an application that is related to a *vector* is allocated similarly to the *vector* itself (§19.4.1).

16.3.10 Helper Functions [org.algo]

Two *vector*s can be compared using == and <:

```
template <class T, class A>
bool std::operator==(const vector<T,A>& x, const vector<T,A>& y);

template <class T, class A>
bool std::operator<(const vector<T,A>& x, const vector<T,A>& y);
```

Two *vector*s *v1* and *v2* compare equal if *v1*.*size*()==*v2*.*size*() and *v1*[*n*]==*v2*[*n*] for every valid index *n*. Similarly, < is a lexicographical ordering. In other words, < for *vector*s could be defined like this:

```
template <class T, class A>
inline bool std::operator<(const vector<T,A>& x, const vector<T,A>& y)
{
    return lexicographical_compare(x.begin(),x.end(),y.begin(),y.end()); // see §18.9
}
```

This means that *x* is less than *y* if the first element *x*[*i*] that is not equal to the corresponding element *y*[*i*] is less than *y*[*i*], or *x*.*size*()<*y*.*size*() with every *x*[*i*] equal to its corresponding *y*[*i*].

The standard library also provides !=, <=, >, and >=, with definitions that correspond to those of == and <.

Because *swap*() is a member, it is called using the *v1*.*swap*(*v2*) syntax. However, not every type has a *swap*() member, so generic algorithms use the conventional *swap*(*a*,*b*) syntax. To make that work for *vector*s also, the standard library provides the specialization:

```
template <class T, class A> void std::swap(vector<T,A>& x, vector<T,A>& y)
{
    x.swap(y);
}
```

16.3.11 Vector<bool> [org.vector.bool]

The specialization (§13.5) *vector*<*bool*> is provided as a compact *vector* of *bool*. A *bool* variable is addressable, so it takes up at least one byte. However, it is easy to implement *vector*<*bool*> so that each element takes up only a bit.

The usual *vector* operations work for *vector*<*bool*> and retain their usual meanings. In particular, subscripting and iteration work as expected. For example:

```
void f(vector<bool>& v)
{
    for (int i = 0; i<v.size(); ++i) cin >> v[i];          // iterate using subscripting

    typedef vector<bool>::const_iterator VI;
    for (VI p = v.begin(); p!=v.end(); ++p) cout<<*p;      // iterate using iterators
}
```

To achieve this, an implementation must simulate addressing of a single bit. Since a pointer cannot address a unit of memory smaller than a byte, *vector*<*bool*>::*iterator* cannot be a pointer. In particular, one cannot rely on *bool** as an iterator for a *vector*<*bool*>:

```
void f(vector<bool>& v)
{
    bool* p = v.begin(); // error: type mismatch
    // ...
}
```

A technique for addressing a single bit is outlined in §17.5.3.

The library also provides *bitset* as a set of Boolean values with Boolean set operations (§17.5.3).

16.4 Advice [org.advice]

[1] Use standard library facilities to maintain portability; §16.1.
[2] Don't try to redefine standard library facilities; §16.1.2.
[3] Don't believe that the standard library is best for everything.
[4] When building a new facility, consider whether it can be presented within the framework offered by the standard library; §16.3.
[5] Remember that standard library facilities are defined in namespace ***std***; §16.1.2.
[6] Declare standard library facilities by including its header, not by explicit declaration; §16.1.2.
[7] Take advantage of late abstraction; §16.2.1.
[8] Avoid fat interfaces; §16.2.2.
[9] Prefer algorithms with reverse iterators over explicit loops dealing with reverse order; §16.3.2.
[10] Use ***base***( ) to extract an ***iterator*** from a ***reverse_iterator***; §16.3.2.
[11] Pass containers by reference; §16.3.4.
[12] Use iterator types, such as ***list<char>::iterator***, rather than pointers to refer to elements of a container; §16.3.1.
[13] Use ***const*** iterators where you don't need to modify the elements of a container; §16.3.1.
[14] Use ***at***( ), directly or indirectly, if you want range checking; §16.3.3.
[15] Use ***push_back***( ) or ***resize***( ) on a container rather than ***realloc***( ) on an array; §16.3.5.
[16] Don't use iterators into a resized ***vector***; §16.3.8.
[17] Use ***reserve***( ) to avoid invalidating iterators; §16.3.8.
[18] When necessary, use ***reserve***( ) to make performance predictable; §16.3.8.

16.5 Exercises [org.exercises]

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

1. (∗1.5) Create a ***vector<char>*** containing the letters of the alphabet in order. Print the elements of that vector in order and in reverse order.
2. (∗1.5) Create a ***vector<string>*** and read a list of names of fruits from ***cin*** into it. Sort the list and print it.
3. (∗1.5) Using the ***vector*** from §16.5[2], write a loop to print the names of all fruits with the initial letter ***a***.
4. (∗1) Using the ***vector*** from §16.5[2], write a loop to delete all fruits with the initial letter ***a***.
5. (∗1) Using the ***vector*** from §16.5[2], write a loop to delete all citrus fruits.
6. (∗1.5) Using the ***vector*** from §16.5[2], write a loop to delete all fruits that you don't like.
7. (∗2) Complete the ***Vector***, ***List***, and ***Itor*** classes from §16.2.1.
8. (∗2.5) Given an ***Itor*** class, consider how to provide iterators for forwards iteration, backwards iteration, iteration over a container that might change during an iteration, and iteration over an immutable container. Organize this set of containers so that a user can interchangeably use iterators that provide sufficient functionality for an algorithm. Minimize replication of effort in the implementation of the containers. What other kinds of iterators might a user need? List the strengths and weaknesses of your approach.

9. (∗2) Complete the ***Container***, ***Vector***, and ***List*** classes from §16.2.2.
10. (∗2.5) Generate 10,000 uniformly distributed random numbers in the range 0 to 1,023 and store them in (a) an standard library ***vector***, (b) a ***Vector*** from §16.5[7], and (3) a ***Vector*** from §16.5[9]. In each case, calculate the arithmetic mean of the elements of the vector (as if you didn't know it already). Time the resulting loops. Estimate, measure, and compare the memory consumption for the three styles of vectors.
11. (∗1.5) Write an iterator to allow ***Vector*** from §16.2.2 to be used as a container in the style of §16.2.1.
12. (∗1.5) Write a class derived from ***Container*** to allow ***Vector*** from §16.2.1 to be used as a container in the style of §16.2.2.
13. (∗2) Write classes to allow ***Vector*** from §16.2.1 and ***Vector*** from §16.2.2 to be used as standard containers.
14. (∗2) Write a template that implements a container with the same member functions and member types as the standard ***vector*** for an existing (nonstandard, non-student-exercise) container type. Do not modify the (pre)existing container type. How would you deal with functionality offered by the nonstandard ***vector*** but not by the standard ***vector***?
15. (∗1.5) Outline the possible behavior of ***duplicate_elements()*** from §16.3.6 for a ***vector<string>*** with the three elements ***don´t do this***.