19

Iterators and Allocators

The reason that data structures and algorithms
can work together seamlessly is ... that they
do not know anything about each other.
– Alex Stepanov

Iterators and sequences — operations on iterators — iterator traits — iterator categories — inserters — reverse iterators — stream iterators — checked iterators — exceptions and algorithms — allocators — the standard ***allocator*** — user-defined allocators — low-level memory functions — advice — exercises.

19.1 Introduction [iter.intro]

Iterators are the glue that holds containers and algorithms together. They provide an abstract view of data so that the writer of an algorithm need not be concerned with concrete details of a myriad of data structures. Conversely, the standard model of data access provided by iterators relieves containers from having to provide a more extensive set of access operations. Similarly, allocators are used to insulate container implementations from details of access to memory.

Iterators support an abstract model of data as sequences of objects (§19.2). Allocators provide a mapping from a lower-level model of data as arrays of bytes into the higher-level object model (§19.4). The most common lower-level memory model is itself supported by a few standard functions (§19.4.4).

Iterators are a concept with which every programmer should be familiar. In contrast, allocators are a support mechanism that a programmer rarely needs to worry about and few programmers will ever need to write a new allocator.

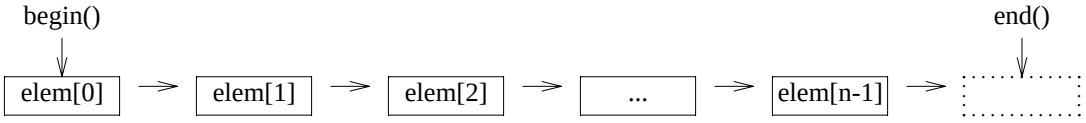
19.2 Iterators and Sequences [iter.iter]

An iterator is a pure abstraction. That is, anything that behaves like an iterator is an iterator (§3.8.2). An iterator is an abstraction of the notion of a pointer to an element of a sequence. Its key concepts are

- ''the element currently pointed to'' (dereferencing, represented by operators `*` and `->`),
- ''point to next element'' (increment, represented by operator `++`), and
- equality (represented by operator `==`).

For example, the built-in type ***int****** is an iterator for an ***int*`[]` and the class ***list*<*int*>`::`*iterator*** is an iterator for a ***list*** class.

A sequence is an abstraction of the notion ''something where we can get from the beginning to the end by using a next-element operation:''



Examples of such sequences are arrays (§5.2), vectors (§16.3), singly-linked lists (§17.8[17]), doubly-linked lists (§17.2.2), trees (§17.4.1), input (§21.3.1), and output (§21.2.1). Each has its own appropriate kind of iterator.

The iterator classes and functions are declared in namespace ***std*** and found in **<*iterator*>**.

An iterator is *not* a general pointer. Rather, it is an abstraction of the notion of a pointer into an array. There is no concept of a ''null iterator.'' The test to determine whether an iterator points to an element or not is conventionally done by comparing it against the *end* of its sequence (rather than comparing it against a ***null*** element). This notion simplifies many algorithms by removing the need for a special end case and generalizes nicely to sequences of arbitrary types.

An iterator that points to an element is said to be *valid* and can be dereferenced (using `*`, `[]`, or `->` appropriately). An iterator can be invalid either because it hasn't been initialized, because it pointed into a container that was explicitly or implicitly resized (§16.3.6, §16.3.8), because the container into which it pointed was destroyed, or because it denotes the end of a sequence (§18.2). The end of a sequence can be thought of as an iterator pointing to a hypothetical element position one-past-the-last element of a sequence.

19.2.1 Iterator Operations [iter.oper]

Not every kind of iterator supports exactly the same set of operations. For example, reading requires different operations from writing, and a ***vector*** allows convenient and efficient random access in a way that would be prohibitively expensive to provide for a ***list*** or an ***istream***. Consequently, we classify iterators into five categories according to the operations they are capable of providing efficiently (that is, in constant time; §17.1):

Iterator Operations and Categories					
Category:	output	input	forward	bidirectional	random-access
Abbreviation:	***Out***	***In***	***For***	***Bi***	***Ran***
Read:		`=*p`	`=*p`	`=*p`	`=*p`
Access:		`->`	`->`	`->`	`-> []`
Write:	`*p=`		`*p=`	`*p=`	`*p=`
Iteration:	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Comparison:		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

Both read and write are through the iterator dereferenced by `*`:

```
*p = x;    // write x through p
x = *p;    // read through p into x
```

To be an iterator type, a type must provide an appropriate set of operations. These operations must have their conventional meanings. That is, each operation must have the same effect it has on an ordinary pointer.

Independently of its category, an iterator can allow ***const*** or non-***const*** access to the object it points to. You cannot write to an element using an iterator to ***const*** – whatever its category. An iterator provides a set of operators, but the type of the element pointed to is the final arbiter of what can be done to that element.

Reads and writes copy objects, so element types must have the conventional copy semantics (§17.1.4).

Only random-access iterators can have an integer added or subtracted for relative addressing. However, except for output iterators, the distance between two iterators can always be found by iterating through the elements, so a ***distance()*** function is provided:

```
template<class In> typename iterator_traits<In>::difference_type distance(In first, In last)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++!=last) d++;
    return d;
}
```

An ***iterator_traits<In>::difference_type*** is defined for every iterator ***In*** to hold distances between elements (§19.2.2).

This function is called ***distance()*** rather than ***operator-()*** because it can be expensive and the operators provided for an iterator all operate in constant time (§17.1). Counting elements one by one is not the kind of operation I would like to invoke unwittingly for a large sequence. The library also provides a far more efficient implementation of ***distance()*** for a random-access iterator.

Similarly, ***advance()*** is provided as a potentially slow `+=`:

```
template <class In, class Dist> void advance(In i, Dist n);    // i+=n
```

19.2.2 Iterator Traits [iter.traits]

We use iterators to gain information about the objects they point to and the sequences they point into. For example, we can dereference an iterator and manipulate the resulting object and we can find the number of elements in a sequence, given the iterators that describe it. To express such operations, we must be able to refer to types related to an iterator such as ''the type of the object referred to by an iterator'' and ''the type of the distance between two iterators.'' The related types of an iterator are described by a small set of declarations in an ***iterator_traits*** template class:

```
template<class Iter> struct iterator_traits {
    typedef typename Iter::iterator_category iterator_category;      // §19.2.3
    typedef typename Iter::value_type value_type;                    // type of element
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::pointer pointer;                  // return type of operator–>()
    typedef typename Iter::reference reference;              // return type of operator*()
};
```

The ***difference_type*** is the type used to represent the difference between two iterators, and the ***iterator_category*** is a type indicating what operations the iterator supports. For ordinary pointers, specializations (§13.5) for **<*T**>** and **<*const T**>** are provided. In particular:

```
template<class T> struct iterator_traits<T*> {          // specialization for pointers
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T& reference;
};
```

That is, the difference between two pointers is represented by the standard library type ***ptrdiff_t*** from **<*cstddef*>** (§6.2.1) and a pointer provides random access (§19.2.3). Given ***iterator_traits***, we can write code that depends on properties of an iterator parameter. The ***count***( ) algorithm is the classical example:

```
template<class In, class T>
typename iterator_traits<In>::difference_type count(In first, In last, const T& val)
{
    typename iterator_traits<In>::difference_type res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

Here, the type of the result is expressed in terms of the ***iterator_traits*** of the input. This technique is necessary because there is no language primitive for expressing an arbitrary type in terms of another.

Instead of using ***iterator_traits***, we might have specialized ***count***( ) for pointers:

```
template<class In, class T>
typename In::difference_type count(In first, In last, const T& val);

template<class In, class T> ptrdiff_t count<T*,T>(T* first, T* last, const T& val);
```

However, this would have solved the problem for ***count***`()` only. Had we used this technique for a dozen algorithms, the information about distance types would have been replicated a dozen times. In general, it is better to represent a design decision in one place (§23.4.2). In that way, the decision can – if necessary – be changed in one place.

Because ***iterator_traits<Iterator>*** is defined for every iterator, we implicitly define an ***iterator_traits*** whenever we design a new iterator type. If the default traits generated from the general ***iterator_traits*** template are not right for our new iterator type, we provide a specialization in a way similar to what the standard library does for pointer types. The ***iterator_traits*** that are implicitly generated assume that the iterator is a class with the member types ***difference_type***, ***value_type***, etc. In ***<iterator>***, the library provides a base type that can be used to define those member types:

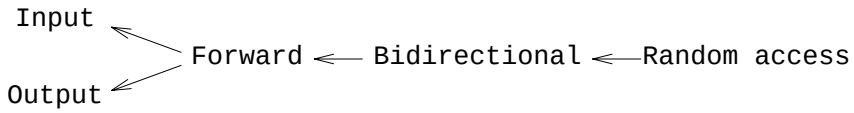
```
template<class Cat, class T, class Dist = ptrdiff_t, class Ptr = T*, class Ref = T&>
struct iterator {
    typedef Cat iterator_category;  // §19.2.3
    typedef T value_type;           // type of element
    typedef Dist difference_type;   // type of iterator difference
    typedef Ptr pointer;            // return type for ->
    typedef Ref reference;          // return type for *
};
```

Note that ***reference*** and ***pointer*** are not iterators. They are intended to be the return types of ***operator****`()` and ***operator***`->()`, respectively, for some iterator.

The ***iterator_traits*** are the key to the simplicity of many interfaces that rely on iterators and to the efficient implementation of many algorithms.

19.2.3 Iterator Categories [iter.cat]

The different kinds of iterators – usually referred to as iterator categories – fit into a hierarchical ordering:



This is not a class inheritance diagram. An iterator category is a classification of a type based on the operations it provides. Many otherwise unrelated types can belong to the same iterator category. For example, both ordinary pointers (§19.2.2) and ***Checked_iter***s (§19.3) are random-access iterators.

As noted in Chapter 18, different algorithms require different kinds of iterators as arguments. Also, the same algorithm can sometimes be implemented with different efficiencies for different kinds of iterators. To support overload resolution based on iterator categories, the standard library provides five classes representing the five iterator categories:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

Looking at the operations supported by input and forward iterators (§19.2.1), we would expect ***forward_iterator_tag*** to be derived from ***output_iterator_tag*** as well as from ***input_iterator_tag***. The reasons that it is not are obscure and probably invalid. However, I have yet to see an example in which that derivation would have simplified real code.

The inheritance of tags is useful (only) to save us from defining separate versions of a function where several – but not all – kinds of iterators can use the same algorithms. Consider how to implement ***distance***:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last);
```

There are two obvious alternatives:

[1] If ***In*** is a random-access iterator, we can subtract ***first*** from ***last***.

[2] Otherwise, we must increment an iterator from ***first*** to ***last*** and count the distance.

We can express these two alternatives as a pair of helper functions:

```
template<class In>
typename iterator_traits<In>::difference_type
dist_helper(In first, In last, input_iterator_tag)
{
    typename iterator_traits<In>::difference_type d = 0;
    while (first++!=last) d++;                          // use increment only
    return d;
}

template<class Ran>
typename iterator_traits<Ran>::difference_type
dist_helper(Ran first, Ran last, random_access_iterator_tag)
{
    return last-first;      // rely on random access
}
```

The iterator category tag arguments make it explicit what kind of iterator is expected. The iterator tag is used exclusively for overload resolution; the tag takes no part in the actual computation. It is a purely compile-time selection mechanism. In addition to automatic selection of a helper function, this technique provides immediate type checking (§13.2.5).

It is now trivial to define ***distance()*** by calling the appropriate helper function:

```
template<class In>
typename iterator_traits<In>::difference_type distance(In first, In last)
{
    return dist_helper(first,last,iterator_traits<In>::iterator_category());
}
```

For a *dist_helper*( ) to be called, the *iterator_traits<In>::iterator_category* used must be a *input_iterator_tag* or a *random_access_iterator_tag*. However, there is no need for separate versions of *dist_helper*( ) for forward or bidirectional iterators. Thanks to tag inheritance, those cases are handled by the *dist_helper*( ) which takes an *input_iterator_tag*. The absence of a version for *output_iterator_tag* reflects the fact that *distance*( ) is not meaningful for output iterators:

```
void f(vector<int>& vi,
     list<double>& ld,
     istream_iterator<string>& is1, istream_iterator<string>& is2,
     ostream_iterator<char>& os1, ostream_iterator<char>& os2)
{
     distance(vi.begin(),vi.end());          // use subtraction algorithm
     distance(ld.begin(),ld.end());          // use increment algorithm
     distance(is1,is2);                      // use increment algorithm
     distance(os1,os2);   // error: wrong iterator category, dist_helper() argument type mismatch
}
```

Calling *distance*( ) for an *istream_iterator* probably doesn't make much sense in a real program, though. The effect would be to read the input, throw it away, and return the number of values thrown away.

Using *iterator_traits<T>::iterator_category* allows a programmer to provide alternative implementations so that a user who cares nothing about the implementation of algorithms automatically gets the most appropriate implementation for each data structure used. In other words, it allows us to hide an implementation detail behind a convenient interface. Inlining can be used to ensure that this elegance is not bought at the cost of run-time efficiency.

19.2.4 Inserters [iter.insert]

Producing output through an iterator into a container implies that elements following the one pointed to by the iterator can be overwritten. This implies the possibility of overflow and consequent memory corruption. For example:

```
void f(vector<int>& vi)
{
     fill_n(vi.begin(),200,7);       // assign 7 to vi[0]..[199]
}
```

If *vi* has fewer than *200* elements, we are in trouble.

In <*iterator*>, the standard library provides three iterator template classes to deal with this problem, plus three functions to make it convenient to use those iterators:

```
template <class Cont> back_insert_iterator<Cont> back_inserter(Cont& c);
template <class Cont> front_insert_iterator<Cont> front_inserter(Cont& c);
template <class Cont, class Out> insert_iterator<Cont> inserter(Cont& c, Out p);
```

The *back_inserter*( ) causes elements to be added to the end of the container, *front_inserter*( ) causes elements to be added to the front, and ''plain'' *inserter*( ) causes elements to be added before its iterator argument. For *inserter*(*c*,*p*), *p* must be a valid iterator for *c*. Naturally, a container grows each time a value is written to it through an insert iterator.

When written to, an inserter inserts a new element into a sequence using ***push_back***( ), ***push_front***( ), or ***insert***( ) (§16.3.6) rather than overwriting an existing element. For example:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi),200,7);    // add 200 7s to the end of vi
}
```

Inserters are as simple and efficient as they are useful. For example:

```
template <class Cont>
class insert_iterator : public iterator<output_iterator_tag,void,void,void,void> {
protected:
    Cont& container;                    // container to insert into
    typename Cont::iterator iter;       // points into the container
public:
    explicit insert_iterator(Cont& x, typename Cont::iterator i)
        : container(x), iter(i) {}

    insert_iterator& operator=(const typename Cont::value_type& val)
    {
        iter = container.insert(iter,val);
        ++iter;
        return *this;
    }

    insert_iterator& operator*() { return *this; }
    insert_iterator& operator++() { return *this; }         // prefix ++
    insert_iterator operator++(int) { return *this; }       // postfix ++
};
```

Clearly, inserters are output iterators.

An ***insert_iterator*** is a special case of an output sequence. In parallel to the ***iseq*** from §18.3.1, we might define:

```
template<class Cont>
insert_iterator<Cont>
oseq(Cont& c, typename Cont::iterator first, typename Cont::iterator last)
{
    return insert_iterator<Cont>(c,c.erase(first,last)); // erase is explained in §16.3.6
}
```

In other words, an output sequence removes its old elements and replaces them with the output. For example:

```
void f(list<int>& li, vector<int>& vi)  // replace second half of vi by a copy of li
{
    copy(li.begin(),li.end(),oseq(vi,vi+vi.size()/2,vi.end()));
}
```

The container needs to be an argument to an ***oseq*** because it is not possible to decrease the size of a container, given only iterators into it (§18.6, §18.6.3).

19.2.5 Reverse Iterators [iter.reverse]

The standard containers provide *rbegin*() and *rend*() for iterating through elements in reverse order (§16.3.2). These member functions return *reverse_iterator*s:

```
template <class Iter>
class reverse_iterator : public iterator<iterator_traits<Iter>::iterator_category,
                                         iterator_traits<Iter>::value_type,
                                         iterator_traits<Iter>::difference_type,
                                         iterator_traits<Iter>::pointer,
                                         iterator_traits<Iter>::reference> {
protected:
    Iter current;    // current points to the element after the one *this refers to.
public:
    typedef Iter iterator_type;

    reverse_iterator() : current() { }
    explicit reverse_iterator(Iter x) : current(x) { }
    template<class U> reverse_iterator(const reverse_iterator<U>& x) : current(x.base()) { }

    Iter base() const { return current; } // current iterator value

    reference operator*() const { Iter tmp = current; return *--tmp; }
    pointer operator->() const;
    reference operator[](difference_type n) const;

    reverse_iterator& operator++() { --current; return *this; }      // note: not ++
    reverse_iterator operator++(int) { reverse_iterator t = current; --current; return t; }
    reverse_iterator& operator--() { ++current; return *this; }      // note: not --
    reverse_iterator operator--(int) { reverse_iterator t = current; ++current; return t; }

    reverse_iterator operator+(difference_type n) const;
    reverse_iterator& operator+=(difference_type n);
    reverse_iterator operator-(difference_type n) const;
    reverse_iterator& operator-=(difference_type n);
};
```

A *reverse_iterator* is implemented using an *iterator* called *current*. That *iterator* can (only) point to the elements of its sequence plus its one-past-the-end element. However, the *reverse_iterator*'s one-past-the-end element is the original sequence's (inaccessible) one-before-the-beginning element. Thus, to avoid access violations, *current* points to the element after the one the *reverse_iterator* refers to. This implies that * returns the value *(*current*-1) and that ++ is implemented using -- on *current*.

A *reverse_iterator* supports the operations that its initializer supports (only). For example:

```
void f(vector<int>& v, list<char>& lst)
{
    reverse_iterator(v.end())[3] = 7;          // ok: random-access iterator
    reverse_iterator(lst.end())[3] = '4';      // error: bidirectional iterator doesn't support []
    *(++++++reverse_iterator(lst.end())) = '4'; // ok!
}
```

In addition, the library provides ==, !=, <, <=, >, >=, + and - for *reverse_iterator*s.

19.2.6 Stream Iterators [iter.stream]

Ordinarily, I/O is done using the streams library (Chapter 21), a graphical user-interface system (not covered by the C++ standard), or the C I/O functions (§21.8). These I/O interfaces are primarily aimed at reading and writing individual values of a variety of types. The standard library provides four iterator types to fit stream I/O into the general framework of containers and algorithms:

- ***ostream_iterator***: for writing to an ***ostream*** (§3.4, §21.2.1).
- ***istream_iterator***: for reading from an ***istream*** (§3.6, §21.3.1).
- ***ostreambuf_iterator***: for writing to a stream buffer (§21.6.1).
- ***istreambuf_iterator***: for reading from a stream buffer (§21.6.2).

The idea is simply to present input and output of collections as sequences:

```
template <class T, class Ch = char, class Tr = char_traits<Ch> >
class ostream_iterator : public iterator<output_iterator_tag, void, void, void, void> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_ostream<Ch, Tr> ostream_type;

    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const Ch* delim); // write delim after each output value
    ostream_iterator(const ostream_iterator&);
    ~ostream_iterator();

    ostream_iterator& operator=(const T& val);          // write val to output

    ostream_iterator& operator*();
    ostream_iterator& operator++();
    ostream_iterator& operator++(int);
};
```

This iterator accepts the usual write and increment operations of an output iterator and converts them into output operations on an ***ostream***. For example:

```
void f()
{
    ostream_iterator<int> os(cout);     // write ints to cout through os
    *os = 7;        // output 7
    ++os;           // get ready for next output
    *os = 79;       // output 79
}
```

The ++ operation might trigger an actual output operation, or it might have no effect. Different implementations will use different implementation strategies. Consequently, for code to be portable a ++ must occur between every two assignments to an ***ostream_iterator***. Naturally, every standard algorithm is written that way – or it would not work for a ***vector***. This is why ***ostream_iterator*** is defined this way.

An implementation of ***ostream_iterator*** is trivial and is left as an exercise (§19.6[4]). The standard I/O supports different character types; ***char_traits*** (§20.2) describes the aspects of a character type that can be important for I/O and ***string***s.

An input iterator for ***istream***s is defined analogously:

```
template <class T, class Ch = char, class Tr = char_traits<Ch>, class Dist = ptrdiff_t>
class istream_iterator : public iterator<input_iterator_tag, T, Dist, const T*, const T&> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef basic_istream<Ch,Tr> istream_type;

    istream_iterator();                          // end of input
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator&);
    ~istream_iterator();

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator& operator++();
    istream_iterator operator++(int);
};
```

This iterator is specified so that what would be conventional use for a container triggers input from an ***istream***. For example:

```
void f()
{
    istream_iterator<int> is(cin);          // read ints from cin through is
    int i1 = *is;           // read an int
    ++is;                   // get ready for next input
    int i2 = *is;           // read an int
}
```

The default ***istream_iterator*** represents the end of input so that we can specify an input sequence:

```
void f(vector<int>& v)
{
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));
}
```

To make this work, the standard library supplies `==` and `!=` for ***istream_iterator***s.

An implementation of ***istream_iterator*** is less trivial than an ***ostream_iterator*** implementation, but it is still simple. Implementing an ***istream_iterator*** is also left as an exercise (§19.6[5]).

19.2.6.1 Stream Buffers [iter.streambuf]

As described in §21.6, stream I/O is based on the idea of ***ostream***s and ***istream***s filling and emptying buffers from and to which the low-level physical I/O is done. It is possible to bypass the standard iostreams formatting and operate directly on the stream buffers (§21.6.4). That ability is also provided to algorithms through the notion of ***istreambuf_iterator***s and ***ostreambuf_iterator***s:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator
    : public iterator<input_iterator_tag, Ch, typename Tr::off_type, Ch*, Ch&> {
public:
    typedef Ch char_type;
    typedef Tr traits_type;
    typedef typename Tr::int_type int_type;
    typedef basic_streambuf<Ch, Tr> streambuf_type;
    typedef basic_istream<Ch, Tr> istream_type;

    class proxy;                                    // helper type

    istreambuf_iterator() throw();                  // end of buffer
    istreambuf_iterator(istream_type& is) throw();  // read from is's streambuf
    istreambuf_iterator(streambuf_type*) throw();
    istreambuf_iterator(const proxy& p) throw();    // read from p's streambuf

    Ch operator*() const;
    istreambuf_iterator& operator++();              // prefix
    proxy operator++(int);                          // postfix

    bool equal(istreambuf_iterator&);               // both or neither streambuf at eof
};
```

In addition, == and != are supplied.

Reading from a ***streambuf*** is a lower-level operation than reading from an ***istream***. Consequently, the ***istreambuf_iterator*** interface is messier than the ***istream_iterator*** interface. However, once the ***istreambuf_iterator*** is properly initialized, *, ++, and = have their usual meanings when used in the usual way.

The ***proxy*** type is an implementation-defined helper type that allows the postfix ++ to be implemented without imposing constraints on the ***streambuf*** implementation. A ***proxy*** holds the result value while the iterator is incremented:

```
template<class Ch, class Tr = char_traits<Ch> >
class istreambuf_iterator<Ch, Tr>::proxy {
    Ch val;
    basic_istreambuf<Ch, Tr>* buf;

    proxy(Ch v, basic_istreambuf<Ch, Tr>* b) :val(v), buf(b) { }
public:
    Ch operator*() { return val; }
};
```

An ***ostreambuf_iterator*** is defined similarly:

```
template <class Ch, class Tr = char_traits<Ch> >
class ostreambuf_iterator : public iterator<output_iterator_tag, void, void, void, void>{
public:
    typedef Ch char_type;
    typedef Tr traits_type;
```

```
    typedef basic_streambuf<Ch,Tr> streambuf_type;
    typedef basic_ostream<Ch,Tr> ostream_type;

    ostreambuf_iterator(ostream_type& os) throw();      // write to os's streambuf
    ostreambuf_iterator(streambuf_type*) throw();
    ostreambuf_iterator& operator=(Ch);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);

    bool failed() const throw();                        // true if Tr::eof() seen
};
```

19.3 Checked Iterators [iter.checked]

A programmer can provide iterators in addition to those provided by the standard library. This is often necessary when providing a new kind of container, and sometimes a new kind of iterator is a good way to support a different way of using existing containers. As an example, I here describe an iterator that range checks access to its container.

Using standard containers reduces the amount of explicit memory management. Using standard algorithms reduces the amount of explicit addressing of elements in containers. Using the standard library together with language facilities that maintain type safety dramatically reduces run-time errors compared to traditional C coding styles. However, the standard library still relies on the programmer to avoid access beyond the limits of a container. If by accident element `x[x.size()+7]` of some container `x` is accessed, then unpredictable – and usually bad – things happen. Using a range-checked *vector*, such as *Vec* (§3.7.1), helps in some cases. More cases can be handled by checking every access through an iterator.

To achieve this degree of checking without placing a serious notational burden on the programmer, we need checked iterators and a convenient way of attaching them to containers. To make a *Checked_iter*, we need a container and an iterator into that container. As for binders (§18.4.4.1), inserters (§19.2.4), etc., I provide functions for making a *Checked_iter*:

```
template<class Cont, class Iter> Checked_iter<Cont,Iter> make_checked(Cont& c, Iter i)
{
    return Checked_iter<Cont,Iter>(c,i);
}

template<class Cont> Checked_iter<Cont,typename Cont::iterator> make_checked(Cont& c)
{
    return Checked_iter<Cont,typename Cont::iterator>(c,c.begin());
}
```

These functions offer the notational convenience of deducing the types from arguments rather than stating those types explicitly. For example:

```
void f(vector<int>& v, const vector<int>& vc)
{
    typedef Checked_iter<vector<int>,vector<int>::iterator> CI;
    CI p1 = make_checked(v,v.begin()+3);
    CI p2 = make_checked(v);            // by default: point to first element

    typedef Checked_iter<const vector<int>,vector<int>::const_iterator> CIC;
    CIC p3 = make_checked(vc,vc.begin()+3);
    CIC p4 = make_checked(vc);

    const vector<int>& vv = v;
    CIC p5 = make_checked(v,vv.begin());
}
```

By default, ***const*** containers have ***const*** iterators, so their ***Checked_iter***s must also be constant iterators. The iterator ***p5*** shows one way of getting a ***const*** iterator for a non-***const*** iterator.

This demonstrates why ***Checked_iter*** needs two template parameters: one for the container type and one to express the ***const***/non-***const*** distinction.

The names of these ***Checked_iter*** types become fairly long and unwieldy, but that doesn't matter when iterators are used as arguments to a generic algorithm. For example:

```
template<class Iter> void mysort(Iter first, Iter last);

void f(vector<int>& c)
{
    try {
        mysort(make_checked(c), make_checked(c,c.end()));
    }
    catch (out_of_bounds) {
        cerr<<"oops: bug in mysort()\n";
        abort();
    }
}
```

An early version of such an algorithm is exactly where I would most suspect a range error so that using checked iterators would make sense.

The representation of a ***Checked_iter*** is a pointer to a container plus an iterator pointing into that container:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    Iter curr; // iterator for current position
    Cont* c;   // pointer to current container

    // ...
};
```

Deriving from ***iterator_traits*** is one technique for defining the desired ***typedef***s. The obvious alternative – deriving from ***iterator*** – would be verbose in this case (as it was for ***reverse_iterator***; §19.2.5). Just as there is no requirement that an iterator should be a class, there is no requirement that iterators that are classes should be derived from ***iterator***.

The *Checked_iter* operations are all fairly trivial:

```
template<class Cont, class Iter = typename Cont::iterator>
class Checked_iter : public iterator_traits<Iter> {
    // ...
public:
    void valid(Iter p)
    {
        if (c->end() == p) return;
        for (Iter pp = c->begin(); pp!=c->end(); ++pp) if (pp == p) return;
        throw out_of_bounds()
    }

    friend bool operator==(const Checked_iter& i, const Checked_iter& j)
    {
        return i.c==j.c && i.curr==j.curr;
    }

    // no default initializer.

    // use default copy constructor and copy assignment.

    Checked_iter(Cont& x, Iter p) : c(&x), curr(p) { valid(p); }

    reference_type operator*()
    {
        if (curr==c->end()) throw out_of_bounds();
        return *curr;
    }

    pointer_type operator->()
    {
        return &*curr;                          // checked by *
    }

    Checked_iter operator+(Dist d)              // for random-access iterators only
    {
        if (c->end()-curr<=d) throw out_of_bounds();
        return Checked_iter(c,curr+d);
    }

    reference_type operator[](Dist d)           // for random-access iterators only
    {
        if (c->end()-curr<=d) throw out_of_bounds();
        return c[d];
    }

    Checked_iter& operator++()                  // prefix ++
    {
        if (curr == c->end()) throw out_of_bounds();
        ++curr;
        return *this;
    }
```

```
    Checked_iter operator++(int)                    // postfix ++
    {
        Checked_iter tmp = *this;
        ++*this;                                    // checked by prefix ++
        return tmp;
    }

    Checked_iter& operator--()                      // prefix --
    {
        if (curr == c->begin()) throw out_of_bounds();
        --curr;
        return *this;
    }

    Checked_iter operator--(int)                    // postfix --
    {
        Checked_iter tmp = *this;
        --*this;                                    // checked by prefix --
        return tmp;
    }

    difference_type index() { return curr-c.begin(); }  // random-access only

    Iter unchecked() { return curr; }

    // +, -, < , etc. (§19.6[6])
};
```

A ***Checked_iter*** can be initialized only for a particular iterator pointing into a particular container. In a full-blown implementation, a more efficient version of ***valid()*** should be provided for random-access iterators (§19.6[6]). Once a ***Checked_iter*** is initialized, every operation that changes its position is checked to make sure the iterator still points into the container. An attempt to make the iterator point outside the container causes an ***out_of_bounds*** exception to be thrown. For example:

```
void f(list<string>& ls)
{
    int count = 0;
    try {
        Checked_iter< list<string> > p(ls, ls.begin());
        while (true) {
            ++p;        // sooner or later this will reach the end
            ++count;
        }
    }
    catch(out_of_bounds) {
        cout << "overrun after " << count << " tries\n";
    }
}
```

A ***Checked_iter*** knows which container it is pointing into. This allows it to catch some, but not all, cases in which iterators into a container have been invalidated by an operation on it (§16.3.8). To

protect against all such cases, a different and more expensive iterator design would be needed (see §19.6[7]).

Note that postincrement (postfix ++) involves a temporary and preincrement does not. For this reason, it is best to prefer *++p* over *p++* for iterators.

Because a ***Checked_iter*** keeps a pointer to a container, it cannot be used for a built-in array directly. When necessary, a ***c_array*** (§17.5.4) can be used.

To complete the notion of checked iterators, we must make them trivial to use. There are two basic approaches:

[1] Define a checked container type that behaves like other containers, except that it provides only a limited set of constructors and its ***begin***( ), ***end***( ), etc., supply ***Checked_iter***s rather than ordinary iterators.

[2] Define a handle that can be initialized by an arbitrary container and that provides checked access functions to its container (§19.6[8]).

The following template attaches checked iterators to a container:

```
template<class C> class Checked : public C {
public:
    explicit Checked(size_t n) :C(n) { }
    Checked() :C() { }

    typedef Checked_iter<C> iterator;
    typedef Checked_iter<C, C::const_iterator> const_iterator;

    typename C::iterator begin() { return iterator(*this, C::begin()); }
    typename C::iterator end() { return iterator(*this, C::end()); }

    typename C::const_iterator begin() const { return const_iterator(*this, C::begin()); }
    typename C::const_iterator end() const { return const_iterator(*this, C::end()); }

    typename C::reference_type operator[](size_t n) { return Checked_iter<C>(*this)[n]; }

    C& base() { return static_cast<C&>(*this); }  // get hold of the base container
};
```

This allows us to write:

```
Checked< vector<int> > vec(10);
Checked< list<double> > lst;

void f()
{
    int v1 = vec[5];            // ok
    int v2 = vec[15];           // throws out_of_bounds
    // ...
    lst.push_back(v2);
    mysort(vec.begin(), vec.end());
    copy(vec.begin(), vec.end(), lst.begin(), lst.end());
}
```

If a container is resized, iterators – including ***Checked_iter***s – into it may become invalid. In that case, the ***Checked_iter*** can be re-initialized:

```
void g()
{
    Checked_iter<int> p(vi);
    // ..
    int i = p.index();                      // get current position
    vi.resize(100);                         // p becomes invalid
    p = Checked_iter<int>(vi,vi.begin()+i); // restore current position
}
```

The old – and invalid – current position is lost. I provided ***index()*** as a means of storing and restoring a ***Checked_iter***. If necessary, a reference to the container used as the base of the ***Checked*** container can be extracted using ***base()***.

19.3.1 Exceptions, Containers, and Algorithms [iter.except]

You could argue that using both standard algorithms and checked iterators is like wearing both belt and suspenders: either should keep you safe. However, experience shows that for many people and for many applications a dose of paranoia is reasonable – especially during times when a program goes through frequent changes that involve several people.

One way of using run-time checks is to keep them in the code only while debugging. The checks are then removed before the program is shipped. This practice has been compared to wearing a life jacket while paddling around close to the shore and then removing it before setting out onto the open sea. However, some uses of run-time checks do impose significant time and space overheads, so insisting on such checks at all times is not realistic. In any case, it is unwise to optimize without measurements, so before removing checks, do an experiment to see if worthwhile improvements actually emerge from doing so. To do such an experiment, we must be able to remove run-time checks easily (see §24.3.7.1). Once measurements have been done, we could remove the run-time testing from the most run-time critical – and hopefully most thoroughly tested – code and leave the rest of the code checked as a relatively cheap form of insurance.

Using a ***Checked_iter*** allows us to detect many mistakes. It does not, however, make it easy to recover from these errors. People rarely write code that is 100% robust against every `++`, `--`, `*`, `[]`, `->`, and `=` potentially throwing an exception. This leaves us with two obvious strategies:

[1] Catch exceptions close to the point from which they are thrown so that the writer of the exception handler has a decent chance of knowing what went wrong and can take appropriate action.

[2] Catch the exception at a high level of a program, abandon a significant portion of a computation, and consider all data structures written to during the failed computation suspect (maybe there are no such data structures or maybe they can be sanity checked).

It is irresponsible to catch an exception from some unknown part of a program and proceed under the assumption that no data structure is left in an undesirable state, unless there is a further level of error handling that will catch subsequent errors. A simple example of this is when a final check (by computer or human) is done before the results are accepted. In such cases, it can be simpler and cheaper to proceed blithely rather than to try to catch every error at a low level. This would be an example of a simplification made possible by a multilevel error recovery scheme (§14.9).

19.4 Allocators [iter.alloc]

An *allocator* is used to insulate implementers of algorithms and containers that must allocate memory from the details of physical memory. An allocator provides standard ways of allocating and deallocating memory and standard names of types used as pointers and references. Like an iterator, an allocator is a pure abstraction. Any type that behaves like an allocator is an allocator.

The standard library provides a standard allocator intended to serve most users of a given implementation well. In addition, users can provide allocators that represent alternative views of memory. For example, we can write allocators that use shared memory, garbage-collected memory, memory from preallocated pools of objects (§19.4.2), etc.

The standard containers and algorithms obtain and access memory through the facilities provided by an allocator. Thus, by providing a new allocator we provide the standard containers with a way of using a new and different kind of memory.

19.4.1 The Standard Allocator [iter.alloc.std]

The standard *allocator* template from <*memory*> allocates memory using *operator new*() (§6.2.6) and is by default used by all standard containers:

```
template <class T> class allocator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    typedef T* pointer;
    typedef const T* const_pointer;

    typedef T& reference;
    typedef const T& const_reference;

    pointer address(reference r) const { return &r; }
    const_pointer address(const_reference r) const { return &r; }

    allocator() throw();
    template <class U> allocator(const allocator<U>&) throw();
    ~allocator() throw();

    pointer allocate(size_type n, allocator<void>::const_pointer hint = 0); // space for n Ts
    void deallocate(pointer p, size_type n);  // deallocate n Ts, don't destroy

    void construct(pointer p, const T& val) { new(p) T(val); }     // initialize *p by val
    void destroy(pointer p) { p->~T(); }            // destroy *p but don't deallocate

    size_type max_size() const throw();

    template <class U>
    struct rebind { typedef allocator<U> other; };  // in effect: typedef allocator<U> other
};

template<class T> bool operator==(const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(const allocator<T>&, const allocator<T>&) throw();
```

An *allocate(n)* operation allocates space for *n* objects that can be deallocated by a corresponding call of *deallocate(p,n)*. Note that *deallocate()* also takes a number-of-elements argument *n*. This allows for close-to-optimal allocators that maintain only minimal information about allocated memory. On the other hand, such allocators require that the user always provide the right *n* when they *deallocate()*.

The default *allocator* uses *operator new(size_t)* to obtain memory and *operator delete(void*)* to free it. This implies that the *new_handler()* might be called and *out_of_memory* might be thrown in case of memory exhaustion (§6.2.6.2).

Note that *allocate()* is not obliged to call a lower-level allocator each time. Often, a better strategy is for the allocator to maintain a free list of space ready to hand out with minimal time overhead (§19.4.2).

The optional *hint* argument to *allocate()* is completely implementation-dependent. However, it is intended as a help to allocators for systems where locality is important. For example, an allocator might try to allocate space for related objects on the same page in a paging system. The type of the *hint* argument is the *pointer* from the ultra-simplified specialization:

```
template<> class allocator<void> {
public:
    typedef void* pointer;
    typedef const void* const_pointer;
    // note: no reference
    typedef void value_type;
    template<class U>
    struct rebind { typedef allocator<U> other; }; // in effect: typedef allocator<U> other
};
```

The *allocator<void>::pointer* type acts as a universal pointer type and is *const void** for all standard allocators.

Unless the documentation for an allocator says otherwise, the user has two reasonable choices when calling *allocate()*:

[1] Don't give a hint.

[2] Use a pointer to an object that is frequently used together with the new object as the hint; for example, the previous element in a sequence.

Allocators are intended to save implementers of containers from having to deal with raw memory directly. As an example, consider how a *vector* implementation might use memory:

```
template<class T, class A = allocator<T> > class vector {
public:
    typedef typename A::pointer iterator;
    // ...
private:
    A alloc;        // allocator object
    iterator v;     // pointer to elements
    // ...
```

```
public:
    explicit vector(size_type n, const T& val = T(), const A& a = A())
        : alloc(a)
    {
        v = alloc.allocate(n);
        for(iterator p = v; p<v+n; ++p) alloc.construct(p,val);
        // ...
    }

    void reserve(size_type n)
    {
        if (n<=capacity()) return;

        iterator p = alloc.allocate(n);
        iterator q = v;

        while (q<v+size()) {                    // copy existing elements
            alloc.construct(p++,*q);
            alloc.destroy(q++);
        }
        alloc.deallocate(v,capacity());         // free old space
        v = p;
        // ...
    }

    // ...
};
```

The ***allocator*** operations are expressed in terms of ***pointer*** and ***reference typedef***s to give the user a chance to supply alternative types for accessing memory. This is very hard to do in general. For example, it is not possible to define a perfect reference type within the C++ language. However, language and library implementers can use these ***typedef***s to support types that couldn't be provided by an ordinary user. An example would be an allocator that provided access to a persistent store. Another example would be a ''long'' pointer type for accessing main memory beyond what a default pointer (usually 32 bits) could address.

The ordinary user can supply an unusual pointer type to an allocator for specific uses. The equivalent cannot be done for references, but that may be an acceptable constraint for an experiment or a specialized system.

An allocator is designed to make it easy to handle objects of the type specified by its template parameter. However, most container implementations require objects of additional types. For example, the implementer of a ***list*** will need to allocate ***Link*** objects. Usually, such ***Link***s must be allocated using their ***list***'s allocator.

The curious ***rebind*** type is provided to allow an allocator to allocate objects of arbitrary type. Consider:

```
typedef typename A::rebind<Link>::other Link_alloc;
```

If ***A*** is an ***allocator***, then ***rebind<Link>::other*** is ***typedef***'d to mean ***allocator<Link>***, so the previous ***typedef*** is an indirect way of saying:

```
typedef allocator<Link> Link_alloc;
```

The indirection frees us from having to mention ***allocator*** directly. It expresses the ***Link_alloc*** type in terms of a template parameter ***A***. For example:

```
template <class T, class A = allocator<T> > class list {
private:
    class Link { /* ... */ };

    typedef typename A::rebind<Link>::other Link_alloc;     // allocator<Link>

    Link_alloc a;   // link allocator
    A alloc;        // list allocator
    // ...
public:
    typedef typename A::pointer iterator;
    // ...

    iterator insert(iterator pos, const T& x )
    {
        Link_alloc::pointer p = a.allocate(1);     // get a Link
        // ...
    }
    // ...
};
```

Because ***Link*** is a member of ***list***, it is parameterized by an allocator. Consequently, ***Link***s from ***list***s with different allocators are of different types, just like the ***list***s themselves (§17.3.3).

19.4.2 A User-Defined Allocator [iter.alloc.user]

Implementers of containers often ***allocate***( ) and ***deallocate***( ) objects one at a time. For a naive implementation of ***allocate***( ), this implies lots of calls of operator ***new***, and not all implementations of operator ***new*** are efficient when used like that. As an example of a user-defined allocator, I present a scheme for using pools of fixed-sized pieces of memory from which the allocator can ***allocate***( ) more efficiently than can a conventional and more general ***operator new***( ).

I happen to have a pool allocator that does approximately the right thing, but it has the wrong interface (because it was designed years before allocators were invented). This ***Pool*** class implements the notion of a pool of fixed-sized elements from which a user can do fast allocations and deallocations. It is a low-level type that deals with memory directly and worries about alignment:

```
class Pool {
    struct Link { Link* next; };

    struct Chunk {
        enum { size = 8*1024-16 };
        Chunk* next;
        char mem[size];
    };
    Chunk* chunks;

    const unsigned int esize;
    Link* head;
```

```
        Pool(Pool&);                // copy protection
        void operator=(Pool&);      // copy protection
        void grow();                // make pool larger
public:
        Pool(unsigned int n);       // n is the size of elements
        ~Pool();

        void* alloc();              // allocate one element
        void free(void* b);         // put an element back into the pool
};

inline void* Pool::alloc()
{
        if (head==0) grow();
        Link* p = head;             // return first element
        head = p->next;
        return p;
}

inline void Pool::free(void* b)
{
        Link* p = static_cast<Link*>(b);
        p->next = head;             // put b back as first element
        head = p;
}

Pool::Pool(unsigned int sz)
        : esize(sz<sizeof(Link*)?sizeof(Link*):sz)
{
        head = 0;
        chunks = 0;
}

Pool::~Pool() // free all chunks
{
        Chunk* n = chunks;
        while (n) {
                Chunk* p = n;
                n = n->next;
                delete p;
        }
}

void Pool::grow()   // allocate new 'chunk,' organize it as a linked list of elements of size 'esize'
{
        Chunk* n = new Chunk;
        n->next = chunks;
        chunks = n;

        const int nelem = Chunk::size/esize;
        char* start = n->mem;
        char* last = &start[(nelem-1)*esize];
```

```
        for (char* p = start; p<last; p+=esize)          // assume sizeof(Link)<=esize
                reinterpret_cast<Link*>(p)->next = reinterpret_cast<Link*>(p+esize);
        reinterpret_cast<Link*>(last)->next = 0;
        head = reinterpret_cast<Link*>(start);
}
```

To add a touch of realism, I'll use ***Pool*** unchanged as part of the implementation of my allocator, rather than rewrite it to give it the right interface. The pool allocator is intended for fast allocation and deallocation of single elements and that is what my ***Pool*** class supports. Extending this implementation to handle allocations of arbitrary numbers of objects and to objects of arbitrary size (as required by ***rebind***( )) is left as an exercise (§19.6[9]).

Given ***Pool***, the definition of ***Pool_alloc*** is trivial;

```
template <class T> class Pool_alloc {
private:
        static Pool mem;        // pool of elements of sizeof(T)
public:
        // like the standard allocator (§19.4.1)
};

template <class T> Pool Pool_alloc<T>::mem(sizeof(T));

template <class T> Pool_alloc<T>::Pool_alloc() { }

template <class T>
T* Pool_alloc<T>::allocate(size_type n, void* = 0)
{
        if (n == 1) return static_cast<T*>(mem.alloc());
        // ...
}

template <class T>
void Pool_alloc<T>::deallocate(pointer p, size_type n)
{
        if (n == 1) {
                mem.free(p);
                return;
        }
        // ...
}
```

This allocator can now be used in the obvious way:

```
vector<int,Pool_alloc> v;
map<string,number,Pool_alloc> m;

// use exactly as usual

vector<int> v2 = v;     // error: different allocator parameters
```

I chose to make the ***Pool*** for a ***Pool_alloc*** static because of a restriction that the standard library imposes on allocators used by the standard containers: the implementation of a standard container is allowed to treat every object of its allocator type as equivalent. This can lead to significant

performance advantages. For example, because of this restriction, memory need not be set aside for allocators in ***Link*** objects (which are typically parameterized by the allocator of the container for which they are ***Link***s; §19.4.1), and operations that may access elements of two sequences (such as ***swap***( )) need not check whether the objects manipulated all have the same allocator. However, the restriction does imply that such allocators cannot use per-object data.

Before applying this kind of optimization, make sure that it is necessary. I expect that many default ***allocator***s will implement exactly this kind of classic C++ optimization – thus saving you the bother.

19.4.3 Generalized Allocators [iter.general]

An ***allocator*** is a simplified and optimized variant of the idea of passing information to a container through a template parameter (§13.4.1, §16.2.3). For example, it makes sense to require that every element in a container is allocated by the container's allocator. However, if two ***list***s of the same type were allowed to have different allocators, then ***splice***( ) (§17.2.2.1) couldn't be implemented through relinking. Instead, ***splice***( ) would have to be defined in terms of copying of elements to protect against the rare cases in which we want to splice elements from a ***list*** with one allocator into another with a different allocator of the same allocator type. Similarly, if allocators were allowed to be perfectly general, the ***rebind*** mechanism that allows an allocator to allocate elements of arbitrary types would have to be more elaborate. Consequently, a standard allocator is assumed to hold no per-object data and an implementation of a standard may take advantage of that.

Surprisingly, the apparently Draconian restriction against per-object information in allocators is not particularly serious. Most allocators do not need per-object data and can be made to run faster without such data. Allocators can still hold data on a per-allocator-type basis. If separate data is needed, separate allocator types can be used. For example:

```
template<class T, class D> class My_alloc {     // allocator for T implemented using D
        D d;        // data needed for My_alloc<T,D>
        // ...
};

typedef My_alloc<int,Persistent_info> Persistent;
typedef My_alloc<int,Shared_info> Shared;
typedef My_alloc<int,Default_info> Default;

list<int,Persistent> lst1;
list<int,Shared> lst2;
list<int,Default> lst3;
```

The lists ***lst1***, ***lst2***, and ***lst3*** are of different types. Therefore, we must use general algorithms (Chapter 18) when operating on two of these lists rather than specialized list operations (§17.2.2.1). This implies that copying rather than relinking is done, so having different allocators poses no problems.

The restriction against per-object data in allocators is imposed because of the stringent demands on the run-time and space efficiency of the standard library. For example, the space overhead of allocator data for a list probably wouldn't be significant. However, it could be serious if each link of a list suffered overhead.

Consider how the allocator technique could be used when the efficiency constraints of the standard library don't apply. This would be the case for a nonstandard library that wasn't meant to deliver high performance for essentially every data structure and every type in a program and for some special-purpose implementations of the standard library. In such cases, an allocator can be used to carry the kind of information that often inhabits universal base classes (§16.2.2). For example, an allocator could be designed to answer requests about where its objects are allocated, present data representing object layout, and answer questions such as ''is this element in this container?'' It could also provide controls for a container that acts as a cache for memory in permanent storage, provide association between the container and other objects, etc.

In this way, arbitrary services can be provided transparently to the ordinary container operations. However, it is best to distinguish between issues relating to storage of data and issues of the use of data. The latter do not belong in a generalized allocator, but they could be provided through a separate template argument.

19.4.4 Uninitialized Memory [iter.memory]

In addition to the standard *allocator*, the *<memory>* header provides a few functions for dealing with uninitialized memory. They share the dangerous and occasionally essential property of using a type name ***T*** to refer to space sufficient to hold an object of type ***T*** rather than to a properly constructed object of type ***T***.

The library provides three ways to copy values into uninitialized space:

```
template <class In, class For>
For uninitialized_copy(In first, In last, For res)     // copy into res
{
     typedef typename iterator_traits<For>::value_type V;

     while (first != last)
          new (static_cast<void*>(&*res++)) V(*first++);      // construct in res (§10.4.11)
     return res;
}

template <class For, class T>
void uninitialized_fill(For first, For last, const T& val)
{
     typedef typename iterator_traits<For>::value_type V;

     while (first != last) new (static_cast<void*>(&*first++)) V(val);      // construct in first
}

template <class For, class Size, class T>
void uninitialized_fill_n(For first, Size n, const T& val)
{
     typedef typename iterator_traits<For>::value_type V;

     while (n--) new (static_cast<void*>(&*first++)) V(val); // construct in first
}
```

These functions are intended primarily for implementers of containers and algorithms. For example, *reserve*( ) and *resize*( ) (§16.3.8) are most easily implemented using these functions

(§19.6[10]). It would clearly be most unfortunate if an uninitialized object escaped from the internals of a container into the hands of general users.

Algorithms often require temporary space to perform acceptably. Often, such temporary space is best allocated in one operation but not initialized until a particular location is actually needed. Consequently, the library provides a pair of functions for allocating and deallocating uninitialized space:

```
template <class T> pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t); // allocate, don't initialize
template <class T> void return_temporary_buffer(T*);                  // deallocate, don't destroy
```

A ***get_temporary_buffer<X>(n)*** operation tries to allocate space for ***n*** or more objects of type ***X***. If it succeeds in allocating some memory, it returns a pointer to the first uninitialized space and the number of objects of type ***X*** that will fit into that space; otherwise, the ***second*** value of the pair is zero. The idea is that a system may keep a number of fixed-sized buffers ready for fast allocation so that requesting space for ***n*** objects may yield space for more than ***n***. It may also yield less, however, so one way of using ***get_temporary_buffer()*** is to optimistically ask for a lot and then use what happens to be available.

A buffer obtained by ***get_temporary_buffer()*** must be freed for other use by a call of ***return_temporary_buffer()***. Just as ***get_temporary_buffer()*** allocates without constructing, ***return_temporary_buffer()*** frees without destroying. Because ***get_temporary_buffer()*** is low-level and likely to be optimized for managing temporary buffers, it should not be used as an alternative to ***new*** or ***allocator::allocate()*** for obtaining longer-term storage.

The standard algorithms that write into a sequence assume that the elements of that sequence have been previously initialized. That is, the algorithms use assignment rather than copy construction for writing. Consequently, we cannot use uninitialized memory as the immediate target of an algorithm. This can be unfortunate because assignment can be significantly more expensive than initialization. Besides, we are not interested in the values we are about to overwrite anyway (or we wouldn't be overwriting them). The solution is to use a ***raw_storage_iterator*** from ***<memory>*** that initializes instead of assigns:

```
template <class Out, class T>
class raw_storage_iterator : public iterator<output_iterator_tag,void,void,void,void> {
    Out p;
public:
    explicit raw_storage_iterator(Out pp) : p(pp) { }

    raw_storage_iterator& operator*() { return *p; }
    raw_storage_iterator& operator=(const T& val)
    {
        T* pp = &*p;
        new(pp) T(val);      // place val in pp (§10.4.11)
        return p;
    }
    raw_storage_iterator& operator++() { return ++p; }
    raw_storage_iterator operator++(int) { return p++; }
};
```

For example, we might write a template that copies the contents of a ***vector*** into a buffer:

```
template<class T, class A> T* temporary_dup(vector<T,A>& v)
{
    T* p = get_temporary_buffer<T>(v.size()).first;
    if (p == 0) return 0;
    copy(v.begin(),v.end(),raw_storage_iterator<T*,T>(p));
    return p;
}
```

Had *new* been used instead of *get_temporary_buffer()*, initialization would have been done. Once initialization is avoided, the *raw_storage_iterator* becomes necessary for dealing with the uninitialized space. In this example, the caller of *temporary_dup()* is responsible for calling *destroy_temporary_buffer()* for the pointer it received.

19.4.5 Dynamic Memory [iter.dynamic]

The functions used to implement the *new* and *delete* operators are declared in *<new>* together with a few related facilities:

```
class bad_alloc : public exception { /* ... */ };

struct nothrow_t {};
extern const nothrow_t nothrow;      // indicator for allocation that doesn't throw exceptions

typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

void* operator new(size_t) throw(bad_alloc);
void operator delete(void*) throw();

void* operator new(size_t, const nothrow_t&) throw();
void operator delete(void*, const nothrow_t&) throw();

void* operator new[](size_t) throw(bad_alloc);
void operator delete[](void*) throw();

void* operator new[](size_t, const nothrow_t&) throw();
void operator delete[](void*, const nothrow_t&) throw();

void* operator new (size_t, void* p) throw() { return p; }      // placement (§10.4.11)
void operator delete (void* p, void*) throw() { }

void* operator new[](size_t, void* p) throw() { return p; }
void operator delete[](void* p, void*) throw() { }
```

The *nothrow* versions of *operator new()* allocate as usual, but if allocation fails, they return *0* rather than throwing *bad_alloc*. For example:

```
void f()
{
    int* p = new int[100000]; // may throw bad_alloc
```

```
        if (int* q = new(nothrow) int[100000]) { // will not throw exception
                // allocation succeeded
        }
        else {
                // allocation failed
        }
}
```

This allows us to use pre-exception error-handling strategies for allocation.

19.4.6 C-Style Allocation [iter.c]

From C, C++ inherited a functional interface to dynamic memory. It can be found in **<*cstdlib*>**:

```
void* malloc(size_t s);             // allocate s bytes
void* calloc(size_t n, size_t s);   // allocate n times s bytes initialized to 0
void  free(void* p);                // free space allocated by malloc() or calloc()
void* realloc(void* p, size_t s);   // change the size of the array pointed to by p to s;
                                    // if that cannot be done, allocate s bytes, copy
                                    // the array pointed to by p to it, and free p
```

These functions should be avoided in favor of ***new***, ***delete***, and standard containers. These functions deal with uninitialized memory. In particular, ***free***( ) does not invoke destructors for the memory it frees. An implementation of ***new*** and ***delete*** may use these functions, but there is no guarantee that it does. For example, allocating an object using ***new*** and deleting it using ***free***( ) is asking for trouble. If you feel the need to use ***realloc***( ), consider relying on a standard container instead; doing that is usually simpler and just as efficient (§16.3.5).

The library also provides a set of functions intended for efficient manipulation of bytes. Because C originally accessed untyped bytes through ***char****** pointers, these functions are found in **<*cstring*>**. The ***void**** pointers are treated as if they were ***char**** pointers within these functions:

```
void* memcpy(void* p, const void* q, size_t n);     // copy non-overlapping areas
void* memmove(void* p, const void* q, size_t n);    // copy potentially overlapping areas
```

Like ***strcpy***( ) (§20.4.1), these functions copy ***n*** bytes from ***q*** to ***p*** and return ***p***. The ranges copied by ***memmove***( ) may overlap. However, ***memcpy***( ) assumes that the ranges do not overlap and is usually optimized to take advantage of that assumption. Similarly:

```
void* memchr(const void* p, int b, size_t n); // like strchr() (§20.4.1): find b in p[0]..p[n-1]
int memcmp(const void* p, const void* q, size_t n); // like strcmp(): compare byte sequences
void* memset(void* p, int b, size_t n);         // set n bytes to b, return p
```

Many implementations provide highly optimized versions of these functions.

19.5 Advice [iter.advice]

[1] When writing an algorithm, decide which kind of iterator is needed to provide acceptable efficiency and express the algorithm using the operators supported by that kind of iterator (only); §19.2.1.

[2] Use overloading to provide more-efficient implementations of an algorithm when given as arguments iterators that offer more than minimal support for the algorithm; §19.2.3.
[3] Use ***iterator_traits*** to express suitable algorithms for different iterator categories; §19.2.2.
[4] Remember to use ++ between accesses of ***istream_iterator***s and ***ostream_iterator***s; §19.2.6.
[5] Use inserters to avoid container overflow; §19.2.4.
[6] Use extra checking during debugging and remove checking later only where necessary; §19.3.1.
[7] Prefer ***++p*** to ***p++***; §19.3.
[8] Use uninitialized memory to improve the performance of algorithms that expand data structures; §19.4.4.
[9] Use temporary buffers to improve the performance of algorithms that require temporary data structures; §19.4.4.
[10] Think twice before writing your own allocator; §19.4.
[11] Avoid ***malloc***(), ***free***(), ***realloc***(), etc.; §19.4.6.
[12] You can simulate a ***typedef*** of a template by the technique used for ***rebind***; §19.4.1.

19.6 Exercises [iter.exercises]

1. (∗1.5) Implement ***reverse***() from §18.6.7. Hint: See §19.2.3.
2. (∗1.5) Write an output iterator, ***Sink***, that doesn't actually write anywhere. When can ***Sink*** be useful?
3. (∗2) Implement ***reverse_iterator*** (§19.2.5).
4. (∗1.5) Implement ***ostream_iterator*** (§19.2.6).
5. (∗2) Implement ***istream_iterator*** (§19.2.6).
6. (∗2.5) Complete ***Checked_iter*** (§19.3).
7. (∗2.5) Redesign ***Checked_iter*** to check for invalidated iterators.
8. (∗2) Design and implement a handle class that can act as a proxy for a container by providing a complete container interface to its users. Its implementation should consist of a pointer to a container plus implementations of container operations that do range checking.
9. (∗2.5) Complete or reimplement ***Pool_alloc*** (§19.4.2) so that it provides all of the facilities of the standard library ***allocator*** (§19.4.1). Compare the performance of ***allocator*** and ***Pool_alloc*** to see if there is any reason to use a ***Pool_alloc*** on your system.
10. (∗2.5) Implement ***vector*** using allocators rather than ***new*** and ***delete***.