SCHAUM'S OUTLINE OF

THEORY AND PROBLEMS

of

FUNDAMENTALS OF COMPUTING WITH C++

•

JOHN R. HUBBARD, Ph.D.

Professor of Mathematics and Computer Science
University of Richmond

•

SCHAUM'S OUTLINE SERIES

McGRAW-HILL

New York San Francisco Washington, D.C. Auckland Bogotá Caracas Lisbon
London Madrid Mexico City Milan Montreal New Dehli
San Juan Singapore Sydney Tokyo Toronto

IPR2025-00647
TankLogix, LLC v. SitePro, Inc.
TankLogix's Exhibit 1007
EX1007, Page 1

JOHN R. HUBBARD is Professor of Mathematics and Computer Science at the University of Richmond. He received his Ph.D. from The University of Michigan (1973) and has been a member of the Richmond faculty since 1983. His primary interests are in numerical algorithms and database systems. Dr. Hubbard is the author of several other books, including *Schaum's Outline of Programming with C++*.

Schaum's Outline of Theory and Problems of

PROGRAMMING WITH C++

Copyright © 1998 by The McGraw-Hill Companies, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 PRS PRS 9 0 1 0 9 8

ISBN 0-07-030868-3

Sponsoring Editor: Barbara Gilson
Production Supervisor: Sherri Souffrance
Editing Supervisor: Maureen Walker

Library of Congress Cataloging-in-Publication Data

Hubbard, J. R. (John R.), date
Schaum's outline of theory and problems of fundamentals of computing with C++ / John R. Hubbard.
p. cm. -- (Schaum's outline series)
Includes bibliographical references and index.
ISBN 0-07-030868-3 (pbk.)
1. C++ (Computer program language) 2. Computer science.
I. Title II. Series
QA76.73.C153H82 1998
005.13'3--dc21 98-16860
CIP

McGraw-Hill
A Division of The McGraw-Hill Companies

Chapter 10

Standard Container Classes

10.1 CONTAINERS

A *container* is an object that contains other objects. The contained objects, called the *elements* of the container, must have the same type. Arrays, stacks, queues, and lists are containers.

A *container class* is a class whose instances are containers. The `Stack` class defined on page 154, the `Queue` class defined on page 163, and the `List` class defined on page 204 are container classes. The new ISO C++ Standard Library defines a collection of container classes including `stack`, `queue`, and `list`. (Note that we capitalized the names of our own container classes to distinguish them from the standard classes.) This chapter summarizes the standard container classes.

10.2 TEMPLATES

The `Stack`, `Queue`, and `List` classes in Chapters 7 and 9 define containers whose element type is `char`. With slight modification, these definitions can be changed for other element types. However C++ provides a better method for being able to change the underlying element type without having to change the container definition. This is done with a template.

A *template* is an outline for the definition of a function or class that uses parameters in place of types or objects. The template can then be *instantiated* by substituting types and objects for the template parameters. In C++, both functions and classes can be defined from templates.

EXAMPLE 10.1 The `swap()` Function Template

```
template <class T>
void swap(T& x, T& y)
{ T temp = x;
  x = y;
  y = temp;
}
```

This function template can be used to define a `swap()` function for any argument type:

```
int m=22, n=44;
cout << "m = " << m << ", n = " << n << endl;
swap(m, n);
cout << "m = " << m << ", n = " << n << endl;
string r="Washington", s="Jefferson";
cout << "r = " << r << ", s = " << s << endl;
swap(r, s);
cout << "r = " << r << ", s = " << s << endl;
```

The output is

```
m = 22, n = 44
m = 44, n = 22
r = Washington, s = Jefferson
r = Jefferson, s = Washington
```


EX1007, Page 3

When the compiler encounters the first call to `swap()`, it uses the fact that the arguments `m` and `n` have type `int` to generate the following function definition:

```
void swap(int& x, int& y)
{ int temp = x;
  x = y;
  y = temp;
}
```

Note that this is done simply by substituting the type `int` for the template parameter `T`.

When the compiler encounters the second call to `swap()`, it uses the fact that the arguments `r` and `s` have type `string` to generate the following function definition:

```
void swap(string& x, string& y)
{ string temp = x;
  x = y;
  y = temp;
}
```

This is done simply by substituting the type `string` for the template parameter `T`.

The effect of the function template is to shorten the source code by having a single template in place of several different function definitions. The compiler uses the template to generate the definitions based upon the calls that it finds in the source code. Note that the efficiency provided by the template occurs only within the source code file. The resulting executable file is no more efficient than if the source code had included two separate non-template definitions.

Note the required code

```
template <class T>
```

that precedes the function header. The symbol `T` is the name of the *template parameter*. Any valid identifier may be used for the name; the letter "T" stands for "type."

EXAMPLE 10.2 A `Stack` Class Template

This template is the same as the definition on page page 154 except with `T` in place of `char`:

```
template <class T>
class Stack
{ public:
    Stack(int m=100) : _max(m), _size(0) { _a = new T[_max]; }
    ~Stack() { delete [] _a; }
    bool is_empty() const { return bool(_size == 0); }
    bool is_full() const { return bool(_size == _max); }
    int size() const { return _size; }
    T top() const { return _a[_size-1]; }
    void push(const T& t) { _a[_size++] = t; }
    T pop() { return _a[--_size]; }
  private:
    T* _a;        // a dynamic array of elements of type T
    int _max;     // the maximum number of elements on the stack
    int _size;   // the number of elements on the stack
};
```

This class template can be used to define a `Stack` class for any element type:

```
Stack<int> int_stack;
int_stack.push(66);
int_stack.push(88);
cout << "int_stack.top() = " << int_stack.top() << endl;
Stack<string> string_stack;
string_stack.push("Madison");
string_stack.push("Monroe");
cout << "string_stack.top() = " << string_stack.top() << endl;
```

The output is

```
int_stack.top() = 88
string_stack.top() = Monroe
```

Note that, unlike a function template, the instantiation of a class template requires the extra < > syntax (*e.g.*, `<int>`, `<string>`) as a suffix onto the template name (*e.g.*, `Stack<int>`, `Stack<string>`) to indicate what type is to be substituted for the template parameter `T`. Function templates don't need this extra syntax because the type is already specified as the argument type.

10.3 THE STANDARD C++ CONTAINER CLASSES AND THEIR OPERATIONS

The C++ Standard Library defines the following ten container class templates:

```
stack<T>
queue<T>
deque<T>
vector<T>
list<T>
priority_queue<T>
set<T>
multitiset<T>
map<K,T>
multimap<K,T>
```

In each case, the template parameter `T` stands for the container's element type. For example the type `vector<double>` would be used to declare a vector whose elements have type `double`.

The `map` and `multimap` templates have two template parameters, `K` and `T`, which stand for the *key type* and the data type. For example the type `map<string,Student>` would be used to declare a map object (*a.k.a.* a *table* or an *associative array*) each of whose elements contains a `string` object paired with a `Student` object. The `string` object would be the student's identification code (*e.g.*, Social Security Number), and the `Student` object would be the data record for the student identified by that code. (The `Student` class would be user-defined, similar to the `Person` class in Problem 7.22 on page 160.) The ID code is called the *key* value for the record.

The common *fundamental operations* for these classes include:

Constructors and Destructors:

`x()`	Default constructor: creates an empty container `x`.
`x(y)`	Copy constructor: creates a duplicate `x` of container `y`.
`x(n)`	Creates a container `x` with `n` copies of the default element.
`x(n,t)`	Creates a container `x` with `n` copies of the element `t`.
`x(p,q)`	Creates a container `x` with the `q-p` elements `*p...*(q-1)`.
`~x()`	Destructor: destroys the container `x` and all of its elements.

Assignment Functions:

`assign(n)`	Replaces current contents with `n` copies of the default element.
`assign(n,t)`	Replaces current contents with `n` copies of the element `t`.
`assign(p,q)`	Replaces current contents with the `q-p` elements `*p...*(q-1)`.
`operator=(y)`	Assignment operator: replaces current contents with those of container `y`.

Iterators:

`begin()`	Returns the position of the first element of the container.
`end()`	Returns the position the dummy element at the end of the container.

Access Functions:

`front()`	Returns the first element of the container.
`back()`	Returns the last element of the container.
`operator[](i)`	Subscript operator: *e.g.*, `x[4] = 99`.
`at(i)`	Range-checked subscript function: *e.g.*, `x.at(4) = 99`.

`size()`	Returns the number of elements in the container.
`empty()`	Returns `true` if the container is empty otherwise, `false`.

Mutator Functions:

`push_front(t)`	Inserts the element `t` at the beginning of the container.
`pop_front()`	Removes the first element in the container.
`push_back(t)`	Inserts the element `t` at the end of the container.
`pop_back()`	Removes the last element in the container.
`insert(p)`	Inserts the default element in front of element `*p`.
`insert(p,t)`	Inserts the element `t` in front of element `*p`.
`insert(p,n,t)`	Inserts `n` copies of the element `t` in front of element `*p`.
`insert(p,q,r)`	Inserts `r-q` elements copied from another container in front of element `*p`.
`erase(p)`	Removes the element `*p`.
`erase(p,q)`	Removes the `q-p` elements `*p...*(q-1)`.
`swap(y)`	Swaps all of its elements with those in container `y`.

Relational Operators:

`operator==(y)`	Returns `true` if the container's contents are the same as those of `y`.
`operator!=(y)`	Returns `true` if the container's contents are not the same as those of `y`.
`operator<(y)`	Returns `true` if the container lexicographically precedes container `y`.

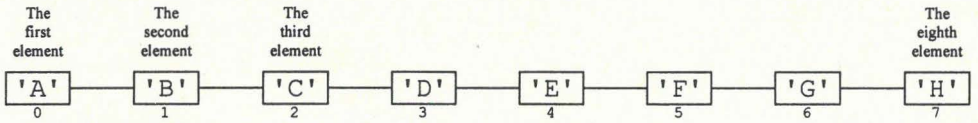
Here, `x` and `y` are container objects, `n` and `i` are non-negative integers, `t` is an element for the container, and `p`, `q`, and `r` are iterators that locate positions in the containers.

Iterators on containers work the same way as pointers on arrays. If `p` and `q` are iterators on the same container and `k` is a positive integer, then:

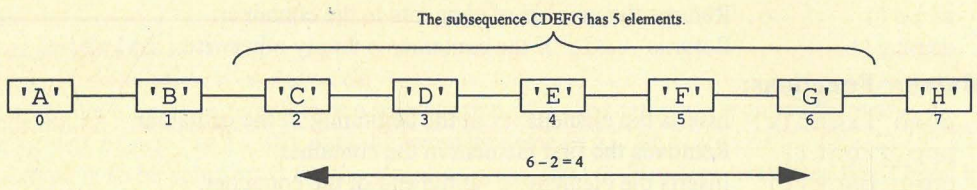
`*p`	is the element at position `p`;
`++p`	advances `p` to the next position;
`--p`	moves `p` back to the preceding position;
`p+k`	locates the position `k` elements after `p`;
`p-k`	locates the position `k` elements before `p`;
`q-p`	is the number of elements in the subsequence from `*p` to `*(q-1)`;

We use the notation `*p...*(q-1)` to denote the subsequence from element `*p` to the element `*(q-1)`. For example, if `p` locates the third element and `q` locates the eighth element of a `list` object `x`, then the value of the expression `q-p` is 5, which is the number of elements in the subsequence from the element `*p` (the third element) to `*(q-1)` (the seventh element).

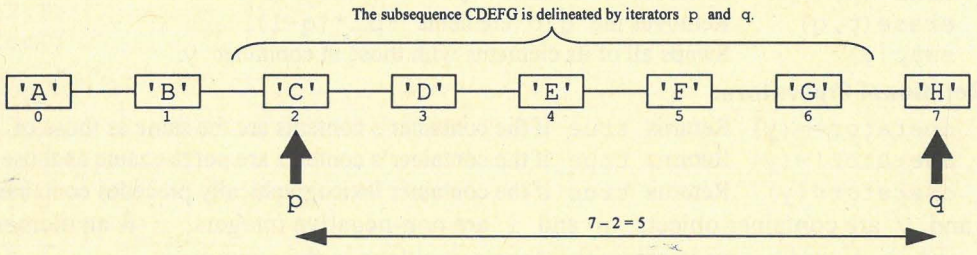
In computer science, elements of sequences are always numbered beginning with the number zero. This is called *zero-based indexing*. The inescapable consequence of this protocol is that the ordinal number of an element is always one more than the index of the element. The 8th element is element number 7:



This is consistent with the fact that the number of elements in a subsequence is one more than the difference between the first and last element numbers. The sequence pictured above has 8 elements, which is one more than the difference 7 – 0 = 7. The subsequence CDEFG has 5 elements, which is one more than the difference 6 – 2 = 4, as shown in the figure below. Because of this intrinsic anomaly between cardinal and ordinal numbers, the C++ Standard Library always delineates a subsequence by specifying the position of its first element and the position of the element that



follows its last element. So the subsequence CDEFG would be delineated by the two iterators `p` and `q` that locate elements C and H:

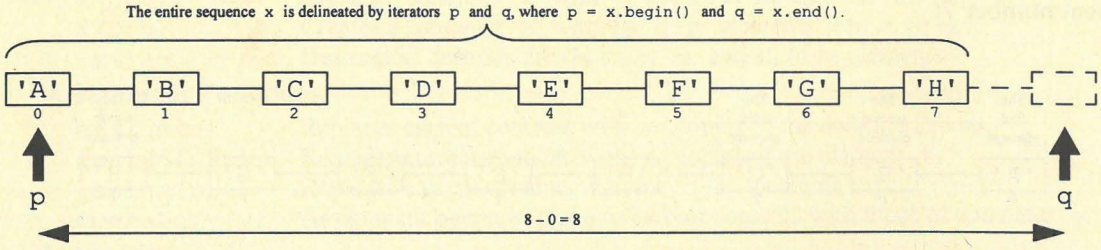


Then the notation `*p...*(q-1)` correctly represents the subsequence CDEFG because `p` locates C and `q-1` locates G.

Consistency may be, as Henry David Thoreau wrote, "the hobgoblin of little minds," but it is essential to good software. It is the hallmark of the C++ Standard Library. To remain consistent to the protocol described above for subsequences, the Standard Library always includes a dummy element appended to the end of every sequence. This element cannot be accessed, but it can be located by an iterator. Indeed, if `ContainerClass<ElementType>` is any container class, then the code

```
ContainerClass<ElementType> x;
ContainerClass<ElementType>::iterator p = x.begin();
ContainerClass<ElementType>::iterator q = x.end();
```

declares `x` to be a container of that type and `p` and `q` to be iterators on `x` that locate the beginning and the end of the sequence:



Then element number `i` can be located as `*(p+i)`.

Note that the access `*(p+i)` appears to be direct access, just like `x[i]`. But the actual implementation of the iterator function `operator+()` depends upon the container class. If `ContainerClass` is `vector`, it will be direct access. But for the container classes that are implemented as linked structures, `p+i` is done by sequential access: `i` calls to `++p`.

10.4 THE C++ STANDARD `stack` CLASS TEMPLATE

A *stack* is a "last-in-first-out" container (see Example 7.17 on page 154). The C++ Standard Library defines the `stack<T>` class template in the `<stack>` file. Here is a (simplified) partial listing of its interface:

```
template <class T>
class stack
{ public:
    stack();                                  // default constructor
    stack(const stack&);                   // copy constructor
    ~stack();                                 // destructor
    stack& operator=(const stack&); // assignment operator
    bool empty() const;                       // returns true iff empty
    int size() const;                         // return number of elements
    T& top();                                 // returns the top element
    void push(const T&);                      // pushes element onto stack
    void pop();                               // removes top element
  private:
    //...
};
```

Note that the `pop()` function does not return the element removed by it. It has to be accessed separately by the `top()` function. Also note that the `top()` function returns by reference (see Section 4.9 on page 76) so it can be used to change the top element without removing it, like this:

```
s.top() = 44;
```

EXAMPLE 10.3 Using the Standard `stack<T>` Class Template

Here is a complete C++ program that uses a `stack` of `strings`:

```
#include <iostream>
#include <stack>
using namespace std;
typedef stack<string> string_stack;

void load(string_stack& s)
{ s.push("alpha");
  s.push("beta");
  s.push("gamma");
  s.push("delta");
  s.push("epsilon");
}
void dump(string_stack& s)
// Postcondition: the stack s is empty
{ int n = s.size();
  for (int k=1; k <= n; k++)
  { cout << '\t' << k << ". " << s.top() << endl;
    s.pop();
  }
}
void transfer(string_stack& s1, string_stack& s2)
{ while (!s2.empty())
  { s1.push(s2.top());
    s2.pop();
  }
}
```

```
int main()
{ string_stack letters;
  load(letters);
  string_stack backup = letters;  // uses the copy constructor
  cout << "In reverse order:\n";
  dump(letters);
  transfer(letters, backup);
  cout << "In order:\n";
  dump(letters);
}
```

The output is

```
In reverse order:
        1. epsilon
        2. delta
        3. gamma
        4. beta
        5. alpha
In order:
        1. alpha
        2. beta
        3. gamma
        4. delta
        5. epsilon
```

The `letters` stack is loaded with the names of the first six Greek letters. Then the copy constructor is used to construct a clone named `backup`. The `dump()` function empties and prints in reverse order the contents of the stack passed to it. Next, the `transfer()` function transfers the contents of stack `s2` into `s1`. Since stack access is last-in-first-out, the transfer from the `backup` stack to the `letters` stack loads the names into the `letters` stack in reverse order. Consequently, the second call to `dump()` prints the letter names in their proper order.

10.5 THE C++ STANDARD `queue` CLASS TEMPLATE

A *queue* is a "first-in-first-out" container (see Problem 7.7 on page 156). The C++ Standard Library defines the `queue<T>` class template in the `<queue>` file. Its class interface is essentially the same as that for the `stack<T>` template, using the same names for its member functions. The difference, of course, is that the `pop()` function for queue classes removes elements from the other end. Here is a (simplified) partial listing of its interface:

```
template <class T>
class queue
{ public:
    queue();                                  // default constructor
    queue(const queue&);                    // copy constructor
    ~queue();                                 // destructor
    queue& operator=(const queue&);// assignment operator
    bool empty() const;                       // returns true iff empty
    int size() const;                         // return number of elements
    T& front();                               // returns the front element
    T& back();                                // returns the back element
    void push(const T&);                      // inserts element at back
    void pop();                               // removes element from front
  private:
    //...
};
```

Note that, like the `stack::top()` function, the `front()` and `back()` functions return by reference so they can be used to change those two elements without removing them.

EXAMPLE 10.4 Using the Standard `queue<T>` Class Template

Here is a complete C++ program that uses a queue of strings:

```
#include <iostream>
#include <queue>
using namespace std;
typedef queue<string> string_queue;

void load(string_queue& s)
{ s.push("alpha");
  s.push("beta");
  s.push("gamma");
  s.push("delta");
  s.push("epsilon");
}

void dump(string_queue& s)
// Postcondition: the stack s is empty
{ int n = s.size();
  for (int k=1; k <= n; k++)
  { cout << '\t' << k << ". " << s.front() << endl;
    s.pop();
  }
}

void transfer(string_queue& s1, string_queue& s2)
{ while (!s2.empty())
  { s1.push(s2.front());
    s2.pop();
  }
}

int main()
{ string_queue letters;
  load(letters);
  string_queue backup = letters;  // uses the copy constructor
  cout << "In order:\n";
  dump(letters);
  transfer(letters, backup);
  cout << "In order:\n";
  dump(letters);
}
```

The output is

```
In order:
        1. alpha
        2. beta
        3. gamma
        4. delta
        5. epsilon
```

```
In order:
        1. alpha
        2. beta
        3. gamma
        4. delta
        5. epsilon
```

This is the program from Example 10.3, except using queues in place of stacks. The only difference in its output is that, reflecting the first-in-first-out nature of queues, this dumps the letters in their proper order the first time.

10.6 THE C++ STANDARD `vector` CLASS TEMPLATE

A *vector* is a sequence of elements that supports direct access. As an abstract type, it simply generalizes the ordinary array type. The C++ Standard Library defines the `vector<T>` class template in the `<vector>` file. Its class interface is the prototype for all the Standard container class templates. With few exceptions, for each member function of the `vector` class there is an equivalent member function for each of the other container classes (`stack`, `queue`, `list`, `set`, `map`, *etc.*). Here is a (simplified) partial listing of its interface:

```
template <class T>
class vector
{   friend bool operator==(const vector&, const vector&);
    friend bool operator<(const vector&, const vector&);
  public:
    typedef T* iterator;
    vector();                           // default constructor
    vector(const vector&);              // copy constructor
    vector(int, const T&);              // auxiliary constructor
    vector(iterator, iterator);         // auxiliary constructor
    ~vector();                          // destructor
    vector& operator=(const vector&);   // assignment operator
    void assign(int, const T&);         // assigns a given value
    void assign(iterator, iterator);    // copies elements from object
    void resize(int);                   // changes size of vector
    void swap(vector&);                 // swaps elements with object
    bool empty() const;                 // returns true iff empty
    int size() const;                   // return number of elements
    iterator begin();                   // locates first element
    iterator end();                     // locates dummy element at end
    T& operator[](int);                 // subscript operator
    T& at(int);                         // range-checked access
    T& front();                         // accesses the first element
    T& back();                          // accesses the last element
    void push_back(const T&);           // inserts element at end
    void pop_back();                    // removes last element
    iterator insert(iterator, const T&);
    void insert(iterator, int, const T&);
    void insert(iterator, iterator, iterator);
    iterator erase(iterator);
    iterator erase(iterator, iterator);
    void clear();                       // removes all the elements
  private:
    //...
};
```

EXAMPLE 10.5 Using the Standard `vector<T>` Class Template

Here is a complete C++ program that uses the Standard `vector<T>` class template::

```
#include <iostream>
#include <vector>      // defines the Standard vector<T> class template
using namespace std;
typedef vector<double> vec;
typedef vector<bool> bits;

template <class T>
void copy(vector<T>& v, const T* x, int n)
{ vector<T> w;
  for (int i=0; i<n; i++)
    w.push_back(x[i]);
  v = w;
}

vec projection(vec& v, bits& b)
{ int v_size = v.size();
  assert(b.size() >= v_size);
  vec w;
  for (int i=0; i<v_size; i++)
    if (b[i]) w.push_back(v[i]);
  return w;
}

void print(vec& v)
{ int v_size = v.size();
  for (int i=0; i<v_size; i++)
    cout << v[i] << "  ";
  cout << endl;
}

int main()
{ double x[8] = { 22.2, 33.3, 44.4, 55.5, 66.6, 77.7, 88.8, 99.9 };
  vec v;
  copy(v, x, 8);
  bool y[8] = { false, true, false, true, true, true, false, true };
  bits b;
  copy(b, y, 8);
  vec w = projection(v, b);
  print(v);
  print(w);
}
```

The output is

```
22.2  33.3  44.4  55.5  66.6  77.7  88.8  99.9
33.3  55.5  66.6  77.7  99.9
```

This illustrates the `vector` class `push_back()` and `size()` member functions.

The purpose of the `projection(v, b)` function is to use the bit vector `b` as a *mask* to remove selected elements of the vector `v`. The resulting vector `w` is called the *projection* of `v` onto the subspace determined by `b`.

10.7 THE C++ STANDARD `list` CLASS TEMPLATE

A *list* is a sequence of elements that do not have direct access. A definition of the ADT **List** is given in Chapter 9 along with an implementation for lists of elements of type `char`.

The C++ Standard Library defines the `list<T>` class template in the `<list>` file. It is distinguished from the `vector<T>` class template class by the following features:

- it lacks the direct access provided by the subscript operator and the `at()` function;
- it includes `push_front()`, `pop_front()`, and `remove()` functions;
- its `insert()` and `erase()` functions are much faster than those for vectors.

These distinctions reflect the fundamental trade-off between the fast direct access provided by contiguous storage (vectors) and the fast insertions and deletions provided by linked storage (lists).

Due to the special nature of its linked implementation, the `list<T>` class template includes some other member functions not included in the `vector<T>` class template:

```
template <class T>
class list
{   // same functions as in vector<T>
  public:
    class iterator;                  // defined elsewhere
    void push_front(const T&);       // inserts element at beginning
    void pop_front();                // removes first element
    void remove(const T&);           // removes elements
    void sort();                     // sorts the list
    void unique();                   // removes all duplicates
    void merge();                    // removes all duplicates
    void splice(iterator, list&);    // inserts elements from other list
    void reverse();                  // reverses the elements
  private:
    //...
};
```

Unlike `vector` iterators which are just pointers to elements, `list` iterators are separate objects, as defined in Chapter 9. The `list` classes can push and pop both at the end (like the `vector` classes) and at the beginning of the sequence. The other member functions declared here for `list` classes are defined as non-member generic functions for the `vector` classes. (See Section 10.8.)

Note that the subscript operator is defined for `list` classes; but it does not give direct access. The reference `x[6]` is implemented by 6 increments `++p` of an iterator initialized at `x.begin()`.

EXAMPLE 10.6 Using the Standard `list<T>` Class Template

This program uses a list of strings to find the most popular fruit among a group of people. The program reads a sequence of names of fruit and stores them in a list. Each item is inserted so that the list is maintained in alphabetical order. No name is inserted more than once. Instead, each name is stored together with its ordinal number within the input. For example, "pear" was the 1st, 15th, and 22nd name input, so it is stored in the list like this:

```
pear      01, 15, 22
```

This is one of the list elements.

```
#include <iomanip.h>
#include <iostream.h>
#include <vector>      // defines the Standard vector<T> class template
using namespace std;

typedef list<string> SL;              // string list type
typedef list<string>::iterator SLI;   // string list iterator type
```

10.7 THE C++ STANDARD `list` CLASS TEMPLATE

A *list* is a sequence of elements that do not have direct access. A definition of the ADT **List** is given in Chapter 9 along with an implementation for lists of elements of type `char`.

The C++ Standard Library defines the `list<T>` class template in the `<list>` file. It is distinguished from the `vector<T>` class template class by the following features:

- it lacks the direct access provided by the subscript operator and the `at()` function;
- it includes `push_front()`, `pop_front()`, and `remove()` functions;
- its `insert()` and `erase()` functions are much faster than those for vectors.

These distinctions reflect the fundamental trade-off between the fast direct access provided by contiguous storage (vectors) and the fast insertions and deletions provided by linked storage (lists).

Due to the special nature of its linked implementation, the `list<T>` class template includes some other member functions not included in the `vector<T>` class template:

```
template <class T>
class list
{   // same functions as in vector<T>
  public:
    class iterator;                  // defined elsewhere
    void push_front(const T&);       // inserts element at beginning
    void pop_front();                // removes first element
    void remove(const T&);           // removes elements
    void sort();                     // sorts the list
    void unique();                   // removes all duplicates
    void merge();                    // removes all duplicates
    void splice(iterator, list&);    // inserts elements from other list
    void reverse();                  // reverses the elements
  private:
    //...
};
```

Unlike `vector` iterators which are just pointers to elements, `list` iterators are separate objects, as defined in Chapter 9. The `list` classes can push and pop both at the end (like the `vector` classes) and at the beginning of the sequence. The other member functions declared here for `list` classes are defined as non-member generic functions for the `vector` classes. (See Section 10.8.)

Note that the subscript operator is defined for `list` classes; but it does not give direct access. The reference `x[6]` is implemented by 6 increments `++p` of an iterator initialized at `x.begin()`.

EXAMPLE 10.6 Using the Standard `list<T>` Class Template

This program uses a list of strings to find the most popular fruit among a group of people. The program reads a sequence of names of fruit and stores them in a list. Each item is inserted so that the list is maintained in alphabetical order. No name is inserted more than once. Instead, each name is stored together with its ordinal number within the input. For example, "pear" was the 1st, 15th, and 22nd name input, so it is stored in the list like this:

```
pear      01, 15, 22
```

This is one of the list elements.

```
#include <iomanip.h>
#include <iostream.h>
#include <vector>     // defines the Standard vector<T> class template
using namespace std;

typedef list<string> SL;              // string list type
typedef list<string>::iterator SLI;   // string list iterator type
```

```
string extract_word(SLI& it)
// returns the word stored in *it:
{ string element = *it;          // the complete element *it
  int k = element.find('\t');    // location of the tab
  return element.substr(0, k);   // the substring ahead of the tab
}
bool found(SL& x, SLI& it, string& new_word)
// searches the list x for the new_word:
{ for (it=x.begin(); it != x.end(); ++it)
  { string word = extract_word(it);
    if (word == new_word) return true;
    if (word > new_word) return false;
  }
}
string literal(int n)
// returns the string literal for the integer n:
{ string s(2);                // a 2-character string
  s[0] = char('0' + n/10);    // the tens digit
  s[1] = char('0' + n%10);    // the ones digit
  return s;
}
void print(SL& x)
// prints the entire list, numering each element:
{ int i=1;
  for (SLI it=x.begin(); it != x.end(); ++it, ++i)
    cout << setw(4) << i << ". " << *it << endl;
}
int main()
{ SL x;                     // x is an empty list of strings
  SLI it = x.begin();       // it is an iterator on x
  string new_word;
  int count=0;
  while (cin >> new_word)
  { ++count;
    if (found(x, it, new_word)) *it += ", " + literal(count);
    else x.insert(it, new_word + "\t" + literal(count));
  }
  print(x);
}
```

Each iteration of the main `while` loop reads one word from `cin`, counts it, and searches for it in the list `x`. If it is found to be already in the list, then the string `*it` at that element is appended with the current `count`. Otherwise the `new_word` and its `count` are inserted into a new element in the list. The `while` loop terminates when the end-of-file is detected. Then the entire list is printed.

Here is the input:

```
pear    peach   prune   orange  plum    lemon   lime    banana
peach   raisin  grape   date    lemon   mango   pear    lime
apple   fig     orange  peach   mellon  pear    orange  apple
cherry  lemon   cherry  grape   raisin  orange  peach   mango
apple   cherry  date    peach
```

And here is the output:

```
   1. apple     17, 24, 33
   2. banana    08
   3. cherry    25, 27, 34
   4. date      12, 35
   5. fig       18
   6. grape     11, 28
```

```
 7. lemon      06, 13, 26
 8. lime       07, 16
 9. mango      14, 32
10. mellon     21
11. orange     04, 19, 23, 30
12. peach      02, 09, 20, 31, 36
13. pear       01, 15, 22
14. plum       05
15. prune      03
16. raisin     10, 29
```

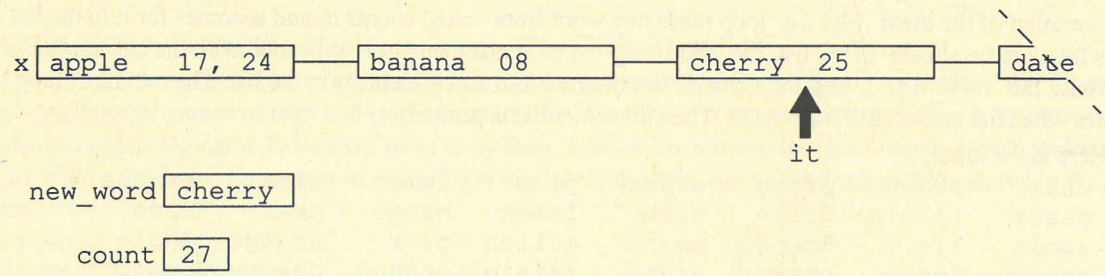
This shows that peaches are the most popular fruit among these 36 people, being the favorite of persons number 2, 9, 20, 31, and 36. Notice that the list elements are in alphabetical order. Also notice that the output forms a *histogram* (a frequency chart), from which the *mode* (most frequent item) is easily discernible.

The `extract_word()` function is passed a reference to the iterator `it`. It assigns the element `*it` to the string `element`, and then uses the `find()` function from the `string` class to locate the position of the tab character `'\t'` within the string. This character follows the last letter of the fruit name in each list element, so its index number `k` can be used to extract that name from the element. This is done with the string class's `substr()` function. The parameters `(0, k)` tell it to extract the first `k` characters (from position `0` to position `0+k`). That is the fruit name which is then returned.

The `found()` function is passed a reference to the list `x`, a reference to the iterator `it`, and a reference to the string `new_word`. It uses a `for` loop to traverse the list, searching for an element that contains `new_word`. It applies the `extract_word()` function to the iterator at each element to extract the fruit name in that element. If that `word` is the same as `new_word`, then the `found()` function returns `true`. Otherwise, it either returns `false` or continues to the next iteration of the `for` loop, depending upon whether the `word` in the current list element is greater (lexicographically) than `new_word`. This way, if the function does return `false`, then the iterator `it` is positioned at the correct location in the list at which the `new_word` should be inserted to maintain alphabetical order. On the other hand, if the function returns `true`, meaning that the `new_word` is already in the list, then again the iterator `it` returns locating that particular element, thereby allowing `main()` to update it. The iterator `it` must be passed by reference for this to work.

The `literal()` function simply constructs a `string` that is the literal equivalent to the integer passed to it. For example, if `n` has the value `27`, then the function returns the string `"27"`. This is done by extracting the ten's digit (the `2`) and the one's digit (the `7`) separately and then assigning them to `s[0]` and `s[1]`, respectively. This returned string is used by `main()`.

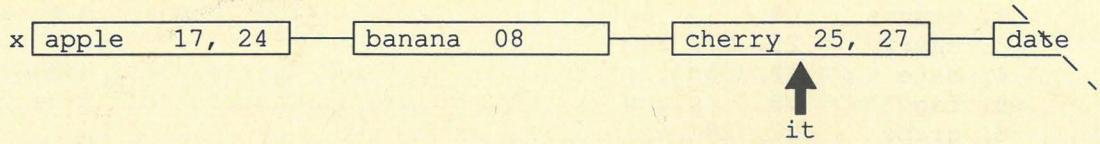
If the `new_word` was `found`, then the element `*it` is updated simply by appending `", "` and the string literal. For example, the 27th word is `cherry`. When that word is input, the list looks like this:



The `found()` function positions the iterator it at the third element and returns `true`. Then the statement

```
*it += ", " + literal(count);
```

appends the string `", 27"` to the string stored in that element, changing the list to



10.8 GENERIC ALGORITHMS

The C++ Standard Library is the result of the 1998 ISO standardization of the C++ programming language, which expanded its library by including a collection of classes and functions that were developed in the early 1990s by Alexander Stepanov and Meng Lee. Before the conclusion of the ISO standardization process, this separate collection was knows as the Standard Template Library. Because it is primarily a collection of classes, iterators, and implemented algorithms, Bjarne Stroustrup suggested the acronym CIA for the library. Regardless of the name, the implementation of the generic algorithms in the library are one of its most distinctive features.

The generic algorithms are defined in the Standard Library file `<algo>`. These algorithms are implemented as non-member functions (also called *free* functions). The file includes over 70 of these functions, most of which apply to most of the standard container classes. Here is a listing of some of the most useful:

```
find(start, stop, t);                                  // find t in subseq
adjacent_find(start, stop);          // find adjacent equal pair in subseq
count(start, stop, t, n);              // count occurrences of t in subseq
equal(start, stop, pos);             // return true iff subseq1 == subseq2
search(start1, stop1, start2, stop2);          // find subseq2 in subseq1
copy(start, stop, pos);                   // replace subseq2 with subseq1
swap(t, u);                                          // interchange t and u
swap_ranges(start, stop, pos);         // interchange subseq1 with subseq2
replace(start, stop, t, u);            // replace every t in subseq with u
fill(start, stop, t);            // replace every element in subseq with t
remove(start, stop, t);                // remove every element t in subseq
unique(start, stop);                    // remove all duplicates in subseq
reverse(start, stop);                    // reverse the elements in subseq
rotate(start, mid, stop);    // shift subseq (mid-start) to left & wrap
random_shuffle(start, stop);                   // randomly permute subseq
sort(start, stop);                                      // sort the subseq
nth_element(start, nth, stop);          // partition the subseq about nth
binary_search(start, stop, t);                       // search subseq for t
min(t, u);                                       // return ( t < u ? t : u )
max(t, u);                                       // return ( t > u ? t : u )
min_element(start, stop);            // return smallest element in subseq
max_element(start, stop);              // return largest element in subseq
next_permutation(start, stop);                        // permute the subseq
```

Here, `start`, `stop`, `pos`, and `mid` are iterators, `t` and `u` are elements, and `n` is a positive integer. In the comments, `subseq` and `subseq1` refer to the elements `*start...*(stop-1)` or `*start1...*(stop1-1)`, and `subseq2` refers to the elements `*start2...*(stop2-1)` or `*pos...*(pos+(stop1-start1)-2)`.

The details of these algorithms are explained in the Problems below.

Review Questions

10.1 What is the difference between a class template and a template class?

10.2 What, if anything, is wrong with the following code. Explain.

```
Stack<Stack<short>> s;
```

10.3 Is the Ratio class (see Chapter 3) a container class?

10.4 What is the difference between a vector and an array?

10.5 What are the differences between C strings and C++ strings?

10.6 What is "fragmented memory" and why is it bad?

10.7 *a.* What is the difference between the subscript operator and the `at()` function?
b. Why is one preferred over the other?

10.8 What is the difference between the `stack`, `queue`, `deque`, and `vector` template classes?

Programming Problems

10.9 Omit the member function `is_full()` in the definition for class template `Stack<T>` in Example 10.2 on page 227. Replace it with a private utility function named `_expand()` that dynamically doubles the size of the stack and copies the entire contents of the old stack into the first half of the new stack. Then use that utility function in the `push()` function to prevent overflow.

10.10 Convert the `Ratio` class (see Chapter 7) into a template that allows ratios of numbers other than `int`s. For example, `Ratio<long>` and even `Ratio<double>` would be valid classes.

10.11 Use the Standard `list` template defined in the `<list>` file to implement a `Deque<T>` class template. (See Problem 7.7 on page 156) Declare a list as the (only) data member and then call the appropriate list member functions to implement the corresponding deque member functions. For example, the deque's `push_back()` function is the same as the `push_back()` function defined in the Standard `list` class.

10.12 Use the Standard `deque` template defined in the `<deque>` file to implement the `Stack<T>` class template. Declare a deque as the (only) data member and then call the appropriate deque member functions to implement the corresponding stack member functions. For example, the stack's `pop()` function is simply the `pop_back()` function defined in the `deque` class. This is how `stack` is defined in the C++ Standard Library.

10.13 Use the Standard `deque` template defined in the `<deque>` file to implement the `Queue<T>` class template. Declare a deque as the (only) data member and then call the appropriate deque member functions to implement the corresponding queue member functions. For example, the queue's `pop()` function is simply the `pop_front()` function defined in the `deque` class. This is how `queue` is defined in the C++ Standard Library.

10.14 Use the Standard `queue` template defined in the `<queue>` file and the `Person` class (Problem 7.22 on page 160) to simulate a line of people waiting for movie tickets.

10.15 Modify the `projection()` function in Example 10.5 on page 235 so that it returns the vector `v` after erasing the elements in the slots where `b[i]` is false. Use the `erase()` function defined in the `vector` class. Use iterators instead of integers.

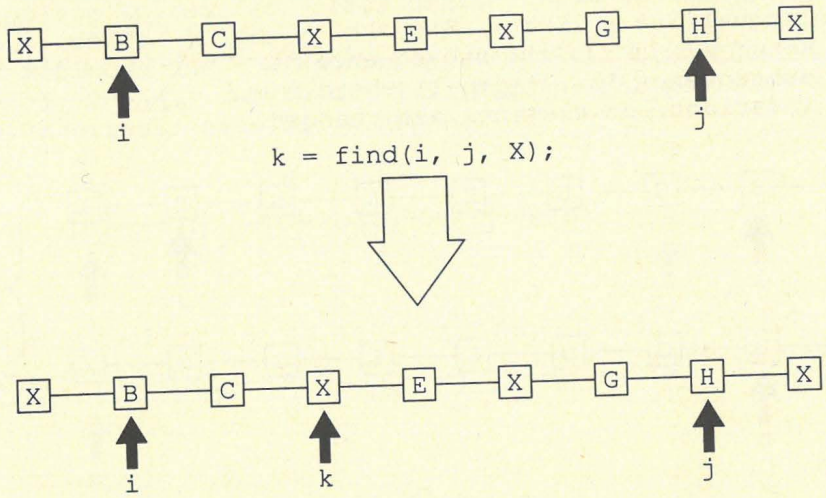
10.16 Add the following function to the program in Example 10.5 on page 235:

```
int freq(SL& x, string& word)
// returns the number of times word appeared in the input
```

Locate the word in the list `x` and then count how many numbers are in that element.

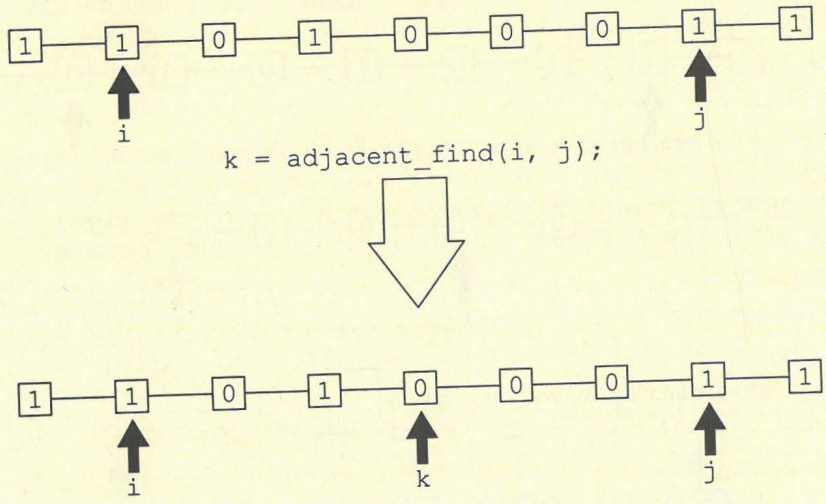
10.17 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class T>
Src find(Src i, Src j, const T& t)
// Searches for t in the subsequence *i...*(j-1); if found, its
// position k is returned; otherwise, j is returned.
// Invariant: no elements are changed.
```



10.18 Implement the following generic Standard Library function defined in <algo>:

```
template <class Src, class T>
Src adjacent_find(Src i, Src j);
// Returns the position k of the first occurrence of a pair of
// adjacent equal elements among the elements *i...*(j-1).
// If no equal adjacent elements are found, j is returned.
// Invariant: no elements are changed.
```

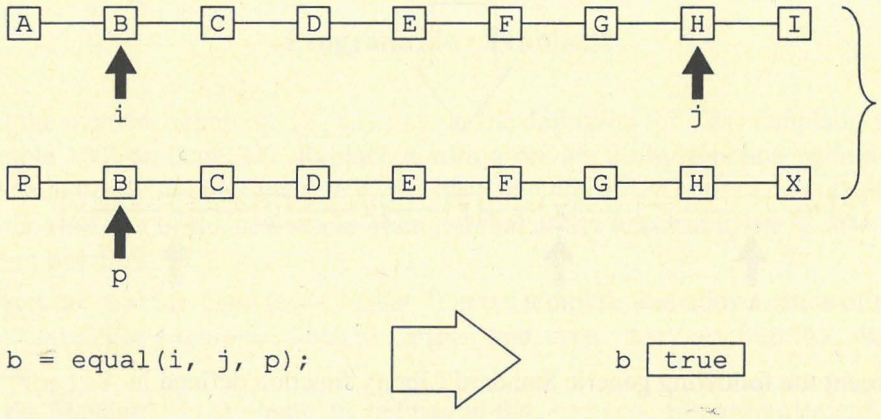


10.19 Implement the following generic Standard Library function defined in <algo>:

```
template <class Src, class T>
void count(Src i, Src j, const T& t, int& n);
// Increments n for each occurrence of the value t among the
// elements *i...*(j-1).
// Invariant: no elements are changed.
```

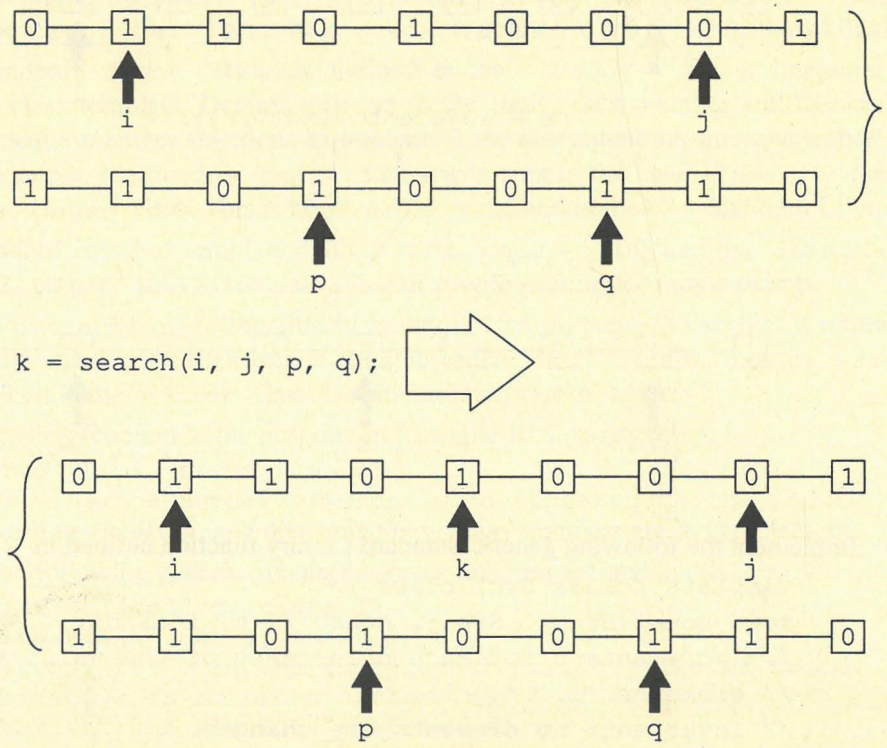
10.20 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src>
bool equal(Src i, Src j, Src p);
// Returns true iff the subsequence *i...*(j-1) is the same as the
// subsequence *p...*(p+n-1) where n = i - j.
// Invariant: no elements are changed.
```



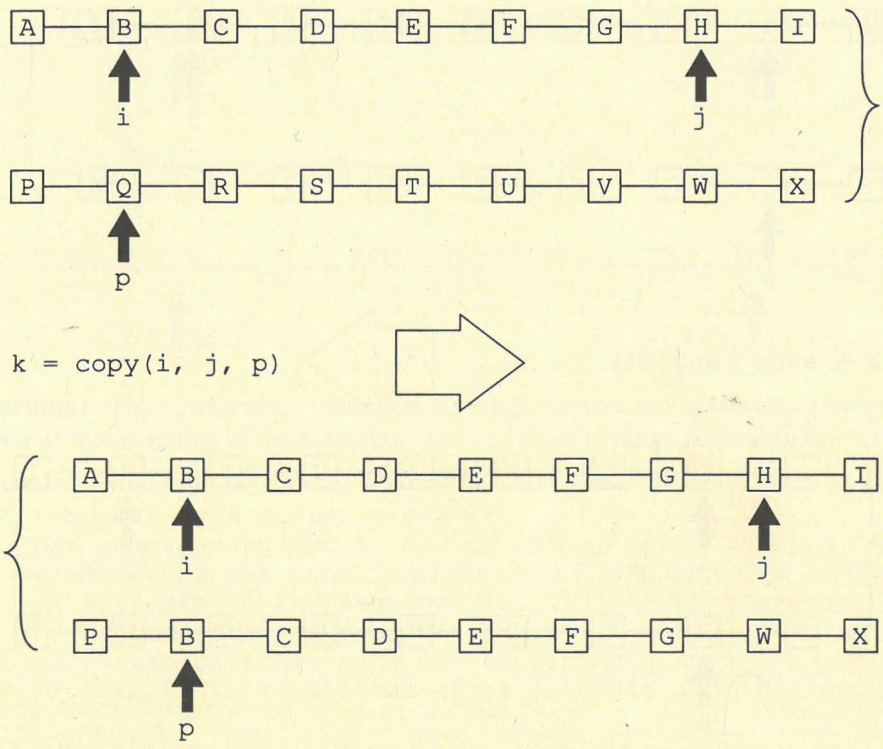
10.21 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src1, class Src2>
Src1 search(Src1 i, Src1 j, Src2 p, Src2 q);
// Searches the sequence *i...*(j-1) for a subsequence that equals
// the sequence *p...*(q-1). If found, position k where it begins
// is returned; otherwise, j is returned.
// Invariant: no elements are changed.
```



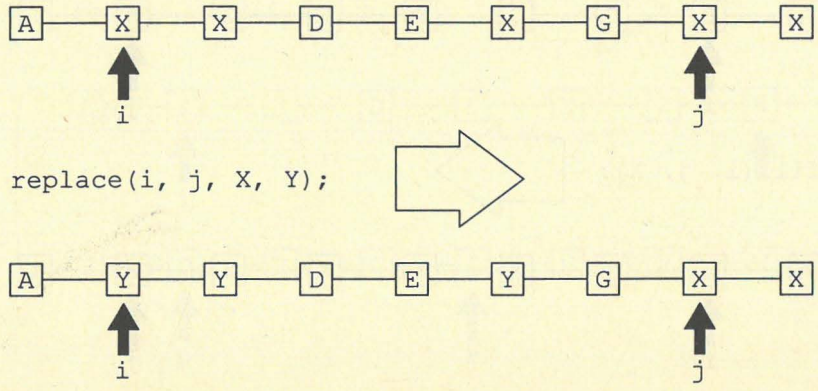
10.22 Implement the following generic Standard Library function defined in <algo>:

```
template <class Src, class Dst>
Dst copy(Src i, Src j, Dst p);
// Replaces the n elements *p...*(p+n-1) with the n elements
// *i...*(j-1), where n = j-i. Returns the position k = p+n.
// Invariant: size is unchanged.
```



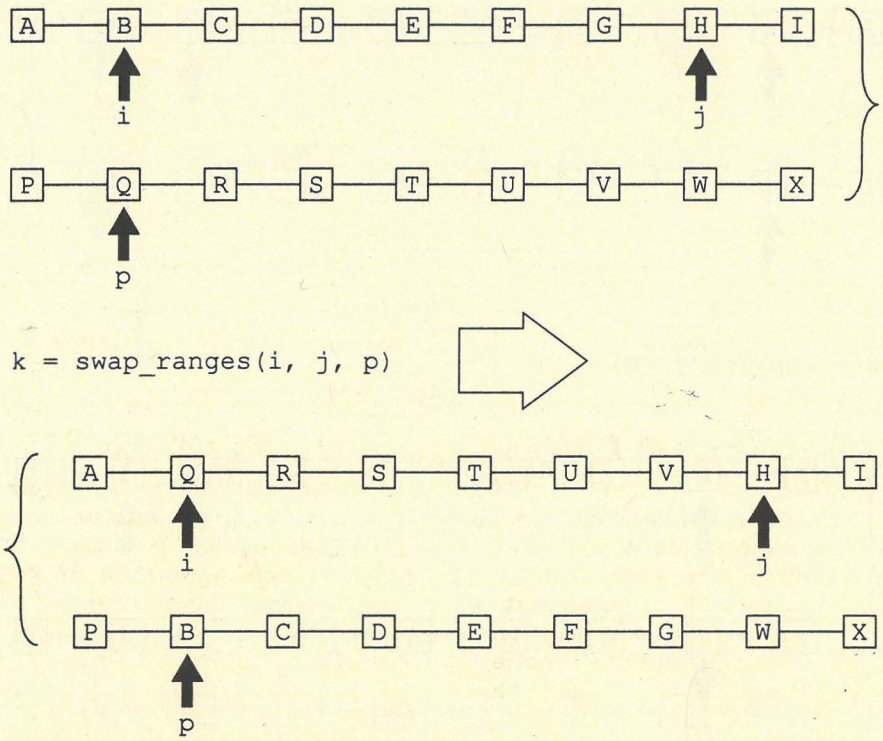
10.23 Implement the following generic Standard Library function defined in <algo>:

```
template <class Src, class T>
void replace(Src i, Src j, const T& t, const T& u);
// Replaces each occurrence of the value t with the value u in
the
// sequence *i...*(j-1).
// Invariant: size is unchanged.
```



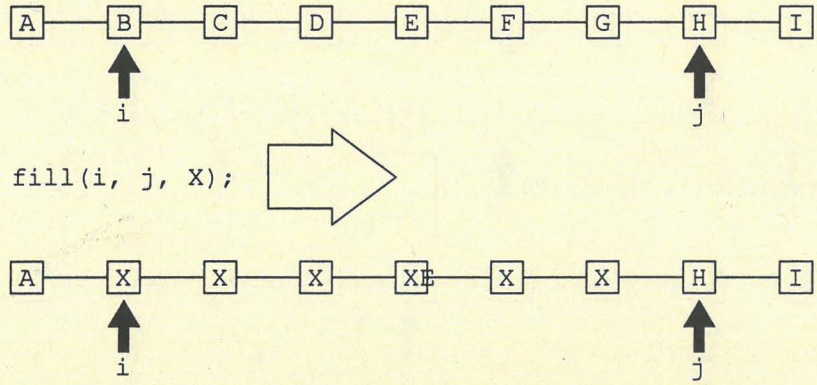
10.24 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class Dst>
Dst swap_ranges(Src i, Src j, Dst p);
// Interchanges the n elements *p...*(p+n-1) with the n elements
// *i...*(j-1), where n = j-i. Returns the position k = p+n.
// Invariant: size is unchanged.
```



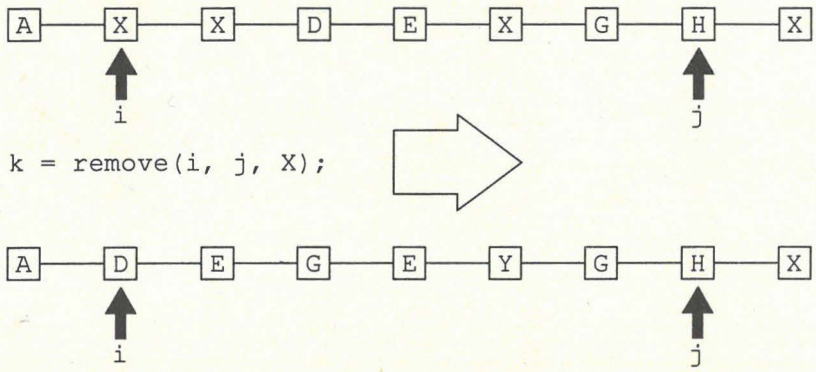
10.25 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class T>
void fill(Src i, Src j, const T& t);
// Replaces each of the elements *i...*(j-1) with a copy of the
// value t.
// Invariant: size is unchanged.
```



10.26 Implement the following generic Standard Library function defined in `<algo>`:

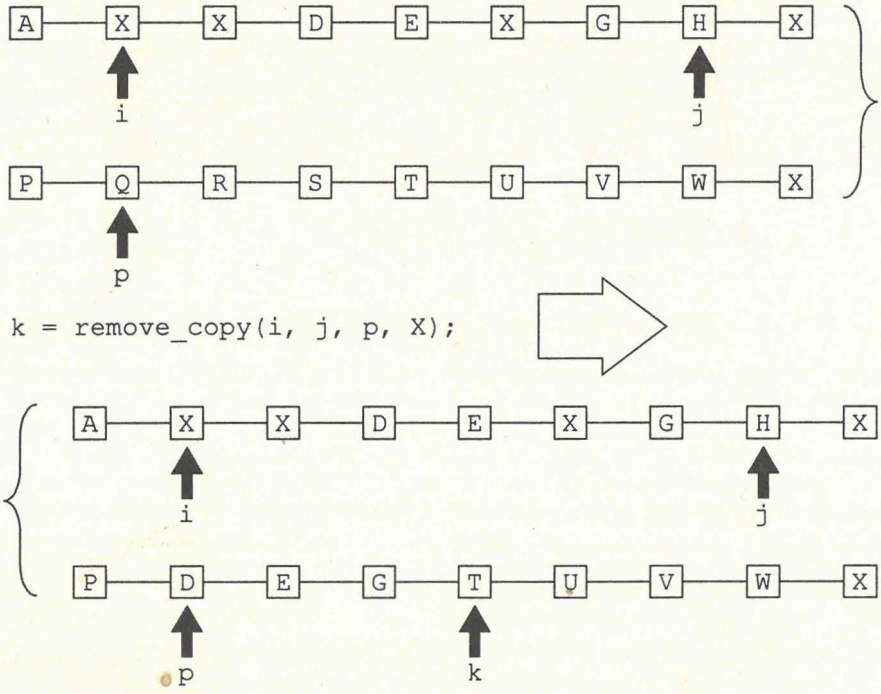
```
template <class Src, class T>
Src remove(Src i, Src j, const T& t);
// Shifts to the left the n elements among *i...*(j-1) that are
// not equal to t. Returns the position k = i+n.
// Postconditions: none of the elements among *i...*(k-1) are
//   equal to t; all the other elements are left unchanged.
// Invariant: size is unchanged.
```



Warning: The `remove()` function does not remove any elements. Instead, it overwrites some of the elements in the subrange `*i...*(k-1)` range removing the `t` values.

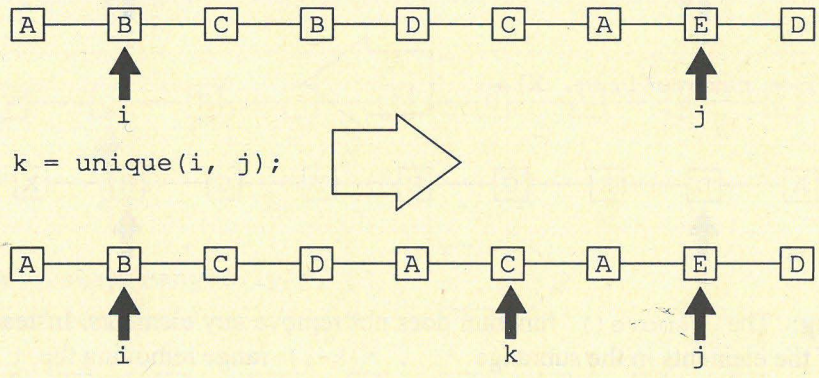
10.27 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class T>
Dst remove_copy(Src i, Src j, Dst p, const T& t);
// Replaces the first n elements of the sequence at *p with the n
// elements in the sequence *i...*(j-1) that are not equal to t.
// Returns the position k = p+n.
// Postconditions: none of the elements among *p...*(k-1) are
//   equal to t; all the other elements are left unchanged.
```



10.28 Implement the following generic Standard Library function defined in `<algo>`:

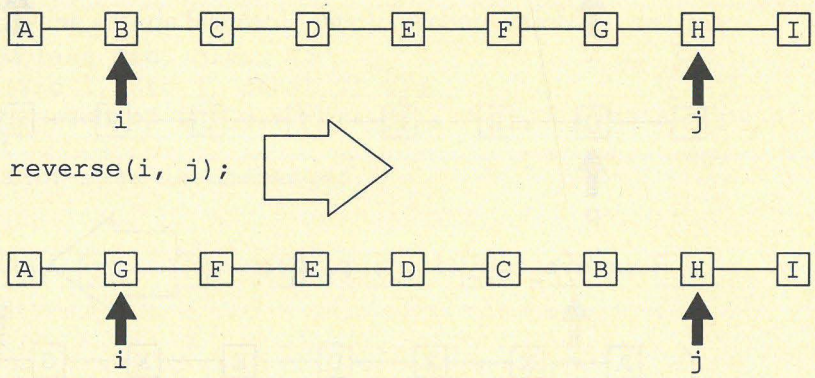
```
template <class Src, class T>
Src unique(Src i, Src j);
// Overwrites up to n-1 elements starting with element *i among
// the sequence *i...*(j-1), where n is the number of distinct
// elements in that range. Returns: the position k = i+n.
// Postconditions: all of the elements among *i...*(k-1) are
//   distinct; all the other elements are left unchanged.
```



Warning: Like the `remove()` function, the `unique()` function achieves its desired result only in the resulting subsequence `*i...*(k-1)`.

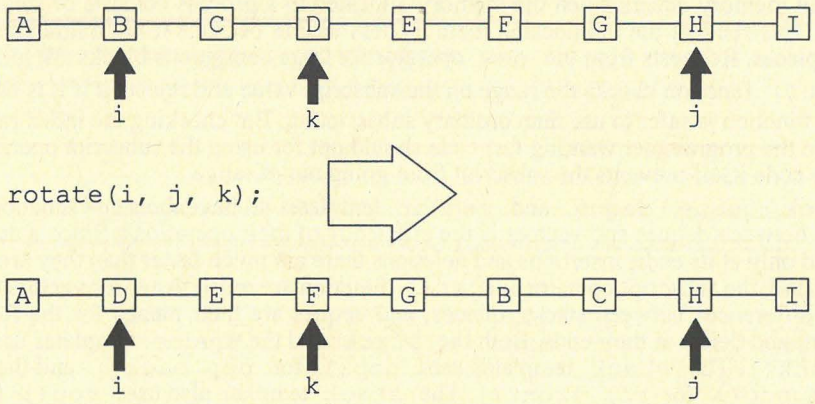
10.29 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class T>
void reverse(Src i, Src j);
// Reverses the sequence *i...*(j-1).
```



10.30 Implement the following generic Standard Library function defined in `<algo>`:

```
template <class Src, class T>
void rotate(Src i, Src k, Src j);
// Replaces the subsequence *i...*(i+n-1) with *k...*(j-1), and
// the subsequence *(i+n)...*(j-1) with *i...*(k-1),
// where n = j-k.
```



Solutions

10.1 A *class template* is only an outline for a class; it includes one or more template parameters. A *template class* is a specific class that is generated by the compiler from a class template; it specifies what type(s) are to be substituted in for the template parameter(s). For example,

```
template<class T> class Stack ( //... };        // a class tem-
plate
Stack<int> s1;     // a template class used to declare the stack
s1
Stack<double> s2; // a template class used to declare the stack
s2
```

Here `s1` is declared to be a stack whose elements have type `int` and `s2` is declared to be a stack whose elements have type `double`. Note that when the declaration of `s1` is compiled, the compiler uses the template to create a complete class definition for the `Stack<int>` type, and when the declaration of `s2` is compiled, the compiler uses the template to create a separate class definition for the `Stack<double>` type.

10.2 There is nothing wrong with this code (although some compilers may not yet be able to compile it). It declares `s` to be a stack whose elements are stacks whose elements are `shorts`. It is equivalent to

```
typedef Stack<short> short_stack_type;
Stack<short_stack_type> s;
```

The object `s` could be used like this:

```
Stack<short> ss1, ss2;
ss1.push(22);  // now ss1 contains 1 element: 22
ss2.push(44);  // now ss2 contains 1 element: 44
s.push(ss1);   // now s contains 1 element: ss1
s.push(ss2);   // now s contains 2 elements: ss1 and ss2
```

The object `s` is a container of containers.

10.3 The `Ratio` class (Chapter 3) a not really a container class. Although `Ratio` objects (*i.e.*, fractions) do contain pairs of numbers, they themselves are used as numbers with the usual arithmetic operations (`+`, `-`, `*`, and `/`) being defined for them. Containers do not interact with other containers that way: one doesn't subtract a stack from a queue or even from another stack. Moreover, a ratio always contains exactly two numbers (its numerator and its denominator), whereas containers allow the insertion and removal of many elements.

10.4 A `vector` is a container object; an array is a built-in structured type. A `vector` has all the extra functionality provided by the `vectot<T>` class template and the free functions defined in the Standard Library. In the context of object-oriented programming, arrays are like linked lists; they provide a method of implementation.

10.5 A C-string is just a null-terminated array of `chars`. A C++ `string` is an object that has all the functionality provided by the Standard C++ `string` class.

10.6 Fragmented memory occurs when the memory allocated to a process consists of small, disconnected segments. This renders the unallocated memory less usable because it is "fragmented;" *i.e.*, broken into little pieces. Requests from the `new` operator for large contiguous blocks are more likely to fail.

10.7 ***a.*** The `at()` function checks the range on the subscript value and rejects it if it is out of range.

b. So this function is safer to use than ordinary subscripting. But checking the index range takes extra time. So the programmer wanting fast code should opt for using the subscript operator and ensure that the code itself prevents the subscript from going out of range.

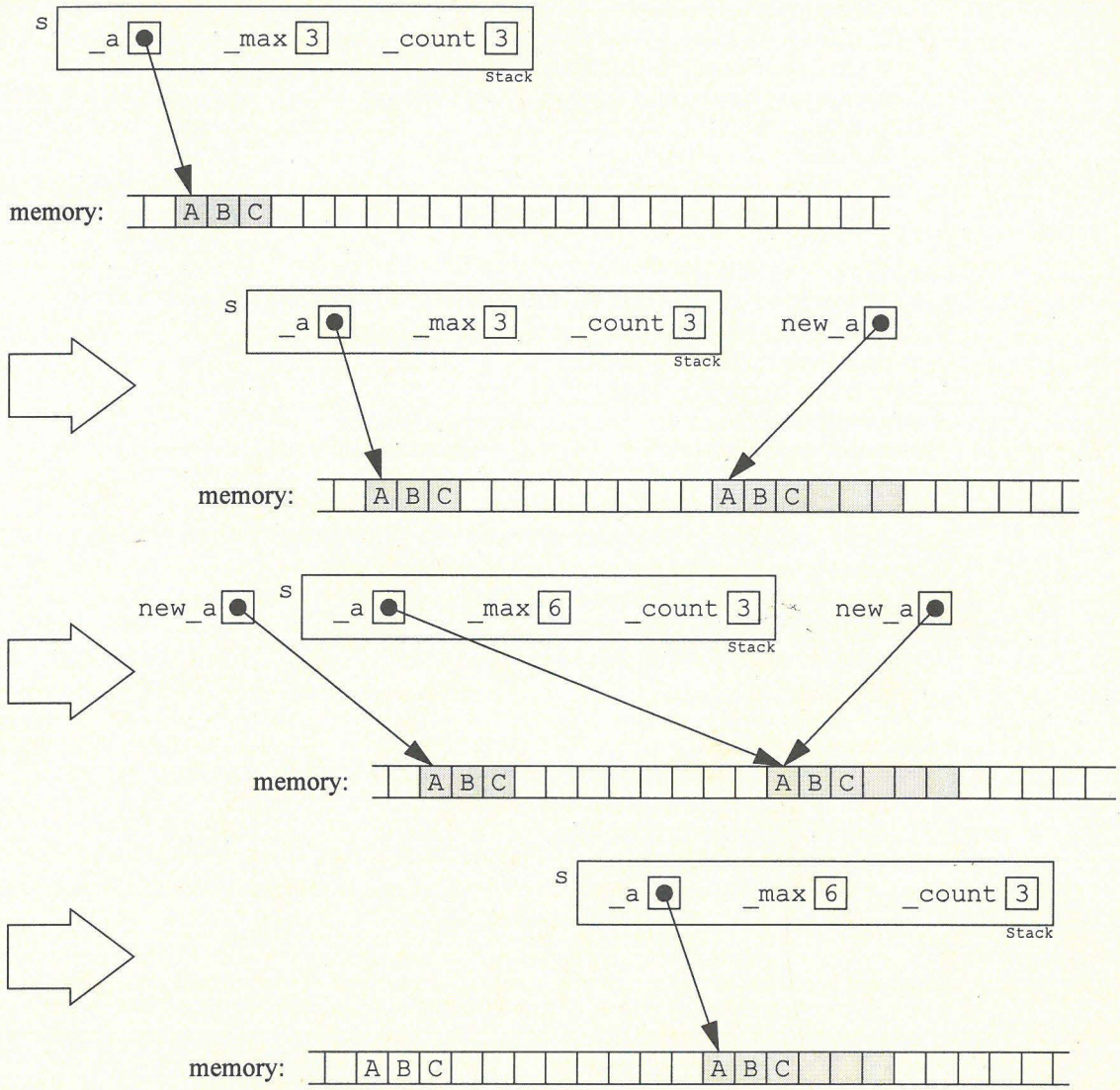
10.8 The `stack`, `queue`, `deque`, and `vector` templates all have the same functionality. The only difference between deques and vectors is the efficiency of their operations. Since a deque is meant to be changed only at its ends, insertions and deletions there are much faster than they are for vectors. On the other hand, the subscript operator and `at()` function are much faster for vectors than for deques. The only differences between stacks, queues, and deques are their names for the functions used to read, insert, and delete at their ends. Both the `stack` and the `queue` templates use `push()` for `push_back()`. The `stack` templates uses `pop()` for `pop_back()` and the `queue` templates uses `pop()` for `pop_front()`. The `stack` template also uses `top()` for the `back()` function. (Technically, the `stack` and `queue` templates are *adaptors* that use the `deque` template as their default implementation. They can also use the `vector` template or the `list` template. These alternate implementations change the efficiency of certain operations.)

10.9 Here are the changes:

```
template<class T>
class Stack
{ public:
    Stack(int);
    ~Stack();
    bool is_empty() const;
    void push(const T);
    T pop();
  private:
    void _expand();  // used to prevent stack overflow
    T* _a;          // the stack itself: a dynamic array of type T
    int _max;       // the maximum number of elements on the stack
    int _count;      // the actual number of elements on the stack
};
template<class T>
void Stack<T>::push(const T x)
{ if (_count == _max) _expand();
  _a[_count++] = x;
}
template<class T>
void Stack<T>::_expand()
{ T* new_a = new T[2*_max];     // allocate storage for array
new_a
  assert(new_a != 0);                // abort if not enough memory
  for (int i=0; i < _count; i++)      // copy all of _a into new_a
    new_a[i] = _a[i];
  T* temp_a = _a;                     // hold address _a in temp_a
  _a = new_a;                        // copy address new_a into _a
  _max *= 2;                                          // double _max
  delete [] temp_a;              // return old storage back to heap
}
```

The execution of the `_expand()` function looks like the picture shown below.
The object `s` would have been created by

```
Stack<char> s;
s.push('A');
s.push('B');
s.push('C');
```



Then `_expand()` is called with `s._a[0]` containing `'A'`, `s._a[1]` containing `'B'`, `s._a[2]` containing `'C'`, `s._max` containing 3, and `s._count` containing 3. The array `new_a` is created and allocated 6 bytes in memory. The 3 elements (`'A'`, `'B'`, and `'C'`) in `_a` are then copied into the first 3 elements of `new_a`. Then the pointer `temp_a` is created and initialized to point to the same memory location to which `_a points`. This is done so that access to that location can be kept after `_a` is reassigned to point to the new array `new_a`. Finally the space used by the old array is returned to the heap so that it can be reused later by other processes.

10.10

```
#include "algo"       // defines swap()
#include <assert.h>   // defines assert()
template <class T>
class Ratio
{   friend Ratio operator+(const Ratio&, const Ratio&);
    friend Ratio operator-(const Ratio&, const Ratio&);
    friend Ratio operator*(const Ratio&, const Ratio&);
    friend Ratio operator/(const Ratio&, const Ratio&);
  public:
    Ratio(T num=0, T den=1): _num(num), _den(den) { _reduce(); }
   Ratio(const Ratio& x): _num(x._num), _den(x._den) {_reduce();}
```

```
    Ratio& operator-() { _num *= -1; }
    T num() const { return _num; }
    T den() const { return _den; }
    operator double() const { return double(_num)/double(_den); }
  private:
    T _num;
    T _den;
    void _reduce();  // reduces fraction to lowest terms
};
template <class T>
Ratio<T> operator+(const Ratio<T>& x, const Ratio<T>& y)
{ T num = x._num*y._den + x._den*y._num;
  T den = x._den*y._den;
  Ratio<T> z(num, den);
  z._reduce();
  return z;
}
template <class T>
Ratio<T> operator-(const Ratio<T>& x, const Ratio<T>& y)
{ T num = x._num*y._den - x._den*y._num;
  T den = x._den*y._den;
  Ratio<T> z(num, den);
  z._reduce();
  return z;
}
template <class T>
Ratio<T> operator*(const Ratio<T>& x, const Ratio<T>& y)
{ T num = x._num*y._num;
  T den = x._den*y._den;
  Ratio<T> z(num, den);
  z._reduce();
  return z;
}
template <class T>
Ratio<T> operator/(const Ratio<T>& x, const Ratio<T>& y)
{ assert(y._num != 0);
  T num = x._num*y._den;
  T den = x._den*y._num;
  Ratio<T> z(num, den);
  z._reduce();
  return z;
}
template <class T>
T gcd(T m, T n)
{ assert(m > 0 && n > 0);
  while (m > 0)
  { if (m < n) swap(m,n);
    m %= n;
  }
  return n;
}
template <class T>
void Ratio<T>::_reduce()
{ if (_num == 0 || _den == 0) { _num = 0; _den = 1; return; }
  if (_den < 0) { _num *= -1; _den *= -1; }
  int sgn = ( _num < 0 ? -1 : 1 );
```

```
    _num *= sgn;
    T g = gcd(_num, _den);
    _num /= g;
    _den /= g;
    _num *= sgn;
  }
```

10.11 This conforms to the Standard C++ `deque` class template defined in `<deque>`:

```
template <class T>
class Deque
{ public:
    Deque(const list& x) : _list(x);      // auxiliary constructor
    int size() const { return _list.size(); }
    bool empty() const { return bool(_list.size() == 0); }
    T& front() const { return _list.front(); }
    T& back() const { return _list.back(); }
    void push_front(const T& t) { _list.push_front(t); }
    void push_back(const T& t) { _list.push_back(t); }
    void pop_front() { _list.pop_front(); }
    void pop_back() { _list.pop_back(); }
  private:
    list<T> _list;
};
```

10.12 This conforms to the Standard C++ `stack` class template defined in `<stack>`:

```
template <class T>
class Stack
{ public:
    Stack(const deque& deque) : _deque(deque); // aux. constructor
    int size() const { return _deque.size(); }
    bool empty() const { return bool(_deque.size() == 0); }
    T& top() const { return _deque.back(); }
    void push(const T& t) { _deque.push_back(t); }
    void pop() { _deque.pop_back(); }
  private:
    deque<T> _deque;
};
```

10.13 This conforms to the Standard C++ `queue` class template defined in `<queue>`:

```
template <class T>
class Queue
{ public:
    Queue(const deque& deque) : _deque(deque); // aux. constructor
    int size() const { return _deque.size(); }
    bool empty() const { return bool(_deque.size() == 0); }
    T& front() const { return _deque.front(); }
    T& back() const { return _deque.back(); }
    void push(const T& t) { _deque.push_back(t); }
    void pop() { _deque.pop_front(); }
  private:
    deque<T> _deque;
};
```

10.14

```
int main()
{ queue<Person> line;
  line.push(Person("Clinton", "William"));
  Address whitehouse("1600 Pennsylvania Avenue", "Washington");
  line.front().set_address(whitehouse);
  line.push(Person("Clinton", "Hillary"));
  line.push(Person("Gore", "Albert"));
```

```
    cout << "There are " << line.size() << " people in line.\n";
    cout << "The name of the first person in line is "
        << line.front().fname() + " " + line.front().lname()<<endl;
    cout << "His street address is "
         << line.front().address().street() << endl;
    cout << "The name of the last person in line is "
         << line.back().fname() + " " + line.back().lname() <<
endl;
}
```

The output from this program is

```
There are 3 people in line.
The name of the first person in line is William Clinton
His street address is 1600 Pennsylvania Avenue
The name of the last person in line is Albert Gore
```

10.15 Note that this alternative function requires that `v` be passed by value because it changes `v`:

```
vec projection(vec v, bits& b)
{ vec::iterator itv = v.begin();
  bits::iterator itb = b.begin();
  while (itv != v.end())
  { if (*itb++) v.erase(itv);
    ++itv;
  }
  return v;
}
```

10.16

```
int number_of_numbers(SLI& it)
{ int count=0;
  string element = *it;
  int k = element.find('\t');
  return (element.length() - k + 1)/4;
}
int freq(SL& x, string& word)
{ string x_word;
  for (SLI it=x.begin(); it != x.end(); ++it)
  { string x_word = extract_word(it);
    if (word == x_word) return number_of_numbers(it);
  }
  return 0;
}
```

10.17

```
template <class Src, class T>
Src find(Src i, Src j, const T& t)
{ while (i != j && *i != t)
    ++i;
  return i;
}
```

10.18

```
template <class Src, class T>
Src adjacent_find(Src i, Src j)
{ if (i == j) return j;
  for (Src k = i+1; k != j; ++i)
  { if (*k == *i) return i;
    k = i;  // with linked lists, this would be faster than ++k
  }
  return j;
}
```

10.19

```
template <class Src, class T>
void count(Src i, Src j, const T& t, int& n)
{ while (i != j)
    if (*i++ == t) ++n;
}
```

10.20

```
template <class Src>
bool equal(Src i, Src j, Src p)
{ while (i != j)
    if (*i++ != *p++) return false;
  return true;
}
```

10.21

```
template <class Src1, class Src2>
Src1 search(Src1 i, Src1 j, Src2 p, Src2 q)
{ while (i != j)
    return equal(p, q, i++);
  return j;
}
```

10.22

```
template <class Src, class Dst>
Dst copy(Src i, Src j, Dst p)
{ while (i != j)
    *p++ = *i++;
  return p;
}
```

10.23

```
template <class Src, class T>
void replace(Src i, Src j, const T& t, const T& u)
{ for (; i != j; ++i)
    if (*i == t) *i = u;
}
```

10.24

```
template <class Src, class Dst>
Dst swap_ranges(Src i, Src j, Dst p)
{ while (i != j)
    swap(*i++, *p++);
  return p;
}
```

10.25

```
template <class Src, class T>
void fill(Src i, Src j, const T& t)
{ while (i != j)
    *i++ = t;
}
```

10.26

```
template <class Src, class T>
Src remove(Src i, Src j, const T& t)
{ Src k=i;
  while (i != j)
  { if (*i != t) *k++ = *i;
    ++i;
  }
  return k;
}
```

10.27

```
template <class Src, class Dst, class T>
Dst remove_copy(Src i, Src j, Dst p, const T& t)
{ while (i != j)
  { if (*i != t) *p++ = *i;
    ++i;
  }
  return p;
}
```