STL Distilled and Generic Programming

Ira Pohl

This eMatter edition is related to *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide*, a paperful book by Addison-Wesley Publishing Company, Inc 1997.

ISBN 0-201-69587-1

About the Author

Ira Pohl, Ph.D., is a professor of Computer Science at the University of California, Santa Cruz. He has over 30 years of experience as a software methodologist. His teaching and research interests are in the areas of artificial intelligence, programming languages, practical complexity problems, heuristic search methods, deductive algorithms, and educational and social issues. He originated error analysis in heuristic search methods and deductive algorithms.

He has lectured at Berkeley, Stanford, the Vrije University in Amsterdam, the Courant Institute, Edinburgh University in Scotland, and Auckland University in New Zealand.

When not programming, he enjoys riding bicycles in Aptos, California, with his wife Debra and daughter Laura.

Ira's web address is *http://www.cse.ucsc.edu/~pohl/*. He can be reached via email at *pohl@cse.ucsc.edu*.

Other Publications by Ira Pohl

Ira Pohl is the coauthor with Al Kelley of a series of books published by Addison-Wesley and Benjamin Cummings on the C programming language:

A Book on C: An Introduction to Programming in C
C by Dissection
Turbo C: The Essentials of C Programming

He is also coauthor will Charlie McDowell of the Addison-Wesley publication due out in Fall of 1999:

Java by Dissection: The Essentials of Java Programming

He is the sole author of Addison-Wesley or Benjamin Cummings publications:

C++ for C Programmers
C++ for Pascal Programmers
C++ for Fortran Programmers
Turbo C++
Object Oriented Programming Using C++
C++ Distilled

His first book, coauthored with Alan Shaw, was a pioneering text on computer science (Computer Science Press, 1981):

The Nature of Computation: An Introduction to Computer Science

eMatter publications available through *FatBrain*:

The C++ Bookshelf: Distilled
Object Oriented Programming Using C++
C++ for Pascal Programmers

Other eMatter books forthcoming.

4 ▼

Contents

Preface

STL Distilled and Generic Programming is an a eMatter companion volume to *C++ Distilled*: Addison-Wesley. It updates and adds material on templates, generic programming and STL as found in ANSI C++. It supplements and brings up-to-date existing literature.

This eMatter book is a concise road map and style guide to generic programming in C++. It distills key ideas and practice for the ANSI standard C++ language and includes many programming tips. It is easily used with any C++ programming book (see Chapter 11, “References,” on page 74, for a selection), but is especially suitable when used with one of the author’s books, such as *Object-Oriented Programming Using C++, 2nd Edition* (OPUS 97) or *C++ for C Programmers, 3rd edition*. (P 99).

Each section has the syntax, semantics, and examples of the language element. There are style and programming tips at the end of most sections. Examples have a consistent professional style to be mimicked by programmers.

This eMatter book in conjunction with *C++ Distilled* is a distillation of the C++ ANSI standard, which is approximately 700 detailed technically dense pages, and rather overwhelming. Fortunately most programmers do not need such detail; indeed, many of the features are highly specialized and little used. Most programmers need to be able to quickly review some syntax or semantics they have not recently used.

C++ has had many recent additions, including STL. These can be used readily by someone already proficient in basic C++, but most books have yet to treat these topics. This eMatter book can provide a handy guide to this important library.

Most programming is done in my imitation of existing code and idioms. These examples use my prescriptions and programming tips (“Dr. P’s Prescriptions”) which are a distillation of considerable professional practice.

Hello World Program

In file hello1.cpp

```
//Traditional first program
#include <iostream.h>

int main()
{
   cout << "HELLO WORLD!" << endl;
}
```

Dr. P's Prescriptions: Style and Rule Tips

- Use the style found in this book.
- Be consistent with whatever style you choose.

Prescription Discussion

Style emphasizes clarity and community norms. Consistency, while the hobgoblin of small minds, is well suited to large computer codes.

Acknowledgments

This eMatter was developed as an extension and to *C++ Distilled*. That book benefited from reviewers Ed Lansinger of General Motors Corporation; Henry A. Etlinger of Rochester Institute of Technology; Glen Deen of Deen Publications, Inc.; Michael Keenan of Columbus State University; and David Gregory. Most importantly, I thank Debra Dolsberry for her invaluable help in the technical editing of this eMatter book, and her careful testing of the code.

Dedication

To Alexander Stepanov and Donald Knuth, who created generic programming and the detailed analysis of best algorithms, respectively.

Chapter 1

Generic Programs

A key problem in programming is programmer productivity. An important technique is code reuse. Generic programming is a critical methodology for enhancing code reuse. Generic programming is about code that can be used over a wide category of types. In C++ there are three different ways to employ generic coding techniques: void* pointers, templates, and inheritance. This chapter will show a simple use of each of these methods. This eMatter book will largely concern C++ templates and how they are employed in STL, the C++ standard template library.

We will start with a simple example of code that can benefit from genericity. This is the everyday application of assigning the contents of one array to a second array.

Array Transfer Function

In file transferArray.cpp

```
//Simple array assignment function

int transfer(int from[], int to[], int size)
{
  for (int i = 0; i < size; i++)
   to[i] = from[i];
  return size;
}
```

This code works for the `int` array type and depends on an appropriate size array being allocated. This piece of code can be readily replicated for different types, but replication has a cost and can introduce errors.

For the following declarations:

```
int a[10], b[10];
double c[20], d[20];

transfer(a, b, 10);            //works fine
transfer(c, d, 20);               //syntax error
```

C++ has a void pointer type that can be used to create generic code. Generic code is code that can work with different types.

Void Array Transfer Function

In file voidTransferArray.cpp

```
//void* generic assignment function

int transfer(void* from, void* to, int elementSize, int size)
{
  int nBytes = size * elementSize;

  for (int i = 0; i < nBytes; i++)
    static_cast<char*>(to)[i] = static_cast<char*>(from)[i];
  return size;
}
```

This code works for any array type. Since `void*` is a universal pointer type any array type can be passed as a parameter. However, the compiler will not catch type errors. For the following declarations:

```
int a[10], b[10];
double c[20], d[20];

transfer(a, b, 10, sizeof(int));       //works fine
transfer(c, d, 20, sizeof(double));    //works fine
transfer(a, c, 10, sizeof(int));          //system dependent
```

C++ has template functions that can be used to create generic code.

Template Array Transfer Function

In file templateTransferArray.cpp

```
//template generic assignment function

template< class T>
int transfer(T* from, T* to, int size)
{
  for (int i = 0; i < size; i++)
    to[i] = from[i];
  return size;
}
```

This code works for the any array type. For the following declarations:

```
int a[10], b[10];
double c[20], d[20];

transfer(a, b, 10);    //works fine
transfer(c, d, 20);    //works fine
transfer(a, c, 10);     //syntax error
```

The template function requires that the type be properly instantiated. It does not allow two distinct types to be used in this form of array transfer. It continues to provide type safety which is important to program correctness.

Dr. P's Prescriptions: General Rules

- Dr. P's first rule of style is "Have a style." (P 97)
- Kernighan and Plauger's first rule of style is "Write clearly—don't be too clever" (KP 74).
- Be consistent in what ever style you choose.
- Use templates instead of `void*` genericity.
- First write a archetypal case and test then recode generically and test.

Prescription Discussion

In this eMatter book we follow the traditional C and C++ style pioneered by Bell Laboratories programmers, such as Kernighan, Ritchie and Stroustrup (KR 88, GRAY 91, ABC 95, P 97).

Several elements of this style can be seen in the our programs. Beginning and ending braces for function definitions line up under each other and under the first character of the function definition. Beginning braces after keywords, such as `do` and `while`, follow the keyword with the ending brace under the first character of that line. This style is in widespread use and makes it easy for others to read your code. The style allows us to distinguish key elements of the program visually, enhancing readability. Style should aim for clarity for both ourselves and others who need to read our code.

Cleverness by its nature is usually obscure. This is the enemy of clarity—hence Kernighan and Plauger's maxim "Write clearly—don't be too clever." Also, inconsistent style tends to obscure.

While `void*` genericity has certain advantages, such as smaller executables than template genericty, it can be more error-prone and less efficient.

It can be difficult to write and test generic code from scratch. Concreteness is a great aid to the program developer. Pick a type that represents the archetypal case. In our example this was the `int` array. Develop the code for this case and test, making sure it's correct. Finally convert this to template code and retest with selected types.

Chapter 2

Iterators and Containers

A container is a data structure that is used to contain a large number of values. The prototypical container is the array. Other familiar containers include the list, queue, stack and map. An iterator is a device for traversing a container. The indexing of an array is a way to sequence through or randomly visit array elements. The C++ pointer is a prototypical iterator.

2.1 A Visitation Example: Accumulate

A standard and important computation is to sum all the elements of an array.

Array Accumulate Function

In file accumulate.cpp

```
//accumulate an integer array of values
int accumulate(int* begin, int* end, int start_value = 0)
{
  int sum = start_value;

  while(begin != end)
    sum += *begin++;
  return sum;
}
```

This algorithm uses pointers in two ways. One: the pointer traverses a given container range. Two: the pointer is dereferenced to obtain the value at a given location. This is a model algorithm for traversing a container and accessing but not mutating or changing their values. Later we will see that STL has very general generic forms of these algorithms.

We convert the above archetypal accumulate algorithm by converting it to a template algorithm. The rules for doing this are usually very simple, namely identify

any types that need to be generic and substitute the template arguments for the concrete types. Retest the code using several representative types.

Template Accumulate Function

In file templateAccumulate.cpp

```
//accumulate an T array of values
#include <iostream>
template <class T>
T accumulate(T* begin, T* end, T start_value = T())
{
  T sum = start_value;

  while (begin != end)
    sum += *begin++;
  return sum;
}

int main()
{
  int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  double x[10] = {1.1, 2.2, 3.3};

  cout << "  sum is  " << accumulate(a, a + 10) << endl;
  cout << "  sum is  " << accumulate(x, x + 10) << endl;
}
```

This code can be instantiated for any array type that can be summed.

Dr. P's Prescriptions: Containers and Iterators

- Use ranges to perform visitation.
- Templatize any standard code to enhance reuse.
- Avoid a commitment to a particular container type.
- Be general where possible.

Prescription Discussion

As in STL, we use a beginning iterator and a sentinel iterator to pass in a range for the traversal of a container. This is a very flexible scheme and is highly efficient. Its flexibility comes from the fact that any contiguous subsection of the container can be specified.

C++ is designed to be template friendly. Code, when designed as a template, benefits from greater abstraction than corresponding specialized code. What I mean by this is that programmers are normally overly clever. This leads to hard to maintain code with possibly subtle bugs. Generic code must be correct over a wide range of types and cannot indulge in cleverness.

Iterators avoid the commitment to a particular container type. In contrast using indices to access arrays does not allow for pointer traversal as used in list and tree containers. Generalization benefits by avoiding commitments. Generalization is the heart and brilliance of the STL library.

Chapter 3

Algorithms

STL is a library of generic algorithms. These algorithm represent best practice (K 97). Given a particular problem such as internal sorting for a sequence container supporting random access, such as an array, what is known is that quicksort is an algorithm with good behavior. The problem for STL is how to capture such an algorithm in its most general form without degrading its performance.

An example of a best practice algorithm is the following code for finding the minimum and maximum element in a container using fewest number of comparisons (P 72). This example was developed when I was trying to understand more complicated examples of sorting and led to one of the first formal proofs based on an adversary strategy. It is intuitively obvious, but it is still hard to see why it is always best in terms of number of comparisons.

We will develop the algorithm in pieces. For its history and influence in the analysis of algorithms (K 98)

```
//minimum of an  array of values
int  min(int* begin, int* end)
{
  int minimum = *begin;

  while(++begin != end)
    if( minimum > *begin)
      minimum = *begin;
  return minimum;
}
```

This is our garden variety find the minimum of a sequence algorithm. As with other STL inspired algorithms we write it in terms of iterator parameters. For n elements it requires n-1 comparisons.

Symmetrically we can immediately write out the maximum algorithm.

```
//maximum of an  array of values
int  max(int* begin, int* end)
{
  int maximum = *begin;

  while(++begin != end)
    if( maximum < *begin)
      maximum = *begin;
  return maximum;
}
```

Now we wish to return from our analysis of an unsorted sequence the minimum and maximum element of a sequence. Do this as a return value requires a new type the `int_pair`.

```
class int_pair{public: int first, second;};
```

Again this is inspired by type `pair<>` found in the STL library, where among its uses are its ability to store map related values (STL 96).

A first attempt at producing a min-max algorithm would be to use both the minimum and maximum algorithms. This would take $2n - 2$ comparisons. We can see that certain comparisons are redundant. in the most elementary case, if we find a minimum we know it will not be simulataneously the maximum unless it is the sole value in the sequence. Indeed any comparison that leads to a minimum produces an element smaller than a maximum, and this element need not be tested for being a maximum. This insight leads to splitting the original sequence in to 2 sequences that contain repectively candidate minima and maxima. The code for this is as follows:

```
void swap(int* p, int* q){int temp = *p; *p = *q; *q = temp;}

//assume an even number of elements
int_pair  min_max(int* begin, int* end, int* middle)
{
  int* mbegin =  begin;
  int* mmiddle = middle;
  int_pair ans;
  while (mbegin != middle) {
    if ( *mbegin > *mmiddle)
      swap ( mbegin, mmiddle);
    mbegin++;
    mmiddle++;
  }
```

```
  ans.first = min(begin, middle);
  ans.second = max(middle, end);
  return ans;
}

//Test Program
int main()
{
  int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int_pair mm;
  mm = min_max(a, a + 10, a + 5);
  cout << " minimum and  maximum is  "
      << mm.first << "  "<< mm.second<< endl;
}
```

This leads to performance that is 3n-2 comparisons. The original n elements are compared pair wise and divided into two groups of n/2 elements. This takes n/2 comparisons; the minimum is found in n/2-1 comparison and likewise the maximum. We leave as an exercise what is needed for an odd number of elements.

Notice this algorithm can be rewritten for any type that allows comparison. Also the original ordering of elements may be changed by the algorithm. This means the code is mutating. These categories of mutating and sorting are used by STL. It remains to templatize the algorithm in order to make it generic. We leave this as an exercise.

Dr. P's Prescriptions: Algorithms

- Read The Art of Computing by Donald Knuth
- Use a best algorithm

Prescription Discussion

The study of algorithms is central to good coding. Knuth in his three volume Art of Computing has exhaustively presented analysis and design of datastructures and algorithms. Studying these volumes for the modern computer scientist and programmer is analagous to an ancient geometer studing Euclid's eements. Without this study you have no basis to understand what a good algorithm is or how to develop one.

As fast as computers become, there is no substitute for using a best algorithm in many instances. Resource use is almost always an issue somewhere in serious code. To appreciate STL as an achievement requires the understanding of best algorithm for different circumstance.

Chapter 4

Templates

The keyword `template` is used to implement parameterized types. Rather than repeatedly recoding for each type, the template feature allows instantiation to generate code automatically for each type. The following code is a simple implementation of a generic stack container class.

Template Stack Program

In file stack_p.cpp

```
template <class T>      //parameterize T
class stack {
public:
  stack();
  explicit stack(int s);
  T&  pop();
  void  push(T);
  .....
private:
  T*   item;
  int  top;
  int  size;
};
```

A template declaration has the form:

`template` < *template arguments* > *declaration*

and a template argument can be:

`class` *identifier*
argument declaration

The `class` *identifier* arguments are instantiated with a type. Other argument declarations are instantiated with constant expressions of a nonfloating type, and can be a function or address of an object with external linkage as shown in the following code.

In file array.cpp

```
template<class T, int n >
class array_n {
   .....
private:
  T  items[n];                 //n explicitly instantiated
  .....
};

array_n<complex, 1000>  w;      //w is an array of complex
```

Member function syntax, when external to the class definition, is as follows.

```
template <class T>
T& stack<T>::pop()
{
  return(item[top--]);
}
```

The class-name used by the scope resolution operator includes the template arguments, and the member function declaration requires the template declaration as a preface to the function declaration.

4.1 Template Parameters

The above template can be rewritten with default parameters for both the `int` argument and the type. For example:

```
template<class T = int, int n = 100>
class array_n {
  .....
};
```

The default parameters can be instantiated when declaring variables, or can be omitted, in which case the defaults will be used.

Templates can use the keyword `typename` in place of `class`. For example:

```
template<typename T = double, double* ptr_dbl>
```

This allows the template code to use a pointer to `double` argument. Ordinary floating-point arguments are not allowed, only pointer and reference to floating-point arguments are allowed.

A template argument can also be a template parameter. For example:

```
template<typename T1, template<class T2> class T3>
```

This allows very sophisticated metatemplates—templates instantiated with templates—to be coded. Libraries such as STL can use such features.

4.2 Function Template

Until 1995 compilers allowed ordinary functions to be parameterized using a restricted form of template syntax. Only `class` *identifier* instantiation is allowed. It must occur inside the function argument list:

Generic Swap Function

In file swap.cpp

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

  temp = x;
  x = y;
  y = temp;
}
```

```
//ANSI C++ but unavailable in many current compilers
template <class T, int n>
T foo()
{
  T  temp[n];
 .....
}

foo<char, 20>();      //use char, 20 and call foo
```

A function template is used to construct an appropriate function for any invocation that matches its arguments unambiguously:

```
swap(i, j);           //i j int - okay
swap(c1, c2);         //c1, c2 complex - okay
swap(i, ch);          //i int ch char - illegal
```

The overloading function selection algorithm is as follows.

Overloaded Function Selection Algorithm

1. Exact match with trivial conversions allowed on a nontemplate function.
2. Exact match using a function template.
3. Ordinary argument resolution on a nontemplate function.

In the previous example, an ordinary function declaration whose prototype was

```
void swap(char, char);
```

would have been invoked on `swap(i, ch)`.

4.3 Friends

Template classes can contain friends. A friend function that does not use a template specification is universally a friend of all instantiations of the template class. A friend function that incorporates template arguments is a friend only of its instantiated class:

```
template <class T>
class matrix {
public:
  friend void  foo_bar();              //universal
  friend vect<T>  product(vect<T> v);  //instantiated
  .....
};
```

4.4 Static Members

Static members are not universal, but are specific to each instantiation:

```
template <class T>
class foo {
public:
  static int  count;
  .....
};

.....
foo<int>     a, b;
foo<double>  c;
```

The static variables `foo<int>::count` and `foo<double>::count` are distinct. The variables `a.count` and `b.count` reference `foo<int>::count`, but `c.count` references `foo<double>::count`. It is preferable to use the form `foo<`*type*`>::count` since this makes it clear that the variable referenced is the static variable.

4.5 Specialization

When the template code is unsatisfactory for a particular argument type it can be specialized. A template function overloaded by an ordinary function of the same type—that is, one whose list of arguments and return type conform to the template declaration—is a specialization of the template. When the specialization matches the call, then it, rather than code generated from the template, is called.

```
void maxelement<char*>(char*a[],char* &max,int size);
//specialized using strcmp() to return max string
```

This would be a specialization of the previously declared template for `template<class T>maxelement()`. Class specializations are also possible, as in:

```
class stack<foobar_obj>   { /*specialize for foobar_obj */ };
```

Templates Program

In file vect_it.cpp

```
//templates for vect with associated iterator class

#include  <iostream.h>
#include  <assert.h>      //for assert

template <class T> class  vect_iterator;
template <class T>
class vect {
public:
  //constructors and destructor
  typedef T* iterator;
  explicit vect(int n = 10);  //default constructor
  vect(const vect& v);        //copy constructor
  vect(const T a[], int n);   //from array
  ~vect() { delete [] p; }
```

```
  iterator begin(){ return p; }
  iterator end(){ return p + size; }
  T&  operator[](int i) const;
  vect& operator=(const vect& v);
  friend vect operator+(const vect& v1, const vect& v2);
  friend ostream& operator<<(ostream& out, const vect<T>& v);
  friend class vect_iterator<T>;

private:
  T*    p;     //base pointer
  int   size;  //number of elements
};

//default constructor
template <class T>
vect<T>::vect(int n): size(n)
{
  assert(n > 0);
  p = new T[size];
  assert(p != 0);
}

//copy constructor
template<class T>
vect<T>::vect(const vect<T>& v)
{
  size = v.size;
  p = new T[size];
  assert (p != 0);
  for (int i = 0; i < size; ++i)
    p[i] = v.p[i];
}

//Initializing vect from an array
template<class T>
vect<T>::vect(const T a[], int n) : size (n)
{
  assert (n > 0);
  p = new T[size];
  assert (p != 0);
  for (int i = 0; i < size; ++i)
    p[i] = a[i];
}
```

```
//overloaded subscript operator
template<class T>
T& vect<T>::operator[](int i) const
{
  assert (i >= 0 && i < size);
  return p[i];
}

//overloaded output operator
template<class T>
ostream& operator<<(ostream& out, const vect<T>& v)
{
  for (int i = 0; i <= (v.size-1); ++i)
    out << v.p[i] << '\t';
  return (out << endl);
}

template<class T>
vect<T>& vect<T>::operator=(const vect<T>& v)
{
  assert(v.size == size);
  for (int i = 0; i < size; ++i)
    p[i] = v.p[i];
  return *this;
}

template<class T>
vect<T> operator+(const vect<T>& v1, const vect<T>& v2)
{
  assert(v1.size == v2.size) ;
  vect<T>  sum(v1.s);

  for (int i = 0; i < s; ++i)
    sum.p[i] = v1.p[i] + v2.p[i];
  return sum;
}
```

```
template<class T>
void init_vect(vect<T>& v, int start, int incr)
{
  for (int i = 0; i <= v.ub(); ++i) {
    v[i] = start;
    start += incr;
  }
}

int main()
{
   vect<double> v(5), t(5);
   vect<double>::iterator p ;
   int i = 0;
   for (p = v.begin() ; p != v.end(); ++p)
    *p = 1.5 + i++;

   do {
    --p;
    cout << *p << " , ";
   } while (p != v.begin());
   t = v; //test assignment
   v = v + t; //test addition
   for (p = v.begin() ; p != v.end(); ++p)
    cout << *p << " , ";
   cout << endl;
}
```

Dr. P's Prescriptions: Templates

- Use templates for containers, such as stack or tree.
- Use template functions in preference to functions acting on `void*` arguments.
- Design templates by first writing a prototype as an ordinary class.
- Use templates in preference to inheritance.

Prescription Discussion

Templates are especially good for code that is repeatedly required with different types. Container class code is usefully generalized by coding with templates. A container is an object whose primary purpose is to store values. A classic example of a container is a stack. Templates allow such code to be reused over arbitrary type with type-safety that is checked at compile-time.

Before templates were used much generic code in C++ was written using `void*` arguments to functions. This generic pointer type can accept any specific pointer type as an argument. This code can largely be replaced with templates. The code is again compile-time type-checked. Also, template functions need not manipulate arguments indirectly with pointers.

Template code is easily developed through generalization of a specific typical case. Develop code with the specific case first; for example, develop code for a stack of integers. Only after all of this code is satisfactory and debugged should it be converted to a general template. Then retest this code over a selection of data types that might represent typical template use.

Templates and inheritance are both techniques for reusing code. When both techniques are possibilities for developing classes templates are preferred to inheritance because they are usually more efficient and lead to simpler class design. Inheritance couples classes.

Chapter 5

STL: Basics and the Container vector

In this section we wish to give the basic ideas behind STL. We will use as our principle example the container class vector and its use. Arguably the most used container class is the vector. Indeed we recommend along with other authorities (S 97) that it wherever possible replace raw arrays. The vector class is safer than then a native array and is more flexible. There might be a slight tradeoff in use of resources, but in most applications it is the superior data structure.

Vector Program

In file test_vector.cpp

```
//Simple STL vector program
#include <iostream>
#include <vector>
using namespace std;

int main ()
{
   vector<int> v(100); //100 is the vector's size

   for (int i = 0; i < 100; i++)
      v[i] = i;
   for (vector<int>::iterator p = v.begin(); p != v.end(); p++)
      cout << *p << '\t';
}
```

The STL container `vector` is used in place of an ordinary `int` array. The first `for`-statement is written in exactly the same manner as a C++ loop on ordinary data. The second `for`-statement is written using the iterator `p`. An iterator behaves as a pointer. STL provides the member functions `begin()` and `end()` as initial and terminal position values for the container. Note that `end()` returns the iterator position (or address) one past the last element of the container. Thus, `end()` is a guard location, or value signaling that you are finished traversing the container.

The iterator is conceptually a pointer. In some cases the template generates a pointer. Regardless of what the template generates, the user of a vector can code as if he had pointer or array indexing as is the case for raw C++ arrays. The vector<> container type is better than an array in several ways. It has more functionality. In the previous example we had `begin` and `end` locations available. We did not have to retain array bounds. The vector<> type resizes automatically. We see some of these properties in the following simple example.

Vector2 Program

In file test_vector2.cpp

```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
  int data[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
  vector<int> v;
  vector<int>::iterator p;

  cout << " size of v is " << v.size();
  cout << " maximum size of v is " << v.size() << endl;
  for (int i = 0; i < 10; i++)
    v.push_back(data[i]);
  for (p = v.begin(); p != v.end(); p++)
    cout << *p << " , " ;
  cout << endl;
  cout << " size of v is " << v.size();
  cout << " maximum size of v is " << v.max_size()<< endl;
}
```

Notice the `vector v` is not declared with any parameters. It will start as a size 0 vector. When we use `push_back()` we automatically add elements to the end of `v` and increase its size as needed. Try this program and see what your system prints for the initial `v.size()` and the final `v.size()` and `v.max_size()`.

The vector is the prototypical sequence container. It can be indexed through. Indeed its elements can be accessed randomly making it very flexible. as we shall see the two other main sequence containers, list and deque, share many but not all of the vector containers properties.

STL vector<> Definitions	
vector<>::value_type	type of value held in the vector<>
vector<>::reference	reference type to value
vector<>::const_reference	const reference
vector<>::pointer	pointer to reference type
vector<>::iterator	iterator type
vector<>::const_iterator	const iterator
vector<>::reverse_iterator	reverse iterator
vector<>::difference_type	represents the difference between two vector<>::iterator values
vector<>::size_type	size of a vector<>

These are typedefs provided for thervector container class. For example, vector<char>::value_type means a character value is stored in the vector container. Such a container could be traversed with a vector<char>::iterator.

Vectors allow equality and comparison operators. They also have an extensive list of standard member functions.

STL Vector<> Members	
vector<>::vector<>()	default constructor
vector<>::vector<>(c)	copy constructor
c.begin()	beginning location of vector<> c
c.end()	ending location of vector<> c
c.rbegin()	beginning for a reverse iterator
c.rend()	ending for a reverse iterator
c.size()	number of elements in vector<>
c.max_size()	largest possible size
c.empty()	true if the vector<> is empty
c.swap(d)	swap two vector<>s

STL vector<> Operators	
== != < > <= >=	equality and comparison operators using vector<>::value_type

So far we have the container vector and the traversal scheme based on iterators. But we also get a variety of algorithms that can be used with the vector container. In the following example we will show how some of these work.

Vector_Algoprithm Program

In file test_vector_algoritm.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
using namespace std;
int main()
{
  vector<int> v(10000);       //size 100 int vector
  vector<double> w(10000);    //size 100  double vector

  generate(v.begin(), v.end(), rand);//use rand() to initialize
  cout << "min element is = "
       << *min_element(v.begin(), v.end()) << endl;
  cout << "min element is = "
       << *max_element(v.begin(), v.end()) << endl;
  for(int i = 0; i < v.size(); i++)
     w[i] = v[i] / static_cast<double>(RAND_MAX);
  double average = accumulate(w.begin(), w.end(), 0.0)/
w.size();
  cout << "average is " << average << endl;
  sort(v.begin(), v.end());
}
```

In this example we can use generic algoritms to find the maximum and minimum element in a large vector. We can generate values and assign them to the elements in

the sequence. In the above example we use the pseudorandom number generator `rand()` from the library. We could as well use our own function. While finding the average element value is not directly in the STL library, `accumulate()` can readily be used to produce the sum of the elements and be divided by `container.size()` to find the average. the sort routine works only on elements that can be compared. It is *O(n ln(n))* and is a version of quicksort optimized for STL containers.

Dr. P's Prescriptions: Basics and the Container `vector`

- Use vectors in place of arrays.
- Use STL algorithms in place of idiosyncratic code.
- Use sequence ranges as arguments instead of the container.

Prescription Discussion

Vectors are better in almost all regards to arrays. The possible small efficiency or resource losses in some situations are not worth using arrays for. Using vectors are 90% of the value of STL. So if you go no further than mastering vector manipulation, you will gain the largest value for study effort.

One is always tempted to design your own version of an algorithm. There is the psychological value of authorship. You need to resist this, as STL reuse technology is very effective. Others can readily maintain or extend such code. You will be more effective. Special cases may exist where it is desireable to use your own form of a sort or accumulate algorithm, but these are rare in professional practice.

It is possible to write algorithms that just pass in the vector and process it. These algorithms will be shown in some of our examples, but it is not STL style. STL style is to use iterator ranges. This is efficient and flexible. Its only downside is that it requires an extra argument over passing in the container and assume that the range is implicitly `(begin(), end())`.

Chapter 6

STL: Containers

The standard template library is the C++ standard library providing generic programming for many standard data structures and algorithms. The library provides containers, iterators, and algorithms that support a standard for generic programming. We present a brief description emphasizing these three components.

The library is built using templates and is highly orthogonal in design. Components can be used with one another on native and user-provided types through proper instantiation of the various elements of the STL library (STL 96 and STLP 96). Different header files are required depending on the system. Our examples conform to the ANSI standard and are encapsulated in `namespace std`.

STL List Container

In file stl_cont.cpp

```
#include <iostream>
#include <list>                          //list container
#include <numeric>                       //for accumulate
using namespace std;

void print(const list<double> &lst)      //using an iterator
{                                        //to traverse lst
  list<double>::const_iterator p;
  for (p = lst.begin(); p !=lst.end(); ++p)
    cout << *p << '\t';
  cout << endl;
}
```

```
int main()
{
  double w[4] = { 0.9, 0.8, 88, -99.99 };
  list<double> z;
  for (int i = 0; i < 4; ++i)
    z.push_front(w[i]);
  print(z);
  z.sort();
  print(z);
  cout << "sum is "
       << accumulate(z.begin(), z.end(), 0.0)
       << endl;
}
```

Here, a list container is instantiated to hold `double`s. An array of `double`s is pushed into the list. The `print()` function uses an iterator to print each element of the list in turn. Notice that iterators work like pointers. They have standard interfaces that include `begin()` and `end()` member functions for starting and ending locations of the container. Also, the list interface includes a stable sorting algorithm, the `sort()` member function. The `accumulate()` function is a generic function in the numeric package that uses `0.0` as an initial value and computes the sum of the list container elements by going from the starting location to the ending location; in the above by going from `z.begin()` to `z.end()`.

6.1 Containers

Containers come in two major families: sequence and associative. Sequence containers include vectors, lists, and deques; they are ordered by having a sequence of elements. Associative containers include sets, multisets, maps, and multimaps; they have keys for looking up elements. The map container is a basic associative array and requires that a comparison operation on the stored elements be defined. All varieties of container share a similar interface.

STL Typical Container Interfaces

- Constructors, including default and copy constructors
- Element access
- Element insertion
- Element deletion
- Destructor
- Iterators

Containers are traversed using iterators. These are pointer-like objects that are available as templates and optimized for use with STL containers.

Deque Traversal Function

In file stl_deq.cpp

```
//A typical container algorithm
double sum(const deque<double> &dq)
{
  deque<double>::const_iterator p;
  double s = 0.0;

  for (p = dq.begin(); p != dq.end(); ++p)
    s += *p ;
  return s;
}
```

The deque (double ended queue) container is traversed using a `const_iterator`. The iterator `p` is dereferenced to obtain each stored value in turn. This algorithm will work with sequence containers and with all types that have `operator+=()` defined.

Container classes will be designated as `CAN` in the following description of their interface.

STL Container Definitions	
`CAN::value_type`	type of value held in the `CAN`
`CAN::reference`	reference type to value
`CAN::const_reference`	`const` reference
`CAN::pointer`	pointer to reference type
`CAN::iterator`	iterator type
`CAN::const_iterator`	`const` iterator
`CAN::reverse_iterator`	reverse iterator
`CAN::const_reverse_iterator`	`const` reverse iterator
`CAN::difference_type`	represents the difference between two `CAN::iterator` values
`CAN::size_type`	size of a `CAN`

All container classes have these definitions available. For example, if we are using the vector container class, then `vector<char>::value_type` means a character value is stored in the vector container. Such a container could be traversed with a `vector<char>::iterator`.

Containers allow equality and comparison operators. They also have an extensive list of standard member functions.

STL Container Members	
`CAN::CAN()`	default constructor
`CAN::CAN(c)`	copy constructor
`c.begin()`	beginning location of `CAN c`
`c.end()`	ending location of `CAN c`
`c.rbegin()`	beginning for a reverse iterator
`c.rend()`	ending for a reverse iterator
`c.size()`	number of elements in `CAN`
`c.max_size()`	largest possible size
`c.empty()`	true if the `CAN` is empty
`c.swap(d)`	swap two `CAN`s

STL Container Operators	
== != < > <= >=	equality and comparison operators using CAN::value_type

6.1.1 Sequence Containers

The sequence containers are vector, list, and deque. They have a sequence of accessible elements. In many cases the C++ array type can also be treated as a sequence container.

Sequence Container Program

In file stl_vect.cpp

```
//Sequence Containers - inserting a vector into a deque

#include <iostream>
#include <deque>
#include <vector>
using namespace std;

int main()
{
  int data[5] = { 6, 8, 7, 6, 5 };
  vector<int> v(5, 6);            //5 element vector
  deque<int> d(data, data + 5);
  deque<int>::iterator p;

  cout << "\nDeque values" << endl;
  for (p = d.begin(); p != d.end(); ++p)
    cout << *p << '\t';          //print:6 8 7 6 5
  cout << endl;
  d.insert(d.begin(), v.begin(), v.end());
  for (p = d.begin(); p != d.end(); p++)
    cout << *p << '\t';          //print:6 6 6 6 6 6 8 7 6 5
}
```

The five-element `vector v` is initialized with the value 6. The `deque d` is initialized with values taken from the array `data`. The `insert()` member function places the `v` values in the specified range `v.begin()` to `v.end()` at the location `d.begin()`.

Sequence classes will be designated as `SEQ` in the following description of their interface; these are in addition to the already described `CAN` interface.

STL Sequence Members	
`SEQ::SEQ(n, v)`	`n` elements of value `v`
`SEQ::SEQ(b_it, e_it)`	starts at `b_it` and go to `e_it - 1`
`c.insert(w_it, v)`	inserts `v` before `w_it`
`c.insert(w_it, v, n)`	inserts `n` copies of `v` before `w_it`
`c.insert(w_it, b_it, e_it)`	inserts `b_it` to `e_it` before `w_it`
`c.erase(w_it)`	erases the element at `w_it`
`c.erase(b_it, e_it)`	erases `b_it` to `e_it`

6.1.2 Associative Containers

The associative containers are set, map, multiset, and multimap. They have key-based accessible elements. These containers have an ordering relation, `Compare`, which is the comparison object for the associative container.

Associative Container Program

In file stl_age.cpp

```
//Associative Containers - looking up ages
#include <iostream>
#include <map>
#include <string>
using namespace std;
```

```
int main()
{
  map<string, int, less<string> > name_age;

  name_age["Pohl,Laura"] = 7;
  name_age["Dolsberry,Betty"] = 39;
  name_age["Pohl,Tanya"] = 14;
  cout << "Laura is  " << name_age["Pohl,Laura"]
       << " years old." << endl;
}
```

In the above example, the `map` `name_age` is an associative array where the key is a `string` type. The `Compare` object is `less<string>`.

Associative classes will be designated as `ASSOC` in the following description of their interface. Keep in mind that these are in addition to the already described `CAN` interface.

STL Associative Definitions	
`ASSOC::key_type`	the retrieval key type
`ASSOC::key_compare`	the comparison object type
`ASSOC::value_compare`	the type for comparing `ASSOC::value_type`

The associative containers have several standard constructors for initialization.

STL Associative Constructors	
`ASSOC()`	default constructor using `Compare`
`ASSOC(cmp)`	constructor using `cmp` as the comparison object
`ASSOC(b_it, e_it)`	uses element range `b_it` to `e_it` using `Compare`
`ASSOC(b_it, e_it, cmp)`	uses element range `b_it` to `e_it` and `cmp` as the comparison object

What distinguishes associative constructors from sequence container constructors is the use of a comparison object.

STL Insert and Erase Member Functions	
`c.insert(t)`	inserts `t`, if no existing element has the same key as `t`; returns `pair <iterator, bool>` with `bool` being `true` if `t` was not present
`c.insert(w_it, t)`	inserts `t` with `w_it` as a starting position for the search; fails on sets and maps if key value is already present; returns position of insertion
`c.insert(b_it, e_it)`	inserts the elements in this range
`c.erase(k)`	erases elements whose key value is `k`, returning the number of erased elements
`c.erase(w_it)`	erases the pointed to element
`c.erase(b_it, e_it)`	erases the range of elements

The insertion works when no element of the same key is already present.

STL Member Functions	
`c.find(k)`	returns iterator to element having the given key `k`, otherwise ends
`c.count(k)`	returns the number of elements with `k`
`c.lower_bound(k)`	returns iterator to first element having value greater than or equal to `k`
`c.upper_bound(k)`	returns iterator to first element having value greater than `k`
`c.equal_range(k)`	returns an iterator pair for `lower_bound` and `upper_bound`

The associative containers are set, map, multiset, and multimap. They have key-based accessible elements. These containers have an ordering relation, `Compare`, which is the comparison object for the associative container.

As a further associate container example we will use a multiset to count the number of times each vegetable enters our diet in the course of 100 meals.

Associative Container Program

In file stl_multiset.cpp

```
//Associative Containers - checking up on your diet
#include <iostream>
#include <set> //used for both set and multiset
#include <vector>
using namespace std;
enum vegetables { broccoli, tomato, carrot, lettuce, beet,
                  radish, potato};

int main() {
  vector<vegetables> my_diet(100);
  vector<vegetables>::iterator pos;
  vegetables veg;
  multiset<vegetables, greater<vegetables> > v_food;
  multiset<vegetables, greater< vegetables> >::iterator vpos;

  for (pos = my_diet.begin(); pos != my_diet.end(); pos++){
    *pos = static_cast<vegetables>( rand() % 7);
    v_food.insert(*pos);
  }

  for (veg = broccoli; veg <= potato; veg++)
    cout << v_food.count(veg) << endl;
}
```

This program generates into vector a random diet of vegetables. it then uses the special properties of multiset to perform a count on how often each vegetable is eaten in our diet.

6.1.3 Container Adaptors

Container adaptor classes are container classes that modify existing containers to produce different public behaviors based on an existing implementation. Three provided container adaptors are `stack`, `queue`, and `priority_queue`.

The `stack` can be adapted from `vector`, `list` and `deque`. It needs an implementation that supports `back`, `push_back` and `pop_back` operations. This is a last-in, first-out data structure.

STL Adapted stack Functions	
`void push(const value_type& v)`	places `v` on the stack
`void pop()`	removes the top element of the stack
`value_type& top() const`	returns the top element of the stack
`bool empty() const`	returns `true` if the stack is empty
`size_type size() const`	returns the number of elements in the stack
`operator== and operator<`	equality and lexicographically less than

The `queue` can be adapted from `list` or `deque`. It needs an implementation that supports `empty`, `size`, `front`, `back`, `push_back` and `pop_front` operations. This is a first-in, first-out data structure.

STL Adapted queue Functions	
`void push(const value_type& v)`	places `v` on the end of the queue
`void pop()`	removes the front element of the queue
`value_type& front() const`	returns the front element of the queue
`value_type& back() const`	returns the back element of the queue
`bool empty() const`	returns `true` if the queue is empty
`size_type size() const`	returns the number of elements in the queue
`operator== and operator<`	equality and lexicographically less than

The `priority_queue` can be adapted from `vector` or `deque`. It needs an implementation that supports `empty`, `size`, `front`, `push_back`, and `pop_back` operations. A `priority_queue` also needs a comparison object for its instantiation. The top element is the largest element as defined by the comparison relationship for the `priority_queue`.

STL Adapted priority_queue Functions	
`void push(const value_type& v)`	places `v` in the priority_queue
`void pop()`	removes top element of the priority_queue
`value_type& top() const`	returns top element of the priority_queue
`bool empty() const`	checks for priority_queue empty
`size_type size() const`	shows number of elements in the priority_queue

We adapt the `stack` from an underlying `vector` implementation.

Container Adaptor Program

In file stl_stak.cpp

```
//Adapt a stack from a vector
#include <iostream>
#include <stack>
#include <vector>
#include <string>
using namespace std;

int main()
{
  stack<string, vector<string> > str_stack;
  string quote[3] =
   { "The wheel that squeaks the loudest\n",
     “Is the one that gets the grease\n",
     "Josh Billings\n" };

  for (int i = 0; i < 3; ++i)
    str_stack.push(quote[i]);
  while (!str_stack.empty()) {
    cout << str_stack.top();
    str_stack.pop();
  }
}
```

Check your vendor’s product for specific system-dependent implementations.

Dr. P’s Prescriptions: STL Containers

- For sequence containers, think vector, first, deque, second and list last.
- Use the most efficient container for a computation.
- When adapting remember the underlying structure determines efficiency.

Prescription Discussion

The vector is generally the easiest container to use. It is a simple generalization of the array and as such is most familar to programmers. It is also often the most efficient over a large class of operations. It should be your default container choice. The deque is the next most useful. Its ability to add to both ends of the data structure in constant time is its greatest strength .It is also supports random access access. The list in many ways is the most expensive container class. Its chief benefit is to give you insertion and deletion of internal elements in constant time without destroying existing iterator values. Again be guided by the most frequent operations required by your problem in making these choices.

There is relative ease in switching among container. One container can be constructed by passing an iterator range from another container. Do not be afraid of using multiple representations for some problems that dictate a combination of space-operation cost tradeoffs. The point of STL is to use a most efficient algorithm. Usually this involves selecting the appropriate container.

Container adaption results in a supported interface, such as a stack or priority queue that hides the underlying container implementation. Nevertheless the different implementations dictate the efficiency of the resulting data structure. Your choice should be sensative to what operations and space constraints are important to your problem. When in doubt profile your program.

Chapter 7

STL: Iterators

Navigation over containers is by iterator. Iterators can be thought of as an enhanced pointer type. They are templates that are instantiated as to the container class type they iterate over. There are five iterator types: input, output, forward, bidirectional, and random access (see Section 7.0.1, “Iterator Categories,” on page 39). Not all iterator types may be available for a given container class. For example, random access iterators are available for vectors but not maps.

The input and output iterators have the fewest requirements. They can be used for input and output and have special implementations called `istream_iterator` and `ostream_iterator` for these purposes. (See Section 7.0.2, “`Istream_iterator`,” on page 40, and Section 7.0.3, “`Ostream_iterator`,” on page 41.) A forward iterator can do everything an input and output iterator can do and can additionally save a position within a container. A bidirectional iterator can go both forward and backward. A random access iterator is the most powerful and can access any element in a suitable container, such as a `vector` in constant time.

Container Iterator Program

In file stl_iter.cpp

```
//Use of an output iterator
#include <iostream>
#include <set>
using namespace std;
```

```
int main()
{
  int primes[4] ={ 2, 3, 5, 7 }, *ptr = primes;
  set<int, greater<int> > s;
  set<int, greater<int> > :: const_iterator  const_s_it;

  while (ptr != primes + 4 )
    s.insert(*ptr++);
  cout << "The primes below 10 : " << endl;
  for (const_s_it = s.begin();
      const_s_it != s.end(); ++const_s_it)
   cout << *const_s_it << '\t';
}
```

The above program uses an iterator for a set container to output one-digit primes. Such an iterator needs to have the ability to autoincrement and to be dereferenced.

7.0.1 Iterator Categories

Input iterators support equality operations, dereferencing, and autoincrement. An iterator that satisfies these conditions can be used for one-pass algorithms that read values of a data structure in one direction. A special case of the input iterator is the `istream_iterator`.

Output iterators support dereferencing restricted to the left-hand side of assignment and autoincrement. An iterator that satisfies these conditions can be used for one-pass algorithms that write values to a data structure in one direction. A special case of the output iterator is the `ostream_iterator`.

Forward iterators support all input/output iterator operations and additionally support unrestricted use of assignment. This allows position within a data structure to be retained from pass to pass. Therefore, general one-directional multipass algorithms can be written with forward iterators.

Bidirectional iterators support all forward iterator operations as well as both autoincrement and autodecrement. Therefore general bidirectional multipass algorithms can be written with bidirectional iterators.

Random access iterators support all bidirectional iterator operations as well as address arithmetic operations such as indexing. In addition, random access iterators support comparison operations. Therefore, algorithms such as quicksort that require efficient random access in linear time can be written with these iterators.

Container classes and algorithms dictate the category of iterator available or needed, so `vector` containers allow random access iterators, but `lists` do not. Sorting generally requires a random access iterator, but finding requires only an input iterator.

7.0.2 Istream_iterator

An istream_iterator is derived from an input_iterator to work specifically with reading from streams. The template for istream_iterator is instantiated with a <*type*, *distance*>. This distance is usually specified by ptrdiff_t. As defined in *cstddef* or *stddef.h*, it is an integer type representing the difference between two pointer values.

Iterators for Streams Program

In file stl_io.cpp

```
//Use of istream_iterator and ostream_iterator

#include <iterator>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   vector<int> d(5);
   int  i, sum ;
   istream_iterator<int, ptrdiff_t> in(cin);
   ostream_iterator<int> out(cout, "\t");

   cout << "enter 5 numbers" << endl;
   sum = d[0] = *in;         //input first value
   for (i = 1; i < 5; ++i) {
      d[i] = *++in;          //input consecutive values
      sum += d[i];
  }
   for (i = 0; i < 5; ++i)
    *out = d[i] ;          //output consecutive values
   cout << " sum = " << sum << endl;
}
```

The istream_iterator in is instantiated with type int and parameter ptrdiff_t. The ptrdiff_t is a distance type that the iterator uses to advance in getting the next element. In the above declaration in is constructed with the input stream cin. The first element is read and cached. The autoincrement operator advances in and

reads and caches a next value of type `int` from the designated input stream. The `ostream_iterator out` is constructed with the output stream `cout` and the `char*` delimiter `"\t"`. Thus the tab character will be issued to the stream `cout` after each `int` value is written. In this program the iterator `out`, when it is dereferenced, writes the assigned `int` value to `cout`.

7.0.3 Ostream_iterator

An `ostream_iterator` is derived from an `output_iterator` to work specifically with writing to streams.

ostream_iterator Program

In file stl_oitr.cpp

```
//Use of as ostream_iterator iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  int d[5] = { 2, 3, 5, 7, 11 };              //primes
  ostream_iterator<int> out(cout, "\t");

  for (int i = 0; i < 5; ++i)
    *out = d[i] ;
}
```

The `ostream_iterator` can be constructed with a `char*` delimiter, in this case `"\t"`. Thus the tab character will be issued to the stream `cout` after each `int` value is written. In this program the iterator `out`, when it is dereferenced, writes the assigned `int` value to `cout`.

The output stream iterator is isomorphic to the input stream iterator. When a value is assigned to the iterator, it is written to the instantiated output stream, using operator >>. As seen in the above example the output stream iterator must specify as a parameter to the constructor, the associated output stream. An optional second parameter to the constructor is a string that will be used as a separator between values.

Simple file manipulations can be coded by using input and output stream iterators and various algorithms in the standard library. The following example reads a

file of integers, removes all occurrences of the value 0, and copies the remaining values separating each value with a comma:

istream_iterator Program

In file stl_ioitr.cpp

```
//Use of both istream_iterator and ostream_iterator iterator
#include <iostream>
#include <iterator>
using namespace std;

void main()
{
  istream_iterator<int, ptrdiff_t> input (cin), eof;
  ostream_iterator<int> output (cout, ",");

  //remove 0 from file redirected to cin
  //print file to cout
  remove_copy (input, eof, output, 0);

}
```

7.0.4 Iterator Adaptors

Iterators can be adapted to provide backward traversal and provide traversal with insertion.

STL Iterator Adaptors

- Reverse iterators—reverse the order of iteration
- Insert iterators—insertion takes place instead of the normal overwriting mode

In the following example we use a reverse iterator to traverse a sequence.

Iterator Adaptor Program

In file stl_iadp.cpp

```
//Use of the reverse iterator

#include <iostream>
#include <vector>
using namespace std;

template <class ForwIter>
void print(ForwIter first, ForwIter last, const char* title)
{
  cout << title << endl;
  while (first != last)
   cout << *first++ << '\t';
  cout << endl;
}

int main()
{
  int     data[3] = { 9, 10, 11};
  vector<int> d(data, data + 3);
  vector<int>::reverse_iterator p = d.rbegin();

  print(d.begin(), d.end(), "Original");
  print(p, d.rend(), "Reverse");
}
```

This program uses a reverse iterator to change the direction in which the `print()` function prints the elements of `vector d`.

We will briefly list adaptors and their purpose as found in this library.

- ```
  template<class BidiIter,
    class T, class Ref = T&,
    class Distance = ptrdiff_t>
  class reverse_bidirectional_iterator;
  ```

  This reverses the normal direction of iteration. Use `rbegin()` and `rend()` for range.
  ```

- ```
  template<class RandAccIter,
    class T, class Ref = T&,
    class Distance = ptrdiff_t>
  class reverse_iterator;
  ```

  This reverses the normal direction of iteration. Use `rbegin()` and `rend()` for range.

- ```
  template <class Can>
    class insert_iterator;
  template <class Can, class Iter>
  insert_iterator<Can>
    inserter(Can& c, Iter p);
  ```

  Insert iterator inserts instead of overwrites. The insertion into `c` is at position `p`.

- ```
  template <class Can>
    class front_insert_iterator;
  template <class Can>
  front_insert_iterator<Can>
    front_inserter(Can& c);
  ```

  Front insertion occurs at the front of the container and requires a `push_front()` member.

- ```
  template <class Can>
    class back_insert_iterator;
  template <class Can>
  back_insert_iterator<Can>
    back_inserter(Can& c);
  ```

  Back insertion occurs at the back of the container and requires a `push_back()` member.

Check your vendor's product for specific system-dependent implementations.

Dr. P's Prescriptions: STL: Iterators

- Use iterator parameters rather than container variables.
- Use the weakest iterator category compatible with the function.

Prescription Discussion

Iterator sequences are not tied to a particular type of container. Container types are a narrower style of representation than iterator ranges. Ergo using iterator sequences leaves algorithms more general and hence the more reusable.

Our modus operandi in generic programming is to make the program as general as possible without degrading efficiency. This leads to rule two, namely use the weakest iterator type compatible with an efficient implementation of a computation.

Chapter 8

STL: Algorithms

The STL algorithms library contains the following four categories.

STL Categories of Algorithms Library

- Sorting algorithms
- Nonmutating sequence algorithms
- Mutating sequence algorithms
- Numerical algorithms

These algorithms generally use iterators to access containers instantiated on a given type. The resulting code can be competitive in efficiency with special-purpose codes.

8.0.1 Sorting algorithms

Sorting algorithms include general sorting, merges, lexicographic comparison, permutation, binary search, and selected similar operations. These algorithms have versions that use either `operator<()` or a `Compare` object. They often require random access iterators.

The following program uses the quicksort function `sort()` from STL.

Sorting Algorithm Program

In file stl_sort.cpp

```
#include <iostream>
#include <algorithm>
using namespace std;

const int N = 5;
```

```
int main()
{
  int    d[N], i, *e = d + N;

  for (i = 0; i < N; ++i)
    d[i] = rand();
  sort(d, e);
  for (i = 0; i < N; ++i)
    cout << d[i] << '\t';
}
```

This is a straightforward use of the library `sort` algorithm operating on the built-in array `d[]`. Notice how ordinary pointer values can be used as iterators.

We present the library prototypes for sorting algorithms.

- ```
  template<class RandAcc>
  void sort(RandAcc b, RandAcc e);
  ```

  This is a quicksort algorithm over the elements in the range `b` to `e`. The iterator type `RandAcc` must be a random access iterator.

- ```
  template<class RandAcc>
  void stable_sort(RandAcc b, RandAcc e);
  ```

  This is a stable sorting algorithm over the elements in the range `b` to `e`. In a stable sort equal elements remain in their relative same position.

- ```
  template<class RandAcc>
  void partial_sort(RandAcc b, RandAcc m, RandAcc e);
  ```

  This is a partial sorting algorithm over the elements in the range `b` to `e`. The range `b` to `m` is filled with elements sorted up to position `m`.

- ```
  template<class InputIter, class RandAcc>
  void partial_sort_copy(InputIter b, InputIter e,
                         RandAcc result_b, RandAcc result_e);
  ```

  This is a partial sorting algorithm over the elements in the range `b` to `e`. Elements sorted are taken from the input iterator range and copied to the random access iterator range. The smaller of the two ranges is used.

- ```
  template<class RandAcc>
  void nth_element(RandAcc b, RandAcc nth, RandAcc e);
  ```

  The `nth` element is placed in sorted order, with the rest of the elements partitioned by it. For example, if the fifth position is chosen, the four smallest elements are placed to the left of it. The remaining elements are placed to the right of it and will be greater than it.

- ```
  template<class InputIter1, class InputIter2, class OutputIter>
  OutputIter merge(InputIter1 b1, InputIter1 e1, InputIter2 b2,
                   InputIter2 e2, OutputIter result_b);
  ```

  The elements in the range `b1` to `e1`, and `b2` to `e2` are merged to the starting position `result_b`.

- ```
  template<class BidiIter>
  void inplace_merge(BidiIter b, BidiIter m, BidiIter e);
  ```

  The elements in the range `b` to `m` and `m` to `e` are merged in place.

We will use a table to briefly list other algorithms and their purpose as found in this library.

STL Sort Related Library Functions	
`binary_search(b, e, t)`	true if `t` is found in `b` to `e`
`lower_bound(b, e, t)`	the first position for placing `t` while maintaining sorted order
`upper_bound(b, e, t)`	the last position for placing `t` while maintaining sorted order
`equal_range(b, e, t)`	returns an iterator pair for the range where `t` can be placed maintaining sorted order
`push_heap(b, e)`	places the location's `e` element into an already existing heap
`pop_heap(b, e)`	swaps the location's `e` element with the `b` location's element and reheaps
`sort_heap(b, e)`	performs a sort on the heap
`make_heap(b, e)`	creates a heap
`next_permutation(b, e)`	produces the next permutation
`prev_permutation(b, e)`	produces the previous permutation
  ```

STL Sort Related Library Functions	
`lexicographical_compare (b1, e1, b2, e2)`	returns `true` if sequence 1 is lexicographically less than sequence 2
`min(t1, t2)`	return the minimum of `t1` and `t2` that are call-by-reference arguments
`max(t1, t2)`	return the maximum
`min_element(b, e)`	return the position of the minimum
`max_element(b, e)`	return the position of the maximum
`includes(b1, e1, b2, e2)`	returns `true` if the second sequence is a subset of the first sequence
`set_union (b1, e1, b2, e2, r)`	returns the union as an output iterator `r`
`set_intersection (b1, e1, b2, e2, r)`	returns the set intersection as an output iterator `r`
`set_difference (b1, e1, b2, e2, r)`	returns the set difference as an output iterator `r`
`set_symmetric_difference (b1, e1, b2, e2, r)`	returns the set symmetric difference as an output iterator `r`

These algorithms have a form that uses a `Compare` object replacing `operator<()`, for example:

- ```
  template<class RandAcc, class Compare>
  void sort(RandAcc b, RandAcc e, Compare comp);
  ```

  This is a quicksort algorithm over the elements in the range `b` to `e` using `comp` for ordering.

8.0.2 Nonmutating Sequence Algorithms

Nonmutating algorithms do not modify the contents of the containers they work on. A typical operation is searching a container for a particular element and returning its position.

In the following program the nonmutating library function `find()` is used to locate the element `t`.
  ```

Nonmutating Sequence Program

In file stl_find.cpp

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;

int main()
{
  string words[5] = { "my", "hop", "mop", "hope", "cope"};
  string*   where;

  where = find(words, words + 5, "hop");
  cout << *++where << endl;                 //mop
  sort(words, words + 5);
  where = find(words, words + 5, "hop");
  cout << *++where << endl;                 //hope
}
```

This uses `find()` to look for the position of the word "hop." We print the word following "hop" before and after sorting the array `words[]`.

We present the library prototypes for nonmutating algorithms.

- ```
  template<class InputIter, class T>
  InputIter find(InputIter b, InputIter e, const T& t));
  ```

  This finds the position of `t` in the range `b` to `e`.

- ```
  template<class InputIter, class Predicate>
  InputIter find(InputIter b, InputIter e, Predicate p));
  ```

  This finds the position of the first element that makes the predicate true in the range `b` to `e`; otherwise the position `e` is returned.

- ```
  template<class InputIter, class Function>
  void for_each(InputIter b, InputIter e, Function f));
  ```

  This applies the function `f` to each value found in the range `b` to `e`.
  ```

We will use a table to briefly list other algorithms and their purpose as found in this library.

STL Nonmutating Sequence Library Functions	
`next_permutation(b, e)`	produces next permutation
`prev_permutation(b, e)`	produces previous permutation
`count(b, e, t, n)`	returns to `n` the count of elements equal to `t`
`count_if(b, e, p, n)`	returns to `n` the count of elements that make predicate `p` true
`adjacent_find(b, e)`	returns the first position of adjacent elements that are equal; otherwise returns `e`
`adjacent_find(b, e, binp)`	returns the first position of adjacent elements satisfying the binary predicate binp; otherwise returns `e`
`mismatch(b1, e1, b2)`	returns an iterator pair indicating the positions where elements do not match from the given sequences starting with `b1` and `b2`
`mismatch (b1, e1, b2, binp)`	as above, with a binary predicate `binp` used instead of equality
`equal(b1, e1, b2)`	returns `true` if the indicated sequences match; otherwise returns `false`
`equal(b1, e1, b2, binp)`	as above, with a binary predicate `binp` used instead of equality
`search(b1, e1, b2, e2)`	returns an iterator where the second sequence is contained in the first, if it is not `e1`
`search (b1, e1, b2, e2, binp)`	as above, with a binary predicate `binp` used instead of equality

8.0.3 Mutating Sequence Algorithms

Mutating algorithms can modify the contents of the containers they work on. A typical operation is reversing the contents of a container.

In the following program the mutating library functions `reverse()` and `copy()` are used.

Mutating Sequence Algorithm Program

In file stl_revr.cpp

```
//Use of mutating copy and reverse
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
  string first_names[5] = { "laura", "ira", "buzz", "debra",
                            "twinkle" };
  string last_names[5] = { "pohl", "pohl", "dolsberry",
                           "dolsberry", "star" };
  vector<string> names(first_names, first_names + 5);
  vector<string> names2(10);
  vector<string>::iterator p;

  copy(last_names, last_names + 5, names2.begin());
  copy(names.begin(), names.end(), names2.begin() + 5);
  reverse(names2.begin(), names2.end());
  for (p = names2.begin(); p != names2.end(); ++p)
    cout << *p <<'\t';
}
```

The first invocation of the mutating function `copy()` places `last_names` in the container `vector names2`. The second call to `copy()` copies in the `first_names`, which had been used in the construction of the `vector names`. The function `reverse()` reverses all the elements that are then printed out.

We present the library prototypes for mutating algorithms.

- ```
  template<class InputIter, class OutputIter>
  OutputIter copy(InputIter b1, InputIter e1, OutputIter b2);
  ```

  This is a copying algorithm over the elements `b1` to `e1`. The copy is placed starting at `b2`. The position returned is the end of the copy.
  ```

- ```
  template<class BidiIter1, class BidiIter2>
  BidiIter2 copy_backward(BidiIter1 b1, BidiIter1 e1,
                          BidiIter2 b2);
  ```

  This is a copying algorithm over the elements `b1` to `e1`. The copy is placed starting at `b2`. The copying runs backward from `e1` into `b2`, which are also going backward. The position returned is `b2 - (e1 - b1)`.

- ```
  template<class BidiIter>
  void reverse(BidiIter b, BidiIter e);
  ```

  This reverses in place the elements `b` to `e`.

- ```
  template<class BidiIter, class OutputIter>
  OutputIter reverse_copy(BidiIter b1, BidiIter e1,
                          OutputIter b2);
  ```

  This is a reverse copying algorithm over the elements `b1` to `e1`. The copy in reverse is placed starting at `b2`. The copying runs backward from `e1` into `b2`, which are also going backward. The position returned is `b2 + (e1 - b1)`.

- ```
  template<class ForwIter>
  ForwardIter unique(ForwIter b, ForwIter e);
  ```

  The adjacent elements in the range `b` to `e` are erased. The position returned is the end of the resulting range.

- ```
  template<class ForwIter, class BinaryPred>
  ForwardIter unique(ForwIter b, ForwIter e, BinaryPred bp);
  ```

  The adjacent elements in the range `b` to `e` with binary predicate `bp` satisfied are erased. The position returned is the end of the resulting range.

- ```
  template<class InputIter, class OutputIter>
  OutputIter unique_copy(InputIter b1, InputIter e1,
                         OutputIter b2);

  template<class InputIter, class OutputIter, class BinaryPred>
  OutputIter unique_copy(InputIter b1, InputIter e1,
                         OutputIter b2, BinaryPred bp);
  ```

  The results are copied to `b2` with the original range unchanged.

The remaining library functions are described in the following tables.

STL Mutating Sequence Library Functions	
swap(t1, t2)	swaps t1 and t2
iter_swap(b1, b2)	swaps pointed to locations
swap_range(b1, e1, b2)	swaps elements from b1 to e1 with those starting at b2; returns b2 + (e1 - b1)
transform(b1, e1, b2, op)	using the unary operator op transforms the sequence b1 to e1, placing it at b2; returns the end of the output location
transform(b1, e1, b2, b3, bop)	uses the binary operator bop on the two sequences starting with b1 and b2 to produce the sequence b3; returns the end of the output location
replace(b, e, t1, t2)	replaces in the range b to e the value t1 by t2
replace_if(b, e, p, t2)	replaces in the range b to e, the elements satisfying the predicate p by t2
replace_copy(b1, e1, b2, t1, t2)	copies and replaces into b2 the range b1 to e1 with the value t1 replacing t2
replace_copy_if(b1, e1, b2, p, t2)	copies and replace into b2 the range b1 to e1 with the elements satisfying the predicate p replacing t2
remove(b, e, t)	removes elements of value t
remove_if, remove_copy, remove_copy_if	similar to replace family except that values are removed
fill(b, e, t)	assigns t to the range b to e
fill_n(b, n, t)	assigns n ts starting at b
generate(b, e, gen)	assigns to the range b to e by calling generator gen
generate_n(b, n, gen)	assigns n values starting at b using gen

STL Mutating Sequence Library Functions	
`rotate(b, m, e)`	rotates leftward the elements of the range `b` to `e`; element in position *i* ends up in position (*i* + *n* - *m*) % *n*, where *n* is the size of the range, `m` is the midposition, and `b` is the first position
`rotate_copy(b1, m, e1, b2)`	as above, but copied to `b2` with the original unchanged
`random_shuffle(b, e)`	shuffles the elements
`random_shuffle(b, e, rand)`	shuffles the elements using the supplied random number generator `rand`
`partition(b, e, p)`	the range `b` to `e` is partitioned to have all elements satisfying predicate `p` placed before those that do not satisfy `p`
`stable_partition(b, e, p)`	as above, but preserving relative order

8.0.4 Numerical Algorithms

Numerical algorithms include sums, inner product, and adjacent difference.

In the following program the numerical function `accumulate()` performs a vector summation, and `inner_product()` performs a vector inner product.

Numerical Algorithm Program

In file stl_numr.cpp

```
//Vector accumulation and innerproduct
#include <iostream>
#include <numeric>
using namespace std;
```

```
int main()
{
  double v1[3] = { 1.0, 2.5, 4.6 },
         v2[3] = { 1.0, 2.0, -3.5 };
  double sum, inner_p;
  sum = accumulate(v1, v1 + 3, 0.0);
  inner_p = inner_product(v1, v1 + 3, v2, 0.0);
  cout << "sum = " << sum
       << ", product = " << inner_p << endl;
}
```

These functions behave as expected on numerical types where + and * are defined.

The library prototypes for numerical algorithms are as follows.

- `template<class InputIter, class T>`
  `T accumulate(InputIter b, InputIter e, T t);`

  This is a standard accumulation algorithm whose sum is initially `t`. The successive elements from the range `b` to `e` are added to this sum.

- `template<class InputIter, class T, class BinOp>`
  `T accumulate(InputIter b, InputIter e, T t, BinOp bop);`

  This is an accumulation algorithm whose sum is initially `t`. The successive elements from the range `b` to `e` are summed with `sum = bop(sum, element)`.

We will use a table to briefly list other algorithms and their purpose as found in this library.

STL Numerical Library Functions	
`inner_product(b1, e1, b2, t)`	returns the inner product from the two ranges starting with `b1` and `b2`; this product is initialized to `t`, which is usually zero
`inner_product(b1,e1,b2,t,bop1,bop2)`	returns a generalized inner product using `bop1` to sum and `bop2` to multiply
`partial_sum(b1, e1, b2)`	produces a sequence starting at `b2`, that is the partial sum of terms from the range `b1` to `e1`
`partial_sum(b1, e1, b2, bop)`	as above, using `bop` for summation
`adjacent_difference(b1, e1, b2)`	produces a sequence starting at `b2`, that is the adjacent difference of terms from the range `b1` to `e1`
`adjacent_difference(b1, e1, b2, bop)`	as above, using `bop` for difference

STL provides the basic computations for many more sophisticated algorithms. By using STL, programmers can easily implement them. We will use numerical integration as an example. The idea is to generate a series of points, using a *generator*. A generator is a class that defines the function by overloading `operator()`, the function call operator. The STL algorithm

```
generate(iterator b, iterator e, generator g)
```

is used to produce a `vector` of values in the range (0, 1) for the function. The *algorithm*, *numeric*, and *vector* libraries are all required.

Numerical Integration Program

In file stl_int1.cpp

```
//Simple integration routine for x*x over (0, 1)
//The function is represented in class gen

class gen {          //generator for function to be integrated
public:
   gen(double x_zero, double increment) : x(x_zero),
   incr(increment) { }
   double operator()() { x += incr; return x*x; }
private:
   double x, incr;
};

double integrate( gen g, int n)   //integrate on (0,1)
{
   vector<double> fx(n);

   generate(fx.begin(),fx.end(), g );
   return(accumulate(fx.begin(), fx.end(), 0.0) / n );
}

int main()
{
   const int n = 10000;

   gen g(0.0, 1.0/n);
   cout << "integration program x**2" << endl;
   cout << integrate(g, n) << endl;
}
```

We approximate the area under the curve by a sequence of rectangles whose height is the value of the function and whose width is the increment. An increment gives us two choices for a height. We could improve the numerical accuracy of integration by bounding the area between rectangles based on the smaller heights and one based on the larger heights.

Integration Function

In file stl_int2.cpp

```
double integrate( gen g, int n, double& diff)
{
   vector<double> fx(n), sm(n), lg(n);
   double s, l;
   generate(fx.begin(),fx.end(), g );
   for (int i = 0; i < n - 1; ++i)
      if (fx[i] > fx[i + 1]) {
         sm[i] = fx[i + 1]; lg[i] = fx[i];
      }
      else {
         sm[i] = fx[i]; lg[i] = fx[i + 1];
      }
   s = accumulate(sm.begin(), sm.end(), 0.0)/n ;
   l = accumulate(lg.begin(), lg.end(), 0.0)/n ;
   diff = l - s;
   return ( s + l ) / 2;
}
```

The preceding code produces a more reliable estimate, with an error estimate calculated in `diff`.

Dr. P's Prescriptions: STL: Algorithms

- Use the most efficient algorithm for a computation.
- Modify or adapt existing STL algorithms.

Prescription Discussion

The STL algorithms are expected to be efficient. The generalized sort is an efficient adaptation of quicksort and compares favorably in most cases to running `qsort()` as found in the C standard library.

As in the above example of numerical integration, STL routines can be readily employed and adapted to perform significant computations without resort to special codes. The use of various function adaptors as discussed in the next chapter will vastlyexpand the applicability of STL.

Chapter 9

STL: Function Objects

It is useful to have function objects to further leverage the STL library. For example, many of the previous numerical functions had a built-in meaning using + or *, but also had a form in which user-provided binary operators could be passed in as arguments. Defined function objects can be found in *function* or built. Function objects are classes that have `operator()` defined. These are inlined and are compiled to produce efficient object code.

Function Object Program

In file stl_fucn.cpp

```
//Using a function object minus<int>
#include <iostream>
#include <numeric>
using namespace std;

int main()
{
  double v1[3] = { 1.0, 2.5, 4.6 }, sum;

  sum = accumulate(v1, v1 + 3, 0.0, minus<int>());
  cout << "sum = " << sum << endl;     //sum = -7
}
```

Accumulation is done using integer minus for the binary operation over the array `v1[]`. Therefore the double values are truncated, with the result being -7.

There are three defined function object classes.

STL Defined Function Object Classes

- Arithmetic objects
- Comparison objects
- Logical objects

We will use tables to briefly list algorithms and their purpose as found in this library.

STL Arithmetic Objects	
`template <class T> struct plus<T>`	adds two operands of type `T`
`template <class T> struct minus<T>`	subtracts two operands of type `T`
`template <class T> struct times<T>`	multiplies two operands of type `T`
`template <class T> struct divides<T>`	divides two operands of type `T`
`template <class T> struct modulus<T>`	modulus for two operands of type `T`
`template <class T> struct negate<T>`	unary minus for one argument of type `T`

Arithmetic objects are often used in numerical algorithms, such as `accumulate()`.

STL Comparison Objects	
`template <class T> struct equal_to<T>`	equality of two operands of type `T`
`template <class T> struct not_equal_to<T>`	inequality of two operands of type `T`
`template <class T> struct greater<T>`	comparison by the greater (`>`) of two operands of type `T`
`template <class T> struct less<T>`	comparison by the less (`<`) of two operands of type `T`
`template <class T> struct greater_equal<T>`	comparison by the greater or equal (`>=`) of two operands of type `T`
`template <class T> struct less_equal<T>`	comparison by the lesser or equal (`<=`) of two operands of type `T`

The comparison objects are frequently used with sorting algorithms, such as `merge()`.

STL Logical Objects	
`template <class T> struct logical_and<T>`	performs logical and (&&) on two operands of type `T`
`template <class T> struct logical_or<T>`	performs logical or (||) on two operands of type `T`
`template <class T> struct logical_not<T>`	performs logical negation (!) on a single argument of type `T`

9.0.1 Function Adaptors

Function adaptors allow for the creation of function objects using adaption.

STL Function Adaptors

- Negators for negating predicate objects
- Binders for binding a function argument
- Adaptors for pointer to a function

In the following example we use a binder function `bind2nd` to transform an initial sequence of values to these values doubled.

Function Adaptor Program

In file stl_adap.cpp

```
//Use of the function adaptor bind2nd
#include <iostream>
#include <algorithm>
#include <functional>
#include <string>
using namespace std;
```

```
template <class ForwIter>
void print(ForwIter first, ForwIter last, const char* title)
{
  cout << title << endl;
  while (first != last)
    cout << *first++ << '\t';
  cout << endl;
}

int main()
{
  int data[3] = { 9, 10, 11 };

  print(data, data + 3, "Original values");
  transform(data, data + 3, data, bind2nd(times<int>(), 2));
  print(data, data + 3, "New values");
}
```

We will use a table to briefly list algorithms and their purpose as found in this library.

STL Function Adaptors	

`template<class Pred>`
`unary_negate<Pred>`
`not1(const Pred& p)`
	returns `!p` where `p` is a unary predicate

`template<class Pred>`
`binary_negate<Pred>`
`not2(const Pred& p)`
	returns `!p` where `p` is a binary predicate

`template<class Op, class T>`
`binder1st<Op>bind1st`
`(const Op& op,const T& t)`
	the binary `op` has a first argument bound to `t`; a function object is returned

`template<class Op, class T>`
`binder2nd<Op>bind2nd`
`(const Op& op,const T& t)`
	the binary `op` has a second argument bound to `t`; a function object is returned

`template<class Arg,class T>` `ptr_fun(T (*f)(Arg))`	constructs a `pointer_to_unary_function<Arg, T>`

`template<class Arg1,`
`class Arg2, class T>`
`ptr_fun(T (*f)(Arg1, Arg2))`
	constructs a `pointer_to_binary_function<Arg,T>`

9.1 Allocators

Allocator objects manage memory for containers. They allow implementations to be tailored to local system conditions while maintaining a portable interface for the container class. Allocator definitions include: `value_type`, `reference`, `size_type`, `pointer`, and `difference_type`.

We will use a table to briefly list allocator member functions and their purpose as found in this library.

STL Allocator Members	
`allocator();` `~allocator();`	constructor and destructor for allocators
`pointer address(reference r);`	returns the address of `r`
`pointer allocate(size_type n);`	allocates memory for `n` objects of `size_type` from free store
`void deallocate(pointer p);`	deallocates memory associated with `p`
`size_type max_size();`	returns the largest value for `difference_type`; in effect, the largest number of element allocatable to a container

Check your vendor's product for specific system-dependent implementations.

Dr. P's Prescriptions: Function Objects

- Understand function composition.

Prescription Discussion

In many cases a lack of understanding of the mathematical concept of function composition prevents a programmer from fully mastering the notion and techniques of adaptation. Many of these concepts are routinely used in functional languages or logic-based languages such as Lisp, ML, Scheme and Prolog. It can be useful to look at examples written in those languages to better understand how these ideas can apply to STL.

Chapter 10

String Library

C++ provides a string type by including the standard header file *string*. It is the instantiation of a template class `basic_string<T>` with `char`. The string type provides member functions and operators that perform string manipulations, such as concatenation, assignment, or replacement. An example of a program using the string type for simple string manipulation follows.

String Library Program

In file stringt.cpp

```
//String class to rewrite a sentence
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string sentence, words[10];
  int pos = 0, old_pos = 0, nwords, i = 0;

  sentence = "Eskimos have 23 ways to ";
  sentence += "describe snow";
```

```
    while (pos < sentence.size()) {
      pos = sentence.find(' ', old_pos);
      words[i++].assign(sentence, old_pos, pos - old_pos);
      cout << words[i - 1] << endl;           //print words
      old_pos = pos + 1;
    }
    nwords = i;
    sentence = "C++ programmers ";
    for (i = 1; i < nwords -1; ++i)
     sentence += words[i] + ' ';
    sentence += "windows";
    cout << sentence << endl;
  }
```

The `string` type is used to capture each word from an initial sentence where the words are separated by the space character. The position of the space characters is computed by the `find()` member function. Then the `assign()` member function is used to select a substring from `sentence`. Finally, a new `sentence` is constructed using the overloaded assignment, `operator+=()`, and `operator+()` functions to perform assignments and concatenations.

We will describe the representation for a string of characters. It is also usual to have the instantiation `basic_string<wchar_t>` for a wide string type `wstring`. Other instantiations are possible as well.

String Private Data Members	
`char* ptr`	for pointing at the initial character
`size_t len`	for the length of the string
`size_t res`	for the currently allocated size, or for an unallocated string its maximum size

This implementation provides an explicit variable to track the string length, thus string length can be looked up in constant time, which is efficient for many string computations.

10.1 Constructors

Strings have six public constructors, which makes it easy to declare and initialize strings from a wide range of parameters.

String Constructor Members	
`string()`	default, creates an empty string.
`string(const char* p)`	conversion constructor from a pointer to `char`
`string(InputIterator b, InputIterator e)`	constructor from the `InputIterator` range from `b` to `e`
`string(const string& str, size_t pos = 0, size_t n = npos)`	copy constructor; `npos` is usually `-1` and indicates no memory was allocated
`string(const char* p, size_t n)`	copy `n` characters where `p` is the base address
`string(size_t n, char c1)`	construct a string of `n` `c`s

These constructors make it quite easy to use the string type initialized from `char*` pointers, which was the traditional C method for working with strings. Also, many computations are readily handled as a vector of characters. This is also facilitated by the `string` interface.

10.2 Member Functions

Strings have some members that overload operators, as briefly described in the next table.

String Overloaded Operator Members	
`string& operator=(const string& s)`	assignment operator
`string& operator=(const char* p)`	assigns a `char*` to a string
`string& operator=(const char c)`	assigns a `char c` to a string
`string& operator+=(const string&s)`	appends string `s`
`string& operator+=(const char* p)`	appends a `char*` to a string
`string& operator+=(const char c)`	appends a `char c` to a string
`char operator[](size_t pos) const`	returns the character at `pos`
`char& operator[](size_t pos)`	returns the reference to the character at `pos`

There is an extensive set of public member functions that let you manipulate strings. In many cases these are overloaded to work with `string`, `char*`, and `char`. We will start by describing `append()`.

- `string& append(const string& s, size_t pos = 0, size_t n=npos);`

  Appends `n` characters starting at `pos` from `s` to the implicit string object.

```
//example s1 "I am " s2 "7 years old"
s1.append(s2);                  // s1 " I am 7 years old"
s2.append(s1,0,4);              //s2 "7 years old I am"
```

- ```
  string& append(const char* p, size_t n);
  string& append(const char* p);
  string& append(size_t n, char c);
  ```

  In each case a `string` object is constructed using the constructor of the same signature and appended to the implicit `string` object.

- `string& assign(const string& s, size_t pos = 0, size_t n=npos);`

  Assigns `n` characters starting at `pos` from `s` to the implicit string object.
  ```

```
//example s1 " I am " s2 "7 years old"
s1.assign(s2);                    // s1 "7 years old"
```

The following signatures with the expected semantics are also overloaded:

```
string& assign(const char* p, size_t n);
string& assign(const char* p);
string& assign(size_t n, char c);
string& assign(InputIterator b, InputIterator e);
```

- `string& insert(size_t pos1, const string& str, size_t pos2 = 0, size_t n = npos);`

  The `insert()` function is an overloaded set of definitions that insert a string of characters at a specified position. It inserts `n` characters taken from `str`, starting with `pos2`, into the implicit string at position `pos1`.

```
//example s1 " I am " s2 " 7 years old"
s1.insert(2,s2);                  // s1"I 7 years old am"
```

The following signatures with the expected semantics are also overloaded:

```
string& insert(size_t pos,const char* p, size_t n);
string& insert(size_t pos, const char* p);
sstring& insert(size_t pos, size_t n, char c);
iterator insert(iterator p, char c);
iterator insert(iterator p, size_t n, char c);
void insert(iterator p, InputIterator b, InputIterator e);
```

The inverse function is `remove()`.

- `string& remove(size_t pos = 0, size_t n = npos);`

  `n` characters are removed from the implicit string at position `pos`.

In the following table, we briefly describe further public string member functions.

String Members	
`string& replace(pos1, n1, str, pos2 = 0, n2 = npos)`	replaces at `pos1` for `n1` characters, the substring in `str` at `pos2` of `n2` characters

`string& replace(pos,n,p,n2);`
`string& replace(pos,n,p);`
`string& replace(pos,n,c);`
	replaces `n` characters at `pos`, using a `char* p` of `n2` characters, or a `char* p` until null, or a character `c`

`size_t length() const;`	returns the string length
`const char* c_str() const;`	converts string to traditional `char*` representation
`const char* data() const;`	returns base address of the string representation
`void resize(n, c);` `void resize(n);`	resizes the string to length `n`; the padding character `c` is used in the first function and the `eos()` character is used in the second
`void reserve(size_t res_arg);` `size_t reserve() const;`	allocates memory for string; returns the size of the allocation
`size_t copy(p, n, pos=0) const;`	the implicit string starting at `pos` is copied into the `char* p` for `n` characters
`string substr(pos=0, n=npos)const;`	a substring of `n` characters of the implicit string is returned

You can lexicographically compare two strings using a family of overloaded member functions `compare()`.

- `int compare(const string& str, size_t pos = 0, size_t n = npos) const;`

  Compares the implicit string starting at `pos` for `n` characters with `str`. Returns zero if the strings are equal; otherwise returns a positive or negative integer value indicating that the implicit string is greater or less than `str` lexicographically. The following signatures with the expected semantics are also overloaded:

  ```
  int compare(const char* p,size_t pos, size_t n) const;
  int compare(const char* p, size_t pos = 0) const;
  ```

Each signature specifies how the explicit string is constructed and then compared to the implicit string.

The final set of member functions perform a find operation. We will discuss one group and then summarize in a table the rest of this group of member functions.

- `size_t find(const string& str, size_t pos=0) const;`

  The string `str` is searched for in the implicit string starting at `pos`. If it is found the position it is found at is returned; otherwise `npos` is returned, indicating failure.

  The following signatures with the expected semantics are also overloaded:

```
size_t find(const char* p, size_t pos, size_t n)const;
size_t find(const char* p, size_t pos= 0) const;
size_t find(char c, size_t pos = 0) const;
```

  Each signature specifies how the explicit string is constructed and then searched for in the implicit string. Further functions for finding strings and characters are briefly described in the following table.

String Find Members	

`size_t rfind(str, pos=npos) const;`
`size_t rfind(p, pos, n) const;`
`size_t rfind(p, pos=npos) const;`
`size_t rfind(c, pos=npos) const;`
	like `find()`, but scans the string backward for a first match

`size_t find_first_of`
`(str, pos = 0) const;`
`size_t find_first_of`
`(p, pos, n) const;`
`size_t find_first_of`
`(p, pos=0) const;`
`size_t find_first_of`
`(c,pos = 0) const;`
	searches for the first character of any character in the specified pattern, either `str`, `char* p`, or `char c`

String Find Members	

`size_t find_last_of (str, pos = npos) const;`
`size_t find_last_of (p, pos, n) const;`
`size_t find_last_of (p, pos= npos) const;`
`size_t find_last_of (c,pos = npos) const;`
	searches backward for the first character of any character in the specified pattern, either `str`, `char* p`, or `char c`

`size_t find_first_not_of (str, pos = 0) const;`
`size_t find_first_not_of (p, pos, n) const;`
`size_t find_first_not_of (p, pos=0) const;`
`size_t find_first_not_of (c,pos = 0) const;`
	searches for the first character that does not match any character in the specified pattern, either `str`, `char* p`, or `char c`

`size_t find_last_not_of (str, pos = npos) const;`
`size_t find_last_not_of (p, pos, n) const;`
`size_t find_last_not_of (p, pos= npos) const;`
`size_t find_last_not_of (c,pos = npos) const;`
	searches backward for the first character that does not match any character in the specified pattern, either `str`, `char* p`, or `char c`

10.3 Global Operators

The string package contains operator overloadings that provide input/output, concatenation, and comparison operators. These are intuitively understandable and are briefly described in the following table.

String Overloaded Global Operators	
`ostream& operator<<(ostream& o, const string& s);`	output operator
`istream& operator>>(istream& in, string& s);`	input operator
`string operator+(const string& s1, const string& s2);`	concatenate `s1` and `s2`
`bool operator==(const string& s1, const string& s2);`	`true` if string `s1` and `s2` are lexicographically equal
`< <= > >= !=`	as expected

The comparison operators and the concatenation `operator+()` are also overloaded with the following four signatures:

```
bool operator==(const char* p, const string& s);
bool operator==(char c, const string& s);
bool operator==(const string& s, const char* p);
bool operator==(const string& s, char c);
```

In effect, a comparison or concatenation of any kind can occur between string and a second argument that is either a string, a character, or a character pointer.

Dr. P's Prescriptions: String Library

- Prefer the C++ standard library replacements to the C library.
  - type is distinct from both `signed char` and `unsigned char`. Functions may be overloaded based on the distinctions, and pointers to the three types are not compatible.

Chapter 11

References

- ABC 95
  Kelley, A., and Pohl, I., *A Book on C, Third Edition*. 1995. Reading, MA: Addison-Wesley.
- ARM 90
  Ellis, M., and Stroustrup, B., *The Annotated C++ Reference Manual*. 1990. Reading, MA: Addison-Wesley.
- C4C 94
  Pohl, I., *C++ for C Programmers, Second Edition*. 1994. Reading, MA: Addison-Wesley.
- C4P 95
  Pohl, I., *C++ for Pascal Programmers, Second Edition*. 1995. Reading, MA: Addison-Wesley.
- DE 94
  Stroustrup, B., *The Design and Evolution of C++*. 1994. Reading, MA: Addison-Wesley.
- DP 95
  Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995. Reading, MA: Addison-Wesley.
- EC 92
  Meyers, S., *Effective C++: 50 Specific Ways to Improve your Programs and Designs*. 1992. Reading, MA: Addison-Wesley.
- GRAY 91
  Stroustrup, B., *The C++ Programming Language, Second Edition*. 1991. Reading, MA: Addison-Wesley.
- IOS 93
  Teale, S., *C++ IO Streams Handbook*. 1993. Reading, MA: Addison-Wesley.
- K 97
  Knuth, Donald E., *The Art of Computer Programming: Volume 1 Fundamental Algorithms: 3rd edition*. 1997. Reading, MA: Addison-Wesley

- K 98
  Knuth, Donald E., *The Art of Computer Programming: Volume 3 Sorting and Searching: 2nd* edition. 1998. Reading, MA: Addison-Wesley
- KR 88
  Kernighan, B., and Ritchie, D., *The C Programming Language, Second Edition*. 1988. Englewood Cliffs, NJ: Prentice Hall.
- KP 74
  Kernighan, B., and Plauger, P., *The Elements of Programming Style*. 1974. New York, NY: McGraw-Hill.
- LIP 91
  Lippman, S., *The C++ Primer, Second Edition*. 1991. Reading, MA: Addison-Wesley.
- OOAD 94
  Booch, G., *Object-Oriented Analysis and Design, Second Edition*. 1995. Reading, MA: Addison-Wesley.
- OPUS 97
  Pohl, I., *Object-Oriented Programming Using C++, Second Edition*. 1997. Reading, MA: Addison-Wesley.OPUS 97
- P 72
  Pohl, I., 1972. "A Sorting Problem and Its Complexity," *CACM*, v. 15, no. 6, June 1972, pp. 462-464.
- P 97
  Pohl, I., *C++ Distilled*. 1997. Reading, MA: Addison-Wesley.
- P 99
  Pohl, I., *C++ for C Programmers*. 1999. Reading, MA: Addison-Wesley.
- S 97
  Stroustrup, B., The C++ Programming Language, 3rd Edition. 1997. Reading, MA: Addison-Wesley
- STL 96
  Musser, D. and Saini, A., *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. 1996. Reading, MA: Addison-Wesley.
- STLP 96
  Glass, G. and Schuchert, B., *The STL <Primer>*. 1996. Upper Saddle River, NJ: Prentice Hall.
- TG 94
  Taligent Inc., *Taligent's Guide to Designing Programs: Well-Mannered Object-Orietned Design in C++*. 1994. Reading, MA: Addison-Wesley.

Chapter 12

Supplemental Programs

We will use this section to give added examples of programs that use ideas in this book. To the extent that this section and the eMatter book prompt further reader queries material will be added to this and other sections.

12.1 Copy Using Conversion Compatible Types

In this first example, we use template variables. The copy routine will work for conversion compatible types.

Generic Copy

In file copy2.cpp

```
/*Filename:  copy2.cpp
  Supplement: Generic Programming and STL
  Compiler:  Borland C++     Version 5.01
  Copyright By Ira Pohl
*/
#include <iostream>
#include <assert>
//using namespace std;
template<class T1, class T2>
void copy(T1 a[], T2 b[], int n)
{
   for (int i = 0; i < n; ++i)
      a[i] = b[i];
}
```

```
template<class TYPE>
void print(TYPE a[], int n)
{
   cout << "\nNEW PRINT =";
   for (int i = 0; i < n; ++i)
      cout << a[i] << " ";
}

int main()
{
   double  f1[50], f2[50];
   char    c1[25], c2[50];
   int     i1[75], i2[75];
   char*   ptr1, *ptr2;
   int i;


   for (i = 0; i < 50; ++i) {
      f1[i] = 1.1 + i;
      f2[i] = 2.2 * i;
      c2[i] = 'A' + i/5;
   }

   for (i = 0; i < 25; ++i)
      c1[i] = 'a' + i/8;

   for (i = 0; i < 75; ++i) {
      i1[i] = 2 * i;
      i2[i] = i * i;
   }

   print(f1, 20);    //print initial values
   print(f2, 20);
   print(i1, 20);
   print(i2, 20);
   print(c1, 20);
   print(c2, 20);
```

```
    copy(f1, f2, 50);
    copy(c1, c2, 10);
    copy(i1, i2, 40);
    copy(ptr1, ptr2, 100);
  //  copy(i1, f2, 50);      //no match on compile
  //  copy(ptr1, f2, 50);    //no match on compile

    print(f1, 20);    //print initial values
    print(f2, 20);
    print(i1, 20);
    print(i2, 20);
    print(c1, 20);
    print(c2, 20);
}
```

12.2 Generic Stack

The next example program implements a simple version of a generic stack. This is a non-STL implementation showing the use of template to develop a simple container. The following code uses this type.

Generic Stack Program

In file stack_t1.cpp

```
/*Filename:  stack_t1.cpp
  Supplement: Generic Programming and STL
  Compiler:  Borland C++     Version 5.01
  Copyright By Ira Pohl
*/

#include <iostream>
#include <assert>
```

```
//template stack implementation
template <class TYPE>
class stack {
public:
   explicit stack(int size = 100)
      : max_len(size), top(EMPTY),s(new TYPE[size])
       { assert(s != 0); }
   ~stack() { delete []s; }
   void  reset() { top = EMPTY; }
   void  push(TYPE c) { s[++top] = c; }
   TYPE  pop() { return s[top--]; }
   TYPE  top_of()const { return s[top]; }
   bool  empty()const { return top == EMPTY;}
   bool  full()const { return top == max_len - 1;}
private:
   enum   { EMPTY = -1 };
   TYPE*  s;
   int    max_len;
   int    top;
};

//Reversing an array of char* represented strings
void reverse(char* str[], int n)
{
   stack<char*>  stk(n);
   int i;

   for (i = 0; i < n; ++i)
      stk.push(str[i]);
   for (i = 0; i < n; ++i)
      str[i] = stk.pop();
}

template <class T1>
void print_and_pop(T1& a, char* comment)
{
   cout << “Printing “ << comment << endl;
   while (!a.empty())
      cout << a.pop() << ‘\t’;
   cout << endl;
}
```

```
//Initializing stack of complex numbers from an array
int main()
{
   stack<char>     stk_ch;         // 1000 char stack
   stack<char*>    stk_str(200);   // 200 char* stack
   char* str[3] = {“Reverse”, “these”, “three”};

   stk_ch.push(‘A’);
   stk_ch.push(‘B’);
   print_and_pop(stk_ch, “char”);
   stk_str.push(“ABCD”);
   stk_str.push(“EFGH”);
   print_and_pop(stk_str, “char*”);
   reverse(str, 2);
   cout << “Reversed 2” << endl << str[0] << str[1]
        << str[2] << endl;
}
```

12.3 Reverse Iterator

The following program uses a reverse iterator. This lets you move backward through the container.

```
#include <iostream>
#include <vector>
using namespace std;

//Use of the reverse iterator
template <class ForwIter>
void print(ForwIter first, ForwIter last, const char* title)
{
   cout << title << endl;
   while ( first != last)
      cout << *first++ << '\t';
   cout << endl;
}
```

```
int main()
{
   int     data[3] = { 9, 10, 11};
   vector<int> d(data, data + 3);
   vector<int>::reverse_iterator p = d.rbegin();

   print(d.begin(), d.end(), "Original");
   print(p, d.rend(), "Reverse");
}
```

Index