VI. Templates

A. Introduction.

The evolution of reusability/genericity — Read pp. 244-6

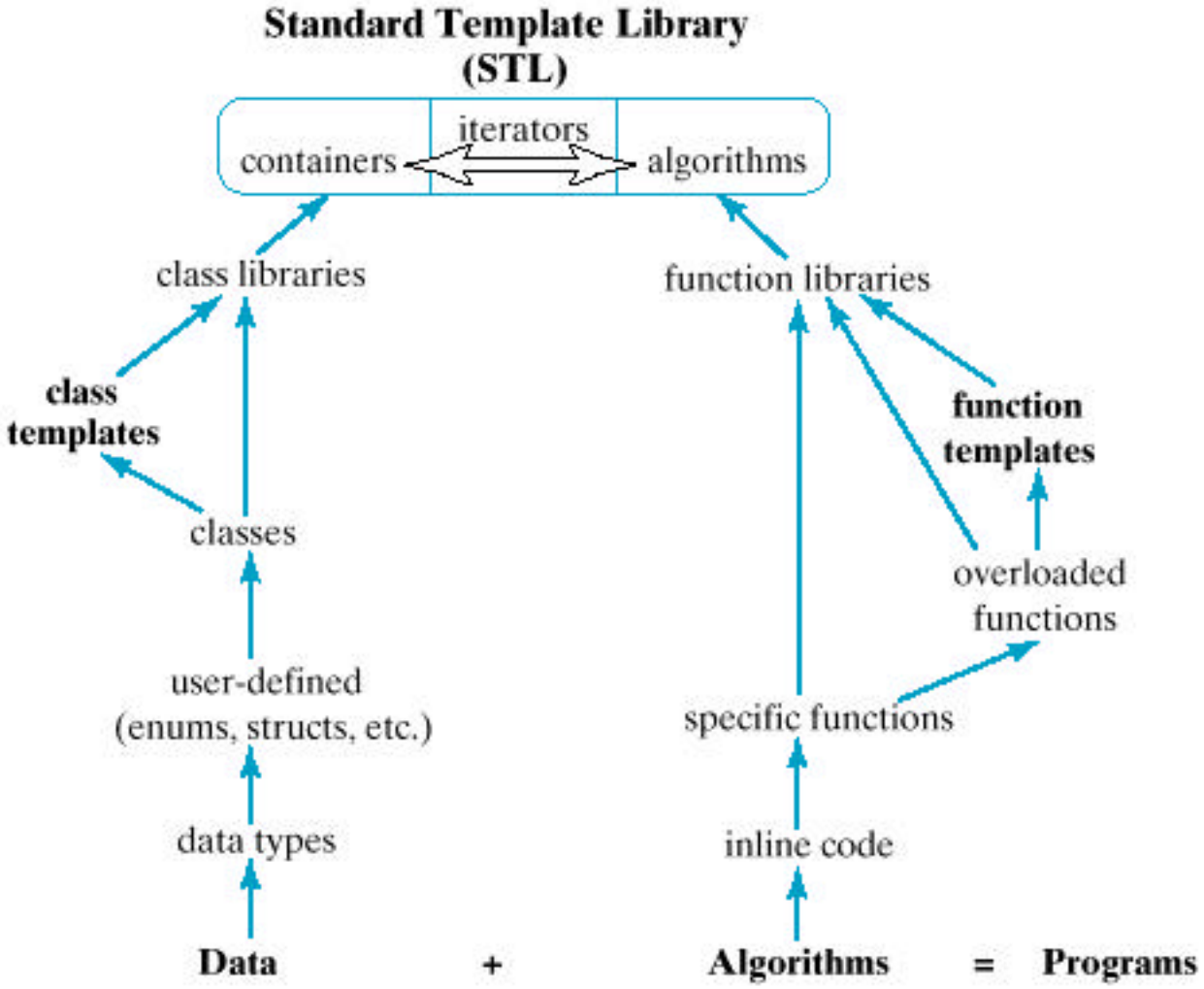


FIGURE 6.1 THE EVOLUTION OF REUSEABILITY/GENERICITY

Templates allow functions and classes to be parameterized, so that the type of data being stored (or operated upon) is received via a parameter. Templates thus provide a means of writing code that is easier to reuse since one template definition can be used to create multiple instances of a class (or function), each storing (operating on) a different type of data.

The template mechanism is important and powerful. It is used throughout the Standard Template Library (STL) to achieve genericity.

B. Function Templates

Main reason for using functions: Make pieces of code reusable by encapsulating them within functions.

1. Example: Interchange problem: To interchange the values of two `int` variables `x` and `y`.

Instead of inline code:

```
int temp = x;
x = y;
y = temp;
```

write a function:

```
/* Function to interchange two integer variables.
   Receive:   Integer variables first and second
   Pass back: first and second with values interchanged.
-----------------------------------------------------------*/
void Swap(int & first, int & second)
{
  int temp = first;
  first = second;
  second = temp;
}
```

→ This gives a general solution to the interchange problem, because this function can be used to exchange the values of any two integer variables:

```
Swap(x, y);
  ...
Swap(w, z);
  ...
Swap(a, b);
```

a. To interchange the values of two `double` variables:

We can't use the preceding function. However, **overloading** allows us to provide multiple definitions for the same function:

```
/* Function to interchange two double variables.
   ...
void Swap(double & first, double & second)
{
  double temp = first;
  first = second;
  second = temp;
}
```

b. To interchange the values of two `string` variables

Again, we can overload function `Swap()`:

```
/* Function to interchange two string variables.
        . . .
void Swap(string & first, string & second)
{
  string temp = first;
  first = second;
  second = temp;
}
```

c. And so on ... for other types of variables

Make a `Swap` library? OK for C++ predefined types, but can't use for user-defined types such as `Time`. We would have to overload `Swap()` for each user-defined type:

```
/* Function to interchange two Time variables.
void Swap(Time & first, Time & second)
{
  Time temp = first;     // assumes that = is
  first = second;        // overloaded for Time
  second = temp;
}
```

d. Observations:

— The logic in each function is exactly the same; the only difference is in the type of the values being exchanged.

— If we could pass the type as an argument, we could write a general solution that could be used to exchange the values of any two variables.

3. The Template Mechanism

It works by declaring a **type parameter** and using it in the function instead of a specific type. This is done using a different kind of parameter list.

A Swap Template

```
* A Swap template for exchanging the values of any two
  objects of the same type, for which the assignment
  operation is defined.
  Receive:   Type parameter Item
             first and second, two objects of the same type.
  Pass back: first and second with values interchanged.
  Assumes:   Assignment (=) is defined for type Item.
-----------------------------------------------------------*/
template <typename Item>      // <- the type parameter
void Swap(Item & first, Item & second)
{
  Item temp;
  temp = first;
  first = second;
  second = temp;
}
```

Notes:

—The word `template` is a C++ keyword specifying that what follows is a pattern for a function, *not a* function definition.

— Whereas “normal” parameters (and arguments) appear within parentheses, type parameters (and arguments for class templates) appear within angle brackets (<> pairs).

— Originally, the keyword `class` was used instead of `typename` in a type-parameter list — "class" as a synonym for "kind" or "category" and specifies the "type/kind" of types.

— Unlike other functions, a template function cannot be split across files, that is, we can't put its prototype in a header file and its definition in an implementation file. It all goes in the header file.

4. How is a Function Template Used?

`<typename Item>` names `Item` as a type parameter — a parameter whose value will be determined (by the compiler) from the type of the arguments when `Swap()` is called.

Example:

```
#include "Swap.h"
  . . .
int i1, i2;
  . . .
Swap(i1, i2);
  . . .
double dub1, dub2;
  . . .
Swap(dub1, dub2);
  . . .
string str1, str2;
  . . .
Swap(str1, str2);
  . . .
Time t1, t2;
  . . .
Swap(t1, t2);
```

Using the `Swap()` template, **the compiler will generate definitions of `Swap()` in which the parameter `Item` is replaced by `int`, `double`, `string`, and `Time`.**

This single function template definition (stored in a header file `Swap.h`) is sufficient to interchange the values of any two variables, provided the assignment operator is defined for their type.

(Simplified) general form:

```
template <typename TypeParam>
FunctionDefinition
```

or

```
template <class TypeParam>
FunctionDefinition
```

where *`TypeParam`* is a type-parameter naming the "generic" type of value(s) on which the function operates, and *`FunctionDefinition`* is the definition of the function, using type *`TypeParam`*.

5. A function template is only a pattern that describes how individual functions can be built from given actual types. This process of constructing a function is called **instantiation**.

   We instantiated `Swap()` four times — once with type `int`, once with type `double`, once with type `string`, and once with type `Time`. In each instantiation, the type parameter is said to be **bound** to the actual type passed to it.

   A template thus serves as a pattern for the definition of an unlimited number of instances. In and of itself, the template does nothing. For example, when the compiler encounters a template like that for `Swap()`, it simply stores it but doesn't generate any machine instructions. Later, when it encounters a call to `Swap()` like

```
Swap(int1, int2);
```

   it generates an integer instance of `Swap()`:

```
void Swap(int & first, int & second)
{
  int temp = first;
  first = second;
  second = temp;
}
```

   For this to be possible, the compiler must "see" the actual definition of `Swap()`, and not just its prototype. This is why:

   - A function template cannot be split across two files (prototype in a header file and definition in an implementation file.)

   Algorithm for instantiation:

   (1) *Search parameter list of template function for type parameters*

   (2) *If one is found, determine type of corresponding argument*

   (3) *Bind these types*

Example:

```
/* Function template to find the largest value of any type
   (for which < is defined) stored in an array.
   Receive: Type parameter ElementType
            array of elements of type ElementType
            numElements, number of values in array
   Return:  Largest value in array
-----------------------------------------------------------*/

template <typename ElementType>
ElementType Largest(ElementType array[], int numElements)
{
  ElementType biggest = array[0];
  for (int i = 1; i < numElements; i++)
    if (array[i] > biggest)
      biggest = array[i];
  return biggest;
}

int main ()
{
  double x[10] = {1.1, 4.4, 3.3, 5.5, 2.2};
  cout << "Largest value in x is " << Largest(x, 5);
  int num[20] = {2, 3, 4, 1};
  cout << "Largest value in num is " << Largest(num, 4);
}
```

Execution:

```
Largest value in x is 5.5
Largest value in num is 4
```

When compiler encounters `Largest(x, 5)`, it:

1. Looks for a type parameter — finds `ElementType`
2. Finds type of corresponding argument (`x`) — `double`
3. Binds these types and generates an instance of `Largest()`:

```
double Largest(double array[], int numElements)
{
  double biggest = array[0];
  for (int i = 1; i < numElements; i++)
    if (array[i] > biggest)
      biggest = array[i];
  return biggest;
}
```

Similarly, it generates an `int` version when `Largest(num, 4)` is encountered.

C. Class Templates

1. What's wrong with `typedef`?

Consider our `Stack` (and `Queue`) class:

```
/* Stack.h contains the declaration of class Stack.
   Basic operations:
       . . .
-------------------------------------------------------*/

const int STACK_CAPACITY = 128;

typedef int StackElement;

class Stack
{
/***** Function Members *****/
public:
   . . .

/***** Data Members *****/
private:
  StackElement myArray[STACK_CAPACITY ];
  int myTop;
};
```

We can change the meaning of `StackElement` throughout the class by changing the type following the `typedef`.

Problems:

- This changes the header file, so any program/library that uses the class must be recompiled.
- (More serious): A name declared using `typedef` can have only one meaning at a time. If we needed two stacks with different elements types, e.g., a `Stack` of ints and a `Stack` of strings, we would need to create two different stack classes with different names.

2. Creating a container class that is truly *type-independent*

Use a **class template**, in which the class is **parameterized** so that it **receives the type of data stored in the class via a parameter** much like function templates :

```
/* StackT.h contains a template for class Stack
   Receives:  Type parameter StackElement
   Basic operations:
      . . .
-------------------------------------------------------*/
...
const int STACK_CAPACITY = 128;

template <typename StackElement>
class Stack
{
{
/***** Function Members *****/
public:
   . . .
/***** Data Members *****/
private:
  StackElement myArray[STACK_CAPACITY ];
  int myTop;
};
```

Here, the type parameter `StackElement` can be thought of as a "blank" type that will be filled in later.

In general:

`template <typename TypeParam >` or **`template <class TypeParam>`**

```
class SomeClass
{
 // ... members of SomeClass ...
}
```

More than one type parameter may be specified:

`template <typename TypeParam`$_1$**`,..., typename TypeParam`**$_n$**`>`**

```
class SomeClass
{  . . .  }
```

3. Instantiating a class

To use a class template in a program/function/library:
Instantiate it by using a declaration of the form

ClassName<Type> object

to pass *Type* as an argument to the class template definition.

Examples:

```
Stack<int> intSt;
Stack<string> stringSt;
```

Compiler will generate two distinct definitions of `Stack` — two **instances** — one for `int`s and one for `string`s.

4. Rules Governing Templates.

1. All definitions of member function outside the class declaration must be template functions.
2. All uses of class name as a type must be parameterized.
3. Member functions must be defined in the same file as the class declaration.

a. Rules don't apply to prototypes of member functions, so no change to them.

```
/* StackT.h provides a Stack template.
 *
 * Receives:  Type parameter StackElement
 * Basic operations:
 *   Constructor:  Constructs an empty stack
 *   empty:   Checks if a stack is empty
 *   push:    Modifies a stack by adding a value at the top
 *   top:     Accesses the top stack value; leaves stack unchanged
 *   pop:     Modifies a stack by removing the value at the top
 *   display: Displays all the stack elements
 * Class Invariant:
 *   1. The stack elements (if any) are stored in positions
 *      0, 1, . . ., myTop of myArray.
 *   2. -1 <= myTop <= STACK_CAPACITY
 ----------------------------------------------------------------*/

#include <iostream>
using namespace std;

#ifndef STACKT
#define STACKT
```

```
const int STACK_CAPACITY = 128;

template <typename StackElement>
class Stack
{

/***** Function Members *****/
public:

  // --- Constructor ---
  Stack();

  // --- Is the Stack empty? ---
  bool empty() const;

  // --- Add a value to the stack ---
  void push(const StackElement & value);

  // --- Display values stored in the stack ---
  void display(ostream & out) const;

  // --- Return value at top of the stack ---
  StackElement top();

  // --- Remove and return value at top of the stack ---
  void pop();
};
...
#endif
```

b. Definitions of member functions operations.

Rule 1: They must be defined as **function templates**:

```
template <typename StackElement>
// ... definition of Stack()

template <typename StackElement>
// ... definition of empty()

template <typename StackElement>
// ... definition of push()

template <typename StackElement>
// ... definition of display()

template <typename StackElement>
// ... definition of top()

template <typename StackElement>
// ... definition of pop()
```

Rule 2: The class name `Stack` preceding the scope operator (`::`) is used **as the name of a type** and must therefore be **parameterized**:

```
template <typename StackElement>
inline Stack<StackElement>::Stack(const StackElement& value)
{
// ... body of Stack()
}

template <typename StackElement>
inline bool Stack<StackElement>::empty(const StackElement& value)
{
// ... body of push()
}

template <typename StackElement>
void Stack<StackElement>::push(const StackElement& value)
{
// ... body of push()
}

template <typename StackElement>
void Stack<StackElement>::display()
{
// ... body of display()
}

template <typename StackElement>
StackElement Stack<StackElement>::top()
{
// ... body of top()
}

template <typename StackElement>
void Stack<StackElement>::pop();
{
// ... body of pop()
}
```

Rule 3: These definitions must be placed **within `StackT.h`**:

```
/* StackT.h provides a Stack template.
   . . .
-----------------------------------------------------*/
#ifndef STACKT
#define STACKT
...
template <typename StackElement>
class Stack
{
  ...
}; // end of class declaration

/***** Function Templates for Operations *****/

//--- Definition of Constructor
template <typename StackElement>
inline Stack<StackElement>::Stack()
{ myTop = -1; }
  ...
#endif
```

c. Friend functions are also governed by the three rules.

For example, suppose we use `operator<<` instead of `display()` for output:

→ Prototype it within the class declaration as a friend:

```
/* StackT.h provides a Stack template.
 ...
-----------------------------------*/
...
const int STACK_CAPACITY = 128;

template <typename StackElement>
class Stack
{
public:
  //--- Output operator -- documentation omitted here
  friend ostream & operator<<(ostream & out,
                              const Stack<StackElement> & st);
...
}; // end of class
```

→ And define it outside the class as a function template:

```
// --- ostream output ----------------------------

template<class StackElement>
ostream & operator<<(ostream & out,
                     const Stack<StackElement> & st)
{
  for (int pos = st.myTop; pos >= 0; pos--)
    out << st.myArray[pos] << endl;
  return out;
}
```

→ Since `Stack` is being used as a type to declare the type of `st`, it must be parameterized.

5. Program to Test the `Stack` Template.

```
#include <iostream>
using namespace std/
#include "StackT.h"

int main()
{
  Stack<int> intSt;       // stack of ints
  Stack<char> charSt;     // stack of chars

  for (int i = 1; i <= 4; i++)
    intSt.push(i);

  while (!intSt.empty())
    {
      i = intSt.top(); intSt.pop();
      cout << i << endl;
    }

  for (char ch = 'A'; ch <= 'D'; ch++)
    charSt.push(ch);

  while (!charSt.empty())
    {
      ch = charSt.top(); charSt.pop();
      cout << ch << endl;
    }
```

Sample run:

```
4
3
2
1
D
C
B
A
```

Sample run:

```
4
3
2
1
D
C
B
A
```

6. An Alternative Version.

*** Templates may have more than one type parameter; they may also have **ordinary parameters**.

```
/* StackT.h provides a Stack template.
    Receives:  Type parameter StackElement
               Integer myCapacity
 . . .
-----------------------------------------------------------*/

#ifndef STACKT
#define STACKT

template <typename StackElement, int myCapacity>

class Stack
{
public:
//... Prototypes of member (and friend) functions ...

private:
  StackElement myArray[myCapacity];
  int myTop;

};

//... Definitions of member (and friend) functions ...

#endif
```

Program to Test the Stack Template.

```
#include <iostream>
using namespace std;
#include "StackT.h"

int main()
{
  Stack<int, 10> intSt;
  Stack<char, 3> charSt;

 // ... same as earlier ...
}
```

Sample run:

```
4
3
2
1
*** Stack is full -- can't add new value ***
Declare a larger one.
C
B
A
```

D. More About Function Templates

Like class templates, more than one type parameter is allowed

template <typename *TypeParam*$_1$, typename *TypeParam*$_2$, ...>
FunctionDefinition

Each of the type parameters *must* appear at least once in the parameter list of the function. Why? Because the compiler must be able to determine the actual type that corresponds to each type parameter from a call to that function.

a. Example:

```
/* Function template to convert a value of any type to
   another type
   Receive:     Type parameters Type1 and Type2
                value1 of Type 1
   Pass back:   value2 of Type2
-----------------------------------------------------------*/

template <typename Type1, typename Type2>
void Convert(Type1 value1, Type2 & value2)
{
  value2 = static_cast<Type2>(value1);
}

#include <iostream>
using namespace std;

int main()
{
  char a = 'a';
  int ia;
  Convert(a, ia);
  cout << a << "  " << ia << endl;

  double x = 3.14;
  int ix;
  Convert(x, ix);
  cout << x << "  " << ix << endl;
}
```

Sample run:

```
a  97
3.14  3
```

b. The following version of function template `Convert` would not be allowed:

```
template <typename Type1, typename Type2>
Type2 Convert(Type1 value1)  // Error--Type2 not used in
{                            // parameter list
  return static_cast<Type2>(value1);
}
```

c. One possible solution would be to provide a dummy second parameter indicating the type of the return value:

```
template <typename Type1, typename Type2>
Type2 Convert(Type1 value1, Type2 value2 = Type2(0))
{
  return static_cast<Type2>(value1);
}
```

Function call:

```
double x = 3.14;
int ix = Convert(x, 0);
```

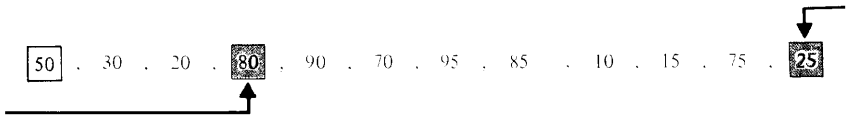
A (Substantial) Application of Template Functions: QuickSort (§11.3)

The **quicksort** method of sorting is one of the fastest methods of sorting and is most often implemented by a recursive algorithm. The basic idea of quicksort is to choose some element called a **pivot** and then perform a sequence of exchanges so that all elements that are less than this pivot are to its left and all elements that are greater than the pivot are to its right. This correctly positions the pivot and divides the (sub)list into two smaller sublists, each of which may then be sorted independently in the *same* way. This **divide-and-conquer** strategy leads naturally to a recursive sorting algorithm.

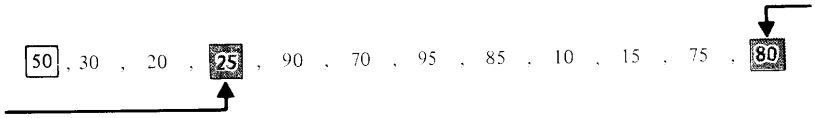
To illustrate this splitting of a list into two sublists, consider the following list of integers:

50, 30, 20, 80, 90, 70, 95, 85, 10, 15, 75, 25

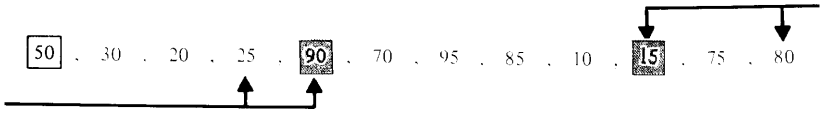
If we select the first number as the pivot, we must rearrange the list so that 30, 20, 10, 15, and 25 are placed before 50, and 80, 90, 70, 95, 85, and 75 are placed after it. To carry out this rearrangement, we search from the right end of the list for an element less than 50 and from the left end for an item greater than 50.



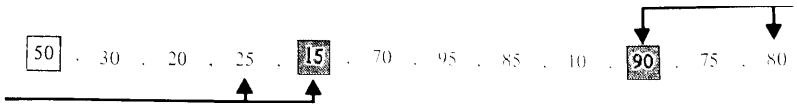
This locates the two numbers 25 and 80, which we now interchange to obtain



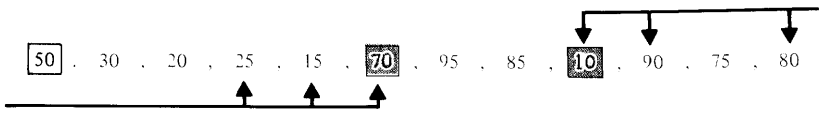
We then resume the search from the right for a number less than 50 and from the left for a number greater than 50:



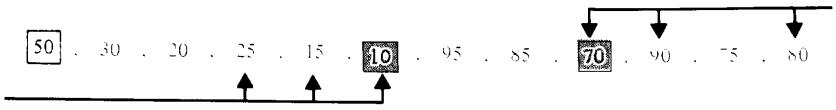
This locates the numbers 15 and 90, which are then interchanged:



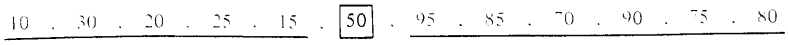
A continuation of the searches locates 10 and 70:



Interchanging these gives



When we resume our search from the right for a number less than 50, we locate the value 10, which was found on the previous left-to-right search. This signals the end of the two searches, and we interchange 50 and 10, giving



The two underlined sublists now have the required properties: All elements in the first sublist are less than 50, and all those in the right sublist are greater than 50. Consequently, 50 has been properly positioned.

Both the left sublist

10, 30, 20, 25, 15

and the right sublist

95, 85, 70, 90, 75, 80

can now be sorted independently. Each must be split by choosing and correctly positioning one pivot element (the first) in each of them.

A function is needed to split a list of items in the array positions given by two parameters `low` and `high`, denoting the beginning and end positions of the sublist, respectively. The following function template carries out the desired splitting.

```
// Need Swap() and operator<<() templates

// Split rearranges x[first], ... , x[last] so that
// the pivot element is properly positioned at
// position pos.
//   Receive:   Type parameter ElementType
//              vector x, indices first, last
//   Pass back: Rearranged x, index pos
//-----------------------------------------------------

template <class ElementType>

void Split(vector<ElementType> & x,
           int first, int last, int & pos)
```

```
{
  ElementType pivot = x[first]; // pivot element
  int left = first,             // index for left search
      right = last;             // index for right search

  while (left < right)
  {
    // Search from right for element <= pivot
    while (x[right] > pivot)
      right--;
    // Search from left for element > pivot
    while (left < right && x[left] <= pivot)
      left++;
    // Interchange elements if searches haven't met
    if (left < right)
      Swap(x[left], x[right]);
  }
  // End of searches; place pivot in correct position
  pos = right;
  x[first] = x[pos];
  x[pos] = pivot;
}
```

Given this function, a recursive function to sort a list is now easy to write.

- The trivial case occurs when the list being examined is empty or contains a single element, in which case the list is in order, and nothing needs to be done.

- The nontrivial case occurs when the list contains multiple elements, in which case the list can be sorted by:
  a. Splitting the list into two sublists;
  b. Recursively sorting the left sublist; and
  c. Recursively sorting the right sublist.

This algorithm is encoded as the following function template `QuickSort()`:

```
// QUICKSORT
//   Receive:   Type parameter ElementType
//              vector x with elements of type ElementType
//              indices first and last
//   Pass back: Rearranged x with x[first], ..., x[last]
//                in ascending order
//-----------------------------------------------------

template <class ElementType>

void Quicksort(vector<ElementType> & x, int first, int last)
{
  int pos;          // final position of pivot
  if (first < last) // list has more than one element
  {
    // Split into two sublists
    Split(x, first, last, pos);
    // Sort left sublist
    Quicksort(x, first, pos - 1);
    // Sort right sublist
    Quicksort(x, pos + 1, last);
  }
  // else list has 0 or 1 element and
  // requires no sorting
}


// Function template interface to QuickSort()
//   Receive:   Type parameter ElementType
//              vector x with elements of type ElementType
//   Pass back: x sorted in ascending order.
//-----------------------------------------------------------

template <typename ElementType>

void QSort(vector<ElementType> & x)
{
  Quicksort(x, 1, x.size() - 1);
}
```

Driver Program:

```
#include <iostream>
using namespace std;
#include "SortLibrary"

int main()
{
  int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
  vector<int> intvec(ints, ints + 8);
  QSort(intvec);
  cout << "Sorted list of integers:\n" << intvec << endl;

  double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
  vector<double> dubvec(dubs, dubs + 7);
  QSort(dubvec);
  cout << "Sorted list of doubles:\n" << dubvec, endl;
}
```

Execution:

```
Sorted list of integers:
1  22  33  66  222  444  555  777

Sorted list of doubles:
0.1  2.2  3.3  22.2  44.4  55.5  77.7
```

E. STL's Containers (pp. 265 - 267)

STL (the Standard Template Library) is a library of class and function templates based on work in generic programming done by Alex Stepanov and Meng Lee of the Hewlett Packard Laboratories in the early 1990s. It has three components:

1. Containers: Generic "off-the-shelf" class templates for storing collections of data
2. Algorithms: Generic "off-the-shelf" function templates for operating on containers
3. Iterators: Generalized "smart" pointers that allow algorithms to operate on almost any container

In 1994, STL was adopted as a standard part of C++.

There are 10 containers in STL:

Kind of container	Containers
Sequential:	**`deque`, `list`, `vector`**
Associative:	**`map`, `multimap`, `multiset`, `set`**
Adapters:	**`priority_queue`, `queue`, `stack`**

F. `vector` (Lab 7 and §6.4)

In Lab 7, we've looked at the `vector` class template in STL and some of the important `vector` operations:

Constructors:

```
vector<T> v,              // empty vector
          v1(100),        // contains 100 elements of type T
          v2(100, val),   // contains 100 copies of val
          v3(fptr,lptr); // contains copies of elements in
                            memory locations fptr up to lptr
```

Copy constructor
Destructor

`v.capacity()`	Number of elements `v` can contain without growing

`v.size()`	Number of elements `v` actually contains
`v.reserve(`*`n`*`)`	Increase capacity (but not size) to *`n`*
`v.empty()`	Check if `v` is empty
Assignment (=)	e.g., `v1 = v2;`
Relational operators	Lexicographic order is used
`v.front()`, `v.back()`, `v[`*`i`*`]`, `v.at(i)`	Access first value, last value, *`i`*-th value without / with range checking (`at` throws out-of-range exception — see p. 272)
`v.push_back(`*`val`*`)`	Add *`val`* at end
`v.pop_back()`	Remove value at end
`v.swap(v1)`	Swap contents with those of vector `v1`

The other operations require knowledge of *iterators*..
Examples:

`v.begin()`	Returns iterator positioned at first element
`v.end()`	Returns iterator positioned immediately after last element
`v.insert(`*`it, val`*`)`	Inserts *`val`* at position specified by iterator *`it`*
`v.erase(`*`it`*`)`	Removes the element at position specified by iterator *`it`*.

Note: `insert()` moves all the array elements from position *`it`* and following one position to the right to make room for the new element.
`erase()` moves all the array elements from position *`it`* and following one position to the left to close the gap.

An iterator declaration for `vector`s has the form:

```
vector<T>::iterator it;
```

Example: Function to display the values stored in a `vector` of `double`s:

```
ostream & operator<<(ostream & out, const vector<double> & v)
{
  for (int i = 0; i < v.size(); i++)
     out << v[i] << "  ";
  return out;
}
```

or using an iterator:

```
ostream & operator<<(ostream & out, vector<double> & v)
{
 for (vector<double>::iterator it = v.begin();
                               it != v.end(); it++)
     out << *it << "  ";
  return out;
}
```

G. A New (But Unnecessary) Revision of Our `Stack` Class Template

Our class `Stack` still has one deficiency, namely, that the stack can become full; it isn't dynamic in that it can grow when necessary. However, we could use `vector` as a container for the stack elements since it can grow automatically as needed, and the `push_back()` and `pop_back()` operations are perfect for stacks.

```
#ifndef STACK_VEC
#define STACK_VEC

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

template<typename StackElement>

class Stack
{
/***** Function Members *****/
public:
  // Don't  need constructor -- let vector's do it
  bool empty() const;
  void push(const StackElement & value);
  void display(ostream & out) const;
  StackElement top() const;
  void pop();

/***** Data Members *****/
private:
  vector<StackElement> myVector;   // vector to store elements
  // don't need myTop -- back of vector is top of stack
}; // end of class declaration

//--- Definition of empty operation
template <typename StackElement>
inline bool Stack<StackElement>::empty() const
{
  return myVector.empty();
}


//--- Definition of push operation
template <typename StackElement>
void Stack<StackElement>::push(const StackElement & value)
{
  myVector.push_back(value);
}
```

```
//--- Definition of display operation
template <typename StackElement>
void Stack<StackElement>::display(ostream & out) const
{
  for (int pos = myVector.size() - 1; pos >= 0; pos--)
    out << myVector[pos] << endl;

/* or using a reverse iterator:
  for (vector<StackElement>::reverse_iterator
         pos = myVector.rbegin(); pos != myVector.rend();  pos++)
    out << *pos << endl;
*/
}


//--- Definition of top operation
template <typename StackElement>
StackElement Stack<StackElement>:: top() const
{
  if (!myVector.empty())
    return myVector.back();
  //else
  cerr << "*** Stack is empty ***\n"; }
}


//--- Definition of pop operation
template <typename StackElement>
void Stack<StackElement>:: pop()
{
  if (!myVector.empty())
    myVector.pop_back();
  else
    cerr << "*** Stack is empty -- can't remove a value ***\n";
}

#endif
```

Basically, all we have done is wrapped a `vector` inside a class template and let it do all the work. Our member functions are essentially just renamings of `vector` member functions.

And there's really no need to do this, since STL has done it for us!

H. STL's `stack` Container

STL includes a `stack` container. Actually, it is an ***adapter*** (as indicated by the fact that its **type parameter is a container type**), which means basically that it is a class that acts as a **wrapper around another class and provides a new user interface for that class**. A *container adapter* such as `stack` uses the members of the encapsulated container to implement what looks like a new container.

For a `stack<C>`, `C` may be any container that supports `push_back()` and `pop_back()` in a LIFO manner; in particular `C` may be a **`vector`**, a **`deque`**, or a **`list`**.

Basic operations:

Constructor `stack< container<T> > st;` creates an empty stack `st` of elements of type `T`; it uses a `container<T>` to store the elements.

Note 1: The space between the two `>`s must be there to avoid confusing the compiler (else it treats it as `>>`); for example, `stack< vector<int> > s;` not `stack< vector<int>> s;`

Note 2: The default container is **`deque`**; that is, if "`container`" is omitted as in `stack<T> st;` a `deque<T>` will be used to store the stack elements Thus `stack<T> st;` is equivalent to `stack< deque<T> > st;`

Destructor

Assignment, relational Operators

`size()`, `empty()`, `top()`, `push()`, `pop()`

Example: Conversion to base two (where our whole discussion of stacks began)
(See Fig. 6.8 on p. 300)

```
/* Program that uses a stack to convert the base-ten
 * representation of a positive integer to base two.
 * Uses the standard C++ stack container.
 *
 * Input:  A positive integer
 * Output: Base-two representation of the number
 ***************************************************/

#include <iostream>
//#include <deque> -- Don't need for default container,
                   // but do need if some other container is used
#include <stack>
using namespace std;

int main()
{
  unsigned number,          // the number to be converted
           remainder;       // remainder when number is divided by 2
  stack<unsigned> stackOfRemainders;
                            // stack of remainders
  char response;            // user response
```

```
  do
  {
    cout << "Enter positive integer to convert: ";
    cin >> number;

    while (number != 0)
    {
      remainder = number % 2;
      stackOfRemainders.push(remainder);
      number /= 2;
    }

    cout << "Base two representation: ";
    while (!stackOfRemainders.empty() )
    {
      remainder = stackOfRemainders.top();
      stackOfRemainders.pop();
      cout << remainder;
    }

    cout << endl;
    cout << "\nMore (Y or N)? ";
    cin >> response;
  }
  while (response == 'Y' || response == 'y');
}
```

I. STL's `queue` Container

Container type `C` may be `list` or `deque`. Why not `vector`?
It can't remove at the front efficiently!

The default container is `deque`.

`queue` has same member functions and operations as `stack` except:

- `front()` (instead of `top()`) retrieves front item
- `pop()` removes front item
- `push()` adds item at back
- `back()` retrieves rear item

Example:

```
#include <string>
#include <queue>
using namespace std;

int main()
{
  queue<int> qint;
  queue<string> qstr;

  // Output number of values stored in qint
  cout << qint.size() << endl;

  // Add 4 positive even integers to qint
  for (int i = 1; i <= 4; i++)
    qint.push(2*i);

  // Change front value of qint to 123;
  qint.front() = 123;

  cout << qint.size() << endl;

  // Dump contents of qint
  while (!qint.empty())
  {
    cout << qint.front() << "  ";
    qint.pop();
  }
  cout << endl;

  // Put strings in qstr and dump it
  qstr.push("STL is"); qstr.push("impressive!\n");
  while (!qstr.empty())
  {
    cout << qstr.front() << ' ';
    qstr.pop();
  }
}
```

Output:

```
0
4
123  4  6  8
STL is impressive!
```

J. Deques (pp. 294-297)

As an ADT, a **deque**, which is an abbreviation for *double-ended queue,* is a sequential container that functions like a queue (or a stack) on both ends. More precisely, it is an ordered collection of data items with the property that items can be added and removed only at the ends. Basic operations are:

- Construct a deque (usually empty):
- Check if the deque is empty
- Push_front: Add an element at the front of the deque
- Push_back: Add an element at the back of the deque
- Front: Retrieve the element at the front of the deque
- Back: Retrieve the element at the back of the deque
- Pop_front:: Remove the element at the front of the deque
- Pop_back:: Remove the element at the back of the deque

STL's **`deque<T>` class template**:

- Has the same operations as `vector<T>` except that there is no `capacity()` and no `reserve()`
- Has two new operations:

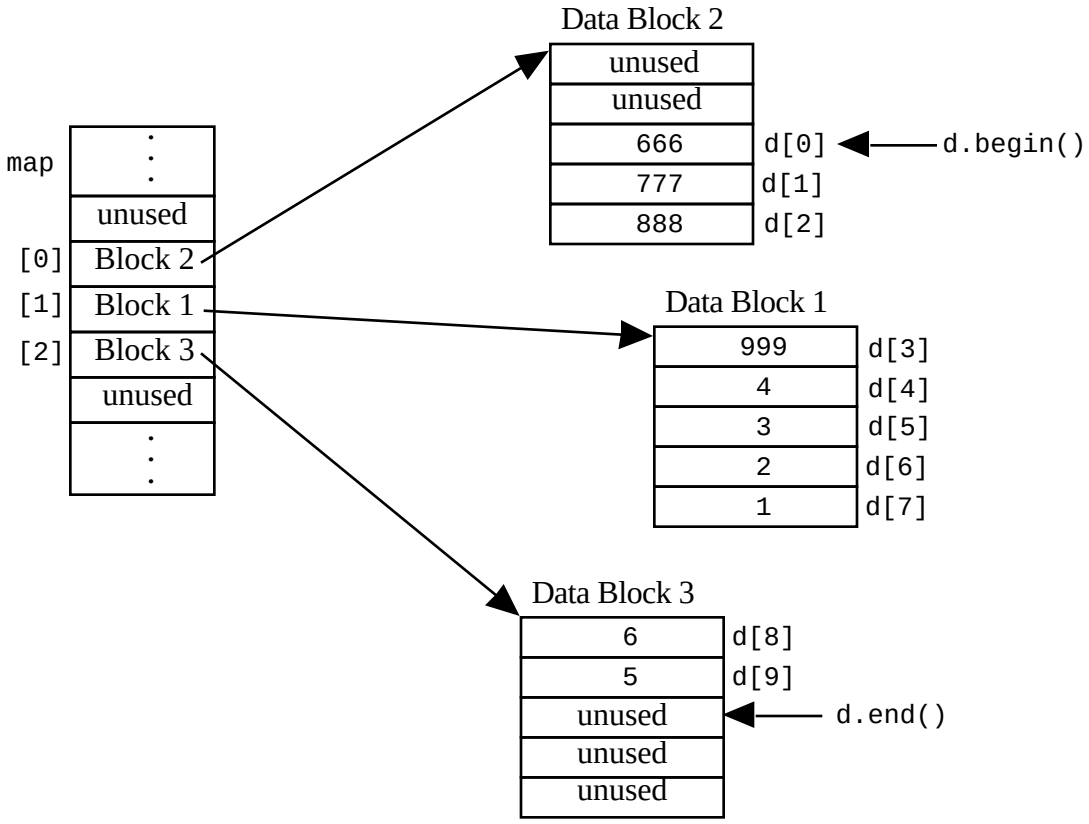
  d`.push_front(`*value*`);` Push a copy of *value* at the front of *d*

  d`.pop_front(`*value*`);` Remove *value* at the front of *d*

- Has several operations like `vector`'s that are not defined for deques as ADTs: `[]`, insert and delete at arbitrary points in the list, same kind of iterators.
  But, insertion and deletion are very inefficient , however, and in fact take longer than for `vector`s. See pp. 296-7 for an explanation of why this is.

One of the bad features of the `vector` container is that when its capacity must be increased, it must copy all the objects from the old vector to the new vector. Then it must destroy each object in the old vector. This is a lot of overhead! With `deque` this copying, creating, and destroying is avoided. Once an object is constructed, it can stay in the same memory locations as long as it exists (provided insertions and deletions take place at the ends of the deque).

The reason for this is that unlike `vector`s, a `deque` is not stored in a single varying-sized block of memory, but rather in a collection of fixed-size blocks (typically, 4K bytes). One of its data members is essentially an array `map` whose elements point to the locations of these blocks. For example, if each block consisted of only five memory locations, we might picture a `deque` containing 999, 888, 777, 666, 4, 3, 2, 1, 6, 5 in this order, from front to back, as follows:



When a data block gets full, a new one is allocated and its address is added to `map`. When `map` gets full, a new one is allocated and the current values are copied into the middle of it.

K. Bitsets and ValArrays (§6.7 & 6.8)

The C++ standard includes **`bitset`** as a container, but it is not in STL. A `bitset` is an array whose elements are bits. It is much like an array whose elements are of type `bool`, but unlike arrays, it does provide operations for manipulating the bits stored in it. They provide an excellent data structure to use to implement sets.

The standard C++ library also provides the **`valarray`** class template, which is designed to carry out (mathematical) vector operations very efficiently. That is, `valarray`s are (mathematical) vectors that have been highly optimized for numeric computations.

L. Algorithms in the STL (Standard Template Library) (§7.5)

Another major parts of STL is its collection of more than 80 generic **algorithms**. They are not member functions of STL's container classes and do not access containers directly. Rather they are stand-alone functions that operate on data by means of *iterators* . This makes it possible to work with regular C-style arrays as well as containers. We illustrate one of these algorithms here: **sort**.

Sort 1: Using <

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

// Add our Display() template for arrays

int main()
{
  int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
  // To use sort, we must supply start and "past-the-end" pointers
  sort(ints, ints + 8);
  cout << "Sorted list of integers:\n";
  Display(Ints, 8);

  double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
  sort(dubs, dubs + 7);
  cout << "\nSorted list of doubles:\n";
  Display(Dubs, 7);


  string strs[] = {"good", "morning", "cpsc", "186", "class"};
  sort(strs, strs + 5);
  cout << "\nSorted list of strings:\n";
  Display(strs, 5);

}

//--- OUTPUT -------------------------------

Sorted list of integers:
1  22  33  66  222  444  555  777

Sorted list of doubles:
0.1  2.2  3.3  22.2  44.4  55.5  77.7

Sorted list of strings:
186  class  cpsc  good  morning
```

Sort 2: Supplying a "less-than" function to use in comparing elements

```
#include <iostream.h>
#include <string>
#include <algorithm>

// Add our Display() function template for arrays

bool IntLessThan(int a, int b)
{ return a > b; }

bool DubLessThan(double a, double b)
{ return a > b; }

bool StrLessThan(string a, string b)
{ return !(a < b) && !(a == b); }

int main()
{
  int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
  sort(ints, ints + 8, IntLessThan);
  cout << "Sorted list of integers:\n";
  Display(ints, 8);

  double dubs[] = {55.5, 3.3, 44.4, 2.2, 22.2, 77.7, 0.1};
  sort(dubs, dubs + 7, DubLessThan);
  cout << "\nSorted list of doubles:\n";
  Display(dubs, 7);

  string strs[] = {"good", "morning", "cpsc", "186", "class"};
  sort(strs, strs + 5, StrLessThan);
  cout << "\nSorted list of strings:\n";
  Display(strs, 5);

}

//-----------------------------------------

Sorted list of integers:
777  555  444  222  66  33  22  1

Sorted list of doubles:
77.7  55.5  44.4  22.2  3.3  2.2  0.1

Sorted list of strings:
morning  good  cpsc  class  186
```

Sort 3: Sorting a vector of stacks using < (defined for stacks)

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
#include "StackT.h"

/* Add operator<() to our Stack class template as a member
   function with one Stack operand or as a friend function with
   two Stacks as operands.
   Or because of how we're defining < for Stacks here,
        st1 < st2   if   top of st1  <  top of st2
   we can use the top() access function and make operator<()
   an ordinary function */

template <typename StackElement>
bool operator<(const Stack<StackElement> & a,
               const Stack<StackElement> & b)
{ return a.top() < b.top();}

int main()
{
  vector< Stack<int> > st(4);     // vector of 4 stacks of ints

  st[0].push(10); st[0].push(20);
  st[1].push(30);
  st[2].push(50); st[2].push(60);
  st[3].push(1);  st[3].push(999);  st[3].push(3);

  sort(st.begin(), st.end());
  for (int i = 0; i < 4; i++)
  {
    cout << "Stack " << i << ":\n";
    st[i].display();
    cout << endl;
  }
}
```

Output

```
Stack 0:
3
999
1

Stack 1:
20
10

Stack 2:
30
```

Stack 3:
70
50