# Standard Template Library

Slides from

www.sethi.org/classes/cet375/daily_lecture_notes/chapter_06-templates.ppt

# Standard Template Library (STL)

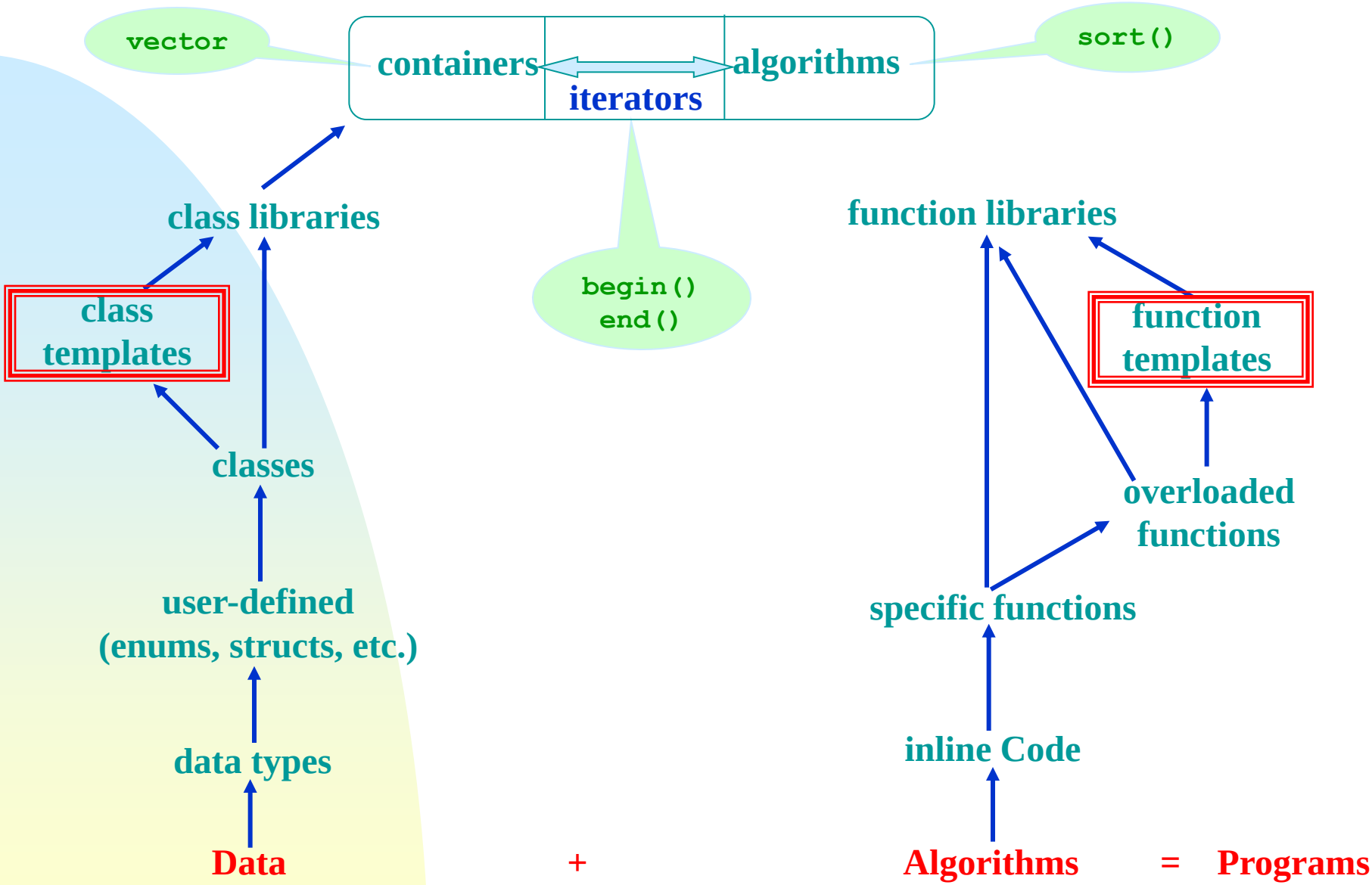



Fig 6.1 The Evolution of Reusability/Genericity

# Templates

Templates allow functions and classes to be ***parameterized*** so that the ***type of data being operated upon (or stored) is received via a parameter*****.**

Templates provide a means to ***reuse code*** — *one template definition* can be used to create ***multiple instances of a function (or class), each operating on (storing) a different type of data***.

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

# Functions

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

Example: Interchange the values of two `int` variables `x` and `y`.

Instead of inline code:

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

Fortran - 1950s "call"

write a function:

```
/* Function to swap two int variables.
   Receive:   int 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;
}
```

# General Solution

This function gives a general solution to the interchange problem for `ints` because this `Swap` function can be used to exchange the values of any two integer variables:

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

In contrast, inline code would have to be rewritten for each pair of integer variables to swap.

# But not for doubles…

Fortran's solution

To interchange the values of two `double` variables:
Can't use the preceding function; it swaps `ints` not `doubles`.
However, ***overloading*** allows us to define multiple versions of the same function:

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

The two different `Swap` functions are distinguished by the compiler according to each function's ***signature*** (name, number, type, and order of parameters).

Sec. 6.2

Signature

# And for strings…

To interchange the values of two `string` variables:

Again, overload function `Swap()`:

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

# What about User Defined Types?

Fortran & 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 swap two Time variables.
   . . .
*/
void Swap(Time & first, Time & second)
{
   Time temp = first;
   first = second;
   second = temp;
}
```

# 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.*

# Template Mechanism

Declare a ***type parameter*** ***(type placeholder)*** and use it in the function instead of a specific type. This requires a ***different kind of parameter list*:**

```
/* Swap template for exchanging the values of any two
   objects of the same type
   Receive:    type parameter DataType
               first and second, two objects of same type
   Pass back:  first and second with values swapped
   Assumes:    Assignment (=) defined for type DataType
*/
template <typename DataType >  // type parameter
void Swap(DataType & first, DataType & second)
{
   DataType temp  = first;
   first = second;
   second = temp;
}
```

# Template Mechanism Comments

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

• "Normal" parameters (& arguments) appear within function parentheses; type parameters (& arguments for class templates) appear within ***angle brackets*** **(< >).**

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

•Unlike other functions, a function template ***cannot be split across files***. That is:

•We can't put prototype in a header
file and definition in an implementation file; it all goes in the header file.

# How is a Template Used?

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

Example:

```
#include "Time.h"
#include "Swap.h"  //ONE function template definition
int main()
{
  int    i1, i2;
  double d1, d2;
  string s1, s2;
  Time   t1, t2;
   ...              // Compiler generates definitions
                    // of Swap() with DataType replaced
  Swap(i1, i2);     //       by int
  Swap(d1, d2);     //       by double
  Swap(s1, s2);     //       by string
  Swap(t1, t2);     //       by Time
}
```

# General Form of Template (simplified)

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

or

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

where:

*TypeParam* is a type-parameter (placeholder) naming the "generic" type of value(s) on which the function operates

*FunctionDefinition* is the definition of the function, using type *TypeParam*.

# Template Instantiation

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 — with types `int`, `double`, `string`, and `Time`. In each instantiation, the type parameter is said to be ***bound*** to the actual type passed.

A template thus serves as a pattern for the definition of an ***unlimited number of instances***.

# Compiler processes template

*In and of itself, the template does nothing.* When the compiler encounters a template like `Swap()`, *it simply* ***stores the template 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;
}
```

Algorithm for instantiation:

(1) *Search parameter list* ***of function template*** *for type parameters.*
(2) *If one is found, determine type of corresponding argument* ***in the fxn call****.*
(3) *Bind these types.*

For this instantiation 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.)

# Instantiation

The previous example illustrates:

Implicit instantiation: *Compiler determines the type by the parameter(s) in the function call.*

For efficiency (and often correctness), compilers frequently recommend:

Explicit instantiation: *Provide a **prototype** of the function to be later instantiated by the compiler.*

```
void Swap(int, int);
void Swap(double, double);
void Swap(string, string);
void Swap(Time, Time);
```

# Example

```
/* Function template to find 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 Max( ElementType array[], int numElements)
{
  ElementType  biggest = array[0];
  for (int i = 1; i < numElements; i++)
    if (biggest < array[i])
      biggest = array[i];
  return biggest;
}
```

# Test Driver

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

**Explicit Instantiation:**

```
double Max(double, int);
int Max(int, int);
```

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

  int num[20] = {2, 3, 4, 1};
  cout <<"Largest value in num is "<< Max(num, 4);
}
```

Execution:

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

# Compiler Processing...

When compiler encounters `Max(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 `Max()`:**

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

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

# Templates with multiple parameters

```
template <typename TypeParam1, typename TypeParam2, ...>
FunctionDefinition
```

Each type parameter *must* appear at least once in the function's parameter list (signature). *Why?* Compiler must be able to determine actual type corresponding to each type parameter *from a function call*.

```
/* 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);
}
```

# Test Driver

```
#include "Convert.h"
#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
```

# Not Allowed!

The following version of function template `Convert` would not be legal:

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

We could provide a dummy second parameter indicating the type of the return value:

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

Function call:

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

# Class Templates

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;
};
```

# What's wrong with `typedef`?

To change the meaning of **`StackElement`** throughout the class, we need only change the type following the **`typedef`**.

*Problems:*

1. Changes the header file

   ⇒ Any program/library that uses the class must be **recompiled**.

2. A name declared using **`typedef`** can have **only one meaning**.

   ⇒ If we need stacks with different element types
   e.g., a **`Stack`** of ints **and** a **`Stack`** of strings,
   we must create **separate** stack classes with **different names**.

Can't overload like functions

# *Type-Independent* Container

Use a **class template**: the class is **parameterized** so that it **receives the type of data stored in the class via a parameter** (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;
};
```

**`StackElement`** is a "blank" type (a type placeholder) to be filled in later.

# General form of class template declaration

```
template <typename TypeParam> or
                template <class TypeParam>
 class SomeClass
 {
     // ... members of SomeClass ...
 };
```

More than one type parameter may be specified:

```
template <typename TypeParam1,...,
          typename TypeParamn>
class SomeClass
{
     // ... members of SomeClass ...
};
```

Can also be set to a default value

# Instantiating class templates

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;
```

"name-mangling" – compiler modifies class/object names

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

See Slide 33

# Rules for class templates

1. **Definitions of *member functions* outside class declaration must be function templates.**
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.**

*For our Stack class*:

a. *Prototypes of function members*? No change — rules don't apply to them since they're declared **within** the class declaration.

# Function DEFINITIONS

b. *Definitions of function members*?

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

```
template <typename StackElement>    // rule #1
// ... definition of Stack()

template <typename StackElement>    // rule #1
// ... 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

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()
{ /* ... body of Stack() ... */ }

template <typename StackElement>
inline bool Stack<StackElement>::empty()
{ /* ... body of empty() ... */ }

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

template <typename StackElement>
void Stack<StackElement>::display(ostream & out)
{ /* ... 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

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
```

# Friend Fxns

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

For example, to 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
```

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

# Contd.

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

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

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

# Test Driver

Program to test this **Stack** template:

```
#include <iostream>
using namespace std;
#include "StackT.h"
int main()
{
  Stack<int> intSt;     // stack of ints
  Stack<char> charSt;   // stack of char

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

  for (char ch = 'A'; ch <= 'D'; ch++)
    charSt.push(ch);
  cout << charSt << endl;
}
```

Output:

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

# Templates with ordinary parameters

*** 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_H
#define __STACKT_H
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
```

# Test Driver

Program to test this modified **Stack** template:

```
...
int main()
{
  Stack<int, 4> intSt;   // stack of 4 ints
  Stack<char, 3> charSt; // stack of 3 chars

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

  for (char ch = 'A'; ch <= 'D'; ch++)
    charSt.push(ch);
  cout << charSt << endl;
}
```

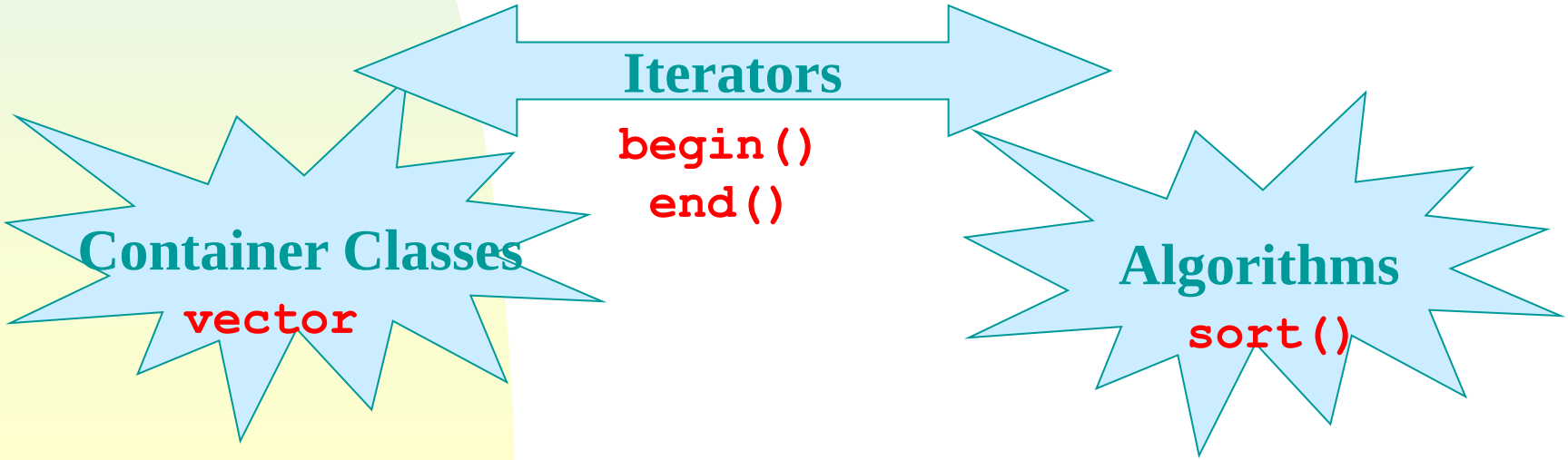
Output:

```
4       *** Stack full -- can't add new value ***
3       C
2       B
1       A
```

# STL (Standard Template Library)

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



# STL's Containers (§ 6.4ff)

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

There are **10** containers in STL:

| Kind of Container | STL Containers |
|---|---|
| Sequential: | `deque, list, vector` |
| Associative: | `map, multimap, multiset, set` |
| Adapters: | `priority_queue, queue, stack` |
| Non-STL: | `bitset, valarray, string` |

# vector (§6.4 & Lab 7)

## 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.max_size()` | Upper limit on the size and capacity |
| `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 |
| `v.push_back(val)` | Add *val* at end |
| `v.pop_back()` | Remove value at end |
| `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 - p. 272) |
| Relational operators | Lexicographic order is used |
| Assignment (=) | e.g., `v1 = v2;` |
| `v.swap(v1)` | Swap contents with those of vector `v1` |

# Iterators – generalized pointers

The other operations require knowledge of *iterators*. For example:

| | |
|---|---|
| **`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 elements from position ***`it`*** and following one position to the right to make room for the new one. **`erase()`** moves all the 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:

```
template<typename T>                                          T
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:

```
                                                              T
  for (vector<double>::iterator it = v.begin();
                                     it != v.end(); it++)
     out << *it << "  ";
```

Go to 54

# `vector` inefficiences

- When its capacity must be increased,
  - $\Rightarrow$ it must copy all the objects from the old vector to the new vector.
  - $\Rightarrow$ it must destroy each object in the old vector.
  - $\Rightarrow$ 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 (if insertions and deletions take place at the ends of the deque).

- Unlike `vectors`, 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.

# A New (But Unnecessary) Revision of Our Stack Class Template

Our class **Stack** still has one deficiency, namely, stacks can become full; they aren't dynamic in that they 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.

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

template<typename StackElement>
class Stack
{
/***** Function Members *****/
public:
  Stack() {};  // let vector's constructor do the work
  bool empty() const;
  void push(const StackElement & value);
  void display(ostream & out) const;
  StackElement top() const;
  void pop();
```

# Contd.

```
/***** 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 (!empty())
    return myVector.back();
  //else
  cerr << "*** Stack is empty ***\n";
}
```

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

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!

# STL's `stack` container

STL includes a **`stack`** container.

Actually, it is an **adapter**, as indicated by the fact that *one of its type parameters is a* **container type.**

Sample declaration:

```
stack<int, vector<int> > st;
```

**Errors in text: pp. 299 & 301**

Basically, it is a class that acts as a **wrapper** around another class, providing 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<T, C<T> >`**, **`C<T>`** 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< T, C<T> > st;` creates an empty stack `st` of elements of type `T`; it uses a container `C<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< int, vector<int> > s;
```

not `stack< int, vector<int>> s;`

Note 2: The default container is `deque`; that is, if `C<T>` 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< T, deque<T> > st;`

Destructor

Assignment, relational Operators

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

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

# Example: convert base 10 to base 2

```
#include <iostream>
//#include <deque> not needed for default container,
        // but do need if some other container is used
#include <stack>
using namespace std;

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

# Example: convert base 10 to base 2

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

# STL's `queue` container

In **`queue<T, C<T> >`**, container type **`C`** may be **`list`** or **`deque`**.

Why not **`vector`**? You can't **remove from the front efficiently!**

The default container is **`deque`**.

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

**`push()`** adds item at back (our **`addQ()`**)
**`front()`** (instead of **`top()`**) retrieves front item
**`pop()`** removes front item (our **`removeQ()`**)
**`back()`** retrieves rear item

# queue 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;

  for (int i = 1; i <= 4; i++)
    qint.push(2*i);

  qint.push(123);

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

# Contd.

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

  qstr.push("STL is"); qstr.push("impressive!\n");
  while (!qstr.empty())
  {
    cout << qstr.front() << ' ';
    qstr.pop();
   }
}
```

Output:

```
0
5
2  4  6  8 123
STL is impressive!
```

# Deques (Read pp. 294-7)

As an ADT, a **deque** — an abbreviation for **double-ended queue** — is a sequential container that functions like a queue (or a stack) on both ends.

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 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 copy of ***`value`*** at front of ***`d`***

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

Like STL's **`vector`**, it has several operations that are not defined for deque as an ADT:

**`[]`**
insert and delete at arbitrary points in the list,
same kind of iterators.

But insertion and deletion are not efficient and, in fact, take longer than for **`vector`**s.

# `vector` **vs.** `deque`

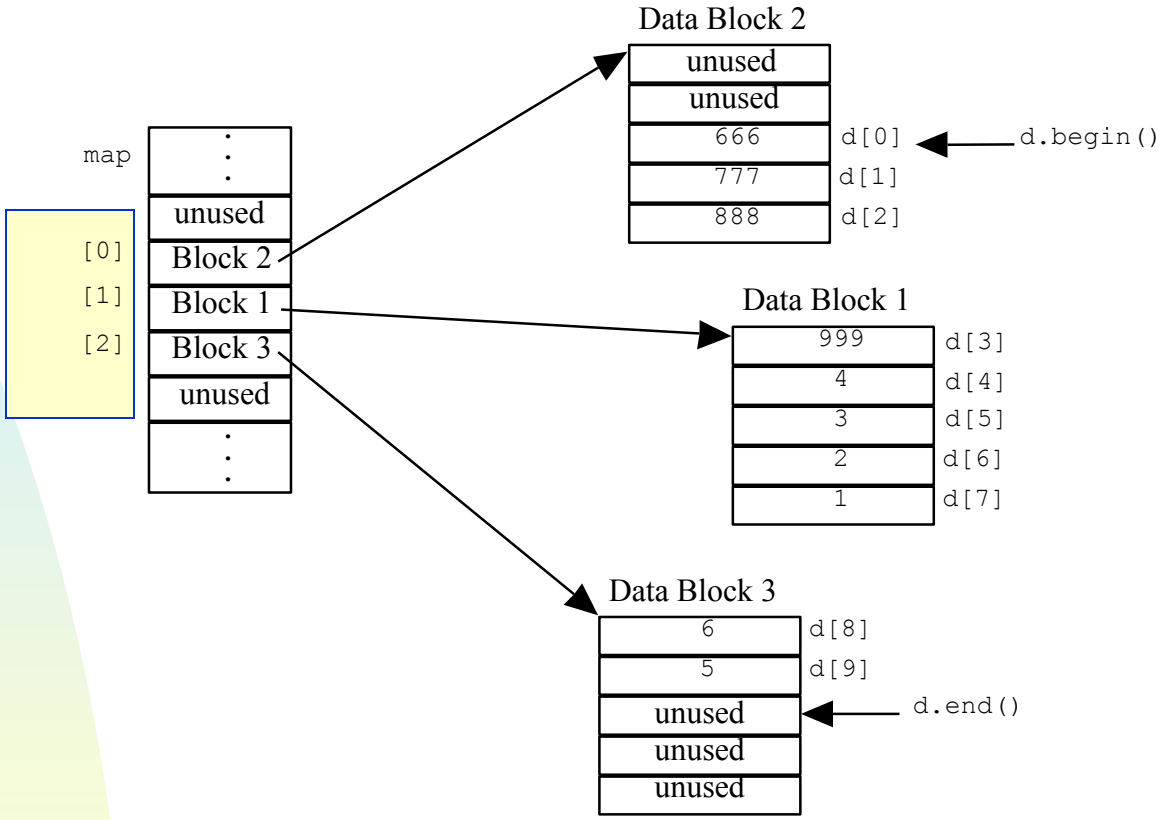
Capacity of a **`vector`** must be increased
⇒ it must copy the objects from the old **`vector`** to the new **`vector`**
⇒ it must destroy each object in the old **`vector`**
⇒ a lot of overhead!

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** (if insertions and deletions take place at the ends of the **`deque`**).

Unlike **`vector`**s, a **`deque`** isn't 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 666, 777, 888, 888, 4, 3, 2, 1, 6, 5 in this order, from front to back, as follows:




⇨ When no room at one end of a data block, 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.

*Inserts and deletes may involve cross-block element-shifting!*

# STL algorithms

do not access containers directly
stand-alone functions that operate on data by means of *iterators*
can work with regular C-style arrays as well as containers.

```
#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};
  // 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
```

# Supply own comparison operator

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

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 << "Sorted list of doubles:\n";
  Display(dubs, 7);

  string strs[] =
{"good","morning","cpsc","186","class"};
  sort(strs, strs + 5, StrLessThan);
  cout << "Sorted 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
```

# 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.

# STL's algorithms (§7.5)

Another major part 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 <

```
template <typename ElemType>
void Display(ElemType arr, int n)
{
  for (int i = 0; i < n; i++)
    cout << arr[i] << "  ";
  cout << endl;
}
```

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

// Add a Display() template for arrays

int main()
{
  int ints[] = {555, 33, 444, 22, 222, 777, 1, 66};
  // 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>
#include <string>
#include <algorithm>
using namespace std;

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 << "Sorted list of doubles:\n";
  Display(dubs, 7);

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

**Output:**

```
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:
60
50
```