Fundamentals of Computer Programming

Chapter 9

Standard Template Library (STL)

Chere L. (M.Tech)
Lecturer, SWEG, AASTU

Outline

- Introduction to Generic Programming
- Basics of Template
  - *What is Template?*
  - *Types of Templates*
  - *Specialized Templates*
- Standard Library Templates(SLT)
  - *Basic concept of SLT (What & Why?)*
  - *Classes of SLT (Standard Library functions & OO Class Library)*
  - *Components of SLT (Containers, Algorithms, Iterators)*
- Exception Handling

Int. to Generic Programming (1/7)

What is Generic Programming?

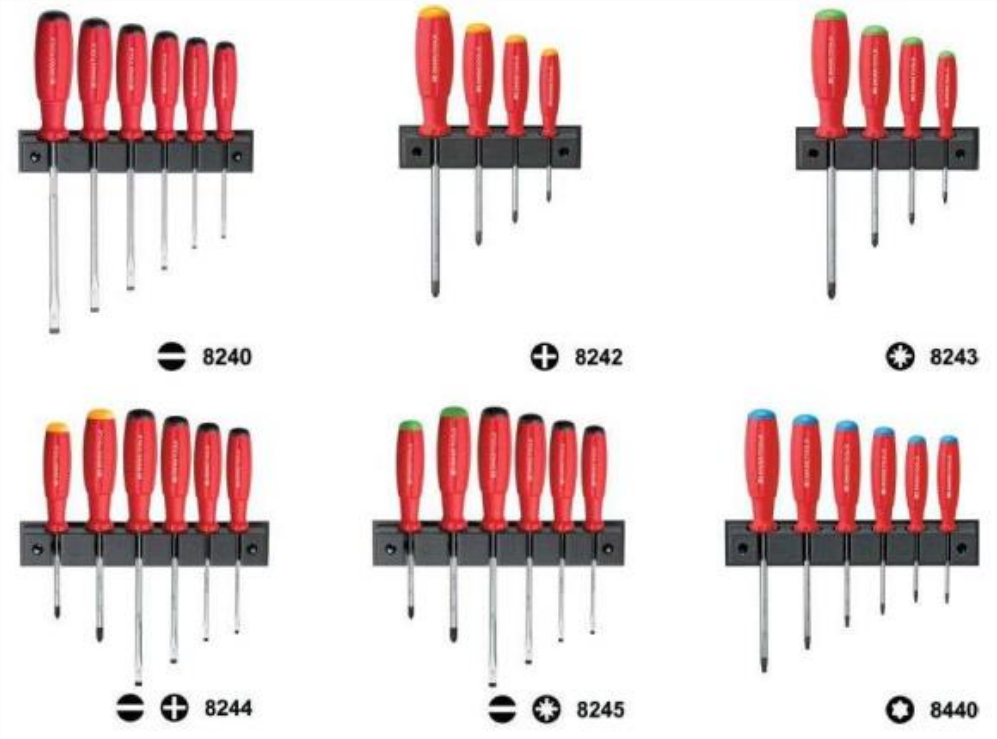
- It is a style of computer programming in which **algorithms** are written in terms of **types to-be-specified-later** that are then **instantiated** when needed for specific types provided as parameters.
- In simple word, it refers to programming/developing algorithms with the abstraction of types

Why Generic Programming?

- ✓ Code reusability - Reuse of algorithmic code is difficult
- ✓ Hence, algorithms are as **insensitive to changes of data structure** as possible.

Int. to Generic Programming (2/7)

- **Real world use cases**



- **Is software any different?**

Int. to Generic Programming (3/7)

- **Case study :** How to solve the following problems in such a way it works with all types (int, float, char, string etc.)?
  - ✓ *Find the largest value of two values*
  - ✓ *Read/print/find the sum of array elements*

➢ **Example 1:**

```
double sum(double * v, int n) {
  double s = 0;
  for(int i = 0; i < n; ++i)
    s += v[i];
  return s;
}
```

➢ is that (re)usable?

- ✓ int * vi; → sum doesn't work . . . vi: wrong value type
- ✓ struct vec3 { double x[3]; ...}; vec3 *v3 = ...;

  → sum doesn't work . . . v3: wrong access pattern

Int. to Generic Programming (4/7)

Example 1: Consider the following function:

```
int biggest (int arg1, int arg2){
        if (arg1 > arg2)
                return arg1;
        else
                return arg2;
}
```

- This function very nicely finds the maximum of two integers.
- What if we also need a function to find the max of two values *of other types (floats, double, char, string)?*
- ***Solution:***
  - ✓ *Rewriting the code or function overloading*
  - ✓ *Using data type of larger class (e.g. double)*

Int. to Generic Programming (5/7)

Function overloading?

```
int biggest (int arg1, int arg2)
{
      if (arg1 > arg2)
         return arg1;
      else
         return arg2;
}
```

```
float biggest (float arg1, float arg2)
{
      if (arg1 > arg2)
         return arg1;
      else
         return arg2;
}
```

- The only thing different about these two functions are the data types of the parameters and the return type

Int. to Generic Programming (6/7)

Can we do better?

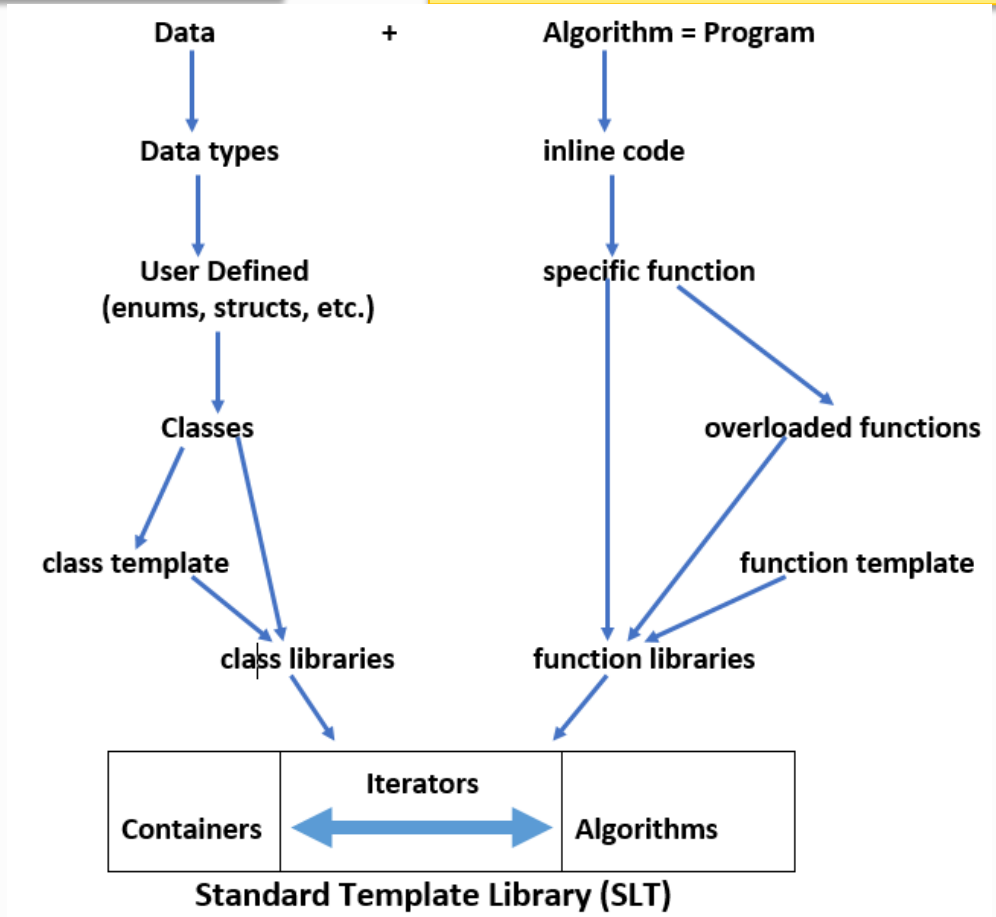
- Yes, Generic Programming
  - ✓ Make implementations as **general as possible** . . . but not more.

Int. to Generic Programming (7/7)

Evolution of Reusability, and Generality

Evolution of **data features** of programming languages

Evolution of **algorithmic features** of programming languages



- Data and algorithms cannot be separated
  Niklaus Wirth (inventor of Pascal)

Basics of Templates (1/2)

What are Templates?

- It is a tool provided by the C++ language to write a **common function/class** that is ***independent of a data type.***
- It defined as a **blueprint or formula** for creating a generic class or a function.
- C++ template is also known as *generic functions or classes.*
- Templates are a ***placeholder for a type*** *and* are not *a data types* by themselves.
- A programmer can create a ***template*** with some or all variables therein having ***unspecified data types.***
- To simply put, programmer can create a ***single function or single class*** to work with different data types using templates.

Basics of Templates (2/2)

- C++ templates come in two flavors:
  - *Functions templates*
  - *Class templates*
- The *template function/class* **embodies** the *common algorithm* for all data types.
- In C++ a keyword ***"template"*** and ***"class/type"*** along with ***angled bracket (<>)*** are used for the template's syntax (definition).
- Generic parameter is defined inside the *angled bracket prefixed with the keyword class/type.*

How template works

- The template *gets expanded* at *compilation time*, just like macros and allows a function or class to work on different data types without being rewritten.

Function Templates (1/6)

- Function templates are a special functions that can operate with *generic types.*
- A template functions can be adapted to more than one type or class without repeating the entire code for each type.
- This can be achieved using ***template parameters*.**
- A template parameter is a special kind of parameter that can be used to **pass a type as argument**.

- ***Syntax:***
  - *template <class identifier> function_declaration;*
  - *template <typename identifier> function_declaration;*

- The **function template** is **defined** just like an **ordinary function** except it start/prefixed with keyword template followed by *generic data-types inside in angular brackets*.

Function Templates (2/6)

Description of the Syntax:

The **template prefix** tells the compiler that this is a template, so treat it differently from a normal function.

- **Generic** type definition
- Instead a keyword **class** the **typename** can be used
- "class" as a synonym for "kind" or "category"
- It can be two or more
  <class T, class U, . . . >

```
template <class T>
T biggest (T arg1, T arg2)
{
   //function body,
   //generic algorithms
}
```

return type is T

The **type parameters**.

- ✓ identifier **T** is most often used, but any identifier is acceptable.
- ✓ arg1 and arg2 are of **type T**

Function Templates (3/6)

Notes:

- A function template cannot be split across files
  - Function template *specification (declaration)* and implementation must be in the same file
- A function template is a pattern that describes how specific function is constructed based on given actual types.
- Type parameter said to be "**bound**" to the actual type passed to it and each of the type parameters must appear at least once in parameter list of the function.
- When a function template is instantiated (called) a compiler finds type parameters in list of function template and determine corresponding argument for each type parameter.
- The compiler uses only *types of the arguments* in the call

Function Templates (4/6)

Example 1: Comparison algorithms

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Function Templates (5/6)

Example 2: Swapping algorithms

```
#include <iostream>
using namespace std;
const unsigned int SIZE = 5;

template <typename T>
void Swap (T &f, T &s);

int main(){
    float n, m;
    char x, y;
    int a, b;
    cout<<"\nInput two float numbers: ";
    cin>>n>>m;
            swap(n,m);
    cout<<"The swapped numbers: ";
    cout<<n<<" "<<m<<endl;
    cout<<"\nInput two float numbers: ";
    cin>>x>>y;          swap(x,y);
    cout<<"The swapped numbers: ";
    cout<<x<<" "<<y<<endl;

    cout<<"\nInput two characters ";
    cin>>a>>b;          swap(x,y);
    cout<<"The Swapped characters: ";
    cout<<a<<" "<<b<<endl;
    }
    template <typename T>
    void Swap (T &first, T &second)
    {
      T temp = first;
       first = second;   second = temp;
    }
```

Function Templates (6/6)

Example 3: algorithm to print array

```
#include <iostream>
using namespace std;
const unsigned int SIZE = 5;

template <typename ElementType>
void printArray(const ElementType list[])
{
    for(unsigned int i = 1; i <= SIZE; i++)
    {
        cout << list[i-1] << '\t';
        if (i % 10 == 0)
            cout << endl;
    }
}

int main()
{
    float array[SIZE];
    cout<<"Please input "<<SIZE<<" Array elements: ";
    for(unsigned int i = 0; i < SIZE; i++)
        cin >> array[i];
    cout<<"\nArray elements are: ";
    printArray(array);
}
```

Note:

Try it by changing the array type to other data types.

Class Templates (1/6)

(a) Class Template

- Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Like function templates, you can also create class templates for generic class operations.
- When a class uses the concept of Template, then the class is known as generic class.
- **Class Template** can also be defined similarly to the Function Template.
- Declaration syntax

```
template<class Ttype>
class class_name{
    //class member data
    //and methods
}
```

instance of a class

```
class_name<type> ob;
```

Where type is a **concrete Type** that you want the class members to be.

Class Templates (2/6)

Example 1:

```
#include <iostream>
using namespace std;
template<class T>
class Adder {
   public:
   T num1,  num2;
    Adder (T x, T y) {
                num1 = x;
                num2 = y;
     }
   void add() {
      cout << "Sum of the numbers:   ";
     cout<< num1+num2<<endl;
   }
};
```

```
int main() {
      A<int> d (4, 5);
      d.add();

      A<int> d (14.5, 65.75);
      d.add();

return 0;
}
```

Class Templates (3/6)

Rules and features of Class Templates

1. Member functions must be defined **in the same file as the class declaration.**
2. All uses of class name as a **type must be parameterized.**
   - *Like a function template, the compiler generates no codes when it encounters a class template, but generates a real class definition when it encounter class instantiation as follow:*
   - **Example:**

     Box <int> myBox ();

     where

     - ✓ ***Box*** **-** *the template name,*
     - ✓ ***int*** **-** *concrete type replaces the template parameter*
     - ✓ ***myBox*** *- the object name*

Class Templates (4/6)

Rules and features of Class Templates

3. If the functions are defined inside the class, then they are defined just like ordinary functions. However, Definitions of member functions outside class declaration **must be function templates**

syntax: template <class T>

returntype classname<T>::function_name(arg-list)

Example:

```
template <class T>
class Box{
  private:
     T dataMember;
  public:
     Box (const T& data)
          {dataMember = data };
     void display ( );
};
```

```
template <class T>
void Box<T>::display( )
{
   cout << dataMember;
}
int main(){
     Box<int> myBox(4);
     Box<float> myBox(5.7);
}
```

Class Templates (5/6)

Rules and features of Class Templates

4. The name of the template parameter cannot be used more than once in the template class's list of template parameters

```
template <class T, class T>   //Error
class X{
        //rest of the class
        };
```

- However, the same name for a template parameter can be used in the list of template parameters of two different template classes.

```
template <class T>
class X{
        //rest of the class
        };
```

```
template <class T>
class Y{
        //rest of the class
        };
```

Class Templates (6/6)

Example 2:

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

template <class T>
class Calculator {
   private: T num1, num2;
   public:
Calculator(T n1, T n2){
          num1 = n1; num2 = n2;
}
void displayResult() {
       cout << "Numbers are: “;
       cout << num1 << " and " << num2 << ".";
       cout << “\nAddition is: " << add();
       cout << “\nSubtraction is: " << subtract();
       cout << “\nProduct is: " << multiply();
       cout << “\nDivision is: " << divide();
}
```

```
T add() {
    return num1 + num2; }
T subtract() {
    return num1 - num2; }
T multiply() {
     return num1 * num2; }
T divide() {
     return num1 / num2; }
 };

int main() {
      Calculator<int> intCalc(2, 1);
      Calculator<float> floatCalc(2.4, 1.2);
      cout << “\nInt results: ";
      intCalc.displayResult();
      cout << “\nFloat results: ";
      floatCalc.displayResult();
      return 0;
       }
```

Specialized Templates (1/2)

- It is possible in C++ to get a **special behavior** for a particular data type?
- **Yes**, using **template specialization**.
- ***Template specialization*** refers to the implementation of a ***function templates or class templates*** for a specific data type.
- **Scenario:**
  - Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all data-types except char.
  - In case of char, total possible values are 256 and counting sort may be a better option.

Specialized Templates (2/2)

Example:

```
#include<iostream>
using namespace std;
template<class T>
void biggest (T arg1, T arg2){
        if (arg1 > arg2)      cout<<"Lager Number is "<<arg1<<endl;
        else if (arg1 > arg2)      cout<<"Lager Number is "<<arg2<<endl;
        else       cout<<“The two numbers is equal”<<endl;
}
template<>
void biggest<string> (string str1, string str2){
        if (str1.compare(str2) > 0)            cout<<str1<<endl;
        else if (str1.compare(str2) < 0)       cout<<str2<<endl;
        else       cout<<"The two string is equal"<<endl;
}
int main(){
        int a = 5, b = 4;     biggest(a, b);      biggest("Addis", "Ababa");   }
```

Multiple template parameters (1/2)

Using multiple template parameters

- Classes templates and function templates can have many template parameters
- **Example 1**: **function template with multiple template parameters**

```
#include <iostream>
using namespace std;
template <typename T, typename U>
T max(T x, U y)
{
    return (x > y) ? x : y;
}

int main()
{
    cout << max(2, 3.5) << '\n';
    return 0;
}
```

Multiple template parameters (2/2)

Example 2: class template with multiple template parameters

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

// Class template with two parameters
template<class T1, class T2>
class Test{
        T1 a;        T2 b;
    public:
        Test(T1 x, T2 y){ a = x;   b = y;   }
        void show() {
            cout << a << " and " << b << endl;
        }
};
int main(){
    // instantiation with float and int type
    Test <float, int> test1 (1.23, 123);
    test1.show();
    // instantiation with float and char type
    Test <int, char> test2 (100, 'W');
    test2.show();

    return 0;
}
```

Standard Template Library (STL)

What is STL?

- Collections of class template for common data structures
- The standard implementation of C++ provide a set of header files where a *large number of useful class templates* have been defined
- These files contain ***definitions of the class templates***, their member functions and a number of global associated functions.
- The ***global associated functions*** implement commonly used algorithms.
- STL provides powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.
- The STL was conceived and designed for performance and flexibility.

STL: Components

- STL provides four components called ***algorithms, containers, functions, and iterators :***

1. Containers

- Generic "off-the-shelf" class templates for storing collections of data

2. Algorithms (function objects)

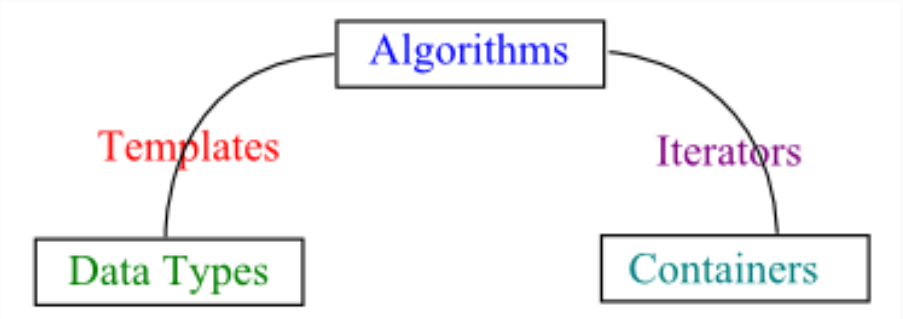
- Generic "off-the-shelf" function templates for operating on containers

3. Iterators

- Generalized "smart" pointers that allow algorithms to operate on almost any container

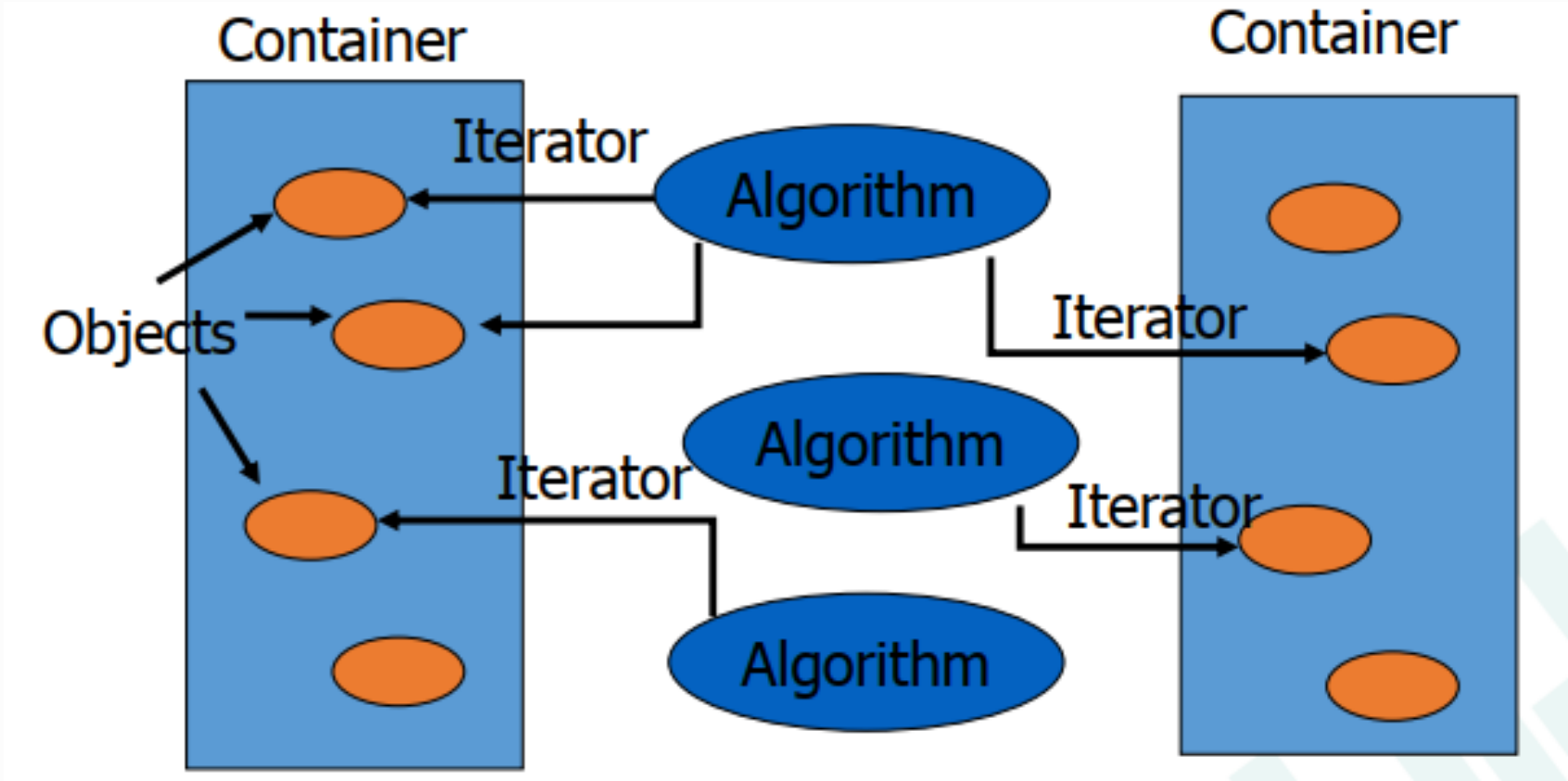
Note:

- The templates and iterators make the algorithms independent of containers respectively.



STL: Components (cont'd)

Containers, Iterators, Algorithms



STL: Components (cont'd)

The common Containers, Iterators and Algorithms:

Containers (holds data)	Iterators (access data)	Algorithms (manipulate data)
➢ *Sequence containers*	➢ *begin()*	➢ *Sorting algorithms*
➢ *Container adapters*	➢ *next()*	➢ *Searching algorithms*
➢ *Associative containers*	➢ *prev()*	➢ *Min & Max Operations*
➢ *Unordered associative containers*	➢ *end()*	➢ *Modifying algorithms*
	➢ *advance()*	➢ *Non modifying algorithms*

STL: Containers

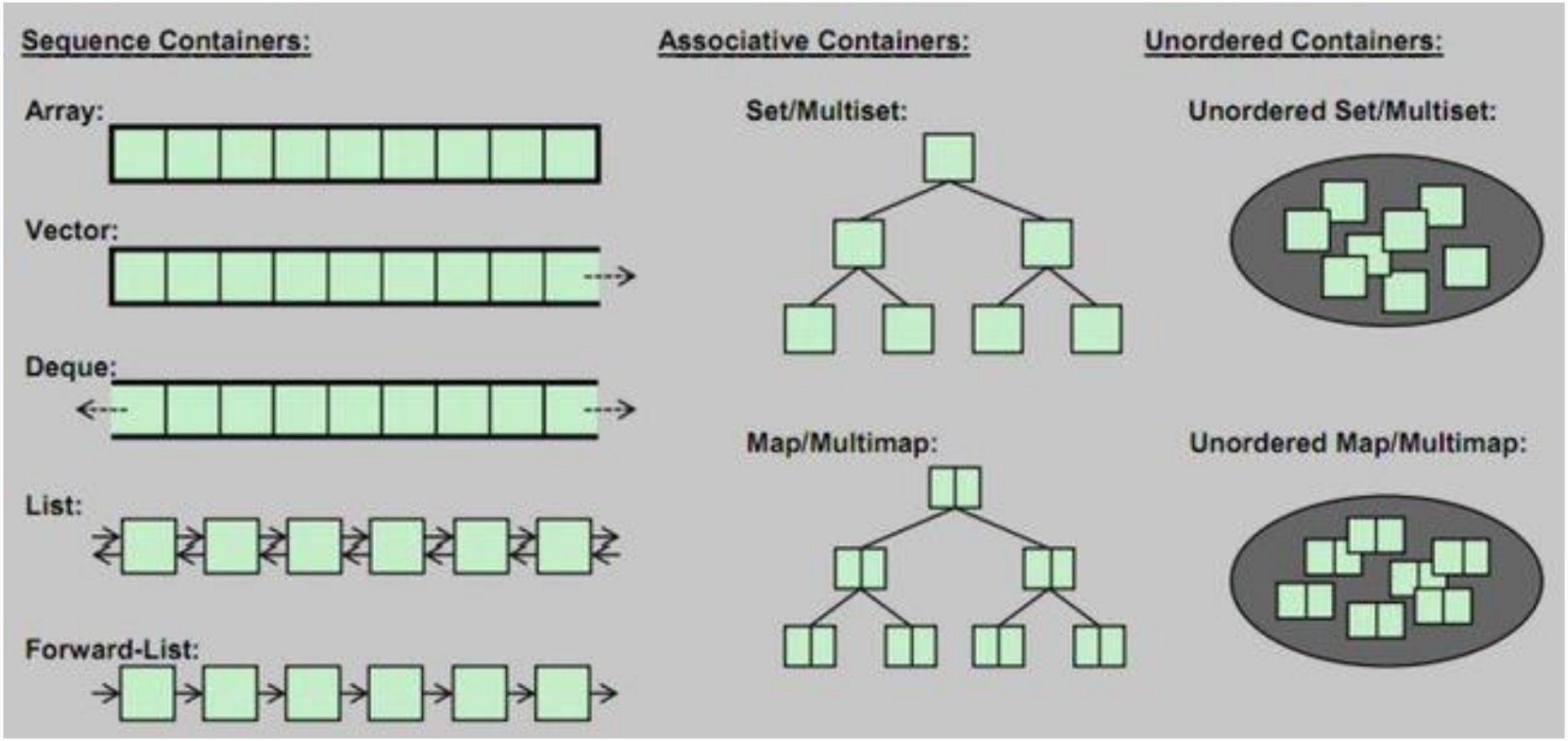
STL Container Categories:

- The C++ container library categorizes containers into four classes:

No	Containers class	Descriptions	Examples
1	*Sequence containers*	– represent linear data structures	– vector, array – deque, list
2	*Container adapters*	– A special type of container class – not full container classes by their own and typically wrap around *sequence containers* – *do not support iterators*	– stack – queue
3	*Associative containers*	– *nonlinear containers* – *typically provide a fast lookup (search) elements using*	– set, multiset – map, multimap
4	*Unordered associative containers*	– implement unordered data structures that can be quickly searched	– undered_set, – unordered_map

STL: Containers (cont'd)

STL Container Categories:



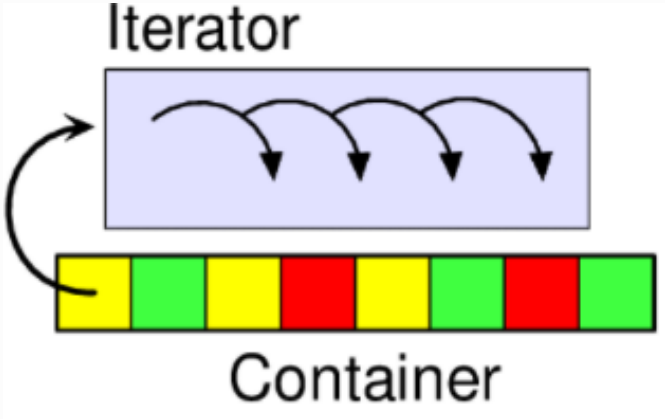
STL: Containers (cont'd)

Containers descriptions:

No	Containers	Descriptions
1	list	➢ Bidirectional/Doubly linked list ➢ Best for rapid insertion and deletion anywhere.
2	vector	➢ "Array" that grows automatically, ➢ Best for rapid insertion and deletion at back. ➢ Support direct access to any element via operator "[]".
3	deque	➢ "Array" that grows automatically. ➢ Best for rapid insertion and deletion at front and back.
4	set and multiset	➢ Set doesn't duplicate element. ➢ Multiset is a set that allows duplicate elements ➢ Elements are automatically sorted. ➢ Best for rapid lookup (searching) of element.
5	map and multimap	➢ Map is collection of (key, value) pairs with non-duplicate key ➢ Multimap is a set that allows duplicate elements ➢ Elements are automatically sorted by key. ➢ Best for rapid lookup of key.

STL: Iterators

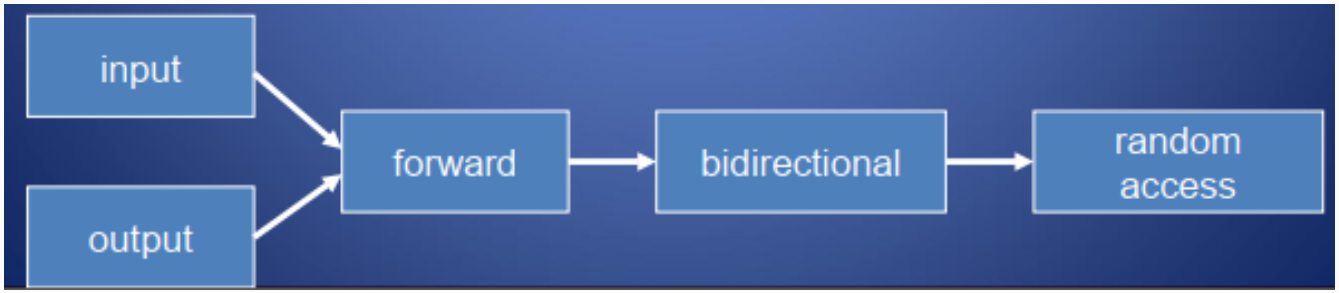
- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often used to ***move sequentially*** from element to element, a process called *iterating* through a container



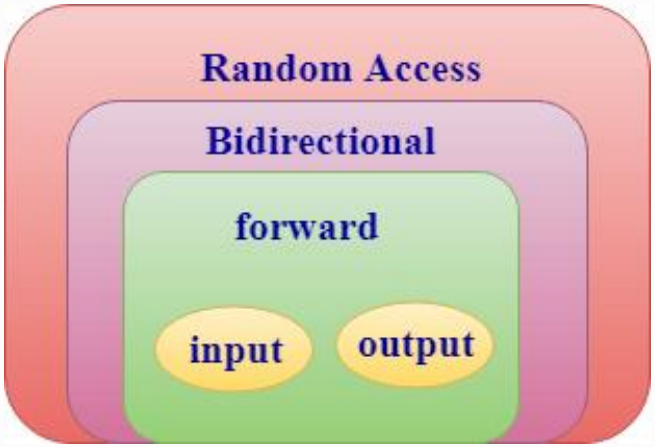
- The **dereferencing operator (*)** dereferences an iterator where **assignment operator (=)** used to assign data.
- The **++/-- operators** moves the iterator to the next item or previous element
- Whereas the **== and != operators** compare two iterators for equality and inequality respectively
- The **begin() and end() method** returns an iterator pointing to the first and the last element of the container

STL: Iterators (cont'd)

- Iterators are divided into five categories



- The higher (more specific) category always subsumes (includes) a lower (more general) category.
- E.g. An algorithm that accepts a forward iterator will also work with a bidirectional iterator and a random access iterator.



STL: Iterators (cont'd)

Iterators and their Characteristics

Iterator	Descriptions	Operators
Input	Used to access the elements from the container, but it does not modify the value of a container.	++, ==, !=, *
Output	Used to modify the value of a container, but it does not read the value from a container	++, =
Forward	used to both read and write to a container.	++, =, !=
Bidirectional	Supports all the features of a forward iterator plus it allows move backward by decrementing an iterator.	++, --, =, !=
Random	Provides random access of an element at an arbitrary location. It has all the features of a bidirectional iterator plus it supports pointer addition and pointer subtraction to provide random access to an element.	++, --, =, !=, +, -

STL: Iterators (cont'd)

Iterators and their Characteristics (cont'd)

Iterator	Access method	Direction of movement	I/O capability
Input	Linear	Forward only	Read-only
Output	Linear	Forward only	Write-only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward & backward	Read/Write
Random	Random	Forward & backward	Read/Write

Note:

- *The "=" operator either used as an assignment operator or equal operator(=)*
- *Every algorithm requires an iterator with a certain level of capability for example to use the [] operator you need a random access iterator*
- *However, not every iterator can be used with every container for example the list class provides no random access iterator*

Containers and corresponding Iterators

- Vector Container
- Set Container
- Map Container

Vector Containers

- Vectors is a sequence container class that implements dynamic arrays capable of *resizing itself automatically* .
- The resizing occurs after an element has been added or deleted from the vector.
- Vectors are not ordered in C++
- Storage is handled automatically by the container.
- Like array a vector stores the elements in contiguous memory locations and allocates the memory as needed at run time
- The elements can be easily accessed and traversed across using iterators.
- **Declaration Syntax:**

```
vector< object_type > vector_variable_name;
```

Vector Containers (cont'd)

Some Iterators of Vector

Function	Description
push_back() and pop_back()	adds a new element at the end and removes a last element from the vector respectively
empty()	determines whether the vector is empty or not.
insert()	inserts new element at the specified position.
resize()	modifies the size of the vector.
clear()	removes all the elements from the vector.
size() and capacity()	determines a number of elements in the vector and the current capacity of the vector respectively
end()	refers to the past-last-element in the vector.
begin()	points the first element of the vector.
max_size()	determines the maximum size that vector can hold.
cend() and cbegin()	refers to the past-last and the firs element in the vector respectively
crbegin() and crend()	It refers to the last character of the vector and element preceding the first element of the vector respectively

Vector Containers (cont'd)

Example: Demonstration of vector and its iterators

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    //vector declaration and initailization
    vector<int> nums = {45, 85, 96, 23, 25};

    //print the size of the vector and capacity
    cout<<"Initial Size of Vector: "<<nums.size()<<endl;
    cout<<"Capacity of Vector: "<<nums.capacity()<<endl;

    //print vector elements using iterators
    cout << "\nOutput from begin and end: ";
    vector <int>:: iterator a;
    for (a = nums.begin(); a != nums.end(); ++a)
        cout << *a << " ";
```

Vector Containers (cont'd)

Example ….

```
cout << "\nOutput from cbegin and cend: ";
for (auto a = nums.cbegin(); a != nums.cend(); ++a)
        cout << *a << " ";

//remove element from the end of the vector
nums.pop_back();
cout<<"\nSize of Vector: "<<nums.size()<<endl;

cout << "\nVector contents After deletion: ";
for (int a = 0; a < nums.size(); a++)
        cout << nums[a] << " ";

//erase the entire vector elements
nums.clear();
cout << "\nSize after clear(): " << nums.size();
```

Vector Containers (cont'd)

Example ….

```
// checks if the vector is empty or not
   if (nums.empty() == false)
      cout << "\nVector is not empty";
   else
      cout << "\nVector is empty";
```

```
vector<int> items;
//print the size of the vector and capacity after insertion
cout<<"\n\n Second Vecor:\n";
cout<<"Size of Vector: "<<items.size()<<endl;
cout<<"Capacity of Vector: "<<items.capacity()<<endl;

//Pushing the values one-by-one in vector using push_back():
for (int a = 1; a <= 5; a++)
        items.push_back(a);
```

Vector Containers (cont'd)

Example ….

```
//Printing the output of vector 'a' using iterators rbegin() and rend()
   cout << "\nOutput of rbegin and rend Function: ";
   for (auto ir = items.rbegin(); ir != items.rend(); ++ir)
      cout << *ir << " ";

   //insert valeu at specified position of vector
         items.insert(items.begin(), 7);
         items.insert(items.begin()+2, 9);
         items.insert(items.end()-1, 15);

   //Printing the output of vector 'a' using iterators crbegin() and crend()
   cout << "\nOutput of crbegin and crend Function: ";
   for (auto ir = items.cbegin(); ir != items.cend(); ++ir)
      cout << *ir << " ";
```

Vector Containers (cont'd)

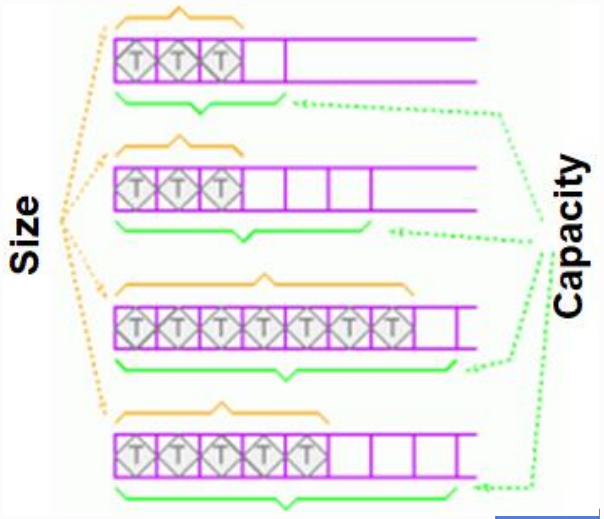
Example ….

```
// resizing  the vector 'a' to  size  4
   items.resize(4);

//Size and capacity of vector after resizing
            cout<<"\nSize of Vector: "<<items.size()<<endl;
            cout<<"Capacity of Vector: "<<items.capacity()<<endl;
            cout << "\nOutput of After resizing : ";

   for (auto ir = items.cbegin(); ir != items.cend(); ++ir)
      cout << *ir << " ";

            return 0;
}
```



Set Containers

- Set is a C++ STL container used to store the **unique elements**
- All the elements are stored in a **sorted** manner.
- Once the value is stored in the set, it cannot be modified within the set.
- However, the elements can be removed and then modified value of the element can be added.
- Sets are **traversed** using the iterators.
- Require to include the two main header files to work with the sets

#include< set > and #include< iterator >

- Instead of all the other header files, you can use the header file that contains all the header files.

#include< bits/stdc++.h >

Set Containers (cont'd)

- **Syntax:**

  set <*key_type*> s;

  where **key_type** is the data types of the key/element

- By default, elements are stored in ***ascending sorted*** order.
- Declaration syntax of the set to store the elements in **decreasing sorted order**.

  set < *key_type*, greater< *key_type* >> s;

- Use **set** when you want a **sorted collection** and you do not need **random access** to its elements**.**
- Duplicates are ignored when elements are inserted

Set Containers (cont'd)

Some Iterators of Set

Function	Description
insert()	Inserts a new element to the set.
begin()	points the first element of the set.
end()	points to the theoretical element that follows last element in the set.
empty()	determines whether the set is empty or not.
erase()	used to delete the elements from the set
find)	Returns an iterator based on the position of the element if it is found, else return iterator at the end
size()	determines a number of elements in the set
clear()	removes all the elements from the vector.
lower_bound(X)	Returns an iterator to upper bound.
upper_bound(X)	Returns an iterator to lower bound
cend() and cbegin()	refers to the past-last and the firs element in the set respectively
crbegin() and crend()	It refers to the last character of the set and element preceding the first element of the set respectively

Set Containers (cont'd)

Example: Demonstration of set and its iterators

```
#include <iostream>
#include <iterator>
#include <set>
#include <algorithm>

using namespace std;

int main(){
        // empty set container
        set<int, greater<int> > s1;

        // insert elements in random order
        s1.insert(40);      s1.insert(30);
        s1.insert(70);      s1.insert(40);
        s1.insert(50);      s1.insert(20);
```

Instead this header file can be used

#include< bits/stdc++.h >

Set Containers (cont'd)

Example ….

```
// printing set s1
set<int, greater<int> >::iterator itr;
cout << "\nThe set s1 is : \n";
for (itr = s1.begin(); itr != s1.end(); itr++)
        cout << *itr<<" ";
cout << endl;

// trying to add duplicate key/element only one value will be added
        s1.insert(50);          s1.insert(10);

// assigning the elements from s1 (descending set) to s2 (ascending)
        set<int> s2(s1.begin(), s1.end());

        // print all elements of the set s2
        cout << "\nThe set s2 after assign from s1 is : \n";
        for (itr = s2.begin(); itr != s2.end(); itr++)
                cout << *itr<<" ";
        cout << endl;
```

Set Containers (cont'd)

Example ….

```
// remove all elements up to 30 in s2
cout<< "\ns2 after removal of elements less than 30 :\n";
s2.erase(s2.begin(), s2.find(30));

for (itr = s2.begin(); itr != s2.end(); itr++)
        cout <<*itr<<" ";

// remove element with value 50 in s2
int num = s2.erase(50);
cout << "\ns2.erase(50) : “<<num << " removed\n";
for (itr = s2.begin(); itr != s2.end(); itr++)
        cout <<*itr<<" ";
cout << endl;

// lower bound and upper bound for set s1
cout << "s1.lower_bound(40) : "<<*s1.lower_bound(40)<< endl;
cout << "s1.upper_bound(40) : "<<*s1.upper_bound(40)<< endl;
```

Set Containers (cont'd)

Example ….

```
//initailize the set at the time of declaration
set<string> fruit {  "orange", "apple", "mango", "peach", "grape"   };
cout << "Size of set container fruit is : " << fruit.size();

//declare multiple set at the same time
set<int> sample1, sample2, sample3;

// set initialization
sample1 = { 1, 2, 3, 4, 5 };          sample2 = { 6, 7, 8, 1 };

// Merge two sets using merge() algorithm and move the result to sample3
merge(sample1.begin(), sample1.end(),  sample2.begin(), sample2.end(),
        inserter(sample3, sample3.begin()));

// copy assignment
sample1 = sample3;
```

Set Containers (cont'd)

Example ….

```
// Print the sets
for (auto it = sample1.begin(); it != sample1.end(); ++it)
        cout << *it << " ";
cout << endl;

// iterator pointing to position where 5 is
auto pos = sample3.find(5);

// prints the set elements
cout << "The set elements after 5 are: ";
for (auto it = pos; it != sample3.end(); it++)
        cout << *it << " ";
cout << endl;

return 0;
}
```

Map Containers

- **Map** is an associative container that store the elements as a combination of key-value pairs (like dictionary).
- Each key is **unique** and it can be inserted or deleted but cannot be altered.
- However, the values associated with keys can be changed.
- By default keys are in ascending order
- **Syntax**:

  map < *key_datatype*, *value_datatype* > *map_name*;

  Here,

  - *key_datatype = datatype of key*
  - *value_datatype = datatypes of value corresponding to key*
  - *map_name = Name of the **map***

Map Containers (cont'd)

Some Iterators of Map

Function	Description
insert()	Insert elements with a particular key in the map container
clear()	removes all the elements from the vector.
begin()	points the first element of the map
end()	points to the theoretical element that follows last element in the map.
operator[]	reference the element present at position given inside the operator
size()	determines a number of elements in the map
empty()	determines whether the vector is empty or not.
max_size()	determines the maximum size that map can hold.
find(loc)	it points to the element if element found at loc.
erase (iterator loc)	it removes the element which is specified at a location by the iterator
upper_bound()	Return iterator to the upper bound
lower_bound()	Return iterator to the lower bound

Map Containers (cont'd)

Example: Demonstration of map and its iterators

```
#include <iostream>
#include <iterator>
#include <map>

using namespace std;

int main(){

    map<int, float> mark; // empty map container

    // insert elements in random order using array index notation
    mark.insert(pair<int, int>(1, 40.5));
    mark.insert(pair<int, int>(2, 73.75));
    mark.insert(pair<int, int>(3, 60.00));
    mark.insert(pair<int, int>(4, 80.7));
    mark.insert(pair<int, int>(5, 50.5));
```

Map Containers (cont'd)

Example ….

```
// printing map mark
map<int, float>::iterator itr;
cout << "\nThe map MARK is : \n";
cout << "\tKEY\tELEMENT\n";
for (itr = mark.begin(); itr != mark.end(); ++itr)
        cout<<'\t'<<itr->first<<'\t'<<itr->second <<endl;

// assigning the elements from mark to result
map<int, float> result(mark.begin(), mark.end());

// print all elements of the map gquiz2
cout << "\n\nThe map RESULT after assign from MARK is : \n";
cout << "\tKEY\tELEMENT\n";
for (itr = result.begin(); itr != result.end(); ++itr) {
        cout<<'\t'<< itr->first<<'\t'<< itr->second<<endl;
}
```

Map Containers (cont'd)

Example ….

```
// remove all elements up to with key=3 in result
cout << "\nRESULT after removal of elements less than key=3 : \n";
cout << "\tKEY\tELEMENT\n";
result.erase(result.begin(), result.find(3));
for (itr = result.begin(); itr != result.end(); ++itr) {
        cout<<'\t'<< itr->first<<'\t'<< itr->second<<endl;

}

// remove all elements with key = 4
int num = result.erase(4);
cout << "\n result.erase(4) : "<< num << " removed \n";
cout << "\tKEY\tELEMENT\n";
for (itr = result.begin(); itr != result.end(); ++itr) {
        cout<<'\t'<< itr->first<<'\t'<< itr->second<<endl;
}
```

Map Containers (cont'd)

Example ….

```
// lower bound and upper bound for map mark key = 5
cout <<"\n mark.lower_bound(5) : \n  KEY \t ELEMENT \n";
cout<<"    "<<mark.lower_bound(5)->first<<'\t';
cout<<"    "<<mark.lower_bound(5)->second<<endl;

cout <<"\n mark.upper_bound(5) : \n KEY \t ELEMENT \n";
cout<<"    "<<mark.upper_bound(5)->first<<'\t';
cout<<"    "<<mark.upper_bound(5)->second<<endl;

//initialize map
map<int, string> Employees = {{101, "Mulalem"}, {105, "John"},
                              {103, "Daniel"}, {104, "Kebede"},
                              {102, "Aman"}};

   //print map element using operator []
   cout<<"Employees[102] = "<<Employees[102]<<endl;
   cout<<"Employees[104] = "<<Employees[104]<<endl;
   cout<<"Map size: "<< Employees.size()<< endl;
```

Map Containers (cont'd)

Example ….

```
    //print map in default order
    cout<<"\nNatural Order:" << endl;
    for(map<int,string>::iterator ii =Employees.begin(); ii!=Employees.end(); ++ii)
      cout << (*ii).first << ": " << (*ii).second << endl;

//print map in reverse order
  cout <<"\nReverse Order:" << endl;
   map<int,string>::reverse_iterator rii;
   for(map<int,string>::iterator rii = Employees.rbegin(); rii!=Employees.rend(); ++rii)
      cout << (*rii).first << ": " << (*rii).second << endl;

   auto it = Employees.find(103);       //find map element
    cout<<"Iterator points to "<<it->first<<" = "<<it->second<<endl;

    //clear a map and checks if the map is empty or not
    Employees.clear();
    if (Employees.empty())        cout << "\nVector is empty"<<endl;
    else        cout << "\nVector is not empty"<<endl;
      return 0;
}
```

Summary (1/2)

- Major theme in development of programming languages involves code reusability and avoid repeatedly reinventing the wheel
- Use of generic codes is one of the trends that contribute to the principles of code reuse.
- Generic programming allows for the abstraction of types
- C++ supports a powerful feature known as a template to implement the concept of generic programming.
- A template allows us to create a family of classes or family of functions to handle different data types.
- C++ has function templates and class templates
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.

Summary (2/2)

- We can also use non-type arguments such as built-in or derived data types as template arguments.
- STL is a set of general-purpose classes and functions which are mainly used for storing and processing data.
- STL can be defined as a library of container classes, algorithms, and iterators.
- The main idea behind STL is to reuse codes already written and tested. It saves time and effort.
- **Vector** is a dynamic array capable of automatically resizing itself when an element is added or deleted from it.
- The elements of a vector are stored in contiguous storage in order to be accessed then traversed using iterators.
- **Set** is a STL container used to store the unique elements, and all the elements are stored in a sorted manner.
- **Map** is an associative container that store the elements as a combination of key-value pairs (like dictionary). By default keys are in ascending order

Practical Exercise (1/2)

1. Create a class Point which has a template parameter of the type of internal data, T, and a template parameter for the dimension of the vector, n. Store a statically allocated, internal array of type T with dimension n.
2. Create a template function which computes the Euclidean distance between 2 points. Also instantiate two points (a) type <double> (b) type <int> and compute their distance.
3. Write a function template palindrome that takes a vector parameter and returns true or false according to whether the vector does or does not read the same forward as backward (e.g., a vector containing 1, 2, 3, 2, 1 is a palindrome, but a vector containing 1, 2, 3, 4 is not).
4. Write a function called floatingPointDivide which accepts two parameters, and will always return the double value result of the first parameter divided by the second parameter. This function should work on all other number types. Write this function as a template and ensure that the division returns a floating point value without using a cast

Practical Exercise (2/2)

5. Write a template function called **findLargestElement** that accepts a vector as a parameter and returns the largest element in that vector. Write a driver program to test your function.
6. Write a program that uses the map template class to compute a histogram of positive numbers entered by the user. The map's key should be the number that is entered, and the value should be a counter of the number of times the key has been entered so far. Use -1 as a sentinel value to signal the end of user input. For example, if the user inputs: 5, 12, 3, 5, 5, 3, 21, -1, then the program should output the following (not necessarily in this order):

number	occurs
3	2
5	3
12	1
21	1

Summary Questions

1. Compare and discuss the advantages of function templates over
    a) *function overloading*
    b) *Function macros*
2. What is friend function? Explain providing an example.
3. Discuss the benefit of using STL.
4. Compare and contrast STL Iterator and smart pointers.
5. Explain the cons and pros of the following
    1. *Templates*
    2. *STL containers*
    3. *STL Iterators*
6. Demonstrate Template *Specialization and Variadic* Templates.
7. Demonstrate how non-type template parameter is used in class and function templates

Reading Assignment

- ***Exception Handling***
- ***Competitive Programming*** ***(very useful for programming consent)***

Reading Resources/Materials

Chapter 8, 17 & 18:

- ✓ Walter Savitch; Problem Solving With C++ [10th edition, University of California, San Diego, 2018

Chapter 15 & 16:

- ✓ P. Deitel , H. Deitel; C++ how to program, 10th edition, Global Edition (2017)

Chapter 16 & 21:

- ✓ Herbert Schildt; C++ from the Ground Up (3rd Edition), 2003 Graw-Hill, California 94710 U.S.A.

Chapter 30 - 33:

- ✓ Bjarne Stroustrup;The C++ Programming Language [4th Edition], Pearson Education, Inc , 2013

Thank You
For Your Attention!!