OBJECT ORIENTED PROGRAMMING IN C++ [UNIT-IV]

Learning Objectives

- Generic Programming
- Types of templates
- Class template
- Multiple parameters in class templates
- Function template
- Overloading of template functions
- Non type template arguments

What is generic programming

- It is an approach where generic types are used as parameters in algorithms so that they work for a variety of data types and DS.
- It eliminates code duplication and makes prog development easy and manageable.(advantage)

Templates

- Feature of C++ is templates which helps to eliminate redundant coding.
- Templates support generic programming, which allows you to develop reusable software components such as functions,classes,etc.,supporting different data types in a single framework.
- A template declared for functions are called function templates and those declared for classes are called class templates.
- Eg:- A function template for swap can be used to swap values for different data types.

Templates

- A template is one of the features added to C++ recently. It is a new concept which enables the programmer to define generic classes and functions; thus provides support for generic programming.
- Generic programming is an approach, where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
- A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array.
- Similarly, we can define a template for a function, say multiply(), that would help us create various versions of multiply() for multiplying int, float and double type values.

Templates

- A template can be considered as a kind of macro.
- When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function. Therefore the templates are sometimes called parameterized classes or functions.

Contd…

Syntax:

```
template <class T>
return-type T function-name( arguments of type T)
{
        // Body of function with type T
}
```

Consider a vector class defined as follows:

```
class vector
{
        private:
                int *v;
                int size;
        public :
                vector(int m)
                {
                        v = new  int[size = m];
                        for(int i=0; i<size; i++)
                        v[i] = 0;
                }
                vector( int *a, int size )
                {
                        for(int i=0; i<size; i++)
                        v[i] = a[i];
                }
                int operator*(vector &y)
                {
                        int sum = 0;
                        for(int i=0; i<size; i++)
                        sum += this->v[i] * y->v[i];
                        return sum;
                }
}
```


Contd…

Now suppose we want to define a vector that can store an array of float values. We can do this by simply replacing the appropriate int declarations with float in the vector class. This means that we have to redefine the entire class all over again.

Assume that, we want to define a vector class with the data type as a parameter and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal:

Template definition of vector class is as shown below:

```
template<class T>
class vector
{
        private:
                T *v;
                int size;
        public :
                vector(int m)
                {
                        v = new  T[size=m];
                        for(int i=0; i<size; i++)
                        v[i] = 0;
                }
                vector(T *a, int size)
                {
                        for(int i=0; i<size; i++)
                        v[i] = a[i];
                }
                T operator*(vector &y)
                {
                        T sum= 0;
                        for(int i=0; i<size; i++)
                        sum += this->v[i] * y->v[i];
                        return sum;
                }
}
```


Contd…

The class template definition is very similar to an ordinary class definition except the prefix **template <class T>** and the use of type **T**. This prefix tells the compiler that we are going to declare a template and use T as a type name in the declaration. Thus, vector has become a parameterized class with the type **T** as its parameter. **T** may be substituted by any other data type including the user-defined types. Now, we can create vectors for holding different data types.

A class created from a class template is called a template class. The syntax for defining an object of a template class is as follows:

```
classname<type> object-name( argument-list );
```

This process of creating a specific class from a class template is called instantiation. The compiler will perform the error analysis only when an instantiation takes place. It is, therefore, advisable to create and debug an ordinary class before converting it into a template.

Example: Program which illustrates the working of templates.

Contd…

```
#include<iostream.h>

template<class T>
T sum( T a, T b )
{
        T c = a + b ;
        return c ;
}

void main()
{
        int x = 10, y = 5 ;
        int z = sum(x, y);
        cout << "\n The sum of two int numbers is: " << z ;

        float f1 = 3.5, f2 = 7.5 ;
        float f = sum(f1, f2);
        cout << "\n The sum of two float numbers is: " << f ;

        char c1 = 'a', c2 = 'b';
        char c = sum(c1, c2);
        cout << "\n The sum of two char variables is: " << c
}
```


Contd…

Output:

```
(Inactive C:\TCWIN45\BIN\NONAME02.EXE)
The sum of two int numbers is: 15
The sum of two float numbers is: 11
The sum of two char variables is: A
```


WITHOUT USING TEMPLATES

```
int sum(int x,int y)
      {
        int z;
        z=x+y;
        return z;
     }

float sum(float x,float y)
       {
         float z;
         z=x+y;
         return z;
       }
```

```
void main()
     {
       sum(3,5);
       sum(6.9,9.0);
     }
```


USING TEMPLATES

```
template <class T>

T sum(T x,T y)
  {
    T s;
    s=x+y;
    return T;
  }
void main()
     {
       sum(2,3);
       sum(5.7,8.9);
     }
```


Function templates

- Allows a single function to be operated upon multiple data types. syntax:

```
keyword          Template type
template<class T,…arguments>
Return type fun_name(arg)        → Atleast one argument of type template
{
        Body of function
}
```

The keyword 'class' above simply means that the identifier fun_name will stand for a datatype

Working of template functions

- Function called normally like any other function with parameters of any data type.
- When compiler encounters a call , it identifies the data types of the parameters and creates a function internally and makes a call to it.
- All future invocations to function template WITH THAT DATA TYPE refer to it.
- Internal function unknown to user.

The following program will illustrate the need for function templates

```
// mswap.cpp: Multiple swap functions
#include <iostream.h>
void swap( char & x, char & y )
{
   char t;  // temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
void swap( int & x, int  & y ) // by reference
{
   int t;   // temporary variable used in swapping

  t = x;
  x = y;
  y = t;
}
```



```
void swap( float & x, float & y ) // by reference
{
   float t; // temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   swap( ch1, ch2 ); // compiler invokes swap( char &a, char &b );
   cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );  // compiler invokes swap( int &a, int &b );
   cout << "On swapping <a, b>: " << a << " " << b << endl;
   float c, d;
   cout << "Enter two floats <c, d>: ";
   cin >> c >> d;
   swap( c, d );  // compiler invokes swap( float &a, float &b );
   cout << "On swapping <c, d>: " << c << " " << d;
}
```

Output:-

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```


Generic function for swapping

```
template <class T>
void swap( T & x, T & y ) // by reference
{
   T t;  // template type temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
```



```
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";

   cin >> ch1 >> ch2;
   swap( ch1, ch2 ); // compiler creates and calls swap( char &x, char &y );
   cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );  // compiler creates and calls swap( int &x, int &y );
   cout << "On swapping <a, b>: " << a << " " << b << endl;
   float c, d;
   cout << "Enter two floats <c, d>: ";
   cin >> c >> d;
   swap( c, d );  // compiler creates and calls swap( float &x, float &y );
   cout << "On swapping <c, d>: " << c << " " << d;
}
```

Output:-

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```


Usage of template argument

- Every template argument specified in template argument list must be used as generic data type for definition of formal parameters.

```
template < class T>
void test(int x)     //error: T not used as argument
{
   T temp;
   test statements;
}
```


- Use of partial no of generic data types or leaving only one argument unused in the argument list also results in an error.

```
template <class T, class U>
void test(T x)       //error: U is not used in argument
{
   U try;
   statements..;
}
```

- Not specifying any argument to be used as generic data types ,if one or more arguments are specified in the template argument list also leads to an error.
- Ex:

```
Template<class T>
T pop(void)              //error:T not used as an argument
{
  return --stack;
}
```


- All the formal parameters need not be of generic type, provided all the template arguments are specified.

Ex:

```
template <class T>
void test(T x, int z)       //no error
{
  function body
}
```


Function vs function template

- Function templates not suitable for handling all data types.
- Hence it is necessary in certain cases to overload function templates for specified data types.

Eg:

max(str1,str2)

Compares address of strings rather than strings.

- If a program has both function template and a normal function, compiler first selects normal function if it matches the requested data type otherwise creates a function using template function.

Bsearch using template function

```
#include<iostream.h>

template<class T>
int bsea(T data[],T ele, int low , int
    high)
{
    if(low>high)    return -1;
    int mid=(low+high)/2;
    if(ele>data[mid])
  return bsea(data,ele,low,mid-1);
else
return bsea(data,ele,mid+1,high);
return mid;
}
```

```
Main()
{
 int ele,size,num[25],index;
Cout<<"program to search integer
    elements…"<<endl;
 cout<<"how many elements?";
   cin>>size;
  cout<<"\nEnter sorted array:";
For(int i=0;i<size;i++)
    cin>>num[i];
Cout<<"Enter element to be searched";
Cin>>ele;
If(index=bsea[num,ele,0,size))==-1)
 cout<<"Element found at
    position:"<<index;
}
```


Output:-

```
Program to search integer elements...
How many elements ? 4
Enter the elements in ascending order for binary search...
1
4
6
8
Enter the element to be searched: 6
Element 6 found at position 2
```

```
// bsort.cpp: template functions for bubble-sort
#include <iostream.h>
enum boolean ( false, true );
template <class T>
void swap( T & x, T & y ) // by reference
{
   T t;    // template type temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
template< class T >
void BubbleSort( T & SortData, int Size )
{
   boolean swapped = true;
   for( int i = 0; ( i < Size - 1 ) && swapped; i++ )
   {
      swapped = false;
      for( int j = 0; j < ( Size - 1 ) - i; j++ )
         if( SortData[ j ] > SortData[ j + 1 ] )
         {
            swapped = true;
            swap( SortData[ j ], SortData[ j + 1 ] );
         }
   }
}
```

```
void main( void )
{
   int IntNums[25];
   float FloatNums[25];
   int i, size;
   cout << "Program to sort elements..." << endl;
   // Integer numbers sorting
   cout << "Enter the size of the integer vector <max-25>: ";
   cin >> size;
   cout << "Enter the elements of the integer vector..." << endl;
   for( i = 0; i < size; i++ )
      cin >> IntNums[i];
   BubbleSort( IntNums, size );
   cout << "Sorted Vector:" << endl;
   for( i = 0; i < size; i++ )
      cout << IntNums[i] << "  ";
   // Floating point numbers sorting
   cout << "\nEnter the size of the float vector <max-25>: ";
   cin >> size;
   cout << "Enter the elements of the float vector..." << endl;
   for( i = 0; i < size; i++ )
      cin >> FloatNums[i];
   BubbleSort( FloatNums, size );
   cout << "Sorted Vector:" << endl;
   for( i = 0; i < size; i++ )
      cout << FloatNums[i] << "  ";
}
```

Function Templates with multiple parameters

We can use more than one generic data type in the template statement, using a comma-separated list as shown below:

```
template < class T1, class T2 >
return-type function-name(argument of types T1, T2,...)
{
        //body of the function
}
```

Example:

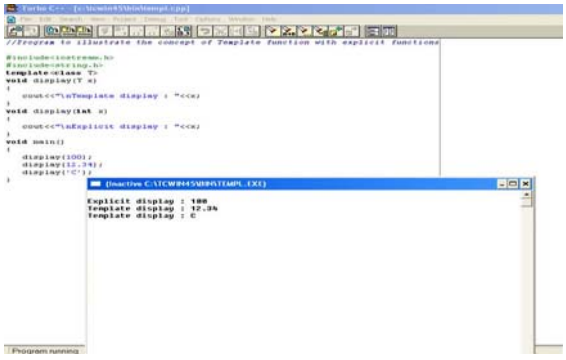
```
//Program to illustrate the concept of using two generic types in template functions.
#include<iostream.h>
#include<string.h>
template<class T1, class T2>
void display(T1 x, T2 y)
{
   cout<<x<<"  "<<y<<"\n";
}
void main()
{
   display(1999,"EBG");
   display(12.5A,"123A");
}
```

```
(Inactive C:\TCWIN45\BIN\TEMPLA.EXE)
1999  EBG
12.5A  123A
```

Function Template Overloading

- A template function may be overloaded either by template functions or ordinary functions. In such cases, the overloading resolution is accomplished using following steps:
-
- Call an ordinary function that has an exact match.
- If a match for an ordinary function is not found, call a template function that could be created with an exact match.
- An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. The program shows how a template function is overloaded with an explicit function.

Contd…



Member Function Template

When we created a class template for vector, in the previous tutorial, all the member functions were defined as inline which was not necessary. We could have defined them outside the class as well. But remember that the member functions of the template classes themselves are parameterized by the type argument (to their template classes) and therefore these functions must be defined by the function templates. It takes the following general form:

```
template<class T>
return-type class name <T> :: function-name(arguments)
{
        // function body
}
```

Contd…

Example:

```
#include<iostream.h>

template<class T>
class vector
{
        private:
                T *v;
                int size;

        public:
                vector(int m);
                vector(T *a, int size);
                T operator*(vector &y);
}

template<class T>
vector<T>::vector(int m)
{
        v=new  T [size = m];
        for(int i=0;i<size;i++)
        v[i] = 0;
}

template<class T>
vector<T>::vector(T *a, int size)
{
        for(int i=0;i<size;i++)
        v[i] = a[i];
}

template<class T>
T vector<T>::operator*(vector &y)
{
        T sum = 0;
        for(int i=0;i<size;i++)
        sum+= this->v[i] * y->v[i];
        return sum;
}

void main()
{
        int x[3] = {1,2,3};
        int y[3] = {4,5,6};
        vector<int> v1;
        vector<int> v2;
        v1=x;
        v2=y;
        int R=v1*v2;
        cout<<"R = "<<R;
}
```


Non-Type Template Arguments

We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument T, we can also use other arguments such as strings, functions, constant expressions, and built-in types.

Consider the following example:

```
template <class T, int size>
class array
{
        T a[size];
        ----------
}
```

This template supplies the size of the array as an argument. This implies that the size of the array is known to the compiler at the compile time itself. The arguments must be specified whenever a template class is created,

Example:

```
array <int, 10> a1 //array of 10 integers
array <float, 5> a2 //array of 5 floats
array <char, 20> a3 // string of size 20
```


CLASS TEMPLATES: using implementation of generic stack

Why? How?

- Writing the same search, sort, or other functionality for all the different data types is tedious and error prone.
- Why not use a solution that allows you to write a function once and then use it for any kind of data?
- In C++ generic programming is implemented with "templates"

WHAT ARE TEMPLATES

- **Templates** are a feature of the C++ programming language that allow functions and classes to operate with generic types.
- This allows a function or class to work on many different data types without being rewritten for each one.

USES

1) Templates allow functions and classes to be ***parameterized*** so that the ***type of data being operated upon (or stored) is received via a parameter*.**
2) 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***.
3) The template mechanism is important and powerful. It is used throughout the Standard Template Library (STL) to achieve ***genericity***.

Class Templates

- Much as function templates allow argument types to be parameterized, class templates allow us to parameterize the types of:
  - member variables
  - arguments of member functions & constructors
  - return values of member functions

What's all this?

- Conceptually, templates are a lot like macros, just without the unpleasant side effects.
  - When you see a template, or use a template, think "text substitution" and you'll be close

Template Instantiation

- A template definition is a recipe that the compiler can use to generate a function (or a class, more on that later)
- The compiler will not use this recipe unless/until you instantiate the template.
  - At that point, the compiler goes and performs the text substitution you asked for, and then compiles the newly generated function as if you'd written that function yourself.
- What happens if I instantiate the same template multiple different ways?
  - Well, with function overloading, we just get two or more functions with the same name, but with different arguments!

Example (template definition)

```
template <class T>
T findMax(T x, T y) {
   if (x < y) { return y; }
   else { return x; }
}
```

- "T" is the template parameter. Since the "T" is specified as a "typename" (i.e., the name of some type), then "T" can be replaced by any type (e.g., "int" or "string"). T can NOT be replaced by any arbitrary text, just by a type.
- We have "defined" this template, which means the compiler now knows the recipe. But there is no machine code for the max function yet. The compiler won't actually compile the max function until we instantiate it.
  - The compiler does do some preliminary syntax checking, so you can get compiler errors in your template definitions even if you don't instantiate them.

Simple Template Functions

```
template <class T>
T findMax(T x, T y) {
   if (x > y) { return x; }
   else { return y; }
}
int main(void) {
  int x = 0;
  cout << findMax(x, ++x); // prints 1
  cout << x; // prints 1
  string s = "hello";
  string t = "world";
  cout << findMax(s, t); // prints "world"
}
```


Template Classes

- C++ also provides template classes.
- Virtually any "data structure" (AKA "collection", AKA "container") will be implemented in C++ as a template
  - The type of data stored in the structure is really not at all relevant to the data structure itself.
- Template classes get "defined" and "instantiated" in analogous ways to template functions with the following caveats
  - The compiler will never guess at the template argument for a template class, you must always explicitly tell the compiler "what T is".
  - Classes cannot be "overloaded", but the compiler will permit you to instantiate the same template class in multiple ways.
    - ✓ Each distinct instantiation results in a completely distinct class! (with its own copy of the static data members, for example).
  - The member functions in a template class are template functions (oh, how confusing!)

Class Templates Syntax

- For template class declaration:

  template<**class** T> class class_name{ … };

- To create an object of template class

  Class_name <**type**> ob;

- For the implementation of the methods outside the

  template<**class** T> return_type
  className<T>::*methodName*(parameter-list){……}

- For the implementation of the constructors outside the class, the syntax is:

  template<**class** T>
  className<T>:: *className*(parameter-list){……}

- Template class functions
  - Declared normally, but preceded by `template<class T>`
    - ✓Generic data in class listed as type **T**
  - Binary scope resolution operator used
  - Template class function definition:

```
template<class T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

    - ✓Constructor definition - creates an array of type **T**

Diff b/w **class template** and **template class**

- Class template is a template used to generate template classes.

```
template <typename T>
class cTemplate {
 int member;
};
```

- Template class is an instance of a class template.

```
cTemplate<int> cti;
```

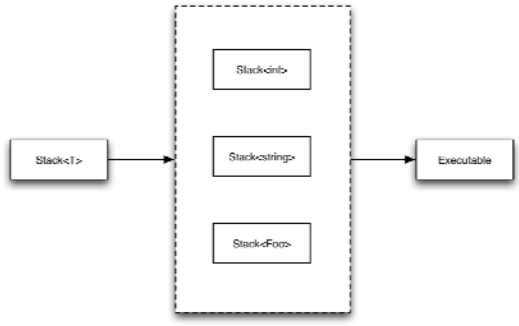
More than 1 Generic Data type

- For template with more than one generic data type.
- **template**<**class** T1,**class** T2> class class_name{ … };

Class_name<**datatype,datatype**> ob;

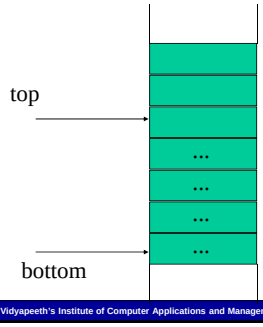
USING STACK EG.

Compile-time Code Generation



Example:

A Generic Stack



- Operations:
- pop
- push
- empty
- full
- initialize
- destroy

A Stack of Integers

```
class Stack {
        int* top, bottom;
        int size;
    public:
        Stack(int);
        bool push(int);
        bool pop(int&);
        bool empty();
        bool full();
        ~Stack();
};
```


A Class Template

```
typedef enum { false = 0, true = 1 } Bool;
template <class T>  class Stack {
        T  *top, *bottom;
        int size;
    public:
        Stack(int);
        bool  push(const T&);
        bool  pop(T&);
        bool empty();
        bool full();
        ~Stack():
};
```

Class implementations

```
template <class T>
Stack<T> :: Stack(int sz)
{
    size = sz;
    top = bottom = new T [sz];
}
```

```
template <class T>
bool Stack<T> :: empty()
{
        if (top == bottom)
                return true;
        return false;
}
```

```
template <class T>
bool Stack<T> :: push(const T& x)
{
        if (full()) return false;
        *top++ = x;
        return true;
}
```

```
template <class T>
bool Stack<T> :: pop(T& x)
{
        if (empty()) return false;
        x = *--top;
        retrun true;
}
```

 U4.58

Cont.

```
template <class T>
Stack<T> :: ~Stack()
{
        delete [] bottom;
}
```

```
template <class T>
bool Stack<T> :: full()
{
        if (top-bottom >= size)
                return true;
        return false;
}
```

 U4.59

Using Stack Template

```
Void main()
{
    Stack<int>    si(100);
    Stack<char>  sc(20);
    Stack<float>  sf(25);
}
```

 U4.60

Function Template Overloading

- You can define multiple functions *and* function templates with the same name
- The “best match” will be used
- You can also overload a function template by having a *different number of template parameters*


```
template<class T>
Void print( T data )
{
Cout<<data<<endl;
}

Template <class T>
Void print(T data, int nTimes)
{
For (int i=0;i<nTimes;i++)
Cout<<data<<endl;
}
Void main()
{
print(1);
Print(19.88);
Print(5,80);
Print(“mca 2nd sem “,11);
}
```


Advantages of templates

- One C++ Class Template can handle different types of parameters.
- Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

Conclusion

- **Templates allow us to generate a family of classes or a family of functions to handle different data types.**
- **It eliminates code duplication for different types and thus makes program development easier, faster and more manageable.**
- **We can use multiple parameters in both the templates.**
- **We may also use non-type parameters such as basic or derived data types as arguments templates.**

Summary

- C++ correspondence with OOPS does not limit to abstraction and encapsulation only, it also implements another crucial feature of inheritance, polymorphism and templates. All these features simulate their relations with real-world-models.
- While inheritance facilitates code reusability, polymorphism is the ability to access different implementation of the same function and operator during compile time and run time and template supports the concept of generic programming.

Standard Template Library (STL)

Learning Objectives

- **Software Evolution**
- **Standard Templates**
- **Standard Template Library**
- **Containers**
- **Types of Containers**
- **Algorithms**
- **Iterators**
- **Functors**

Introduction

- In order to help the C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett Packard developed a set of general-purpose templatized classes (data structures) and functions (algorithms) that could be used as a standard approach for storing and processing of data.
- The collection of these generic classes and functions is called the Standard Template Library (STL).
- Part of the ANSI standard C++ class library.

Contd…

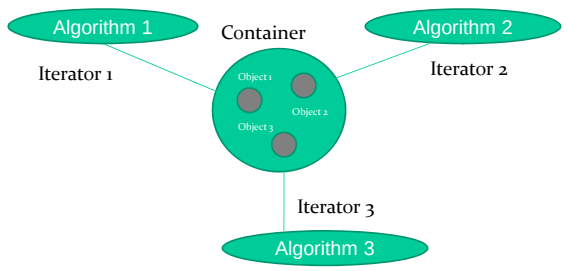
- Using STL save considerable time and effort, and lead to high quality programs.
- These benefits are possible because we basically "reusing" the well-written and well-tested components defined in the STL.
- STL components are defined in the namespace std. Therefore we use:

using namespace std;

Components of STL

- Containers
- Algorithms
- Iterators

Relationship between the three STL components



Contd…

- Algorithms employ iterators to perform operations stored in containers.
- Container: Object that actually stores data. Implemented by template classes.
- Algorithm: Procedure that is used to process the data contained in the containers. Implemented by template functions.
- Iterators: Object (like a pointer) that points to an element in a container. It can be incremented or decremented. It connects algorithms with containers.

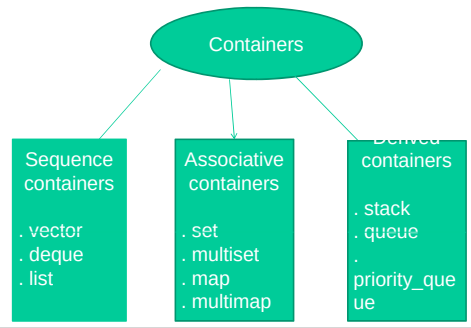
Containers

- Objects that hold data (of same type).
- 3 categories:
  - Sequence containers
  - Associative containers
  - Derived containers

Common container operations

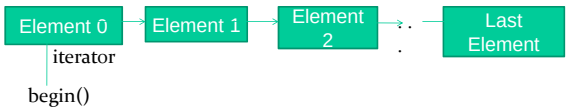
Some constructors, destructor
== , != , < , > , <= , >= , =
c.size( )
c.empty( )
swap(c1,c2)
c.begin( ) , c.rbegin( )
c.end( ) , c.rend( )
c.insert(pos,element)
c.earase(beg,end)
c.clear( )
c.max_size( )

Three major categories of containers



Sequence Containers

- Store elements in a linear sequence.
- Each element is related to other elements by its position along the line



Elements of sequence container

Contd…

- The STL provides three types of sequence containers:
  - vector
  - list
  - deque
- Elements in all these can be accessed using an iterator.

Contd…

Container	Description	Header file	Iterator
vector	A dynamic array. Allows insertion and deletions at back. Permits direct access to any element.	<vector>	Random access
list	A bidirectional, linear list. Allows insertions and deletions anywhere.	<list>	Bidirectional
deque	A double-ended queue. Allows insertions and deletions at both the ends. Permits direct access to any element.	<deque>	Random access

Comparison of sequence containers

Container	Random access	Insertion or deletion in the middle	Insertion or deletion at the ends
vector	Fast	Slow	Fast at back
list	Slow	Fast	Fast at front
deque	Fast	Slow	Fast at both the ends

Associative Containers

- Designed to support direct access to elements using keys.
- Not sequential
- 4 types:
  - set
  - multiset
  - map
  - multimap

Contd…

- All these containers store data in a structure called *tree* which facilitates fast searching, deletion and insertion.
- Very slow for random access and inefficient for sorting.

Contd…

Container	Description	Header file	Iterator
set	An associate container for storing unique sets. Allows rapid lookup. (No duplication allowed)	<set>	Bidirectional
multiset	An associate container for storing non unique sets. (Duplicates allowed)	<set>	Bidirectional
map	An associate container for storing key/value pairs. Each key is associated with only one value (one-to-one mapping). Allows key-based lookup.	<map>	Bidirectional
multimap	An associate container for storing key/value pairs in which one key may be associated with more than one value (one-to-many mapping). Allows key-based lookup.	<map>	Bidirectional

Derived Containers

3 derived containers:

stack

queue

priority_queue

Also known as *container adaptors.*

Created from different sequence containers.

Do not support iterators, therefore we cannot use them for data manipulation.

Support 2 member functions pop() and push() for implementing deleting and inserting operations.

Contd…

Container	Description	Header file	Iterator
stack	A standard stack. Last-in-first-out (LIFO).	<stack>	No iterator
queue	A standard queue. First-in-first-out (FIFO).	<queue>	No iterator
priority_queue	A priority queue. The first element out is always the highest priority element.	<queue>	No iterator

Algorithms

- Functions that can be used generally across a variety of containers for processing their contents.
- STL provides more than sixty standard algorithms.
- Permit us to work with two different types of containers at the same time.
- Standalone template functions.
- Reusability
- To access STL algorithms include <algorithm> in program.

Categories of Algorithms

STL algorithms, based on the nature of operations they perform, may be categorized as under:

Retrieve or non-mutating algorithms- Eg: count(), find(), equal() etc…

Mutating algorithms – Eg: remove(), replace(), swap() etc…

Sorting algorithms – Eg: binary_search(), sort_heap() etc…

Set algorithms – Eg: set_union, set_intersection etc…

Relational algorithms - Eg: max(), min() etc…

Numeric algorithms – Eg: inner_product(), partial_sum() etc… (include header file <numeric>)

Iterators

Behave like pointer

Used to access container elements

Used to traverse from one element to another – Iterating through the container

5 types:

Input

Output

Forward

Bidirectional

Random

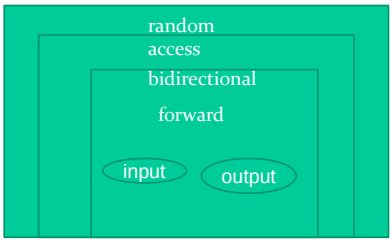
Iterators and their characteristics

Iterator	Access method	Direction of movement	I/O capability	Remark
Input	Linear	Forward only	Read only	Cannot be saved
Output	Linear	Forward only	Write only	Cannot be saved
Forward	Linear	Forward only	Read/Write	Can be saved
Bidirectional	Linear	Forward and backward	Read/Write	Can be saved
Random	Random	Forward and backward	Read/Write	Can be saved

Contd…

- Different types of iterators must be used with different types of containers.
- Only sequence and associative containers are traversable with iterators.
- The level of functionality provided by different categories of iterators can be represented by functionality Venn diagram of iterators.

Functionality Venn diagram of iterators



Contd…

Input and output iterators support the least functions. used only to traverse in a container.

Forward iterator support all operations of input and output iterators and retains its position in the container.

Bidirectional iterator, while supporting all forward iterator operations, provides the ability to move in the backward direction in the container.

Random access iterator combines the functionality of a bidirectional iterator with ability to jump to an arbitrary location.

Operations for Input iterators

*iter
iter -> member
++iter
iter++
iter1 == iter2
iter1 != iter2

You can read every thing just once and you can just read

Operations for output iterators

- *iter = value
- ++iter
- iter++

- You can write every thing just once and you can just write and nothing else

Operation of forward iterators

*iter
iter->member
++iter
iter++
iter1 == iter2
iter1 != iter2
iter1 = iter2

Operations for bidirectional iterators

--iter
iter--

Plus all operations of forward iterator

Operations of random access iterator

All operations of bidirectional iterators
iter[n]
iter += n
iter -= n
iter+n
n+iter
iter-n
iter1 – iter2
iter1 < iter2
iter1 > iter2
iter1 <= iter2
iter1 >= iter2

```
vector<int> coll;
vector<int>::iterator pos;
pos = coll.begin( );
```


Container Classes

Bitset	<bitset>
Deque	<deque>
List	<list>
Map	<map>
Multimap	<map>
Multiset	<set>
Priority queue	<queue>
Queue	<queue>
Set	<set>
Stack	<stack>
vector	<vector>

Theory Of Operations

- Elements can be added or removed in number of ways eg: insert(), erase() ,push_back() ,pop_back() ,(push_front(), pop_front() – lists and dequeues)
- Access : Through an iterator function begin() or end() , find()-associative container

Vector

- A vector models a dynamic array
- The elements always have a certain order
- Vectors provide random access
- Good performance for appending or deleting at the end but not middle
- Pointers are interpreted by insertion in middle

Vector operations

C.capacity( )
C.reserve(int)
C.at(idx)
C[idx]
C.front( )
C.back( )
C.insert(pos,elem) C.insert(pos,n,elem)
C.push_back(elem)
C.pop_back(elem)
C.earase(pos) C.earase(beg,end)
C.resize(num) C.resize(num,elem)
C.clear( )

Vectors

- Vector class supports dynamic array
- Template specification :

template<class T,class Allocator=allocator<T>>class vector

- < and == have to be defined (compiler based)
- Can operate with ==, <, <=, !=, >, >=, []
- Commonly used functions : size(), begin(), end(), push_back(), insert() and erase()

Declaration of a vector

- Vector<int> iv; //Zero Length
- Vector<char> cv(5); //5-element vector
- Vector<char> cv(5,'x'); //Initialization
- Vector<int> iv2(iv); //Create vector from vector


```
#include<iostream>
#include<vector>
int main()
{
  vector<char> v(10);
  vector<char> v2;
  char str[]="vector";
  unsigned int i;
  for(i=0;i<10;i++)          v[i]=i+'a';
  for(i=0;str[i];i++)        v2.push_back(str[i]);
  cout<<"\nOroginal contents : ";
  for(i=0;i<v.size();i++)    cout<<v[i]<<" ";
```



```
vector<char>::iterator p=v.begin();
  p+=2;
  v.insert(p,10,'x');
  cout<<"\nSize after insertion "<<v.size()<<endl;
   p=v.begin();
  p+=2;
  v.erase(p,p+10);
  cout<<"\nSize "<<v.size();
  v.insert(p,v2.begin(),v2.end());
  cout<<"\nSize after v2's insertion : "<<v.size();
  return 0;
}
```

Remove and Replace Algorithm

```
#include<iostream>
#include<vector>
#include<algorithm>
int main()
{
  char str[]="The STL is power programming";
  vector<char> v,v2(30);
  unsigned int i;
  for(i=0;str[i];i++)              v.push_back(str[i]);
  cout<<"\nInput Sequence  ";
  for(i=0;i<v.size();i++)       cout<<v[i];
  remove_copy(v.begin(),v.end(),v2.begin(),' ');
```



```
  cout<<"\nResult after removing spaces  ";
   for(i=0;i<v2.size();i++)         cout<<v2[i];
   cout<<"\nInput sequence  ";
   for(i=0;i<v.size();i++)           cout<<v[i];
   replace_copy(v.begin(),v.end(),v2.begin(),' ',':');
   cout<<"\n Result after replacing ";
   for(i=0;i<v2.size();i++)          cout<<v2[i];
   return 0;
  }
```


Deques

- Very similar to vector.
- Dynamic array, random access, same interface
- But dynamic array in deque open at both ends

Deques compared with Vectors

- Fast insertion and removing at both beginning and the end
- A bit slower than vectors
- Iterators are smart pointers of a special type because they can jump between blocks
- For systems have limited block sizes of memory deques are larger
- We can not avoid reference interpretation in deques
- The container releases the memory if it is no farther used

Extended deque operations

C.push_back(elem)
C.push_front(elem)
C.pop_back(elem)
C.pop_front(elem)
C.front( )
C.end( )

Lists

- A list manages its elements as a doubly linked list
- The list is different in several ways with a vector or deque.

Abilities of lists

- Does not provide random access
- Inserting and removing elements is fast at each positions and not just the ends
- Inserting and deleting pointers does not invalidate pointers, references, and iterators or other elements
- A list supports exception handling in a way that every operation succeeds or is no-op.

Lists compared with vec & deq

- There is no at( ) operation in lists
- Don't provide operation for capacity
- Lists provide many special member functions for moving elements.
- Function c1.splice(pos,c2,c2pos)
- c1.merge(c2) c1.merge(c2,op)
- c1.sort( ) c1.sort(op)
- c1.unique( ) c1.unique(op)
- c1.reverse( )

Classifying algorithms

Nonmodifying algorithms
- ✓ Neither change the value nor the order of elements

Modifying algorithms
- ✓ Change the value of elements

Removing algorithms
- ✓ A special case of modifying algorithms that can remove elements

Mutating algorithms
- ✓ They change the order of elements (and not their values) by assigning and swapping their values

Sorting algorithms
- ✓ Are also special kind of mutating algorithms because change the order

Sorted range algorithms
- ✓ They require that the ranges on which they operate are sorted according to their sorting criterion

Numeric algorithms
- ✓ These algorithms combine numeric elements in different ways

Nonmodifyings

for_each( )
count( )
count_if( )
min_element( )
max_element( )
find( )
find_if( )
search_n( )
search( )
find_end( )
find_first_of( )
adjacent_of( )
equal( )
mismatch( )
lexicographical_compare( )

Modifyings

for_each( )
copy( )
copy_backward( )
transform( )
merge( )
swap_ranges( )
fill( )
fill_n( )
generate( )
generate_n( )
replace( )
replace_if( )
replace_copy( )
replace_copy_if( )

Removings

remove( )
remove_if( )
remove_copy( )
remove_copy_if( )
unique( )
unique_copy( )

Mutatings

reverse( )
reverse_copy( )
rotate( )
rotate_copy( )
next_permutation( )
prev_permutation( )
random_shuffle( )
partition( )
stable_partition( )

Sortings

sort( )
stable_sort( )
partial_sort( )
partial_sort_copy( )
nth_element( )
partition( )
stable_partition( )
make_heap( )
push_heap( )
pop_heap( )
sort_heap( )

Sorted ranges

binary_search( )
includes( )
lower_bound( )
upper_bound( )
equal_range( )
merge( )
set_union( )
set_intersection( )
set_difference( )
set_symmetric_difference( )
inplace_merge( )

Numerics

accumulate( )
inner_product( )
adjacent_difference( )
partial_sum( )

Implementation of Sequence and Associative Containers for different Algorithms using their Iterators

Algorithms

- **Algorithms** are used to process the elements of collections.
- For example, they can search, sort, modify, or simply use the elements for different purposes.
- Algorithms use iterators.

- To Access the STL algorithms we must include **<algorithm>** in our program.
- STL Algorithms are the Standalone Template Functions.
- STL provides more than 60 Standard Algorithms.

Generic Type names used by the Algorithms.

Generic Name	Represents
BiIter	Bidirectional Iterator
ForIter	Forwad Iterator
InIter	Input Iterator
OutIter	Output Iterator
RandIter	Random access Iterator
T	Some type of data
Size	Some type of Integer
Func	Some Type of Func
Generator	A function that generates objects
BinPred	Binary predicate
UnPred	Unary Predicate
Comp	Comparison Function

1. Binary_Search()

1)template<class ForIter, T>
Bool binary_search(ForIter start ,ForIter end, const T& val);

2)template<class ForIter , class T, class Comp>
Bool binary_search(ForIter start ,ForIter end, const T& val , Comp cmpfn);

Performs Binary Search on ordered sequence ranging from start to end. It returns true if **val** is found, false otherwise.

First checks for equality and Second allows you to specify your own comparison function.

2. copy()

template<class InIter, class OutIter>
OutIter copy(InIter start, InIter end, OuIter Result);

Copies a sequence beginning at start & ending at end.
Puts the result into the sequence pointed to by Result.
It Returns an Iterator to the end of the resulting sequence.

3. Copy_backward()

template<class BiIter1, class BiIter2>
BiIter2 copy_backward(BiIter1 start, BiIter1 end, BiIter2 Result);

Same as that of copy() except that it start copying from the end of the sequence first.

4. count()

template<class InIter , class T>
ptrdiff_t count(InIter start, InIter end , const T & val);

Returns the number of elements in the sequence beginning at star and ending at end that match **val.**

Example on Next Slide……

```
#include<iostream.h>
#include<vector>
#include<cstdlib>
#include<algorithm>
int main()
{
    vector<bool> v;
    unsigned int I;
    for(i=0; i<5; i++)
    {
        If( rand() % 2)v.push_back(true);
        else v,push_back(false);
    }
    cout<< “Sequence : \n”;
    for(i=0; i<v.size(); i++)
    cout<< v[i]<< “  “;
    cout<< endl;
    i= count(v.begin(), v.end(), true);
    cout<< i << “ Elements are true.\n ”;
    return 0;
}
```


Output of count()

Sequence :

True true false false true

3 elements are true.

5. count_if()

template<class InIter , class UnPred>

Ptrdiff_t count_if(InIter start , InIter end, Unpred pfn);

This algo returns the number of elements in the sequence starting from start and ending with end for which the unary predicate **pfn** returns true.

The **ptrdiff_t** is some integer .

Example on next slide…….

```
#include<iostream.h>
#include<vector>
#include<algorithm>
bool dividesby3(int i)
{
    if(i%3==0)      return true;
    else return false;
}
int main()
{
    vactor<int > v;
    int I;
    for(i=0;i<20;i++) v.push_back(i);
    cout<<" Sequence :\n";
    vector<int> :: iterator p;
    for( p= v.begin() ; p!= v.end() ; ++p)
    cout<< *p<<" ";
    cout<<endl;
    i=count_if(v.begin(), v.end(), dividesby3);
    cout<<I << " numbers are divisible by 3. \n";
    return 0;
```


Output of count_if()

Sequence:

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

6 numbers are divisible by 3.

6. equal()

1) template<class InIter1, class InIter2>
bool equal(InIter1 start, InIter1 end , InIter2 start2);

2) template<class InIter1, class InIter2, class BinPred>
bool equal(InIter1 start, InIter1 end , InIter2 start2, BinPred pfn);

Range determined by start1 and end1 is tested against the sequence pointed by start2 for equality. If same ,true is returned. Otherwise false is returned.

The Second form allows you to specify a binary predicate that determines when two elements are equal.

8. find()

template<class InIter, class T>
InIter find(InIter start, InIter end, const & val);

This algorithm searches the range start to end for the **val .** It returns an iterator to the first occurrence of the element or to end if value is not found.

9. find_end()

1) template<class ForIter1 , class ForIter2>
ForItet1 find_end(ForIter1 start1, ForIter1 end1, ForIter2 start2, ForIter2 end2);

2) template<class ForIter1 , class ForIter2, class BinPred>
ForItet1 find_end(ForIter1 start1, ForIter1 end1, ForIter2 start2, ForIter2 end2, BinPred pfn);

It finds the last subsequence defined by start2 & end2 within the range start1 & end1. If found ,iterator to the first element in sequence is returned , else the iterator to end1 is returned.

In Second form you can also specify a binary predicate that determines when elements match.


```
void f (vector<int>& v, int x)      // find an int in a vector
{
    vector<int>::iterator  p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

11. generate() and generate_n()

1) template<class FortIter, class Generator>
Void generate(ForIter start, ForIter end , Generator fngen);

2) template<class FortIter,class size, class Generator>
Void generate(ForIter start, size num , Generator fngen);

These algorithms assigns the elements in a range the values returned by a generator function.
The generator function is passed in **fngen**
generate() – range is start to end
Generate_n()- range Is start to **num**

12. includes()

1)template<class InIter1, class InIter2>
Bool includes(InIter1 start1, InIter1 end1, InIter2 start2, InIter2 end2);

2) template<class InIter1, class InIter2,class comp>
Bool includes(InIter1 start1, InIter1 end1, InIter2 start2, InIter2 end2, comp cmpfn);

This algo checks if the sequence defined by start1 and end1 includes all the elements in the sequence defined by start2 and end2.
If yes then returns true else, returns false

12. inplace_merge()

template<class BiIter>
Void inplace_merge(BiIter start, BiIter mid, BiIter end);

template<class BiIter, class comp>
Void inplace_merge(BiIter start, BiIter mid, BiIter end, comp cmpfn);

Within a single sequence , the inplace_merge() algo merges the range defined by start and mid with the range defined by mid and end. Both ranges must be stored in increasing order.

14. make_heap()

```
template<class RandIter>
Void make_heap( RandIter start, RandIter end);

template<class RandIter, class comp>
Void make_heap(RandIter start, RandIter end, comp
   cmpfn);
```

This makes the heap from the sequence defined by styart and end.

Second allows you to specify a comparison function that determines which element is less than another.

15. max()

```
template<class T>
Const T &max( const T &I,const T& j);

template<class T , class comp>
Const T &max(const T &i ,const T &j, comp cmpfn);
```

It returns the maximum of two values

16. max_element()

```
template<class ForIter>
ForIter max_element(ForIter start, FortIter end);

template<class ForIter, class comp>
ForIter max_element(ForIter start, FortIter end, comp
   cmpfn);
```

This returns the iterator to the maximum element within the range start and last.

17. merge()

template<class InIter2, class InIter2, class OutIter>
OutIter merge(InIter1 start1, InIter1 end1, InIter2 start2, InIter2 end2, OutIter Result);

template<class InIter2, class InIter2, class OutIter,class comp>
OutIter merge(InIter1 start1, InIter1 end1, InIter2 start2, InIter2 end2, OutIter Result, comp cmpfn);

This merges the two ordered sequences, placing the result into a third sequence. And an iterator to the end of the third sequence is returned.

18. mismatch()

template<class InIter1, class InIter2>
pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1, InIter2 start2);

template<class InIter1, class InIter2,class BinPred>
pair<InIter1, InIter2> mismatch(InIter1 start1, InIter1 end1, InIter2 start2, BinPred pfn);

This algo finds the first mismatch between the elements in two sequences . Iterators to the two elements are returned. If no mismatch Is found then iterator to the end of the sequence are returned.

The pair template class contaibs two data members called first and second that hold the pair of values.

19. nth_element()

template<class RandIter>
void nth_element(RandIter start, RandIter element,RandIter end);

template<class RandIter, class comp>
void nth_element(RandIter start, RandIter element ,RandIter end, comp cmpfn);

This arranges the sequence such that all elements less than **element** come before that element and all elements>element come after it.

20. partition()

template<class BiIter, class UnPred>
BiIter partition(BiIter start, biiter end, UnPred pfn);

This algorithm arranges the sequence such that all elements for which the predicate returns specified by pfn return true come before those for which the predicate returns false. It returns an iterator to the beginning of the elements for which the predicate is false.

21. pop_heap()

template<class RandIter >
void pop_heap(RandIter, RandIter end);

template<class RandIter , class comp>
void pop_heap(RandIter, RandIter end, comp cmpfn);

This algo exchanges the first and last-1 elements and then rebuilds the heap.

Second function has comp func that determine some element is greater than another.


```
#include<iostream.h>
#include<vector>
#include<algorithm>
Int main()
{
    char str[]=" The STL is power Programming ";
    Vector<char>v,v2(30);
    unsigned int;
    for(i=0; str[i] ; i++) v.push_back(str[i]);
    //** demonstrate remove_copy**//
    cout<<"Input Sequence :\n";
    for(i=0;i<v.size();i++) cout<< v[i];
    cout<<endl;
    // remove all spaces
    remove_copy(v.begin(),v.end(), v2.begin(), ' ');
    cout<<" Result after removing spaces : \n";
    for(i=0;i<v.size();i++)  cout<< v2[i];
    cout<<endl<<endl;
```

```
// * *demonstrates replace_copy **//
    Cout<<"Input Sequence":\n;
    for(i=0; i<v.size(); i++)  cout<< v[i];
    cout<<endl;

    // replace spaces with colons
    replace_copy(v.begin(),v.end(),' ',':');
    cout<<" Rsult after replacing spaces with colns:\n";
    cout<<endl << endl;
    return 0;
}
```


Output

Input sequence:
The STL is power programming.
Result after removing spaces:
TheSTLispowerprogramming.

Input Sequence:
The STL is power programming.
Result after removing spaces with colons:
The:STL:is:power:programming.

24. search()

Template<class ForIter1,class ForIter2>
FortIter1 search(ForIter1 start1, ForIter1 end1, ForIter2 star2, ForIter2 end2);

Template<class ForIter1,class ForIter2, class BinPred>
FortIter1 search(ForIter1 start1, ForIter1 end1, ForIter2 star2, ForIter2 end2, BinPred pfn);

This algo searches a subsequence within a sequence. The sequence being searched is defined by start1 & end at end1 . The subsequence searched is satrt2 & end2. if success ->iterator to its beginning is returned. Otherwise end1 is returned.

25. sort()

```
template<class RandIter>
void sort(RandIter start, RandIter end);

template<class RandIter, class comp>
void sort(RandIter start, RandIter end, comp cmpfn);
```

It sorts the rang specifed by start and end

26. Sort_heap()

```
template<class RandIter>
void sort_heap(RandIter start, RandIter end);

template<class RandIter, class comp>
void sort_heap(RandIter start, RandIter end, comp
  cmpfn);
```

This algo sorts the heap within the range specified by start and end.

27. Stable_partition()

```
template<class BiIter, class Unpred>
BitIter stable_partition(BiIter start, BiIter end, Unpred
  pfn);
```

This algo arranges the seuence defined by start and end such that al elemnets for which the predicate is specified by pfn is true come before those for which predicated came to be false.

It returns an iterator to the beginning of the element for which the pfn is false.

Examples STL

USING VECTORS

```
 # include  < iostream>
#include  < vector>                    //  Vector header file
using namespace  std ;
 void  display ( vector < int >  & v)
{          for ( int  i = 0 ; i <v . size ( ) ; i ++ )
              cout  <<  v [ i ]  <<  "  " ;
           cout  <<  " \ n " ;
}
 int  main ( )
{  vector < int >  v ;                      //  Create  a  vector of type int
   cout  <<  " Initial size  = " <<  v. size ( ) << "\ n " ;
 //  Putting values into the vector
int  x ;
cout  <<   " Enter five integer values : " ;
for ( int  i = 0 ; i < 5 ;  i ++)
{        cin >> x ;
         v. push _back ( x ) ;
}
```


USING VECTORS

```
cout  << " Size after adding  5  values :  " ;
cout  <<  v . size ( )  <<  " \ n " ;
 // Display the contents
cout  <<  "  Current contents :  \ n " ;
display  ( v ) ;
 // Add one more value
v . push _ back ( 6 . 6 ) ;             //  float value truncated  to  int
//   Display size and contents
cout  <<  " \ n Size  =  "  <<  v. size  ( )  <<  " \ n " ;
cout   <<  " Contents now : \n " ;
display ( v ) ;
//Inserting elements
vector < int>  : :  iterator  itr  = v . begin ( ) ;     //  iterator
itr  =  itr  +  3 ;                                      // itr points  to 4th element
v. insert ( itr , 1 , 9 ) ;
//Display the contents
cout  <<  "\ n Contents after inserting :  \n " ;
display ( v ) ;
```

USING VECTORS

```
// Removing 4th and 5th elements
v . erase ( v. begin ( ) +3 , v . begin ( ) + 5 ) ;        // Removes 4th and 5th element
 //Display the contents
cout  <<  " \n Contents after deletion :  \n " ;
display ( v ) ;
cout  <<  " END \ n " ;
return ( 0 ) ;
}
```


USING VECTORS

OUTPUT:

```
Initial Size  =  0
Enter five integer values :   1  2  3  4  5
Size after adding  5  values :    5
Current contents :
2  3  4  5

 Size  =  6
Contents now :
1  2   3   4   5   6

Contents after inserting:
2  3  9  4  5  6

Contents after deletion
2   3    5    6
END
```


Conclusion

- **A collection of generic classes and functions is called Standard Template Library. STL components are part of the C++ standard library.**
- **STL consists of three main components: containers, algorithms and iterators.**
- **Containers are divided into sequential, associative and derived containers. Containers classes define a large number of functions that can be used to manipulate their contents.**

Conclusion

- **Iterators are like pointers. It is used as an arguments for algorithm and are defined for specific containers.**
- **A function object is created by a class that contains only one overloaded operator.**

Summary

- **In order to help C++ users in generic programming, Alexander Stepanov and Meng Lee of Hewlett Packard developed a set of general purpose templatized classes (data structures) and functions (algorithms) that could be used as a standard approach for storing and processing of data.**
- **The collection of these generic classes and functions is called Standard Template Library (STL). The STL has now become a part of the ANSI standard C++ class library.**

Review Questions

- What are manipulators?
- Do templates provide reusability of a different kind?
- Distinguish between list and vectors.
- When do we use sequence diagram?

Review Questions [Short Answer Types]

- When do we need multiple catch handlers?
- How do we return an error value from the constructor?
- How are the STL algorithm implemented?
- Distinguish maps and multimaps concept?
- Briefly describe all UML notations?
- Illustrate the use of function object greater<>() in sort() algorithm.
- Using MAPS Enter three sets of name and telephone numbers through a C++ program.

Review Questions [Long Answer Types]

- What is an algorithm? How STL algorithms are different from the conventional algorithms?
- Illustrate the use of function object greater<>() in sort() algorithm.
- Using MAPS Enter three sets of name and telephone numbers through a C++ program.
- Draw all components and functionalites using State Machine Diagrams.

Recommended Books

TEXT:

1. A..R.Venugopal, Rajkumar, T. Ravishanker "Mastering C++", TMH, 2009.
2. S. B. Lippman & J. Lajoie, "C++ Primer", 6th Edition, Addison Wesley, 2006.

REFERENCE:

1. R. Lafore, "Object Oriented Programming using C++", Galgotia Publications, 2008.
2. D . Parasons, "Object Oriented Programming with C++", BPB Publication.
3. Steven C. Lawlor, "The Art of Programming Computer Science with C++", Vikas Publication.
4. Schildt Herbert, "C++: The Complete Reference", 7th Ed., Tata McGraw Hill, 2008.