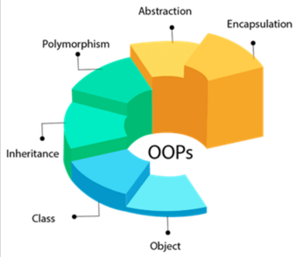
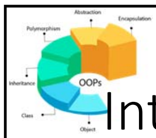


Templates

Chapter 8

Object Oriented Programming

By DSBaral



Introduction

- Template is one of the most sophisticated, flexible and powerful feature of C++.
- It was not the original feature of C++ but was added later.
- Templates provides generic programming, allowing development of reusable software components with functions and classes, supporting different data types in a single framework.
- Template support generic programming by taking type as their argument.
- As template helps to make function and class to support any type of data it reduce the code redundancy.
- With template feature we can make function template also called generic function and class template also called generic class.

Function Template

- The function that is designed to operate on one particular data type cannot be used to operate on another data type.
- A function template or generic function can be used to operate on different types of data by taking data type as an argument.
  - For example a single sort function template can be created which can operate sorting operation with any type of data.
- Without templates function overloading can solve the problem but it add code redundancy.
- With template, we create a generic function that can automatically overload itself.

Function Template (Contd…)

- Let’s see a scenario to implement an algorithm to find maximum among two numbers of different data types using function overloading.

```
int find_max(int a, int b)
{
   int result;
   if(a>b)
       result=a;
   else
       result=b;
   return result;
}
```

Function Template (Contd…)

```
float find_max(float a, float b){
   float result;
   if(a>b)
       result=a;
   else
       result=b;
   return result;
}
char find_max(char a, char b){
   char result;
   if(a>b)
       result=a;
   else
       result=b;
   return result;
}
```

Function Template (Contd…)

```
int main()
{
    int i1=15,i2=20;
    cout<<"Greater is "<< find_max(i1,i2)<<endl;
    float f1=40000.05, f2=38000.44;
    cout<<"Greater is "<< find_max(f1,f2)<<endl;
    char c1='a',c2='A';
    cout<<"Greater is "<< find_max(c1,c2)<<endl;
    return 0;
}
```

- Here the logic of finding greater value is same for all function but they differs only in terms of data types.

Function Template (Contd…)

- This type of redundant code can be eliminated by writing a single generic function that takes data type as argument.
- The C++ template feature enables substitution of a single piece of code for all these overloaded function.
- The general format of a function template definition is as follows

```
template <class template_type,...>
return_type func_name(parameter_list)
{
   //function body
}
```

  There can be more than one template parameter in template definition
- Instead of using the keyword `class` to specify a generic type in template definition, keyword `typename` can also be used.

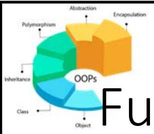
Function Template (Contd…)

- The above overloaded function `find_max()` can be replaced with template function as follows:

```
template <class T>
T find_max(T a, T b)
{
   T result;
   if(a>b)
       result=a;
   else
       result=b;
   return result;
}
```

Function Template (Contd…)

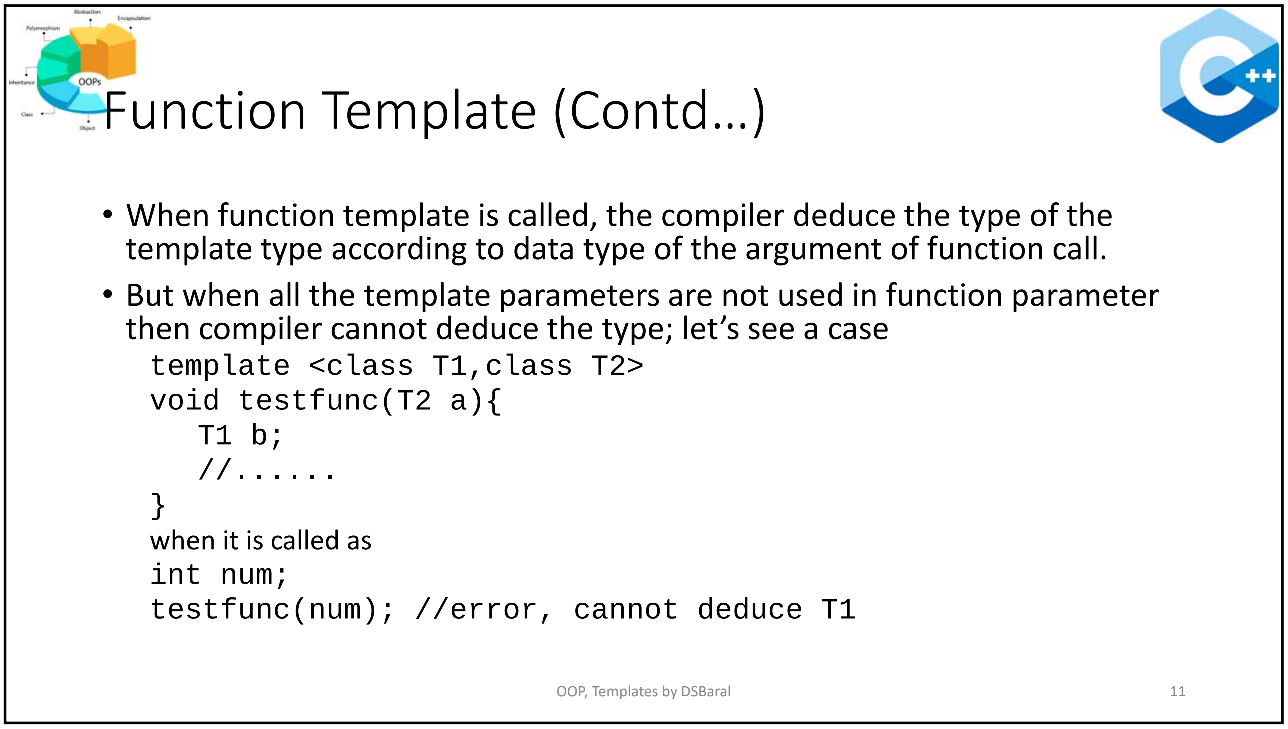
- After making the function template, when function template is called with any data types as argument, the compiler creates a function internally without the user intervention and invokes it.
  - If the function is called again with the same data type as argument the compiler does not generate other function but the same one is called.
- The process of generating a function for specific type is called instantiating the function template.
- A version of a template for a particular template argument is called a *specialization*.
- The function template can be called by passing user-defined data as well, provided that the operators for user-defined types are already defined.



Function Template (Contd…)

- There can be more than one template parameter in function template.
- Lets see an example that uses more than one template parameter

```
template <class T1,class T2>
void testfunc(T1 a, T2 b){
    cout<<a<<" "<<b<<endl;
}
int main(){
    int inum=5;
    float fnum=7.5;
    testfunc(inum,fnum);
    testfunc(20,"Let's make a better World");
    testfunc(15.3,17L);
}
```



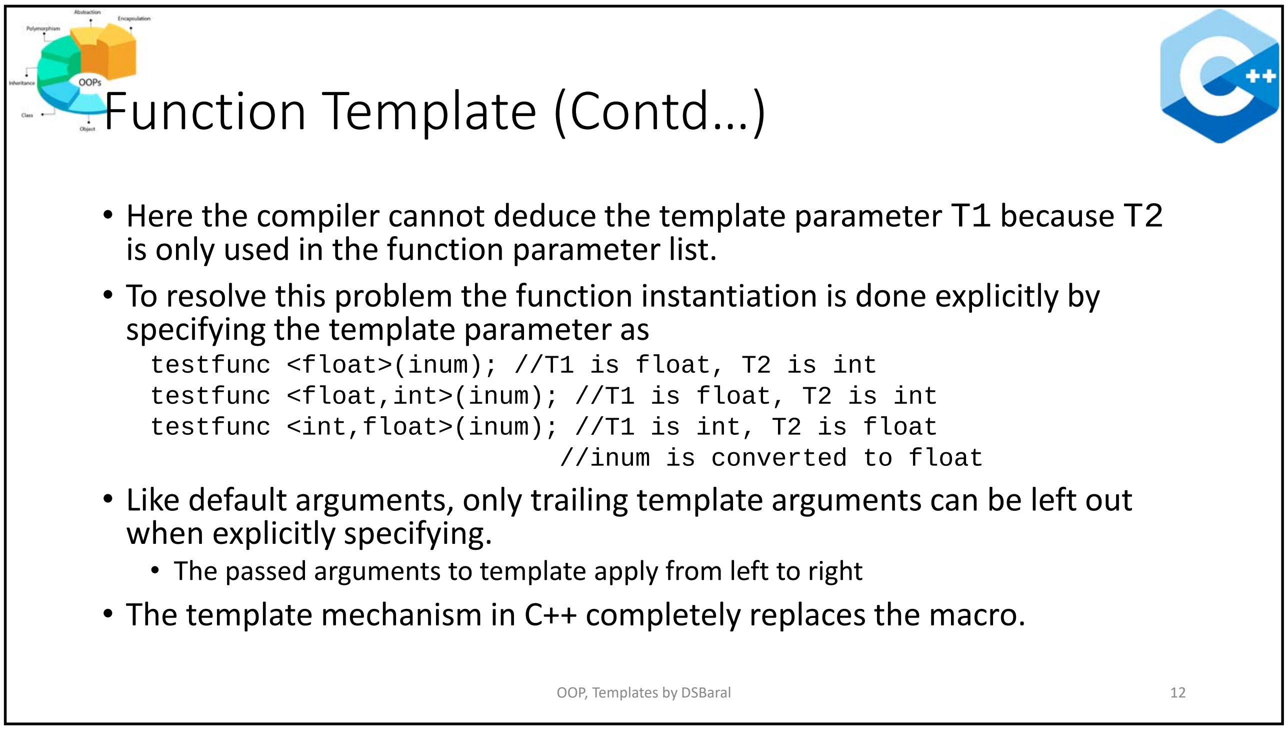
Function Template (Contd…)

- When function template is called, the compiler deduce the type of the template type according to data type of the argument of function call.
- But when all the template parameters are not used in function parameter then compiler cannot deduce the type; let’s see a case

```
template <class T1,class T2>
void testfunc(T2 a){
   T1 b;
   //......
}
```

when it is called as

```
int num;
testfunc(num); //error, cannot deduce T1
```



Function Template (Contd…)

- Here the compiler cannot deduce the template parameter T1 because T2 is only used in the function parameter list.
- To resolve this problem the function instantiation is done explicitly by specifying the template parameter as

```
testfunc <float>(inum); //T1 is float, T2 is int
testfunc <float,int>(inum); //T1 is float, T2 is int
testfunc <int,float>(inum); //T1 is int, T2 is float
                           //inum is converted to float
```

- Like default arguments, only trailing template arguments can be left out when explicitly specifying.
  - The passed arguments to template apply from left to right
- The template mechanism in C++ completely replaces the macro.

Macro

```
#define N 50
#define SQR1(X)   X*X
#define SQR2(X)   ((X)*(X))

c=SQR1(a);   //replaced as a*a
c=SQR1(b+c); //replaced as b+c*b+c

c=SQR2(a);   //replaced as ((a)*(a))
c=SQR2(b+c);  //replaced as ((b+c)*(b+c))
```

Overloading Function Template

- Similar to normal functions, function template can also be overloaded.
- When using function templates, we can declare several function templates with the same name and even declare a combination of function templates and non template functions with the same name.
- The function template can be overloaded as
  - Overloading function template with other functions
  - Overloading function template with other function templates

Overloading Function Template with other Functions

- If the function templates definition is not suitable for some specific data type then it is necessary to override the function template by defining a normal function for specific type.
- When we are defining both function template and normal function of same name with exactly matching parameters in the same scope then the compiler selects the more specific function.
- Let’s see the following example.

```
template<class T>
T find_max(T a, T b)
{
  T result;
```

Overloading Function Template with other Functions (Contd…)

```
  if(a>b)
      result=a;
  else
      result=b;
  return result;
}
```

When this function template is called with string (`char *`) data as

```
char *st1,*st2;
char st3=find_max(st1,st2);
```

It does the comparison between the pointers instead of values so it does not solve our problem of comparing strings

Overloading Function Template with other Functions (Contd…)

- So for `char *` type we have to define a normal function along with the function template as

```
char *find_max (char *a, char *b)
{
    char *result;
    if(strcmp(a,b)>0)
        result=a;
    else
        result=b;
    return result;
}
```

Overloading Function Template with other Functions (Contd…)

- After defining this function for `char *`, when `char *` is passed, the normal function is called and when other data types are passed the function template is called.
- When we explicitly specify the template arguments, the effect is as follows

```
find_max<float> (2,7); //both arguments are converted to float
find_max(str1, str2); //calls the normal function
find_max<char *>(str1, str2); //calls the function template
```

Overloading Function Template with other Function Templates

- Along with overloading the function template with normal function, the function template can be overloaded with other function templates.
- Similar to overloading of normal functions, overloaded function templates must differ either in terms of number of parameters or their type.
- The function template overloading with other function template is illustrated by the following example.

Overloading Function Template with other Function Templates (Contd…)

```
template <class T>
void display(T data)
{
   cout<<data<<endl;
}
template <class T>
void display(T data, int n)
{
   for(int i=0;i<n;i++)
         cout<<data<<endl;
}
```

Overloading Function Template with other Function Templates (Contd…)

```
int main()
{
    display(1);
    display(1.5);
    display(420,2);
    display("Let’s make our world a better place",3);
    return 0;
}
```

- Let’s see another example that uses different types of argument

Overloading Function Template with other Function Templates (Contd…)

```
template<class T>
void func(T a, T b){
  cout<<"func(T a, T b):"<<a<<','<<b<<endl;
}
template <class T1, class T2>
void func(T1 a, T2 b){
  cout<<"func(T1 a,T2 b):"<<a<<','<<b<<endl;
}
```

- In this example, when both the arguments supplied are of same type then the first function template is called and when the argument supplied are of different types the second function template is called.

Class Template

- Function templates help to design a function (implementation of algorithm) that operates on any data types.
- Similar to function template there can be cases where we need to create classes that work on different types of data, that is abstract data type to work on any data type.
- A class that operates on any type of data is called class template.
- Consider an example of creating a class for stack operation.
- A stack is data structure in which data that is stored at last must be removed first; that is in Last In First Out (LIFO) discipline.
- Let’s declare the stack class that stores integer data type

Class Template (Contd…)

```
class intStack
{
private:
   int arr[25];
   int top;
public:
   intStack();
   void push(int data);
   int pop();
   int size();
};
```

- If we need to create a stack for `float` data type then this class will not work so we have to create another class for `float` data type as follows

Class Template (Contd…)

```
class floatStack
{
private:
   float arr[25];
   int top;
public:
   floatStack();
   void push(float data);
   float pop();
   int size();
};
```

- If we need to create stack for other types then we need to declare stack class for each and every data type.

Class Template (Contd…)

- Since the stack operation is same for any type of data, we would create a single class specification that would work for any data type eliminating repeated code.
- The C++ provides a construct called class template, that can be used to create class template to work on generic data type.
- The general form of declaring class template is as follows:

```
template <class template_type,...>
class class_name{
private:
   //data member of template type or non template type
   //......
public:
   //function members with template type argument
   // and return type
};
```

Class Template (Contd…)

- There can be more than one template parameters with comma separated list with each template type preceded with keyword `class`.
- Once a class template is declared, we create a specific instance of the class using the following syntax:

```
class_name<data_type> object;
```

- The process of generating a class definition from a class template is called template instantiation.
- A version of template for a particular template argument is called a specialization.
- Unlike function templates, we must pass the template arguments during object declaration.
- The class template for the above stack class can be created as follows

Class Template (Contd…)

```
template <class T>
class Stack{
private:
   T arr[25];
   int top;
public:
   Stack();
   void push(T data);
   T pop();
   int size();
};
```

- The object of this class are declared by passing two template arguments as

```
Stack<int> istack;
```

This declaration creates a class `Stack` with `int` as the template argument.

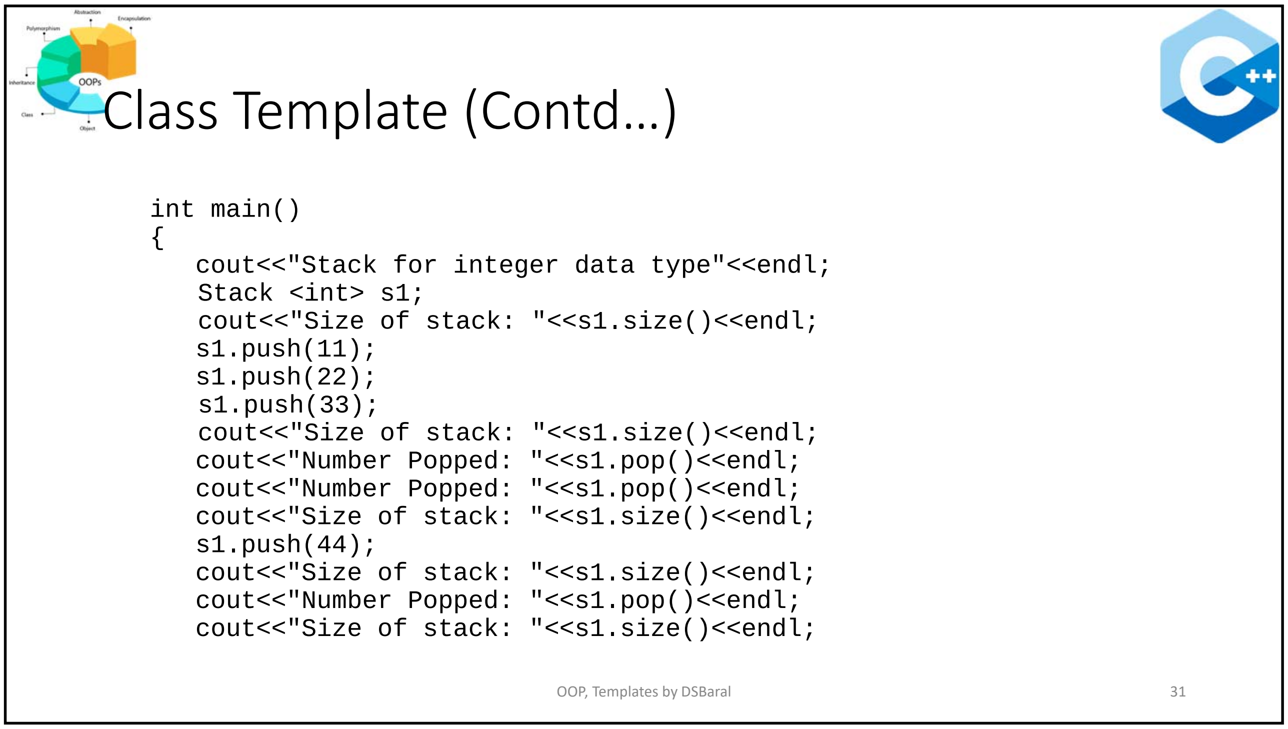
Class Template (Contd…)

- Like functions, class cannot be overloaded.
- The example of the class template when functions are defined inside the class is as follows

```
template <class T>
class Stack
{
private:
   T arr[25];
   int top;
public:
   Stack()
   { top=-1;}
```

Class Template (Contd…)

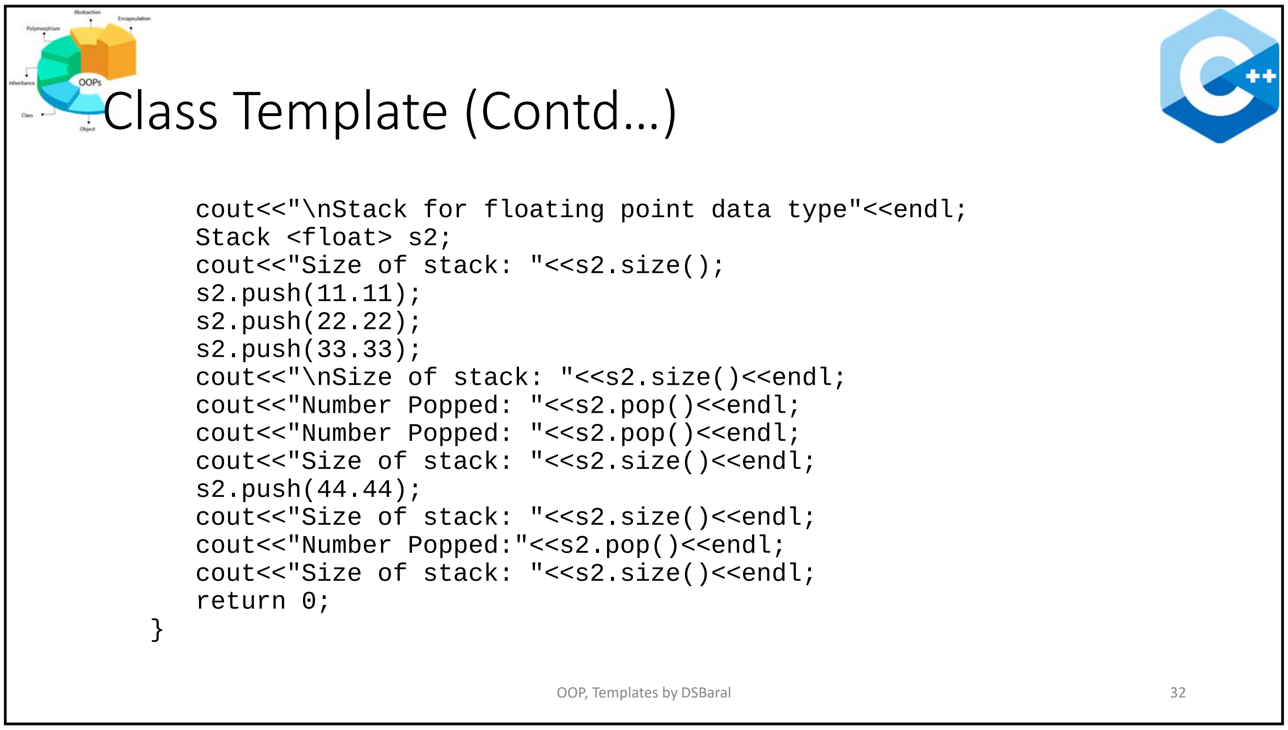
```
        void push(T data)
        {
            arr[++top]=data;
        }
        T pop()
        {
            return arr[top--];
        }
        int size()
        {
            return (top+1);
        }
};
```



Class Template (Contd…)

```
int main()
{
   cout<<"Stack for integer data type"<<endl;
   Stack <int> s1;
   cout<<"Size of stack: "<<s1.size()<<endl;
   s1.push(11);
   s1.push(22);
   s1.push(33);
   cout<<"Size of stack: "<<s1.size()<<endl;
   cout<<"Number Popped: "<<s1.pop()<<endl;
   cout<<"Number Popped: "<<s1.pop()<<endl;
   cout<<"Size of stack: "<<s1.size()<<endl;
   s1.push(44);
   cout<<"Size of stack: "<<s1.size()<<endl;
   cout<<"Number Popped: "<<s1.pop()<<endl;
   cout<<"Size of stack: "<<s1.size()<<endl;
```

OOP, Templates by DSBaral 31



Class Template (Contd…)

```
   cout<<"\nStack for floating point data type"<<endl;
   Stack <float> s2;
   cout<<"Size of stack: "<<s2.size();
   s2.push(11.11);
   s2.push(22.22);
   s2.push(33.33);
   cout<<"\nSize of stack: "<<s2.size()<<endl;
   cout<<"Number Popped: "<<s2.pop()<<endl;
   cout<<"Number Popped: "<<s2.pop()<<endl;
   cout<<"Size of stack: "<<s2.size()<<endl;
   s2.push(44.44);
   cout<<"Size of stack: "<<s2.size()<<endl;
   cout<<"Number Popped:"<<s2.pop()<<endl;
   cout<<"Size of stack: "<<s2.size()<<endl;
   return 0;
}
```

OOP, Templates by DSBaral 32

Class Template (Contd…)

- A class template can have more than one template parameters.
- The following example illustrates the use of multiple templates in a class template.

```
template <class T1,class T2>
class test {
private:
    T1 a;
    T2 b;
public:
    test(){}
    test(T1 n1,T2 n2){
        a=n1;b=n2;
}
```

Class Template (Contd…)

```
    void display(){
        cout<<"Data: "<<a<<" and "<<b<<endl;
    }
};
int main()
{
    test<int,float> myobj(5,7.39);
    myobj.display();
    return 0;
}
```

Function Definition of Class Template outside the Class

- The member function of a class template can also be defined outside the class.
- The member functions of a class template are function templates parameterized by the parameter as that of their class template.
- When a member function of a class template is defined outside the class the member function must explicitly be declared as a template.
- The member function of class template is defined outside the class in the following form.

Function Definition of Class Template outside the Class (Contd…)

```
template<class template_type,…>
class class_name
{
private:
   template_type variable_name;
   //......
public:
   return_type function_name(template_type arg);
   //......
};
template <class template_type,…>
return_type class_name<template_type>
         ::function_name(template_type arg)
{
   //body of function template
}
```

Function Definition of Class Template outside the Class (Contd...)

- For the Stack class discussed above, the class template declaration and the function definition outside the class is done as follows:

```
template <class T>
class Stack{
private:
   T arr[25];
   int top;
public:
   Stack();
   void push(T data);
   T pop();
   int size();
};
```

Function Definition of Class Template outside the Class (Contd...)

```
template <class T>
Stack<T>::Stack(){
   top=-1;
}
template <class T>
void Stack<T>::push(T data){
   arr[++top]=data;
}
template <class T>
T Stack<T>::pop(){
   return arr[top--];
}
template <class T>
int Stack<T>::size(){
   return (top+1);
}
```

Non-Template Type Arguments

- Along with the generic type template parameter, the template specification for a generic class can have non template type parameters too.
- Let’s see the following example

```
template<class T,int size>
class test
{
private:
   T arr[size];
   public:
   //......
};
```

Non-Template Type Arguments (Contd…)

- The non template type parameter size must be known at compile time and must be a constant expression.
- Suppose class template is instantiated as follows

```
test <int,5> t1;
```

- Now, the compiler creates the class as per the non template type argument supplied as follows:

```
class test
{
private:
   int arr[5];
public:
   //......
};
```

Default Arguments with Class Template

- Like the default argument with normal functions the class template can have a default argument associated with template parameter.
- The default template class argument is specified in the template declaration in the following form:

```
template <class template_type=default_data_type>
class class_name
{
   //data and function declarations
};
```

Default Arguments with Class Template (Contd...)

- The actual class template declaration with default template argument is as

```
template <class T=int>
class test
{
   //......
};
```

- When the template argument is not supplied in the object declaration then `int` type will be assumed.
- The object declaration of this class template is done as

```
test<float> ft;//template argument is float
test<> it;    //template argument int when not supplied
```

- Even if when we are not supplying the template argument we must use the empty angle brackets which states that test is a class template.

Default Arguments with Class Template (Contd…)

- The non template argument can also takes default values.
- The default value for non template type parameters are specified in the same way as default argument for function parameters as.
- The Array class in the above program can be declared with default argument as follows

```
template<class T=int,int size=10>
class Array
{
  //......
};
```

Default Arguments with Class Template (Contd…)

- The default type for template parameter T is `int` and the default value for non-template type parameter `size` of type `int` is 10.
- The instantiation of this class template `Array` can be done by specifying both parameters as

```
    Array<float,5> ftarr;
```

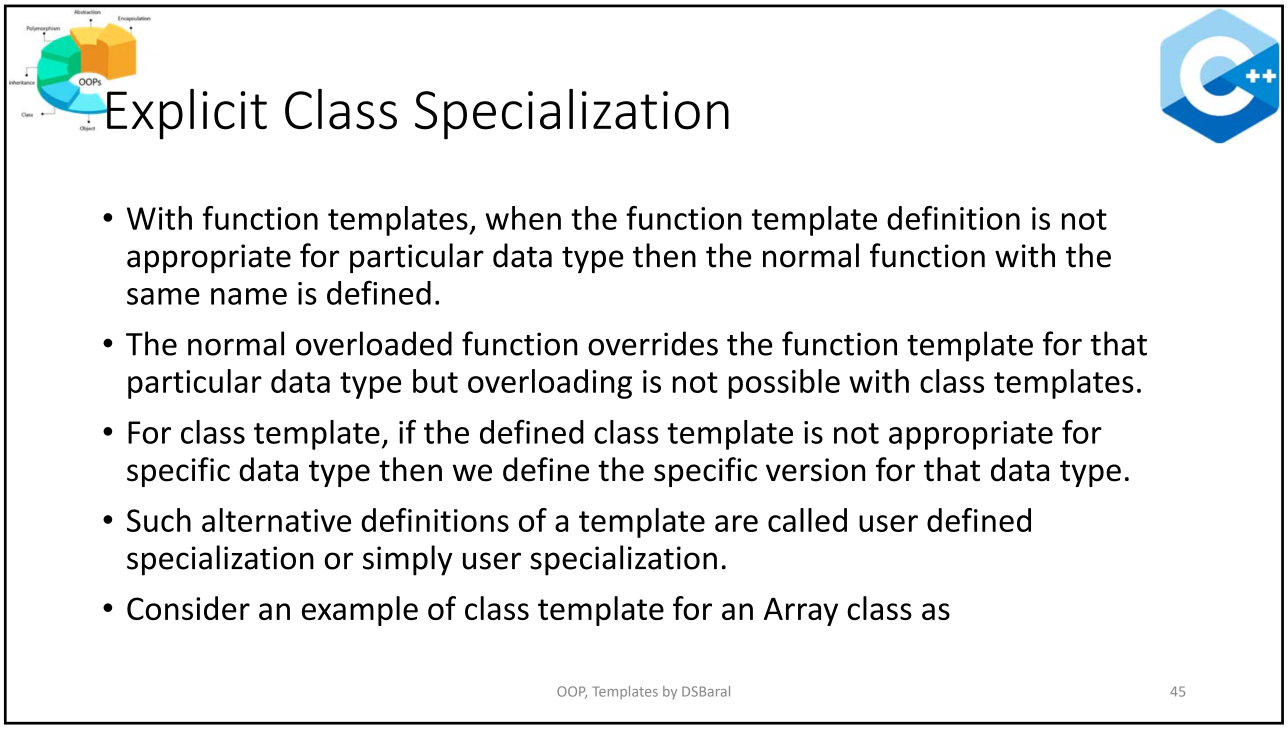
  or by passing the type argument and not passing `size` as

```
    Array<double> dbarr;
```

  or by not passing both type T and `size` as

```
    Array<> inarr;
```

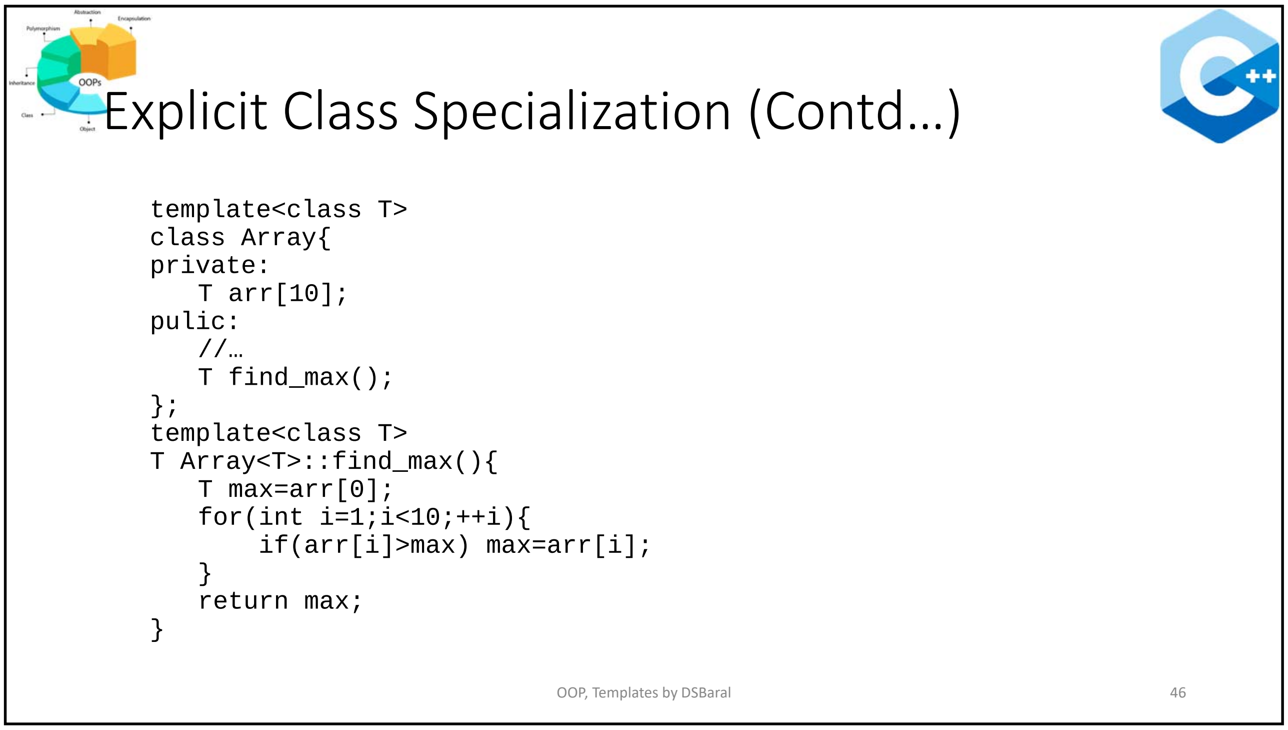
- As with default function arguments, only the trailing default template arguments can be left out of the list of arguments.



Explicit Class Specialization

- With function templates, when the function template definition is not appropriate for particular data type then the normal function with the same name is defined.
- The normal overloaded function overrides the function template for that particular data type but overloading is not possible with class templates.
- For class template, if the defined class template is not appropriate for specific data type then we define the specific version for that data type.
- Such alternative definitions of a template are called user defined specialization or simply user specialization.
- Consider an example of class template for an Array class as

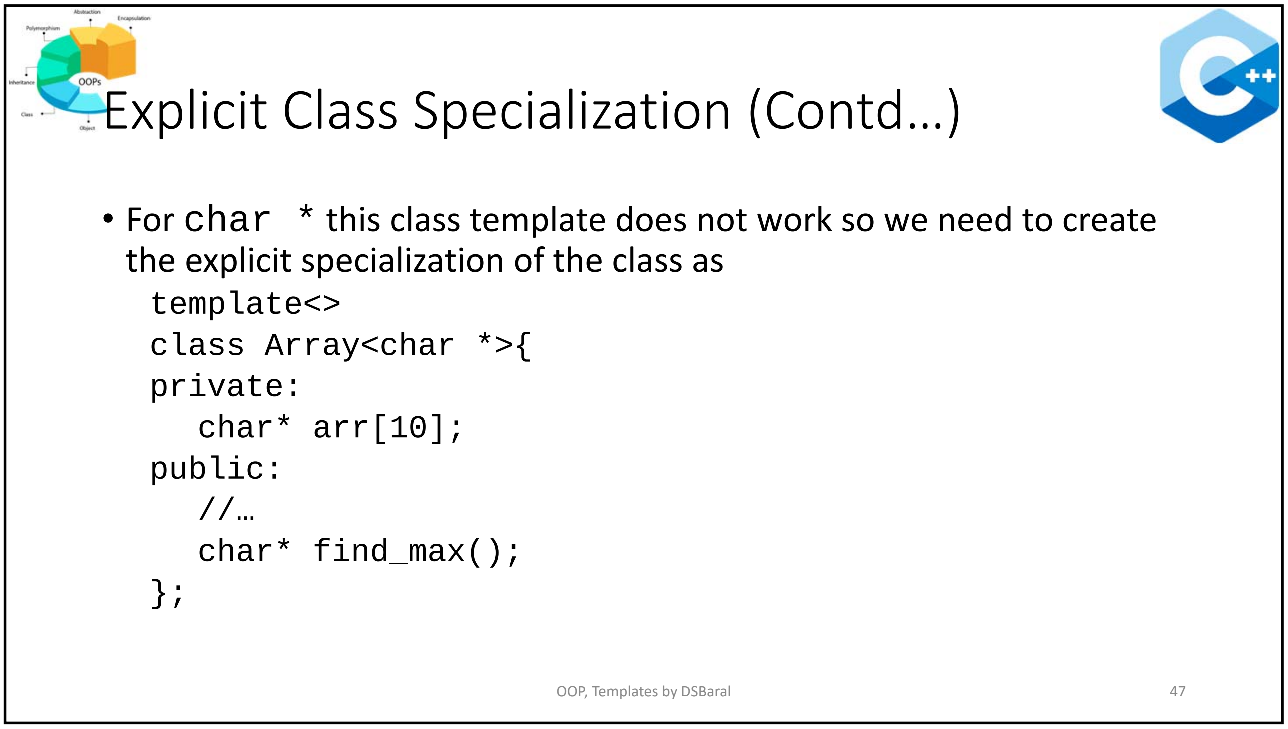
OOP, Templates by DSBaral 45



Explicit Class Specialization (Contd…)

```
template<class T>
class Array{
private:
   T arr[10];
pulic:
   //…
   T find_max();
};
template<class T>
T Array<T>::find_max(){
   T max=arr[0];
   for(int i=1;i<10;++i){
       if(arr[i]>max) max=arr[i];
   }
   return max;
}
```

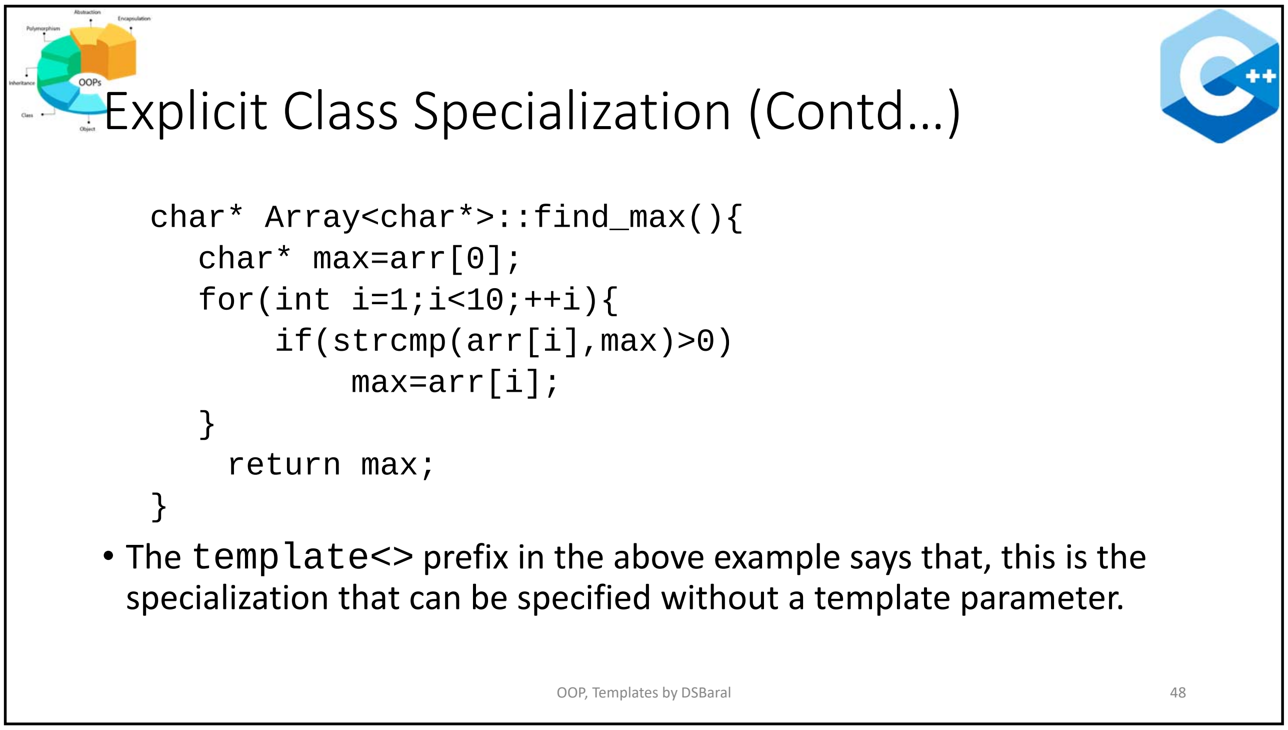
OOP, Templates by DSBaral 46



Explicit Class Specialization (Contd…)

- For `char *` this class template does not work so we need to create the explicit specialization of the class as

```
template<>
class Array<char *>{
private:
   char* arr[10];
public:
   //…
   char* find_max();
};
```



Explicit Class Specialization (Contd…)

```
char* Array<char*>::find_max(){
   char* max=arr[0];
   for(int i=1;i<10;++i){
        if(strcmp(arr[i],max)>0)
             max=arr[i];
   }
     return max;
}
```

- The `template<>` prefix in the above example says that, this is the specialization that can be specified without a template parameter.

Explicit Class Specialization (Contd…)

- When the array is declared with `char *` as

  ```
  Array<char*> starr;
  ```

  then the explicit specialization of the class is used.

- When the member function of the specialized class is to be defined outside the class, we do not need to specify the member function as template as

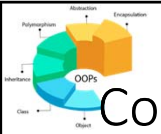
  ```
  char* Array<char*>::find_max()
  {
    //definition
  };
  ```

Standard Template Library

- Since the introduction of C++, a lot of data structures and algorithms are implemented in templates for generic programming by different developers/programmers.
- The importance of templatized library for data structures and algorithm was realized by the C++ standard committee, so, the committee decided to add the templatized library as Standard Template Library (STL) to the C++ standard library.
- The STL defines powerful, template-based, reusable components that implement many common data structures, and algorithms used to process those data structures.

Standard Template Library (Contd…)

- The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and is based on their research in the field of generic programming, with significant contributions from David Musser.
- The sole purpose of designing STL is to increase performance and flexibility for C++ programmer.
- STL was incorporated in C++ in October 1994.
- The STL is a generic library and almost every component in the STL is a template, so, any data type can be used with any algorithm or data structure (class) of library.
- STL components are defined in the namespace `std`.



Components of STL

- The main components of STL are
  - Container -> templatized classes
  - Iterator -> pointer like templatized objects
  - Algorithms -> templatized functions
- The STL containers are data structures (class) capable of storing objects of any data type in an organized way in memory.
- STL containers are implemented as template classes.
- Containers are of three types
  - Sequence Container -> vector, deque, list
  - Associative Container -> set, map, multiset, multimap
  - Derived Container -> stack, queue, priority_queue

Components of STL (Contd…)

- STL iterators, which have properties similar to those of pointers, are used by programs to manipulate the STL-container elements.
- Iterators are templatized classes whose instances (object) are used to traverse (navigate) through elements in the container.
- Some common types of iterator are Random access, Bidirectional, Forward, Input, and Output.
- STL algorithms are function templates that perform common data manipulations as searching, sorting, copying and comparing data.
- Algorithms are standalone global functions that work with array of any data type or containers.
- Some of them are find(), search(), swap(), count(), merge(), fill().
- Most of algorithms use iterators to access container elements.

Components of STL (Contd…)

- A container's supported iterator type determines whether the container can be used with a specific algorithm.

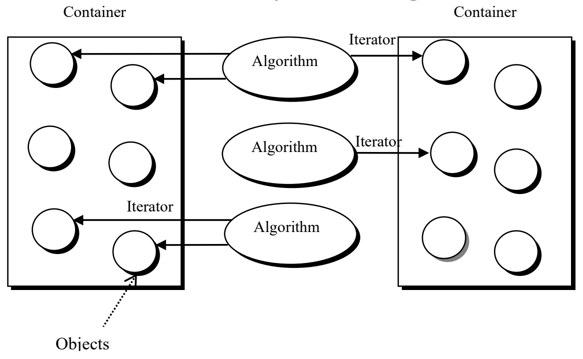
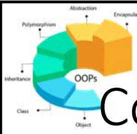
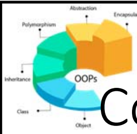


Figure: Relationship between container, algorithm and objects



Container

- A container is a data structure that can be use to store user defined types or built-in types like data int, float, char etc.
- Different Containers are as follows
- Vector:
  - A vector is a sequence container that stores and manages its objects in a dynamic array and can be accessed with [ ] operator.
  - It supports various operations such as random access to elements, insertion and removal of elements at the end, at the beginning or in the middle.
  - Insertion and removal at the beginning or middle of an array is time consuming.
  - The number of elements in a vector may vary dynamically.
  - Memory management in vector is automatic.
  - Vector is the simplest, the most efficient and most frequently used containers among the STL container classes.



Container (Contd…)

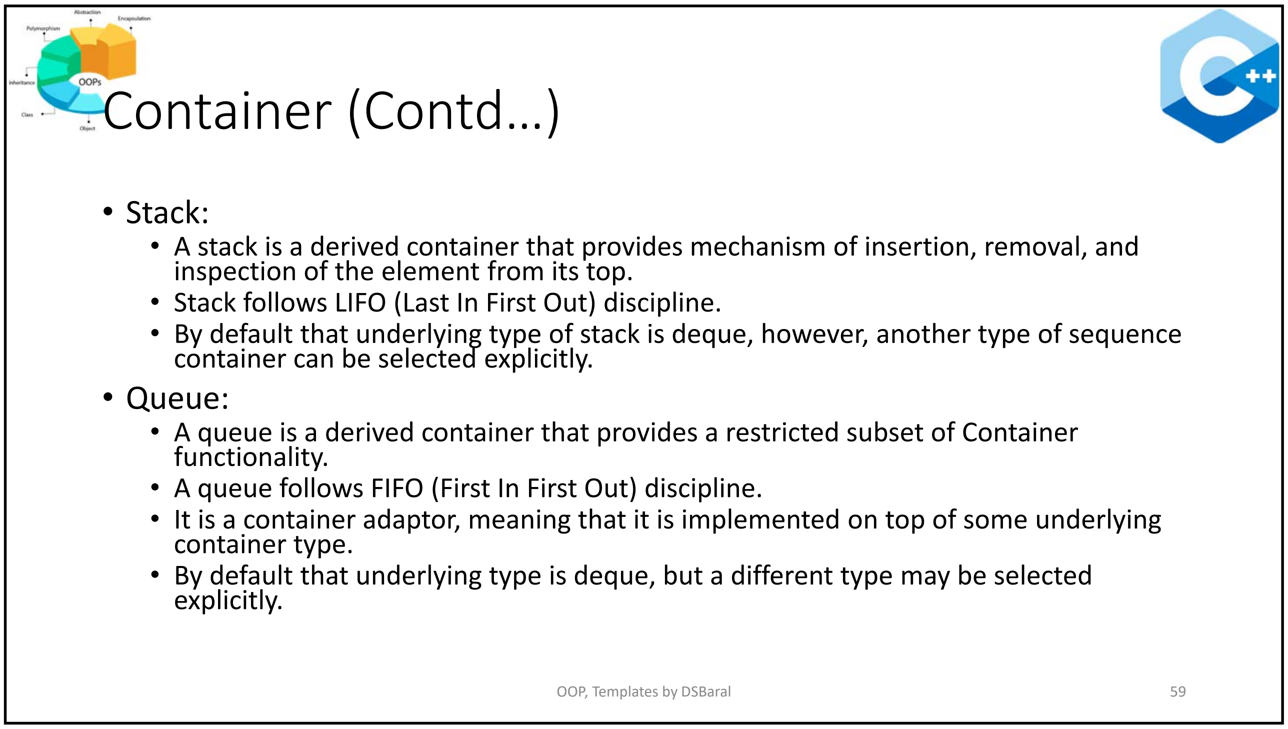
- List:
  - A list is a bidirectional linear list, that is, doubly linked list.
  - list has the important property that insertion and removal can be done in any position easily without any extra overhead like shifting operations.
  - The ordering of iterators may be changed, that is list might have a different predecessor or successor after a list operation than it did before.
  - Since the list stores address of front and back elements, it can be accessed from both ends.
- Deque:
  - A deque is very much like a combination of vector and list.
  - Like vector, it is a sequence container that supports random access to elements using the index operator [ ] and like list it can be accessed at the front and back as well.
  - The main difference between deque and vector is that deque also supports removal of elements at the beginning of the sequence.

Container (Contd…)

- Set:
  - It is an associated container which stores unique values.
  - It stores a number of items containing keys.
  - In set the value is itself the key.
  - The value of the elements in a set cannot be modified (the elements are always const), but they can be inserted or removed from the container.
  - Internally, the elements in a set are always sorted in a specific order (default ascending).
- Multiset:
  - It is an associative container that stores objects of type key.
  - It is also a multiple associative container so that two or more elements may be identical.
  - Internally, the elements in a multiset are always sorted
  - Its members and usage is like set.

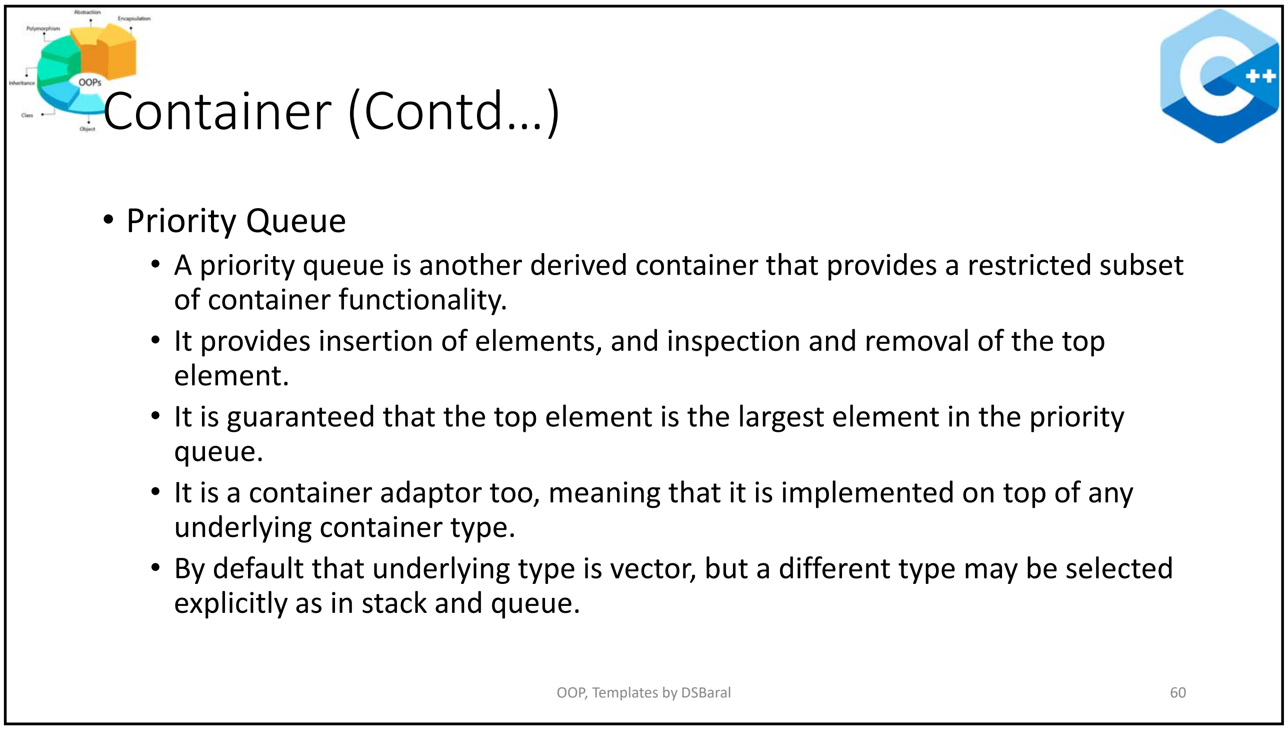
Container (Contd…)

- Map:
  - It stores pair of objects; one key object and another associated value object.
  - It can be used as a container that resembles an array.
  - Unlike array we can access map using key object that serves as the index and the related value object is the value at that index.
  - Internally, the elements in a map are always sorted by its key in specific order (default ascending).
- Multimap:
  - Like map, multimap associates key objects with value objects, unlike map multiple key values can be stored.
  - It is a sorted associative container that associates objects of the type key with other objects.
  - It is also called multi associative container meaning that there is no limit on the number of elements with the same key.
  - Its members and usage is like map.



Container (Contd…)

- Stack:
  - A stack is a derived container that provides mechanism of insertion, removal, and inspection of the element from its top.
  - Stack follows LIFO (Last In First Out) discipline.
  - By default that underlying type of stack is deque, however, another type of sequence container can be selected explicitly.
- Queue:
  - A queue is a derived container that provides a restricted subset of Container functionality.
  - A queue follows FIFO (First In First Out) discipline.
  - It is a container adaptor, meaning that it is implemented on top of some underlying container type.
  - By default that underlying type is deque, but a different type may be selected explicitly.



Container (Contd…)

- Priority Queue
  - A priority queue is another derived container that provides a restricted subset of container functionality.
  - It provides insertion of elements, and inspection and removal of the top element.
  - It is guaranteed that the top element is the largest element in the priority queue.
  - It is a container adaptor too, meaning that it is implemented on top of any underlying container type.
  - By default that underlying type is vector, but a different type may be selected explicitly as in stack and queue.

The Container Examples

- Vector:
  - The following example shows the use of vector

```
#include<vector>
int main()
{   vector <int> v;
    v.push_back(10);
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);
    v[0]=20;
    v[3]=23;
    cout<<"The elements in the vector are:"<<endl;
    for(int i=0;i<v.size();++i)
        cout<<v[i]<<" ";
    return 0;
}
```

The Container Examples (Contd…)

- It is wise to use vector instead of simple array
- The uses of other member of vector are shown below

```
vector <int> v1,v2;
v2.push_back(); //adds element at last
v2.back();         //returns last element
v2.pop_back();     //remove last element
v1.swap(v2); //swap vector v1 and v2
v2.empty();        //checks if empty and returns bool
v1.size();         //returns the size of vector
v2.insert(v2.begin()+2,15);//insert value 15 at position 2
v1.erase(v1.begin()+2);   //erase element at position 2
```

- Vectors can also be initialized with arrays as

```
int arr[]={10,20,30,40};
vector <int>v(arr,arr+4);
```

The Container Examples (Contd…)

- List:
  - The following example shows the use of list

```
#include<list>
int main()
{   list <int> ilist;
    ilist.push_back(30);
    ilist.push_back(40);
    ilist.push_front(20);
    ilist.push_front(10);
    int n=ilist.size();
    for(int i=0;i<n;i++){
        cout<<ilist.front()<<' ';
        ilist.pop_front();
    }
}
```

The Container Examples (Contd…)

- The [] operator is not defined with list container
- List traversal is possible with iterators only
- List is useful if frequent insertion and deletion is required in the middle of the list
- The usage of some useful functions is shown below

```
list <int> l1,l2;
l1.push_back(11);         //push 11 on back of list
l1.back();                //returns the last elements
l1.push_front(22);        //push 22 on front of list
l1.front();               //return front item
l1.pop_back();            //pop back item
l1.pop_front();           //pop front item
l1.reverse();             //reverse the list
l1.merge(l2)              //merge list l2 into l1
l1.unique();              //removes duplicate items
```

The Container Examples (Contd…)

- Deque:
  - The following example shows the use of deque

```
#include<deque>
int main()
{   deque <float> deq;
    deq.push_back(33.33);
    deq.push_back(44.44);
    deq.push_front(22.11);
    deq.push_front(11.11);
    cout<<"Elements in deque are: "<<endl;
    for(int i=0;i<deq.size();++i)
        cout<<deq[i]<<" ";
    cout<<"\nThe first element of deque is: "<<deq.front();
    cout<<"\nThe last element of deque is: "<<deq.back();
}
```

  - Deque also supports functions like `pop_front()`, `front()`.

The Container Examples (Contd…)

- Set:
  - The following example shows the use of set

```
#include<set>
int main()
{   string names[N] ={"Afganistan", "Bangaladesh", "Bhutan",
          "India", "Maldives", "Nepal","Pakistan","Srilanka"};
    string search;
    set<string>nameSet(names,names+N);
    set<string>::iterator iter;
    cout<<"Enter name of country:";  cin>>search;
    iter=nameSet.find(search);
    if(iter==nameSet.end())
        cout<<"The country "<<search<<" is not in SAARC";
    else
        cout<<"The country "<<search<<" is in SAARC";
}
```

The Container Examples (Contd…)

- Map:
  - The following example shows the use of map

```
#include<map>
int main()
{   string country[N] ={"Afganistan", "Bangaladesh", "Bhutan",
          "India", "Maldeevs", "Nepal","Pakistan","Srilanka"};
    string capital[N]={"Kabul", "Dhaka", "Thimpu",
     "New-Delhi", "Male", "Kathmandu", "Islamabad", "Colombo"};
    string name;
    map<string,string>nameMap;
    for(int i=0;i<N;i++)
        nameMap[country[i]]=capital[i];
    cout<<"Enter Country: "; cin>>name;
    cout<<"Capital of "<<name<<" is "<<nameMap[name];
}
```

The Container Examples (Contd…)

- Stack
  - The following example shows the use of the stack

```
#include<stack>
int main()
{   stack <int> s;
    s.push(11);
    s.push(22);
    s.push(33);
    cout<<"Items poped are: ";
    while(!s.empty()){
        cout<<s.top()<<" ";
        s.pop();
    }
    cout<<endl;
}
```

The Container Examples (Contd…)

- Stack container objects are instantiated as

```
stack <char>s1;
      //uses deque<char>to store elements of type char
stack <int,vector<int> > s2;
      //uses a vector <int> to store elements of type int
```

- It is possible to initialize the existing container to initialize a stack as:

```
vector <int> v;
//......
stack <int,vector<int> >st(v);
```

Iterators

- Iterators are pointer like entities that are used to access individual data items in a container.
- The iterators can be regarded as smart pointer which are templatized classes specially designed to point objects of different types.
- If an iterator points to one element in a range, then it is possible to increment or decrement the iterator so that it can points to the next or previous element respectively.
- Algorithms usually take iterators as arguments, so the elements of the containers can be accessed.
- There are generally five types of iterators which are tabulated as follows

Iterators (Contd…)

- Different types of iterators

Iterator	Access Method	Direction	I/O capability	Remarks
Output	Linear	Forward	Write Only	Cannot be saved
Input	Linear	Forward	Read Only	Cannot be saved
Forward	Linear	Forward	Read/Write	Can be saved
Bidirectional	Linear	Forward and Backward	Read/Write	Can be saved
Random Access	Random	Forward and Backward	Read/Write	Can be saved

- Different classes of iterators must be used with different types of containers.
- Only sequence and associative containers are transferable with iterators.
- Input and output iterators support least function and are only used to traverse in a container.
- Forward iterator supports all operations of input and output, and also retains its position in container.

Iterators (Contd…)

- Bidirectional iterators supports two ways traverse.
- Random Access Iterator supports accessing of elements of any location by making use of [ ] operator.
- The following figure illustrates the functionality of

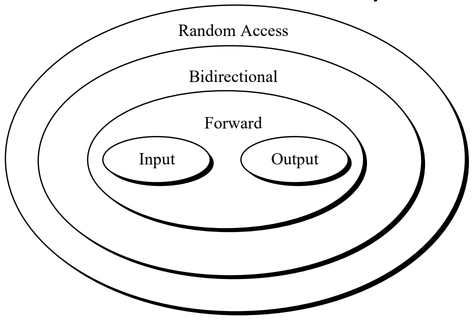


Figure: Relationship of all iterators

Iterators (Contd…)

- The operations supported by iterators are tabulated as follows

Iterator	Element Access	Read	Write	Increment Operation	Comparison
Output			*p=v	++	
Input	->	v=*p		++	==,!=
Forward	->	v=*p	*p=v	++	==,!=
Bidirectional	->	v=*p	*p=v	++,--	==,!=
Random Access	->,[]	v=*p	*p=v	++,--,+,-,+=,-=	==,!=,<,>,<,<=,>=

- Iterator makes programming easy by providing the feature like a smart pointer.
- In container vector and queue it is really very easy to do random access by using the [] operator.
- Same process of accessing is not possible in container list where random access is not possible.

Iterators (Contd…)

- Every container classes have their associated iterator types.
- The derived (adaptor) container have no iterators.
- The containers vector and deque have random access iterator as their default iterator.
- The container list and associative containers have bidirectional iterator as their default iterator.
- Any such iterator is obtained by declaring an object in a manner analogous to the declaration

```
vector<int>::iterator iter;
```

Iterator Example

- Following example shows the use of iterator to traverse through the list.

```
#include<list>
int main()
{   list <int> lst(5);
    list <int>::iterator itr;
    int value=0;
    for(itr=lst.begin();itr!=lst.end();++itr)
        *itr=++value;
    cout<<"Elements in the list are: ";
    itr=lst.begin();
    while(itr!=lst.end()){
        cout<<*itr<<" ";
        itr++;
    }
}
```

  - Iterators are frequently used with algorithms to access container elements.

Algorithm

- The algorithms are the part of standard template library that can be used across a variety of containers, basic data types and even user defined classes.
- STL algorithms are the global standalone templatized function.
- STL provides many algorithms that we can frequently be used to manipulate containers.
- Algorithms are used for various purposes such as inserting, deleting, searching, and sorting.
- The algorithms operate on container and array elements using iterators.
- Algorithms often return iterators that indicate the results of the algorithms.
- By using the rich collection of various common algorithms, we can save much time and programming effort.

Algorithm (Contd…)

- Algorithms are categorized as mutating and non-mutating algorithms.
- Those algorithms that changes the value of containers are called mutating algorithms and those that do not change the value are called non-mutating algorithms.
- Different algorithms and their uses are discussed below
- `find()`: This function finds first occurrence of a value in a sequence.
  - The following example illustrates the use of function `find()`.

```
#include<algorithm>
using namespace std;
int main()
{
    int array[]={10,20,30,10,20};
    int *ptr,num;
```

Algorithm (Contd…)

```
    cout<<"Enter number to find: ";
    cin>>num;
    ptr=find(array,array+5,num);
    if(ptr!=array+5)
             cout<<"Number "<<num<<" found in position "
            <<(ptr-array)<<endl;
    else
      cout<<num<<" is not in the array";
}
```

- When using algorithms with containers we use it as follows

```
int array[]= {10,20,30,10,20};
int num;
vector <int> v(array,array+5);
vector <int>::iterator itr;
```

Algorithm (Contd…)

```
cout<<"Enter number to find: ";
cin>>num;
itr=find(v.begin(),v.end(),num);
if(itr!=v.end())
   cout<<"Number "<<num<<" found in position "
       <<distance(v.begin(),itr)<<endl;
else
   cout<<num<<" is not in the array";
```

- The algorithm `distance()` returns the number of elements between first argument and second argument.
- When using it with list it can be used similar to vector as

```
list <int> lst(array,array+5);
list <int>::iterator itr;
itr=find(lst.begin(),lst.end(),num);
```

Algorithm (Contd…)

- `search()`:This function finds a sequence of value within the range.
  - It returns an iterator pointing to the beginning of that subsequence, or the last iterator if no such subsequence exists.
  - The following example shows the use of the function `search()`

```
#include<algorithm>
int main()
{   int array[]={10,20,30,10,20,80};
    int pat[]={20,30};
    int *ptr,num;
    ptr=search(array,array+6,pat,pat+2);
    if(ptr!=array+6)
        cout<<"Pattern matched at positions "<<(ptr-array);
    else
        cout<<"Pattern is not matched";
}
```

Algorithm (Contd…)

- For algorithm `search()` can be used with list as

```
list<int>::iterator itr;
itr=search(lst.begin(),lst.end(),pat.begin(),pat.end());
if(itr!=lst.end())
    //pattern matched
else
   //pattern not matched
```

- `count()`: This function counts the number of occurrence in a sequence and return the counted number.
  - The following code illustrates the use of function `count()`:

```
cin>>num;
n=count(array,array+6,num);
cout<<"Number "<<num<<" found "<<n<<" times"<<endl;
```

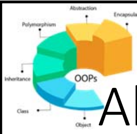
Algorithm (Contd…)

- `equal()`: This function returns true if two ranges are the same.
  - It is used as follow:

```
result=equal(array1,array1+size,array2);
```

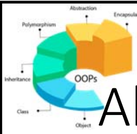
  - The function returns `1` if `array1` and `array2` are equal, otherwise `0`.
- `for_each()`: This algorithm allows to do some operation to every item in a container.
  - This function does not change the elements in the container but it can use or display the value.
  - Following example shows the use of function `for_each()`.

```
void num_to_char(int ch){
    cout<<ch<<"="<<static_cast<char>(ch)<<"\t";
}
for_each(arr,arr+26,num_to_char);
```



Algorithm (Contd…)

- `copy()`: This function copies one sequence into another.
  - The following example copies one list to another
    ```
    list<int> lst(arr,arr+5);
    //…
    list<int> lstcp(lst.size());
    copy(lst.begin(), lst.end(), lstcp.begin());
    ```
  - For array the copy algorithm is used as
    ```
    copy (arr, arr+5,arr2);
    ```
- `swap()`: The `swap()` functions are used to interchange between two objects in various conditions if any.
  - The function `swap()` exchanges two elements.
  - The following code shows the use of function `swap()`.
    ```
    swap(lst1,lst2);
    ```



Algorithm (Contd…)

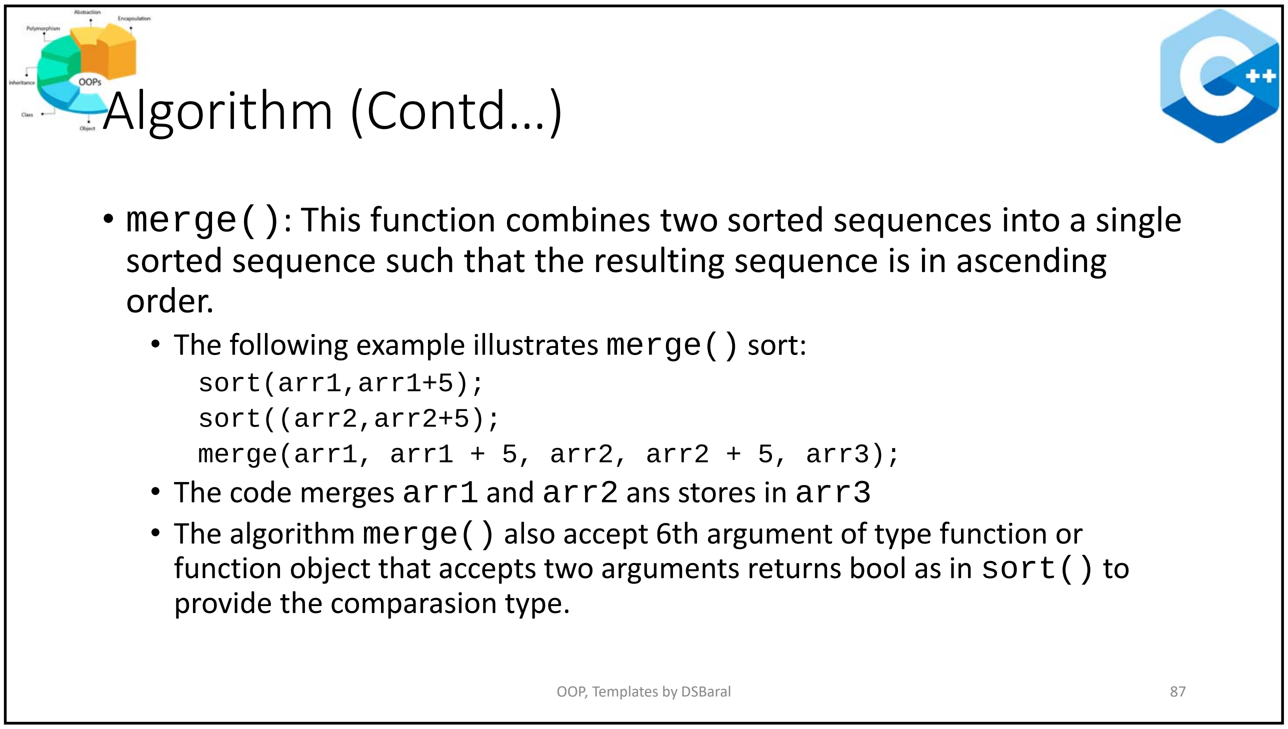
  - The function swap() can also be use to exchange two variables as
    ```
    int v1, v2;
    swap(v1,v2);
    ```
- `replace()`: The function related to replace are used to replace every elements in the range under various terms and condition if any.
  - The function replace() replaces each elements with specified value.
  - Following code shows the use of function `replace()`.
    ```
    replace(v.begin(), v.end(), 1, 99);
    ```
  - This code replaces each occurrence of `1` by `99`.
- `fill()`: This function populates the sequence with a specified value.
  - The following code segment illustrates the use of function `fill()`.
    ```
    vector<int> v(3);
    fill(v.begin(), v.end(),5);
    ```
  - The above statement fills the all the location of vector `v` with value `5`.

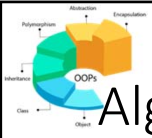
Algorithm (Contd…)

- `remove()`: The functions remove all elements that matches the condition.
  - The function `remove()` deletes all elements of a specified value.
  - The following code shows the use of remove.
    ```
    remove(v.begin(), v.end(), 1);
    ```
  - The code removes all elements having value `1` from the vector `v`.
- `unique()`: This algorithm removes all duplicate elements except the first one.
  - The following code snippet shows the use of `unique()`.
    ```
    unique(v.begin(),v.end());
    ```
  - This statement removes duplicate elements from vector `v`.

Algorithm (Contd…)

- `reverse()`: This function reverses the order of elements.
  - The use of the function is shown below.
    ```
    reverse(v.begin(), v.end());
    ```
  - This statement reverses the elements of the vector `v`.
- `sort()`: This algorithm sorts the range specified in ascending order.
  - The use of the function is shown below
    ```
    sort(arr, arr+5);//sorts array in ascending order
    ```
  - Or for container it is used as below
    ```
    sort(v.begin(),v.end()); //sorts vector in ascending order
    ```
  - We can also provide sort criteria as function of function object as third argument as
    ```
    bool myfunc(int i,int j) { return (i>j); }
    sort(v.begin(), v.end(), myfunc); //sorts in descending order
    ```





Algorithm (Contd…)

- `merge()`: This function combines two sorted sequences into a single sorted sequence such that the resulting sequence is in ascending order.
  - The following example illustrates `merge()` sort:

    ```
    sort(arr1,arr1+5);
    sort((arr2,arr2+5);
    merge(arr1, arr1 + 5, arr2, arr2 + 5, arr3);
    ```
  - The code merges `arr1` and `arr2` ans stores in `arr3`
  - The algorithm `merge()` also accept 6th argument of type function or function object that accepts two arguments returns bool as in `sort()` to provide the comparasion type.