OOP

TEMPLATES

CHAPTER 10

KABIN DEVKOTA

UDITA BISTA

TABLE OF CONTENTS

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

TEMPLATES

Templates are a relatively new addition to C++. They allow you to write generic classes and functions that work for several different data types. The result is that you can write generic code once and then use it over again for many different uses. C++ uses the templates to enable generic techniques. Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

FUNCTION TEMPLATES

```
int sum( int a, int b)
{
 return (a+b);
}
```

The above code is an example of a normal function. It takes two integer values, sums them and returns the sum. However, If we try to call the sum function as sum(2.3,4.5), the compiler will throw an error as the sum function is designed to work on integer values only. We cannot pass other data types to the function. To make the above function flexible, C++ has introduced the concept of templates.
In C++, function overloading allows defining multiple functions with the same names but with different arguments. In a situation, where we need to define functions with the same body but of different data types we can use function overloading. But while overloading a function, different data types will need different functions, and hence the function needs to be rewritten several times for each data type. This is time consuming and space consuming.

FUNCTION TEMPLATES

So C++ introduced the new concept of function template. Function templates are special functions that can operate with generic types. It is the function which is written just for once, and have it worked for many different data type. The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, float, etc., but as a name that can stand for any type.

```
template <class T>
return_type function_name(T type argument){
// body
 }
```

The template keyword informs the compiler that we’re about to define a function template. The keyword class, within angle brackets, acts as the type. We can define our own data type using classes, so there’s no distinction between types and classes. The variable following keyword class (‘T’ in above syntax) is called the template argument.

FUNCTION TEMPLATES

```
#include <iostream>
using namespace std;
template <class T1>
T1 sum(T1 a, T1 b)
{
      return (a+b);
}
```

```
int main()
 {
      cout << sum(10, 20) << endl; //calling the function the function template sum for the integer values
       cout << sum(11.5, 13.1) << endl; //calling the function the function template sum for the float values

      return 0;
 }
```

In the above example, sum() is a template function. It takes two arguments of T1 type and returns a T1 type data equal to the sum of two arguments. Defining a function as a template doesn't generate code for different types, it is just a message to the compiler that the function can be called with different data types. Thus, in main(), when sum(10,20) is called, the template generates a function by substituting template argument (T1) with int. This is called instantiating the function template, and each instantiating version of the function is called template function. When sum(11.5,13.1) is invoked, instantiating of the template by float takes place and another template function of float type is created. Here the function template sum can take arguments of any type(int, float,char etc....)

FUNCTION TEMPLATE WITH MULTIPLE ARGUMENTS

We can use more than one generic type in a function template. They are declared as a comma-separated list within the template specification.

```
template <class T1, class T2, ...>
 returntype functionname(arguments of type T1, T2, T3...)
{
 // body
 }
```

OVERLOADING TEMPLATES

Overloading with function templates

```
#include<iostream>
using namespace std;
//template
template<class T>
T find_max( T a, T b)
{
   if(a>b)
      return a;
   else
      return b;
}
```

```
//normal function having same name as template
float find_max(float x, float y)
{
      if(x>y)
      return x;
      else
      return y;
}
```

```
int main()
{
   int p = 15, q = 12,r;
   float c = 18.9, d = 25.009,e;
   r = find_max(p,q);  //template find_max is called.
   e = find_max(c,d);  //normal function find_max is called.
   cout<<"The greatest integer value is "<<r<<endl;
   cout<<"The greatest float value is "<<e;
   return 0;
}
```

Output:

```
The greatest integer value is 15
The greatest float value is 25.009
```

In the above example, the find_max() function is overloaded. Two functions of same name and same arguments are declared, one as a function template and other of type float. The function template generates functions for every data type but nontemplate functions take precedence over template function. Thus, find_max(c,d) invokes normal function as c and d are of type float and find_max(x,y) invokes template function.

OVERLOADING TEMPLATES

Overloading with other template

```
#include<iostream>
using namespace std;
template<class T>
void display( T p )
{
   cout<< p <<endl;
}
/ overloading template with other template having two arguments
template<class T>
void display( T p, int q)
{
 for(int i = 0;i < q; i++)
 cout<< p <<endl;
}
```

```
int main()
{
   display(5);  // the first template "display" is called for integer value.
   display("hey", 3); //the second template "display" is called for string.
   display(10.8,2);  //the second template "display" is called for float value.
}
```

Output:

```
5
hey
hey
hey
10.8
10.8
```

The above example has two template functions that are overloaded. The display() function chosen depends on number of arguments as the two functions are distinguished by the number of arguments. Thus display(5) uses the first template function whereas display("hey", 3) and display(10.8,2) uses the second template function.

CLASS TEMPLATES

Class template is a template used to generate template classes. Class templates are used to construct classes having the same functionality for different data types. A class template provides specification for generating classes based on parameters.

```
template <class T>
 class classname
 {
  /*class member specification with anonymous type T wherever appropriate*/
};
```

Template class is an instance of a class template as a template function is an instance of a function template.

Syntax(for defining the object of template class):

```
classname<type> objectname;
```

CLASS TEMPLATES

Function Definition of Class Template

If the functions are defined outside the template class body, they should always be defined with the full template definition. They must be written like normal functions with scope resolution operator.

```
template <class T>
 returntype classname <T>::functionname(arglist)
{
 // body of function
 }
```

CLASS TEMPLATES

Function Definition of Class Template

```
#include<iostream>
using namespace std;
template<class T1>
class numbers
{
   T1 a, b;
   public:
   numbers( T1 x, T1 y)
   {
      a = x;
      b = y;
   }
   T1 getmax();
};
```

```
template<class T>
T numbers<T>::getmax()   //getmax() must be defined outside the class using full template definition
{
   if(a>b)
   {
      return a;
   }
   else
   {
      return b;
   }

int main()
{
 numbers<int> obj1(5,9); //obj1 is of type int
 numbers<float> obj2(3.4,1.5); //obj2 is of type float
 int x = obj1.getmax();
 float y = obj2.getmax();
 cout<<"The greatest integer value is "<<x<<endl;
 cout<<"The greatest float value is "<<y<<endl;
 return 0;
}
```

Output:

```
The greatest integer value is 9
The greatest float value is 3.4
```

CLASS TEMPLATES

Class Template with multiple arguments

Like in function template, we can use more than one generic type in class template. Each type are declared in Template specification, and they are separated by comma operator.

Syntax (for defining class template):

template <class T1, class T2, … >
class classname
{
//body of class
};

Syntax (for defining object of the class):

template <type1, type2, … > objectname;

Syntax (for defining member function outside class):

template <class T1, class T2, … >
returntype classname<T1, T2, …>::functionname (argumentlist)
{
// body of function
}

CLASS TEMPLATES

Function Definition of Class Template

```
#include <iostream>
using namespace std;
template<class T1, class T2>
class Record
 {
 T1 a;
 T2 b;
public:
  Record(T1 x, T2 y)
 {
  a = x;
  b = y;
 }
 void show();
};
```

```
template<class T1, class T2>
void Record<T1,T2>::show()
{
 cout << a << " and " << b << endl;
}
int main()
 {
 Record<int, char> Obj1(1,'A');
 Record<float, char> Obj2(12.3,'B');
 Obj1.show();
  Obj2.show();
 return 0;
}
```

Output:

```
1 and A
12.3 and B
```

The class ‘Record’ can take two generic types to store the records of the two types. Obj1 stores an int and a char and Obj2 stores a float and a char.

NON-TEMPLATE TYPE ARGUMENTS

A nontype template argument provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions. addresses of functions or objects with external linkage, or addresses of static class members. Nontemplate arguments are normally used to initialise a class or to specify the sizes of class members.

Syntax:

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

NON-TEMPLATE TYPE ARGUMENTS

```
#include <iostream>
using namespace std;
template <class T, int s>
class Array
 {
       T a[s];
       public:
             void input()
             {
                     cout << "enter numbers " << endl; for(int i=0; i<s; i++)
                    cin >> a[i];
             }
              void display();
};
```

```
template <class T, int s>
void Array<T, s>::display()
{
       for(int i=0; i<s; i++)
        cout << a[i] << endl;
}
int main()
{
       Array<int,4>a1;
       a1.input();
       a1.display();
       Array<float,3>a2;
       a2.input();
       a2.display();
       return 0;
}
```

Output:

```
enter numbers
2 3 4 5
2
3
4
5
enter numbers
2.5 3.2 2
2.5
3.2
2
```

NON-TEMPLATE TYPE ARGUMENTS

Default Arguments with Class Template

Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrates this:

```
template<class T = int, int size = 10>
class Array
{
   //body
};
int main()
{
 //....
 Array<float, 5> ftarray; //float array with size 5
 //....
 Array<double> darray; //double array with the default size of 10
 //....
 Array<> intarr; //default integer array with the default size of 10
 //....
}
```

NON-TEMPLATE TYPE ARGUMENTS

Default Arguments with Class Template

// WAP to show the implementation of default argument with class template

```
 #include <iostream>
 using namespace std;
 template <class T=float, int s=2>
 class Array
 {
       T a[s];
      public:
       void input();
       void sum();
};
template <class T, int s>
void Array<T,s>::input()
{
       cout << "enter numbers " << endl;
       for(int i=0; i<s; i++)
        cin >> a[i];
}
```

```
template <class T, int s>
void Array<T, s>::sum()
{
       T sum = 0;
        for(int i=0; i<s; i++)
        sum += a[i];
        cout << "The sum is : " << sum << endl;
}
int main()
 {
       cout << "For integer array of size 5" << endl;
       Array<int,5> a1;
        a1.input();
        a1.sum();
       cout << "For default values" << endl;
       Array<>a2; //by default a2 object contains a float array with the size of 2
        a2.input();
        a2.sum();
       return 0;
}
```

Output:

```
enter numbers
1 2 3 4 5
The sum is : 15
For default values
enter numbers
1.1
2.2
The sum is : 3.3
```

DERIVED CLASS TEMPLATE

1. We can create a derived class which is a non-template class from a base class which is a template class.
2. We can create a derived class which is a template class from a base class which is not a template class.
3. We can create derived class which is a template class from a base class which is also a template class with the same template parameters as in the base class
4. We can create a derived class which is a template class from a base class which is also a template class with additional template parameters in the derived class than that of the base class

DERIVED CLASS TEMPLATE

1. If we don't add extra template parameter and supply the template argument of base class with data type, we create a non-template derived class as:

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
   base(){}
   base(T a){data = a;}
   void display()
 {
 cout<<"data: "<<data<<endl;
 }

};
```

```
class derived1: public base<int>
{
 public:
 derived1(){}
 derived1(int a): base<int>(a){}
};
int main()
{
 derived1 obj1(5);
 obj1.display();
}
```

DERIVED CLASS TEMPLATE

2. If we add extra template parameter and supply the template argument of base class with data type, we create a derived class template as:

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
   base(){}
   base(T a){data = a;}
   void display()
 {
 cout<<”data: ”<<data<<endl;
 }

};
```

```
template <class T>
class derived2: public base<int>
{
 public:
 derived2(){}
 derived2(int a, T b): base<int>(a),data(b){}
 void display()
 {
  cout<<”in base”;
  base<int>::display();
  cout<<””in derived, data: ”<<data<<endl;
 }
};
int main()
{
 derived2<float> obj2(10, 12.34);
 obj2.display();
}
```

DERIVED CLASS TEMPLATE

3. If the base class template parameter is still useful in derived class, the derived class is created as class template i.e., base and derived template classes have the same template parameter.

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
   base(){}
   base(T a){data = a;}
   void display()
 {
 cout<<”data: ”<<data<<endl;
 }

};
```

```
template <class T>
class derived3: public base<T>
{
 public:
 derived3(){}
 derived3(T a): base<T>(a){}
};
int main()
{
 derived3<int> obj3(5);
 obj3.display();
}
```

DERIVED CLASS TEMPLATE

4. We can also add an extra template parameter in the derived class along with the base class template parameter

```
#include<iostream>
using namespace std;
template <class T>
class base
{
 T data;
 public:
   base(){}
   base(T a){data = a;}
   void display()
 {
 cout<<”data: ”<<data<<endl;
 }

};
```

```
template <class T1, class T2>
class derived4: public base<T1>
{
 T2 data;
 public:
 derived4(){}
 derived4(T1 a, T2 b): base<T1>(a),data(b){}
 void display()
 {
  cout<<”in base”;
  base<T1>::display();
  cout<<””in derived, data: ”<<data<<endl;
 }
};
int main()
{
 derived4<int. float> obj4(10, 12.34);
 obj4.display();
}
```

DERIVED CLASS TEMPLATE

5. The derived class template can be created from the base class which is not a class template. In this case, a template parameter is added in the derived class during inheritance.

```
#include<iostream>
using namespace std;
class base
{
 int data;
 public:
   base(){}
   base(int a){data = a;}
   void display()
 {
 cout<<”data: ”<<data<<endl;
 }

};
```

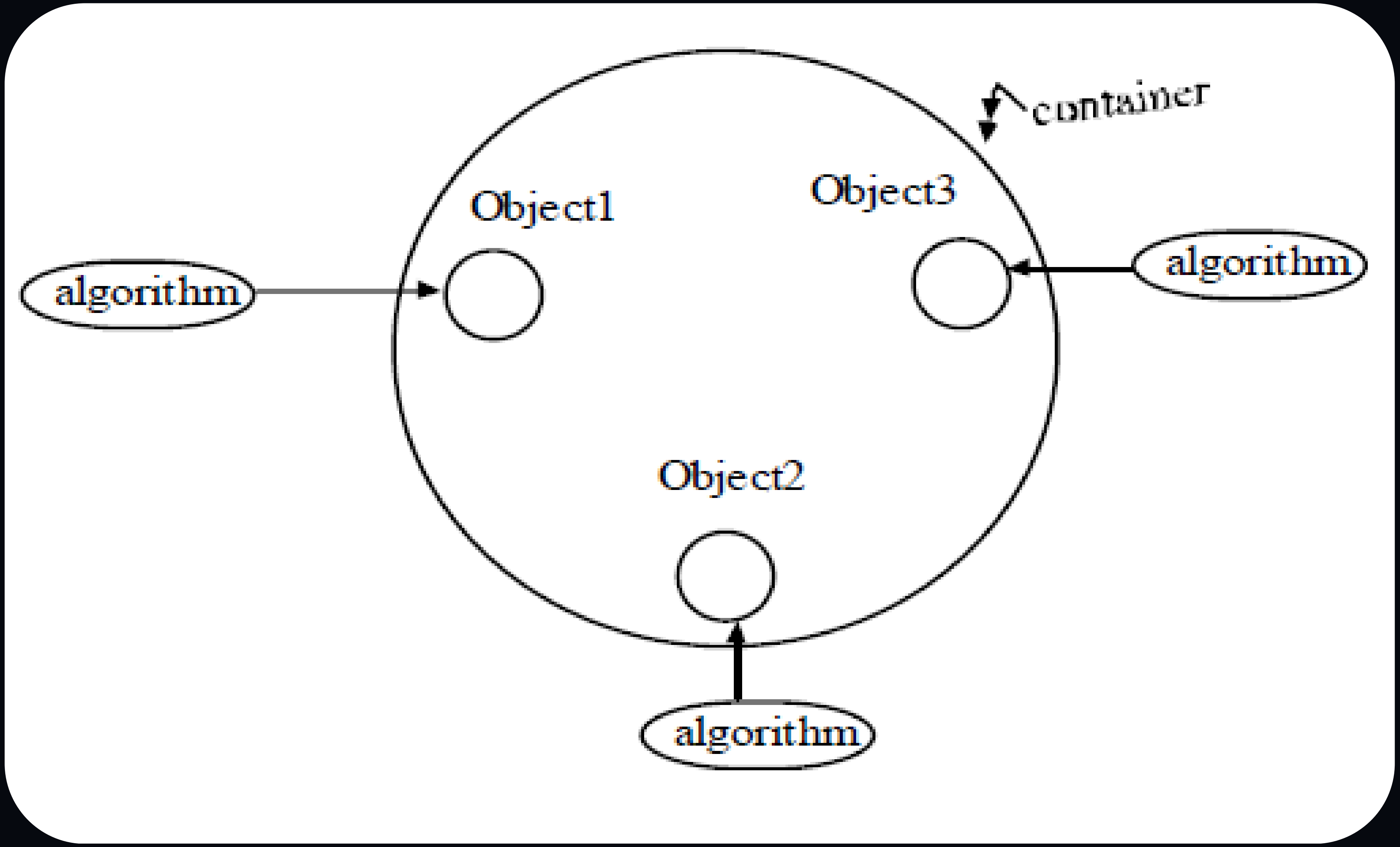
```
template <class T>
class derived5: public base
{
 public:
 derived5(){}
 derived5(int a, T b): base(a),data(b){}
 void display()
 {
  cout<<”in base”;
  base::display();
  cout<<””in derived, data: ”<<data<<endl;
 }
};
int main()
{
 derived5<float> obj5(25, 10.5);
 obj5.display();
}
```

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

We learned about generic programming by using template class and template function in the early section. During the standardization of C++, Standard Template Library(STL) was included. It provides general purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, for example, vector, stack queue, list map, etc. As the STL is constructed from template classes, the algorithm and data structure can be applied to any type of data.

STL is large and complex and it is difficult to discuss all the features. STL components that are now part of the Standard C++ library are defined in the namespace std. So we should use this directive to use STL.

The STL contains several components. But in core, it contains three fundamental components: containers, algorithms, and iterators. Their relationship is shown in the figure



INTRODUCTION TO STANDARD TEMPLATE LIBRARY

1. Containers

Containers are objects that hold other objects. It is a way in which data is organized in memory. The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data. The STL defines ten containers that are grouped into three categories:

a) Sequence containers

The sequence container is a variable-sized container whose elements are arranged in a strict linear order.It supports insertion and removal of elements. Every data, user defined or built-in, has a specific position in the container. Sequence containers store elements in a linear list. Each element is related to one other element by its position along the line. They are decided into the following:

i) Vector
ii) List
iii) Dequeue

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

b) Associative Containers

An associative container is a variable-sized container that supports efficient retrieval of elements based on keys. Like sequence container, it supports insertion and removal of elements, but differs from a sequence in that it does not provide a mechanism for inserting an element at a specific position. They are not sequential. There are four types of associative containers:

i) Set
ii) Multiset
iii) Map
iv) Multimap

c) Derived Containers

These containers are derived from sequential container. These are also known as container adaptors. The STL provides three derived containers:

i) Stack
ii) Queue
iii) Priority_queue

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

2. Algorithms

An algorithm is a procedure that is used to process the data contained in the containers. STL provides more than sixty standard algorithms to support more extended or complex operations. Standard algorithms also permit us to work with two different types of containers at the same time. STL algorithms are not member functions or friends of containers. They are standalone template functions. STL algorithm based on the nature of operations they perform may be categorized as:

a) Mutating algorithms
b) Sorting algorithms
c) Set algorithms
d) Relational algorithms
e) Retrieve or non mutating algorithms

INTRODUCTION TO STANDARD TEMPLATE LIBRARY

3. Iterators

Iterators behave like pointers and are used to access individual elements in containers. They are often used to traverse from one element to another, a process known as iterating through the container. That means if the iterator points to one element in the range then it is possible to increase or decrease the iterator so that we can access the next element in the range. Iterators connect algorithms and play a key role in the manipulation of data stored in the containers.

There are five types of iterators: input, output, forward, bidirectional and random.

THANK YOU