Templates

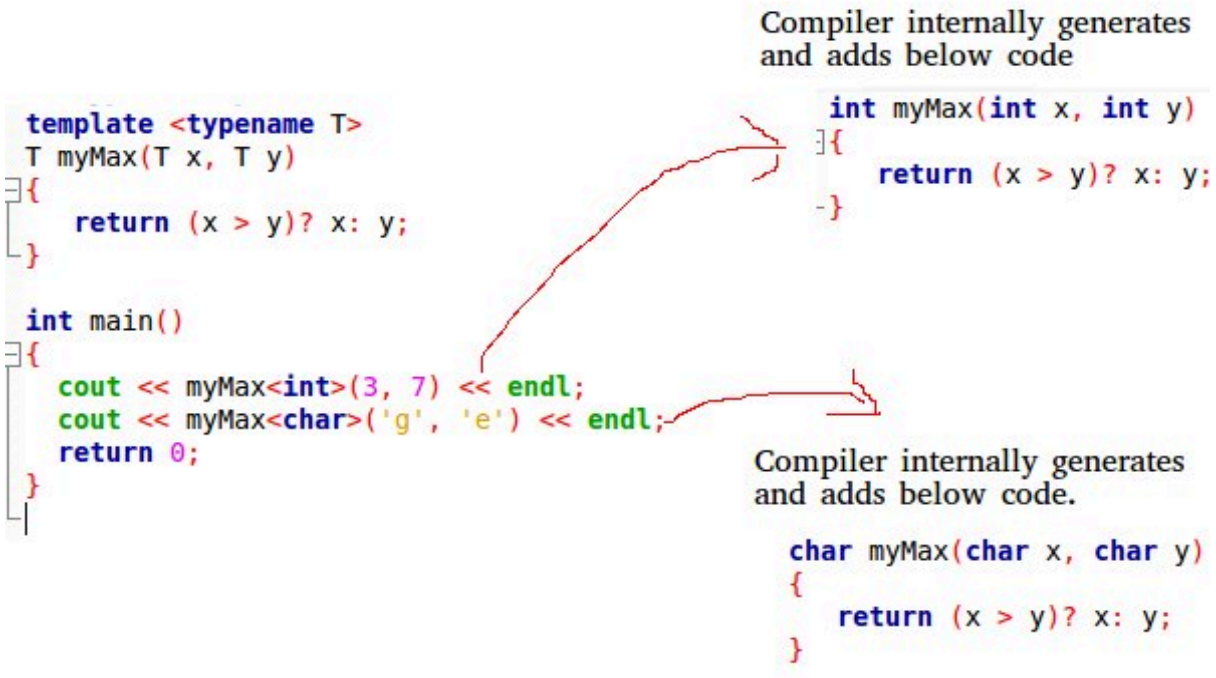
- A **template** is a simple yet very powerful tool in C++
- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.
- A template is a blueprint or formula for creating a generic class or a function.
- The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.
- C++ adds two new keywords to support templates: ***'template'*** and ***'type name'***. The second keyword can always be replaced by the keyword **'class'**.

Templates can be represented in two ways:

- **Function templates**
- **Class templates**

How Do Templates Work?

- Templates are expanded at compile time.
- This is like macros.
- The difference is, that the compiler does type-checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.



template < class Ttype> ret_type func_name(parameter_list)

{

// body of function.

}

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

Function Templates

- We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().
- Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Function Templates with Multiple Parameters

- We can use more than one generic type in the template function by using the comma to separate the list.

template<**class** T1, **class** T2,.....>

return_type function_name (arguments of type T1, T2....)

{

// body of function.

}

The typename and class are keywords used in templates in C++. There is no difference between the typename and class keywords. Both of the keywords are interchangeably used by the C++ developers as per their preference. There is no semantic difference between class and typename in a type-parameter-key

```
#include <iostream>

using namespace std;

template<class X,class Y> void fun(X a,Y b)

{

   std::cout << "Value of a is : " <<a<< std::endl;

   std::cout << "Value of b is : " <<b<< std::endl;

}

int main()

{

  fun(15,12.3);


  return 0;

}
```

Overloading a Function Template

- We can overload the generic function means that the overloaded template functions can differ in the parameter list.

E.g. fun(10);

fun(20,30.5);

Restrictions of Generic Functions

- Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

```
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}
void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }
```

Class Templates

- **Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.
- Class templates like function templates, class templates are useful when a class defines something that is independent of the data type.
- Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
template<class Ttype>
class class_name
{ ….
}
```

- **Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.
- Now, we create an instance of a class
- class_name<type> ob;
- where **class_name**: It is the name of the class.
- **type**: It is the type of the data that the class is operating on.
- **ob**: It is the name of the object.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

```
template<class T1, class T2, ......>
class class_name
{
  // Body of the class.
}
```

Nontype Template Arguments

- The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

```
template<class T, int size>
class array
{
     T arr[size];          // automatic array initialization.
};
```

- Here the nontype template argument is size

Arguments are specified when the objects of a class are created

- Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template.
- The important thing to note about non-type parameters is, that they must be **const**.

- The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time.

default value for Template arguments

- **Can we specify a default value for template arguments?**
- Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

  template <class T, class U = char> class A
- **What is the difference between function overloading and templates?**
- Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

Template Specialization

- We write code once and use it for any data type including user defined data types.
- For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.
- *What if we want a different code for a particular data type?*
- Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?
- *It is possible in C++ to get a special behavior for a particular data type. This is called template specialization*.

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
   // code to implement Quick Sort
}

// Template Specialization: A function
// specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
```

```
    // code to implement counting sort
}
```

- ***How does template specialization work?***
- When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).
- If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Template Argument Deduction

- Template argument deduction automatically deduces the data type of the argument passed to the class or function templates. This allows us to instantiate the template without explicitly specifying the data type.
- For example, consider the below function template to multiply two numbers:

```
template <typename t>
t multiply (t num1,t num2) { return num1*num2; }
```

In general, when we want to use the multiply() function for integers, we have to call it like this:

- multiply<int> (25, 5);

But we can also call it:

- multiply(23, 5);
- We don’t explicitly specify the type ie 1,3 are integers.
- The same is true for the template classes(since C++17 only).
- Suppose we define the template class as:

```
template<typename t>
class student{
   private:
     t total_marks;
   public:
     student(t x) : total_marks(x) {}
};
```

- If we want to create an instance of this class, we can use any of the following syntax:
- student<int> stu1(23);

- or
- student stu2(24);
- ***Note:*** *It is important to note the the template argument deduction for a classes is only available since C++17*

The C++ Standard Template Library (STL)

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as vectors, lists, stacks, arrays, etc.
- The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.
- It is a library of container classes, algorithms, and iterators.
- All the three components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.
- It is a generalized library and so, its components are parameterized.
- One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.
- The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.
- **Some of the key components of the STL include:**
- Containers: The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.
- Algorithms: The STL provides a range of algorithms, such as sort, find, and binary_search, which can be used to manipulate data stored in containers.
- Iterators: Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward_iterator, bidirectional_iterator, and random_access_iterator, that can be used with different types of containers.
- Function Objects: Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.
- Adapters: Adapters are components that modify the behavior of other components in the STL. For example, the reverse_iterator adapter can be used to reverse the order of elements in a container.
- By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.

- **STL has 4 components:**
- **Algorithms**
- **Containers**
- **Functors**
- **Iterators**

Algorithms

- The header algorithm defines a collection of functions specially designed to be used on a range of elements. They act on containers and provide means for various operations for the contents of the containers.

Algorithm

- Sorting -
- Searching
- Important STL Algorithms
- Useful Array algorithms
- Partition Operations
- Numeric
- Sorting - There is a builtin function in C++ STL by the name of sort().
    - sort(startaddress, endaddress)
- Searching - Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.

binary_search(startaddress, endaddress, valuetofind)

- STL has an ocean of algorithms, Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows :
- **Non-Manipulating Algorithms**
- sort(first_iterator, last_iterator) – To sort the given vector.
- sort(first_iterator, last_iterator, greater<int>()) – To sort the given container/vector in descending order
- reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
- *max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
- *min_element (first_iterator, last_iterator) – To find the minimum element of a vector.

- accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements

Containers

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.
- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
- **Sequence containers -** Sequence containers implement data structures that can be accessed sequentially
    - array: Static contiguous array (class template)
    - vector: Dynamic contiguous array (class template)
    - deque: Double-ended queue (class template)
    - forward_list: Singly-linked list (class template)
    - list: Doubly-linked list (class template)
- **Associative containers -**Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).
    - Set: Collection of unique keys, sorted by keys (class template)
    - Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).
    - multiset: Collection of keys, sorted by keys (class template)
    - multimap: Collection of key-value pairs, sorted by keys (class template)
- **Unordered associative containers -** Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched.
    - unordered_set: Collection of unique keys, hashed by keys. (class template)
    - unordered_map: Collection of key-value pairs, hashed by keys, keys are unique. (class template)
    - unordered_multiset: Collection of keys, hashed by keys (class template)
    - unordered_multimap: Collection of key-value pairs, hashed by keys (class template)
- **Container adapters -** Container adapters provide a different interface for sequential containers.
    - stack: Adapts a container to provide stack (LIFO data structure) (class template).
    - queue: Adapts a container to provide queue (FIFO data structure) (class template).
    - priority_queue: Adapts a container to provide priority queue (class template).

Functors

- The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.
- Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?
- One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.
- **Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs

Iterators

- As the name suggests, iterators are used for working on a sequence of values. They are the major feature that allows generality in STL
- An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer.
- Now each one of these iterators are not supported by all the containers in STL, different containers support different iterators, like vectors support Random-access iterators, while lists support bidirectional iterators. The whole list is as given below:
- Types of iterators: Based upon the functionality of the iterators, they can be classified into five major categories:
- Input Iterators: They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.
- Output Iterators: Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
- Forward Iterator: They are higher in the hierarchy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.
- Bidirectional Iterators: They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions, that is why their name is bidirectional.

- Random-Access Iterators: They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.

Benefits of Iterators

- There are certainly quite a few ways which show that iterators are extremely useful to us and encourage us to use it profoundly. Some of the benefits of using iterators are as listed below:
- **Convenience in programming:** It is better to use iterators to iterate through the contents of containers as if we will not use an iterator and access elements using [ ] operator, then we need to be always worried about the size of the container, whereas with iterators we can simply use member function end() and iterate through the contents without having to keep anything in mind.
- A vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows −
- Functions used in eg contEg1 prog

The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.

The size( ) function displays the size of the vector.

The function begin( ) returns an iterator to the start of the vector.

The function end( ) returns an iterator to the end of the vector.

- The C++ Standard Library can be categorized into two parts −
- The Standard Function Library − This library consists of general-purpose,stand-alone functions that are not part of any class. The function library is inherited from C.
- The Object Oriented Class Library − This is a collection of classes and associated functions.
- I/O,
- String and character handling,
- Mathematical,
- Time, date, and localization,
- Dynamic allocation,
- Miscellaneous,
- Wide-character functions,

The Object Oriented Class Library

- Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following −
- The Standard C++ I/O Classes

- The String Class
- The Numeric Classes
- The STL Container Classes
- The STL Algorithms
- The STL Function Objects
- The STL Iterators
- The STL Allocators
- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library

RTTI (Run-Time Type Information) in C++

- In C++, **RTTI (Run-time type information)** is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.
- **Runtime Casts -** checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:
- **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
- **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.
- **Using '*dynamic_cast*':** In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.