Week 15: Generic Programming

By, Mahrukh Khan

Generic Programming

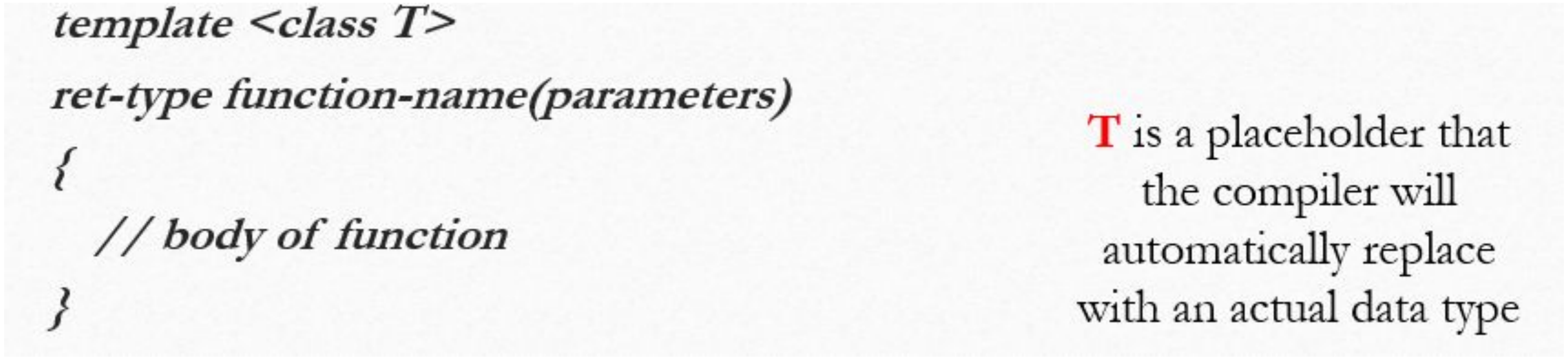
- Generic Programming approach permits writing common functions/**classes** that differ in the types of values they operate upon when used.The idea is to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- The method of Generic Programming 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.

Advantages

- Code Reusability
- Avoid Function Overloading
- Once written it can be used for multiple times and cases.

Templates

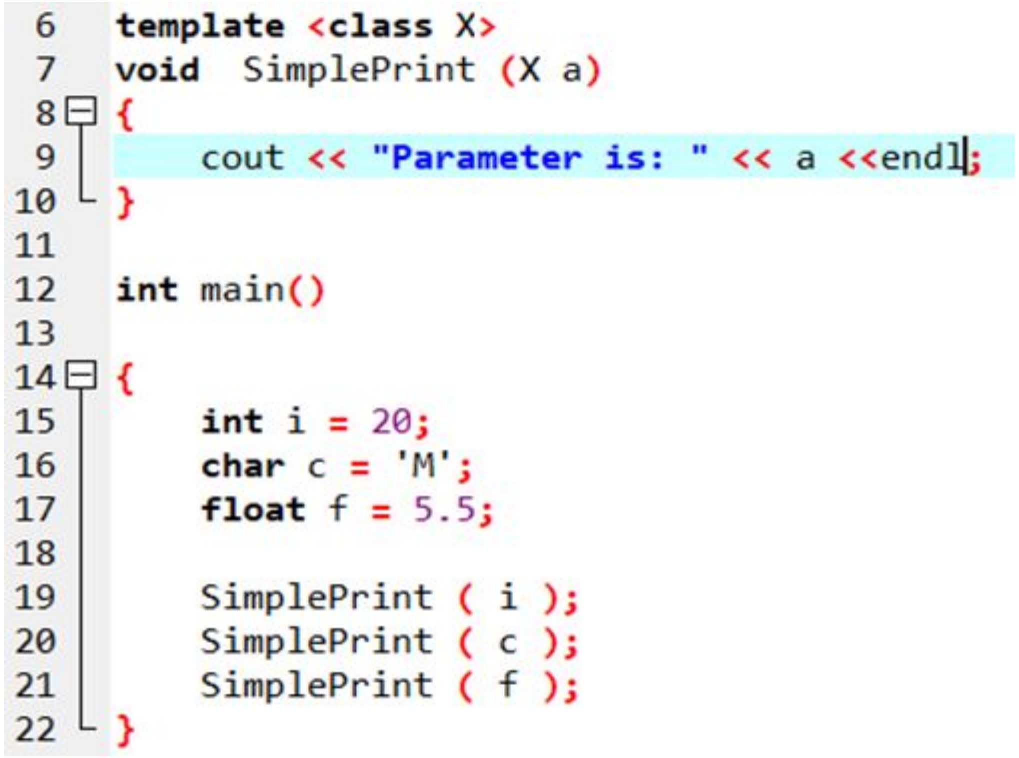
- Generics can be implemented in C++ using **Templates**.
- The general form of a template function definition is:



```
template <class T>
ret-type function-name(parameters)
{
   // body of function
}
```

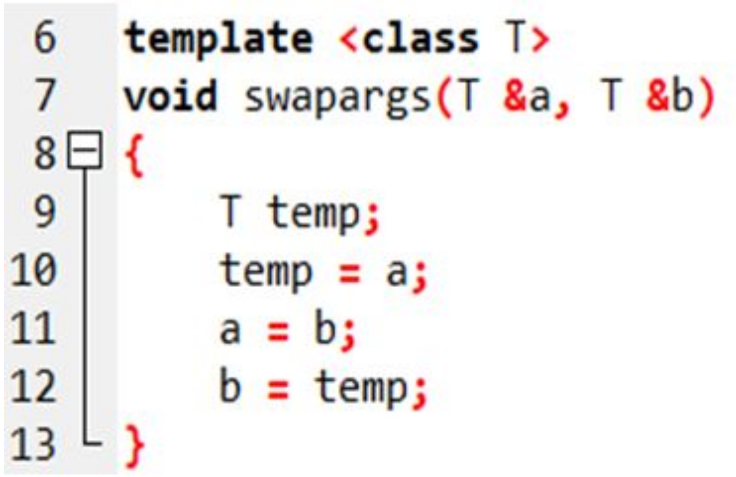
T is a placeholder that the compiler will automatically replace with an actual data type

Example

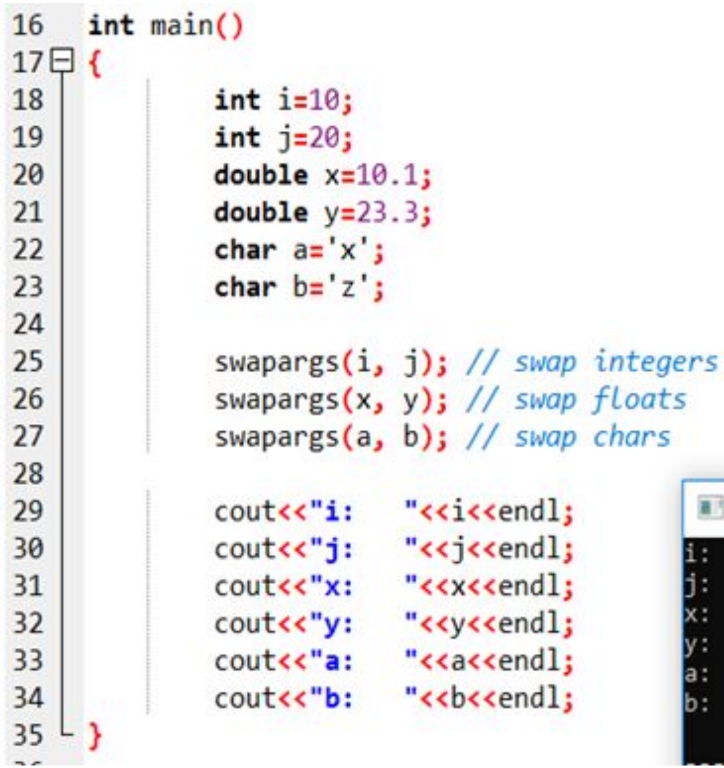


```
template <class X>
void  SimplePrint (X a)
{
    cout << "Parameter is: " << a <<endl;
}

int main()

{
    int i = 20;
    char c = 'M';
    float f = 5.5;

    SimplePrint ( i );
    SimplePrint ( c );
    SimplePrint ( f );
}
```

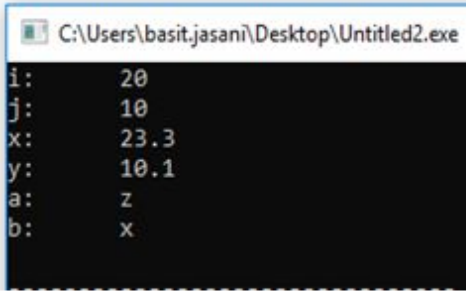
Example



```cpp
template <class T>
void swapargs(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```



```cpp
int main()
{
        int i=10;
        int j=20;
        double x=10.1;
        double y=23.3;
        char a='x';
        char b='z';

        swapargs(i, j); // swap integers
        swapargs(x, y); // swap floats
        swapargs(a, b); // swap chars

        cout<<"i:   "<<i<<endl;
        cout<<"j:   "<<j<<endl;
        cout<<"x:   "<<x<<endl;
        cout<<"y:   "<<y<<endl;
        cout<<"a:   "<<a<<endl;
        cout<<"b:   "<<b<<endl;
}
```

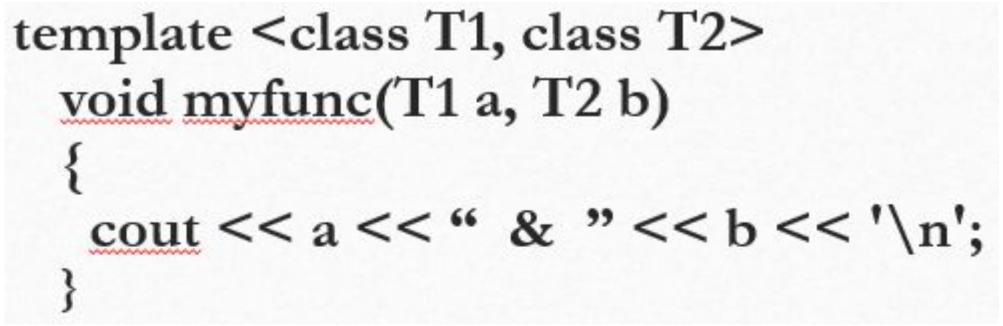


C:\Users\basit.jasani\Desktop\Untitled2.exe

```
i:      20
j:      10
x:      23.3
y:      10.1
a:      z
b:      x
```

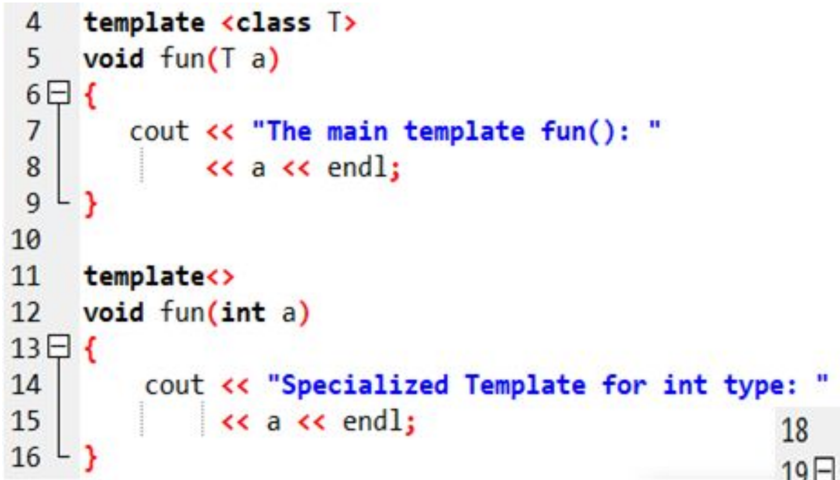
Template Function with two Generic types

- You can define more than one generic data type in the template statement by using a comma-separated list

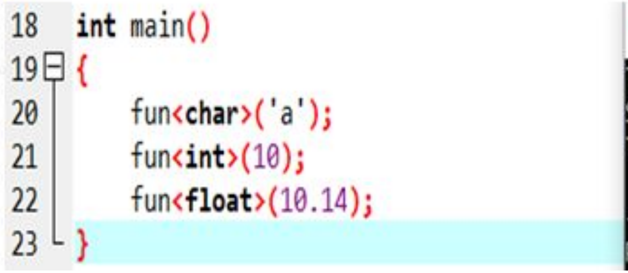


```
template <class T1, class T2>
  void myfunc(T1 a, T2 b)
  {
   cout << a << "  &  " << b << '\n';
  }
```

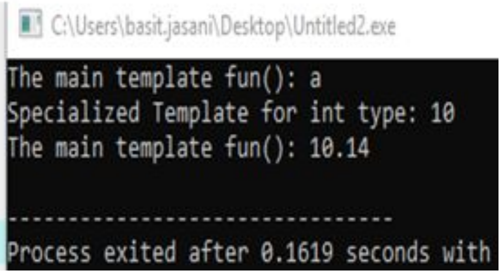
Specialized Template



```
template <class T>
void fun(T a)
{
    cout << "The main template fun(): "
         << a << endl;
}

template<>
void fun(int a)
{
     cout << "Specialized Template for int type: "
          << a << endl;
}
```



```
int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(10.14);
}
```

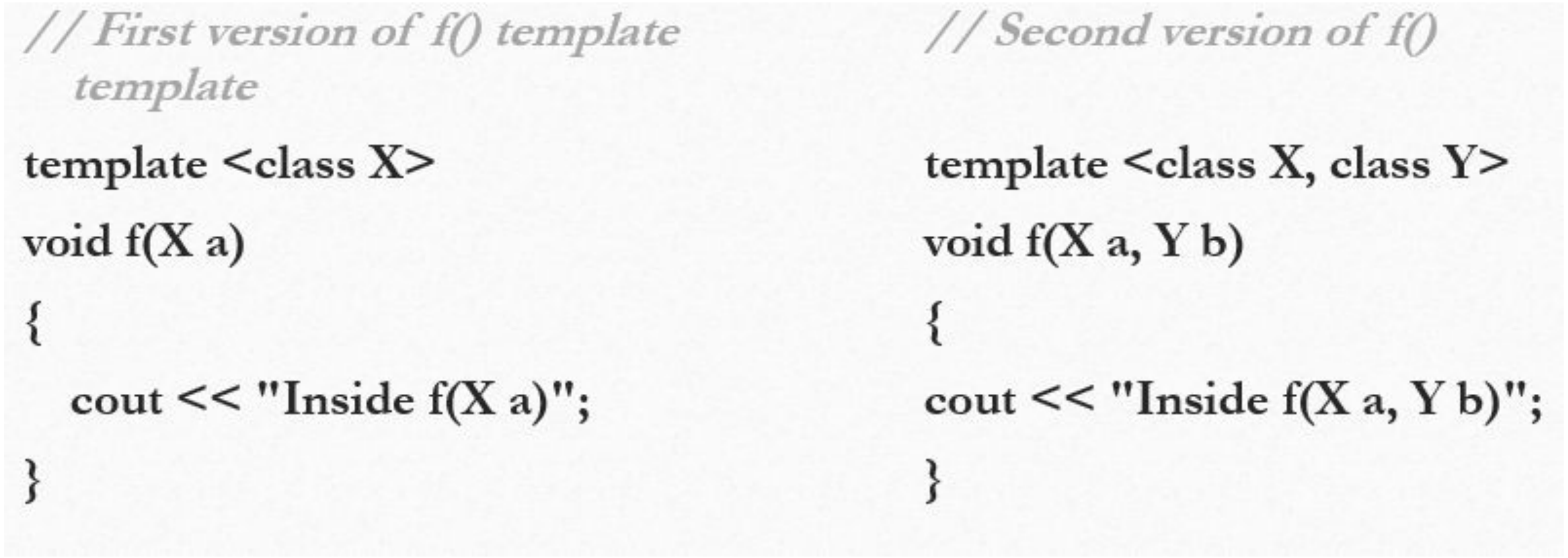


```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14

--------------------------------
Process exited after 0.1619 seconds with
```

Overloading a Generic Template

- In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself
- To do so, simply create another version of the template that differs from any others in its parameter list

Example

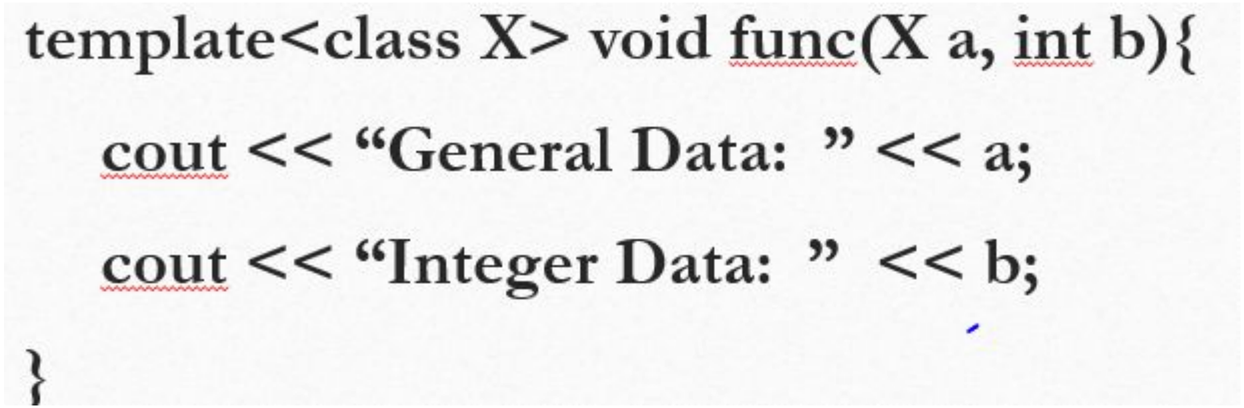


```
// First version of f() template
    template
template <class X>
void f(X a)
{
    cout << "Inside f(X a)";
}
```

```
// Second version of f()

template <class X, class Y>
void f(X a, Y b)
{
cout << "Inside f(X a, Y b)";
}
```

Using normal parameters in Generic Functions

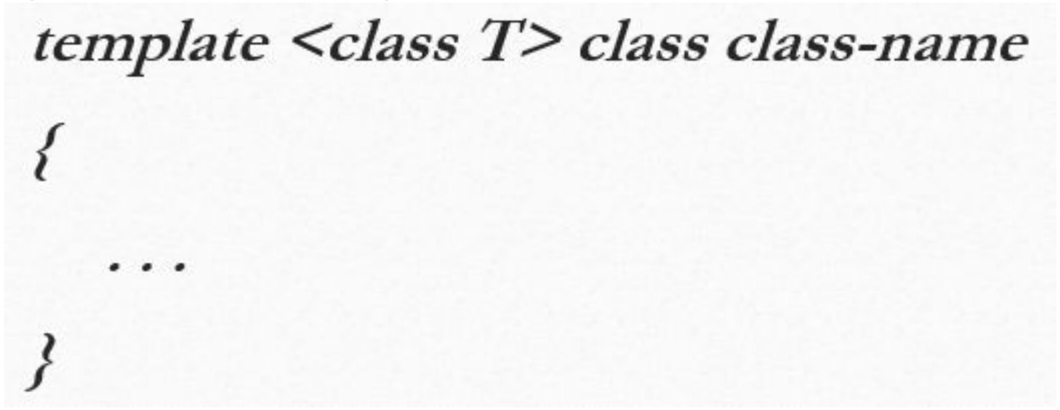
- You can mix non-generic parameters with generic parameters in template functions.



```
template<class X> void func(X a, int b){
    cout << “General Data:  ” << a;
    cout << “Integer Data:  ”  << b;
}
```

Generic Classes

- In addition to generic functions, you can also define a *generic class*
- The actual type of the data being used (in class) will be specified as a parameter when objects of that class are created
- Generic classes are useful when a class uses logic that can be generalized e.g. Stacks, Queues
- The general form of a generic class declaration is shown here:

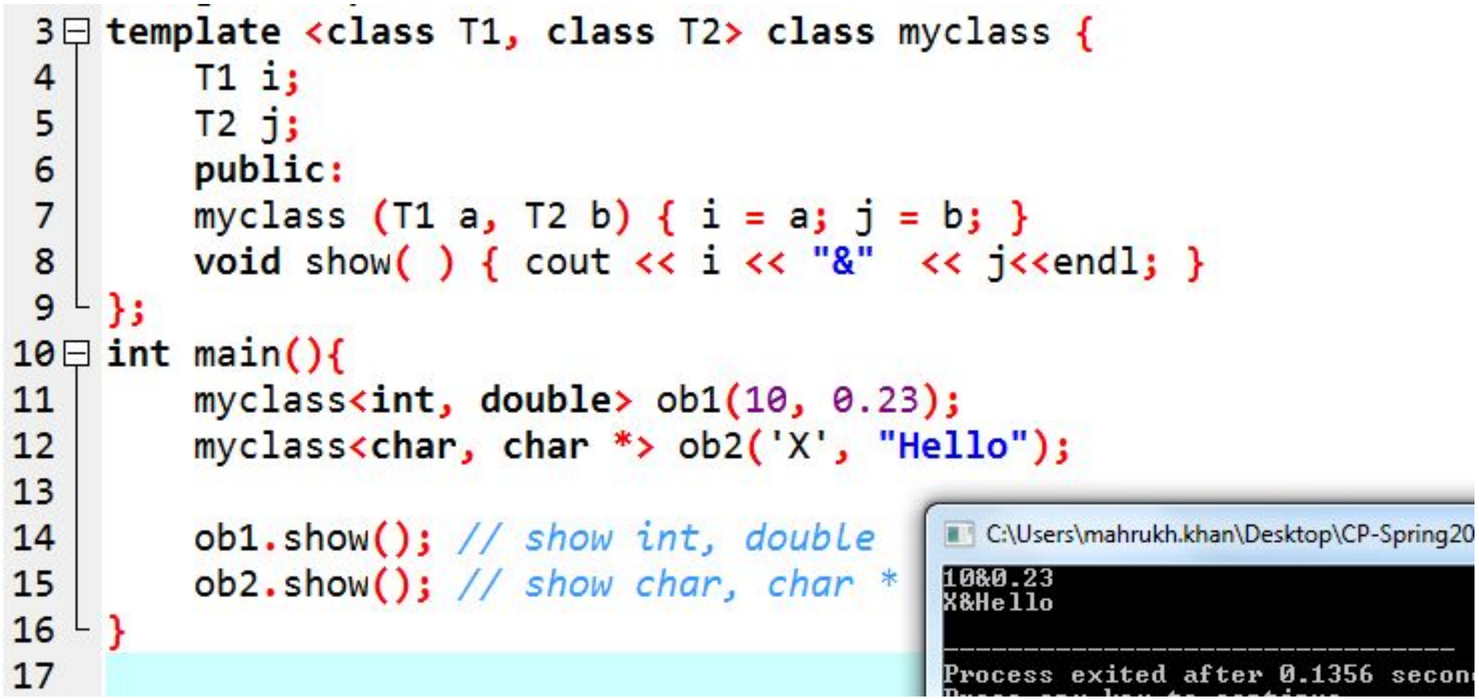


```
template <class T> class class-name
{
    . . .
}
```

Generic Classes

- If necessary, we can define more than one generic data type using a comma-separated list
- We create a specific instance of that class using the following general form:
- ***class-name <type> ob;***

Example



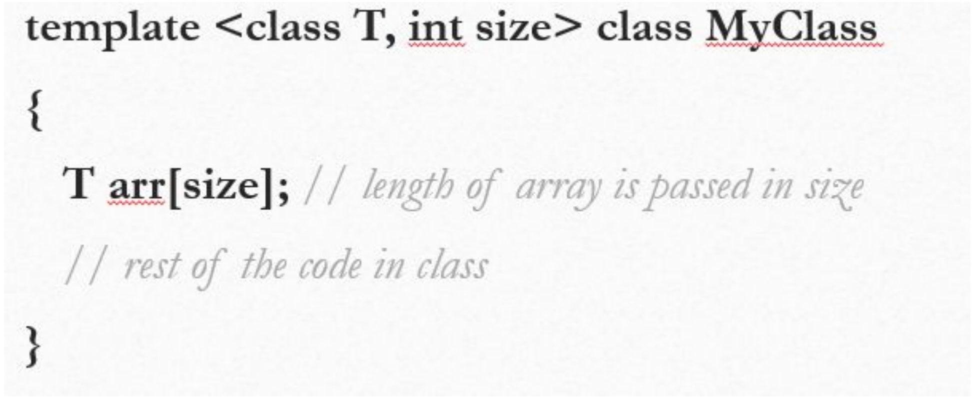
```
template <class T1, class T2> class myclass {
    T1 i;
    T2 j;
    public:
    myclass (T1 a, T2 b) { i = a; j = b; }
    void show( ) { cout << i << "&"  << j<<endl; }
};
int main(){
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Hello");

    ob1.show(); // show int, double
    ob2.show(); // show char, char *
}
```

C:\Users\mahrukh.khan\Desktop\CP-Spring20

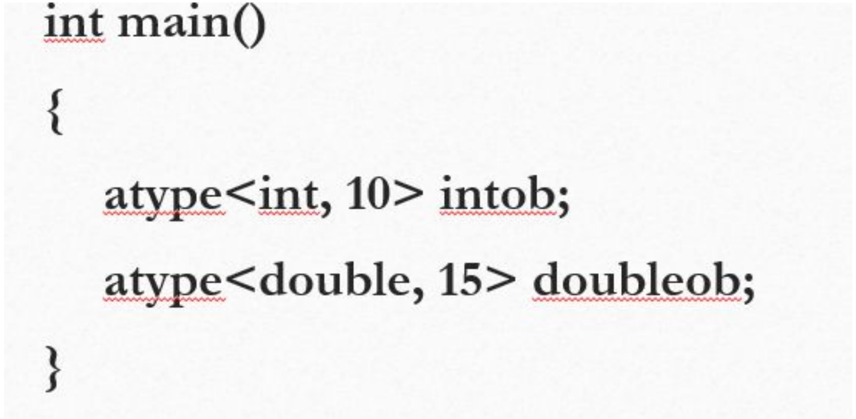
```
10&0.23
X&Hello

--------------------------------
Process exited after 0.1356 secon
```

Using non type arguements in Generic Classes

- In a generic class, we can also specify non-type arguements.



```
template <class T, int size> class MyClass
{
   T arr[size]; // length of  array is passed in size
   // rest of  the code in class
}
```



```
int main()
{
     atype<int, 10> intob;
     atype<double, 15> doubleob;
}
```

Standard Template Library

- The Standard Template Library provides a set of well structured generic C++ components that work together in a seamless way
- A collection of composable class and function templates
  - Helper class and function templates: operators, pair
  - Container and iterator class templates
  - Generic algorithms that operate over iterators
  - Function objects
  - Adaptors

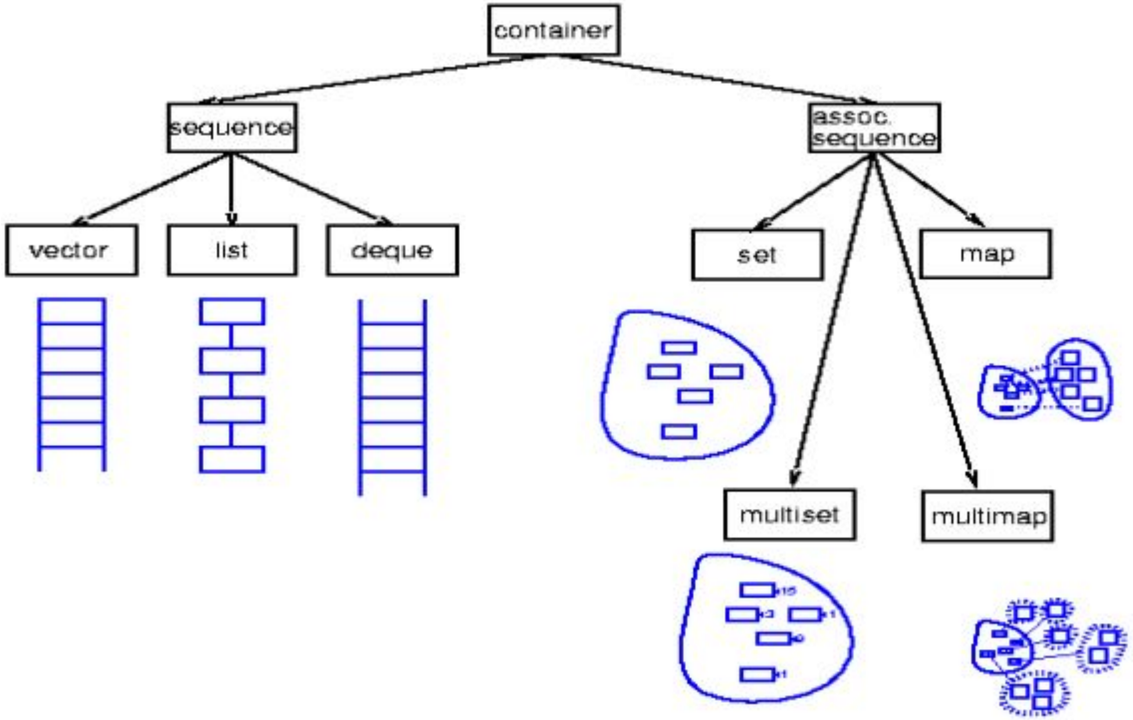
Advantages

- Enables generic programming in C++
- Each generic algorithm can operate over any iterator for which the necessary operations are provided
- Extensible: can support new algorithms, containers, iterators
- Reuse: “write less, do more”
  - The STL hides complex, tedious and error prone details
  - The programmer can then focus on the problem at hand
  - Type-safe plug compatibility between STL components
- Flexibility
  - Iterators decouple algorithms from containers
  - Unanticipated combinations easily supported
- Efficiency
  - Templates avoid virtual function overhead
  - Strict attention to time complexity of algorithms

STL

- Containers:
    - Sequence: vector, deque, list
    - Associative: set, multiset, map, multimap
- Iterators:
    - Input, output, forward, bidirectional, random access
    - Each container declares a trait for the type of iterator it provides
- Generic Algorithms:
    - Sequence (mutating and non-mutating), sorting, numeric

STL Container



STL Container

- STL containers are Abstract Data Types (ADTs)
- All containers are parameterized by the type(s) they contain
- Sequence containers are ordered
- Associative containers are unordered
- Each container declares an iterator typedef
- Each container provides special factory methods for iterators

Sequence Container

- Vector
  - can be used as an array and a stack but can grow itself as necessary
  - provides reallocation, indexed storage, push back, pop back
- Deque
  - A deque is a double ended queue
  - adds efficient insertion and removal at the beginning as well as at the end of the sequence
- List
  - has constant time insertion and deletion at any point in the sequence (not just at the beginning and end)
  - performance trade-off: does not offer a random access iterator

Associative Container

- A set is an unordered collection of unique keys
  - e.g., a set of student id numbers
- A map associates a value with each unique key
  - e.g., a student’s first name
- A multiset or a multimap can support multiple equivalent (nonunique) keys
  - e.g., student last names

Iterators

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.Iterators are used to point at the memory addresses of STL containers.

Operations of iterators :-

1. **begin()** :- This function is used to return the **beginning position** of the container.

2. **end()** :- This function is used to return the ***after* end position** of the container.

3.**advance()** :- This function is used to **increment the iterator position** till the specified number mentioned in its arguments.

Iterators

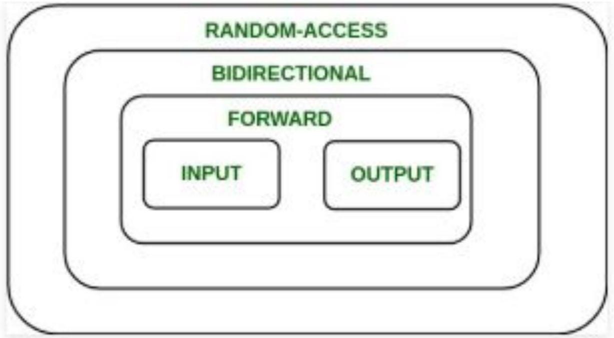
4. next() :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.

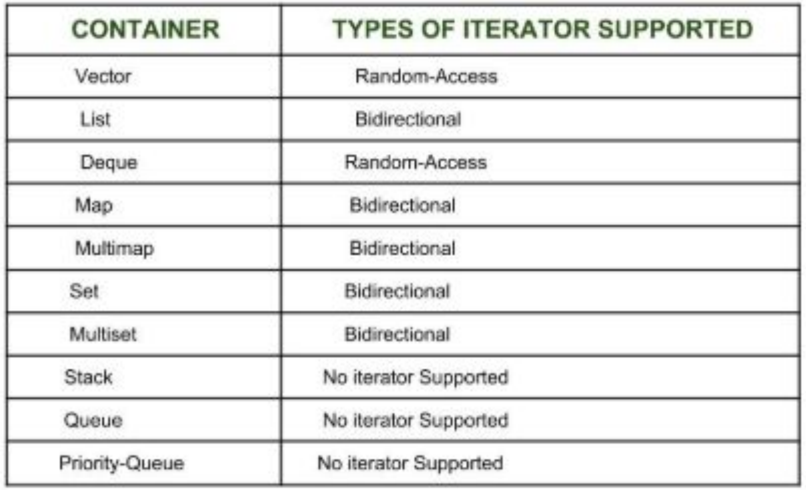
5. prev() :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.

6. inserter() :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.

Iterators

Depending upon the functionality of iterators they can be classified into five categories, as shown in the diagram below with the outer one being the most powerful one and consequently the inner one is the least powerful in terms of functionality.





CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

Iterators

1. **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.
2. **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.
3. **Forward Iterator**: They are higher in hierarachy 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 forward direction and that too one step at a time.

Iterators

4. **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.

5. **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 is same as pointers.