Templates & STL

By

Bhimashankar Takalki

Overview

- Templates
- Function Templates
- Class Templates
- How Templates Work
- STL (Standard Template Library)
- Containers
- Iterators

Templates

Templates are functions or classes that are written for one or more types not yet specified. A class template is like a macro which produces an entire class as its expansion.

A template is not compiled once to generate code usable for any type; instead, it is compiled for each type or combination of types for which it is used.

- ❑ Function Templates
- ❑ Class Templates

Function Templates

Function templates are used to use a common code to specify an entire range of overloaded functions.

- Syntax

```
template <class T>
returntype FunctionName( T param1, …..
{
    .
    .
    .
};
```

A generic swap function to swap two objects of the same data type.

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

int main()
{
    int i=10,j=100;
    char c = 'A', d = 'S';

    Swap(i,j);
    Swap(c,d);
    cout << i  << " " << j <<endl;
    cout << c  << " " << d << endl;
    return 0;
}
```

Output:
100 10
S A

Q: What is the output of the following program?

```
template <class T>
void Swap(T& a, T&b)
{
    T temp = a;
    a = b;
    b = temp;
}
int main()
{
   char str1[8],str2[8];

   strcpy(str1,"Hello");
   strcpy(str2,"World");

   Swap(str1,str2);
   cout << str1 << " " << str2;

   return 0;
}
```

Output:
Compilation Error

Q: How do we call a template function which does not have any parameters of its template argument types

```
template<class T>
T* newArray(int size = 10)
{
    T *p_array = new T[size];
    if(p_array == 0)
            cout << "Error allocating memory";
    return p_array;
}
```

Ans:

```
int*    iArr = newArray<int>();
char* cArr = newArray<char>(20);
```

Q: What is the output of the following program?

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

```
void Swap(int& a, int&b)
{
    cout << "In int Swap\n";
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main()
{
    int i=10,j=100;
    char c = 'A', d = 'S';

    Swap(i,j);
    Swap(c,d);

    return 0;
}
```

Output:
In int Swap
In template Swap

Q: What is the output of the following program?

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

```
void Swap(int& a, int&b)
{
    cout << "In int Swap\n";
    int temp = a;
    a = b;
    b = temp;
}
```

Template Function Specialization

```
template <>
void Swap<int>(int& a, int&b)
{
        cout << "In int template Swap\n";
        int temp = a;
        a = b;
        b = temp;
}
```

```
int main()
{
    int i=10,j=100;
    char c = 'A', d = 'S';

    Swap(i,j);
    Swap(c,d);

    return 0;
}
```

Output:

In int template Swap

In template Swap

Class Templates

- Syntax

```
template <class T>
class ClassName
{
    .
    .
    .
};
```

A generic array class

```
template <class T>
class Array
{
public:
    Array(int size_i = 10)
    {
            m_size = size_i;
            p_array = new T[m_size];
    }
private:
    T * p_array;
    int m_size;
};
```

```
int main()
{
        Array<int> iArray;
        Array<char> cArray(20);

        return 0;
}
```

How do templates work?

A template is rather like a macro which produces an entire class/function as its expansion.

```
#define Array(T)       \
class       Array_##T              \
{                      \
private:               \
    T * p_array;       \
    int m_size;        \
public:                \
    Array_##T(int size_i = 10)   \
    {                            \
        m_size = size_i;         \
        p_array = new T[m_size];    \
    }                            \
              \
    int Size()                   \
    {                            \
        return m_size;           \
    }                            \
};                               \
Array(int) /* Macro expands to class Array_int */
Array(char) /* Macro expands to class Array_char */
```

```
int main()
{
    Array_int iArr1;
    Array_char cArray;

    printf("Size of Int array= %d", iArr1.Size());
    printf("Size of Char array= %d", cArray.Size());

    return 0;
}
```

Q: Point out errors in the program if any

```
template <class T>
template <class T = char>
class Array
{
public:
    Array(int size_i = 10)
    {
            m_size = size_i;
            p_array = new T[m_size];
    }
    int Size()
    {
            return m_size;
    }
private:
    T * p_array;
    int m_size;
};
```

```
int main()
{
   Array<int> iArr;
   Array<> cArr;   // Defaults to a char.
   return 0;
}
```

Ans: error C2976: 'Array' : too few template arguments

Q: What is the output of the following program?

```
template <class T, class P>
class A
{
protected:
    T m_data1;
    T m_data2;
    P m_data2;
public:
    A(T value1,P value2)
    {
        cout << "Constructor of A\n";
        m_data1 = value1;
        m_data2 = value2;
    }

    void PrintData()
    {
        cout << "Data1=" << m_data1 << endl;
        cout << "Data2=" << m_data2 << endl;
    }
};
```

```
int main()
{
    A<int, char> objA(1000, 'Z');
    objA.PrintData();
    return 0;
}
```

Output:

Constructor of A
Data1=1000
Data2=90
Data2=Z

Template Class as a member variable

```
template <class T>
class Base
{
protected:
    T m_data;
public:
    friend ostream & operator << (ostream& out, Base<T> b);

    Base (T value)
    {
            cout << "Constructor of Base\n";
            m_data = value;
    }
    T GetData()    { return m_data; }
};
```

```
template <class T>
class A
{
protected:
    Base<T> m_base;
public:
    A(T value):m_base(value)
    {
        cout << "Ctor of A\n";
    }
    void PrintData()
    {
        cout << m_base;
    }
};
```

```
int main()
{
    A<int> objA(1000);
    objA.PrintData();
    return 0;
}
```

```
template <class T>
ostream & operator << (ostream& out, Base<T> b)
{
    out << "Value @ Base=";
    out << b.m_data << endl;
    return out;
}
```

Nesting of Templates classes

```
template <class T>
class Base
{
protected:
    T m_data;
public:
    Base (T value)
    {
        cout << "Ctor of Base\n";
        m_data = value;
    }
    void SetData(T data) { m_data = data; }
    T GetData() { return m_data; }

    friend ostream & operator << (ostream& out, Base<T> b);
};
```

```
template <class T>
ostream & operator << (ostream& out, Base<T> b)
{
    out << "Value @ Base=";
    out << b.m_data << endl;
    return out;
}
```

```
template <class T>
class A
{
protected:
    T data;
public:
    A():data(0)
    { cout << "Ctor of A\n"; }
    void PrintData()
    {
        cout << data;
    }
};
```

```
int main()
{
    A< Base<int> > objA;
    objA.PrintData();
    return 0;
}
```

Q: Is it possible to define a base class containing derived class objects/pointers as members?

```
template <class T>
class Base
{
protected:
    T m_data;
public:
    Base (T value):m_data(value)
    {
        cout << "Constructor of Base\n";
        m_data = value;
    }
    void SetData(T data) { m_data = data; }
   T GetData() { return m_data; }
};
```

```
template <class T>
class Derived: public Base <T>
{
public:
    Derived(T value):Base<T> (value)
    {
        cout << "Constructor of Derived\n";
    }
};
```

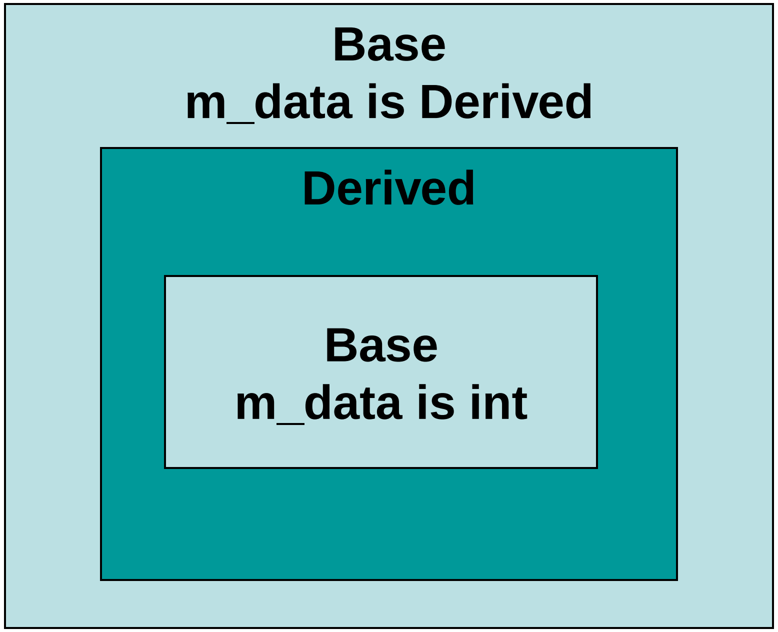
```
int main()
{
    Base< Derived<int> > objB(1000);
    objB.GetData();
    return 0;
}
```

Q: What is the sequence of constructors?

Output:
Constructor of Base
Constructor of Derived
Constructor of Base

This design is called : curiously recurring template pattern (CRTP)

Memory Map of objB



Use Case

This technique achieves a similar effect to the use of virtual functions, without the costs of dynamic polymorphism. (This polymorphism is static)

```
template<class T>
class Currency
{
protected:
    T* mp_type;
    int value;
public:
    Currency(int i):value(i)
    { mp_type = (T*) this; }
    int GetValue()
    {
        return mp_type->GetValue();
    }
};

template<typename T>
void CurrencyValue(Currency<T>& a)
{
    cout << "Currency Value = " << a.GetValue() <<
endl;
}
```

Output:

Currency Value = 500

Currency Value = 630

```
class Dollar : public Currency<Dollar>
{
public:
    Dollar(int i) : Currency<Dollar>(i)
    {
    }
    int GetValue()
    {
        return value*50;
    }
};

class Euro : public Currency<Euro>
{
public:
    Euro(int i) : Currency<Euro>(i)
    {
    }
    int GetValue()
    {
        return value*63;
    }
};
```

```
int main()
{
    Dollar oD(10);
    Euro oE(10);
    CurrencyValue(oD);
    CurrencyValue(oE);
    return 0;
}
```

Advantages & Disadvantages of Templates

Advantages

- ❑ Code is generic and easy to maintain.
- ❑ If templates are compared to macros
  - ❑ Templates are "type-safe".
  - ❑ Macros are always expand as inline, whereas templates are only expanded inline when the compiler deems it appropriate.

Disadvantages

- ❑ Inappropriate usage may lead to code bloating.
- ❑ Extra overhead at compile-time lead to longer build times.
- ❑ All the function definitions must be placed in the header files.

Solutions for making Templates as libraries

- If your compiler supports 'export' keyword. (Currently a very few like 'Comeau' do).

For Eg:

```
------ Source File ------
template<class T>
export int Vector<T>::Size()
{
    return m_size;
}
```

- Add "template class ClassName<datatype>;" to the template source file for each type or combination of types for which it may be used. (Inappropriate usage may result code bloating)

For Eg:

```
----- Source File ------
template<class T>
int Vector<T>::Size()
{
    return m_size;
}
template class Vector<int>;
template class Vector<char>;
```

STL (Standard Template Library)

The standard template library (STL) is a generic template based library that provides solutions to managing collections of data with modern and efficient algorithms.

- ❑ Containers
- ❑ Iterators

Containers

Containers are used to manage collections of objects of a certain kind.

- **Sequence Containers**

Sequence containers are ordered collections in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element.

  - **Vectors**
  - **Deques**
  - **Lists**

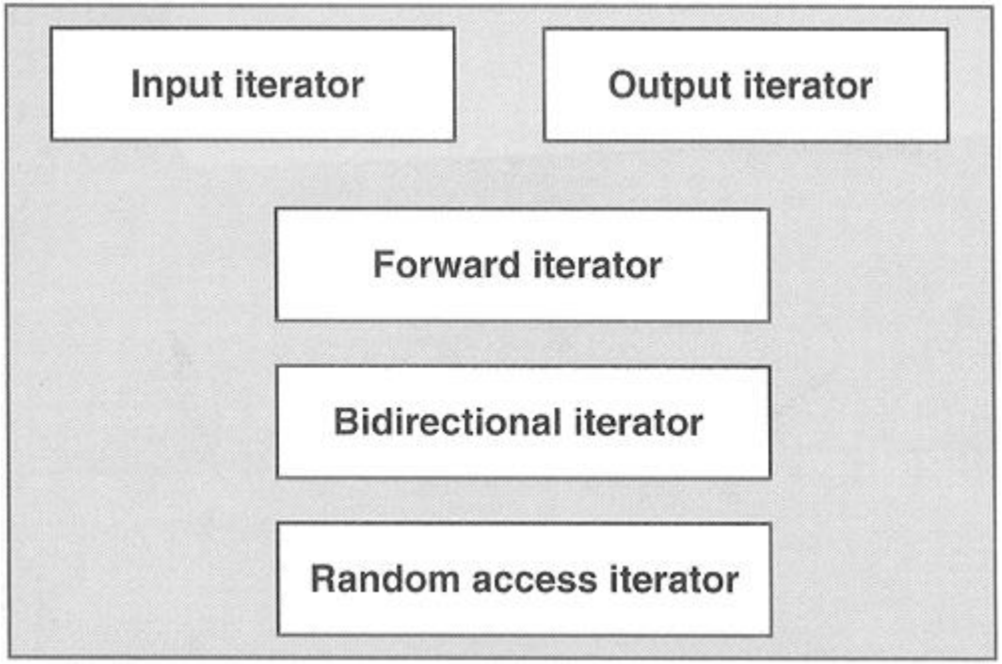
- **Associative Containers**

Associative containers are sorted collections in which the actual position of an element depends on its value due to a certain sorting criterion.

  - **Set**
  - **MultiSet**
  - **Map**
  - **Multimap**

Iterators

Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.



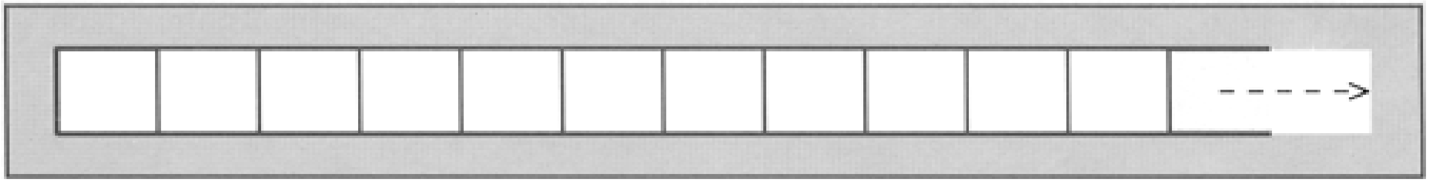
Abilities of Iterator Categories

Iterator Category	Ability	Providers
Input iterator	Reads forward	istream
Output iterator	Writes forward	ostream, inserter
Forward iterator	Reads and writes forward	
Bidirectional iterator	Reads and writes forward and backward	list, set, multiset, map, multimap
Random access iterator	Reads and writes with random access	vector, deque string, array

Sequence Containers

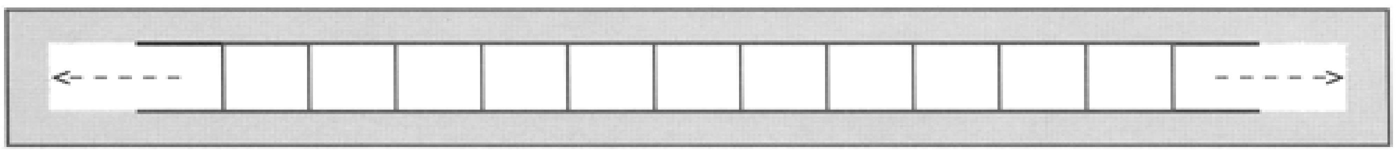
Internal Structure of Sequence Containers

Vectors



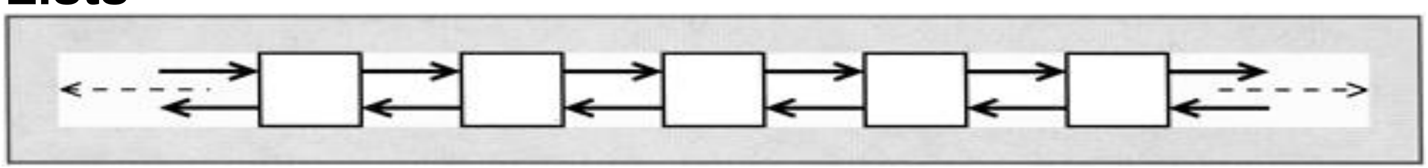
Supported Operations:
add, delete, random access, swap contents, copy, resize, erase, clear etc.

Deques



Supported Operations:
add front, add back, delete, random access, swap contents, copy, resize, erase, clear etc.

Lists



Supported Operations:
add , insert, delete, swap contents, copy, resize, erase, clear etc.

Vectors

A vector manages its elements in a dynamic array.
It enables random access, which means you can access each element directly with the corresponding index. Appending and removing elements at the end of the array is very fast. However, inserting an element in the middle or at the beginning of the array takes time because all the following elements have to be moved to make room for it while maintaining the order.
Example:

```
int main()
{
    vector<string> obj;
    obj.push_back("This");
    obj.push_back("is");
    obj.push_back("a");
    obj.push_back("Vector");

    for(int i=0;i<obj.size();i++)
            cout << obj[i] << " ";
    cout << endl;
    return 0;
}
```

<u>Output:</u>
This is a Vector

Deques

It manages its elements with a dynamic array, provides random access, and has almost the same interface as a vector. The difference is that with a deque the dynamic array is open at both ends. Thus, a deque is fast for insertions and deletions at both the end and the beginning.
Example:

```
int main()
{
    deque<string> obj;
    obj.push_front(string("is"));
    obj.push_back(string("a"));
    obj.push_back(string("deque"));
    obj.push_front(string("this"));

    for(int i=0; i< obj.size();i++)
        cout << obj[i] << " ";
    cout << endl;

    return 0;
}
```

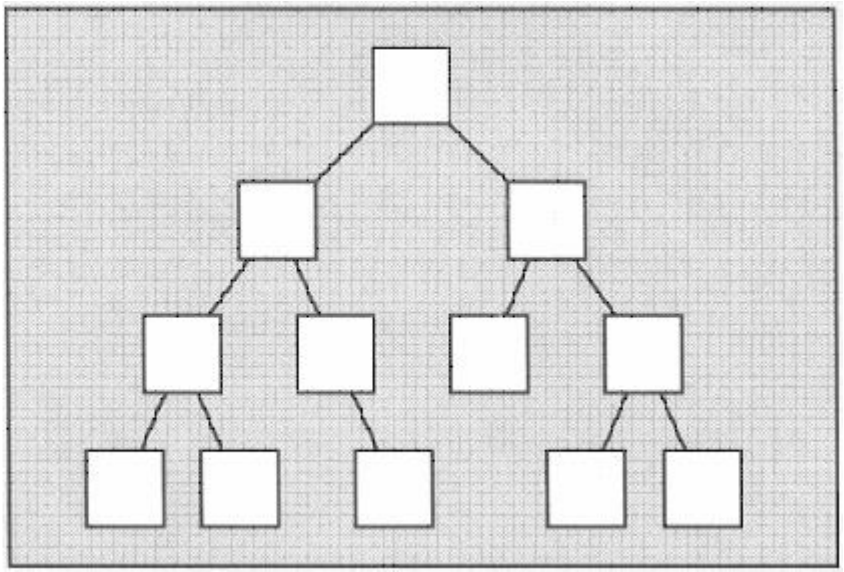
Output:
this is a deque

Lists

A list manages its elements as a doubly linked list.
It does not provide random access. For example, to access the fifth element, you must navigate the first four elements following the chain of links.
Inserting and removing elements is fast at each position, and not only at one or both ends. You can always insert and delete an element in constant time because no other elements have to be moved. Internally, only some pointer values are manipulated.

Associative Containers

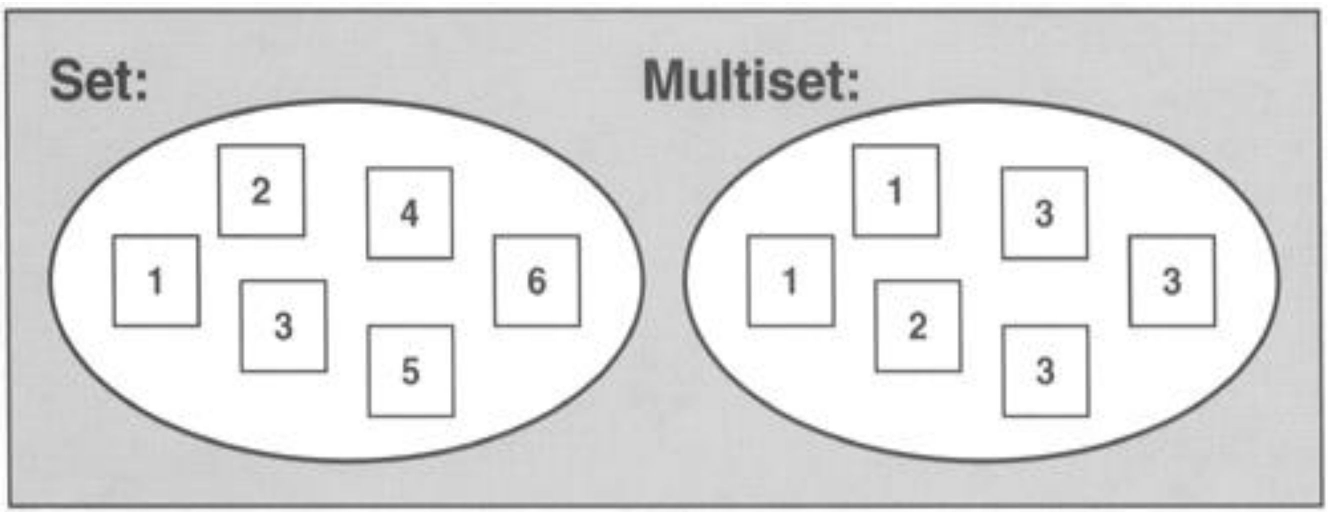
Internal Structure of Associative Containers



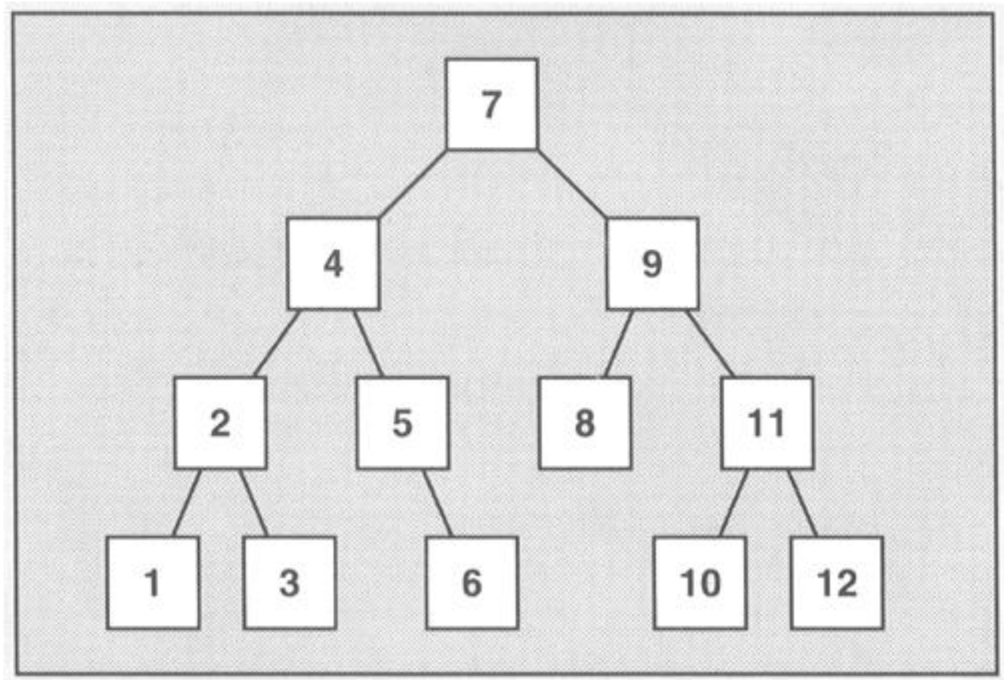
Sets & MultiSets

A set is a collection in which elements are sorted according to their own values. Each element may occur only once, thus duplicates are not allowed.

A multiset is the same as a set except that duplicates are allowed.



Internal structure of Sets and MultiSets



Set Example

Used to maintain any data in sorted form

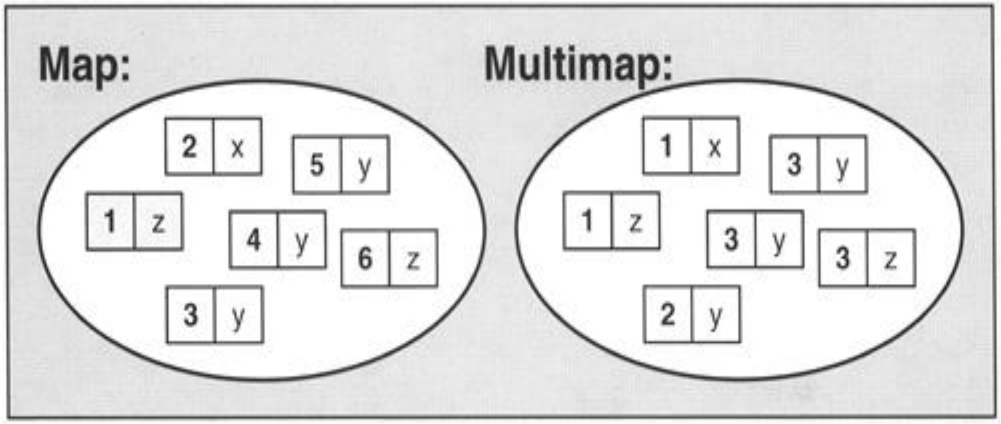
```
int main()
{
    set<int> obj;
    obj.insert(4);
    obj.insert(2);
    obj.insert(3);
    obj.insert(1);

    set<int>::iterator pos;
    for (pos = obj.begin(); pos != obj.end(); ++pos)
            cout << *pos << ' ';
    cout << endl;

    return 0;
}
```

Output:
1 2 3 4

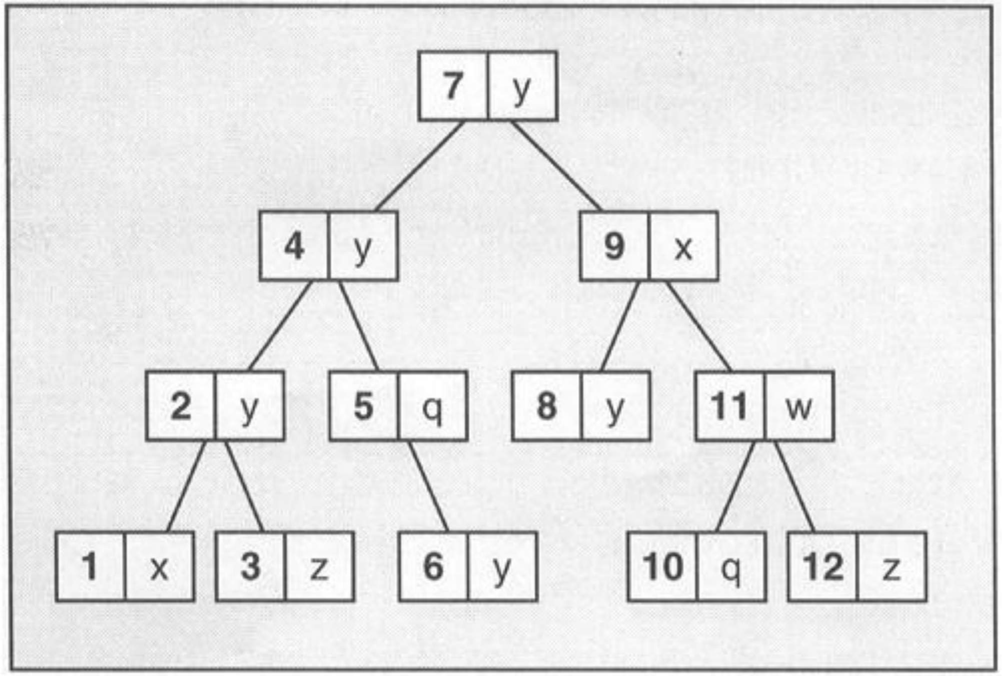
Maps & MultiMaps

A map contains elements that are key/value pairs. Each element has a key that is the basis for the sorting criterion and a value. Each key may occur only once, thus duplicate keys are not allowed.

A multimap is the same as a map except that duplicates are allowed.



Internal structure of Maps and MultiMaps



Examples: Stock and Dictionary

Use case for Map

The map is used as a variable to hold stock values :

```
int main()
{
    map<string,float> stocks;
    stocks["GRASIM"] = 1326.20;
    stocks["TATA STEEL"] = 182.00;
    stocks["RANBAXY"] = 209.00;
    stocks["ACC LIMITED"] = 531.50;

    map<string,float>::iterator pos;
    for(pos = stocks.begin(); pos != stocks.end(); ++pos)
    {
        cout << "stock: " << pos->first << "\t"
                << "price: " << pos->second << endl;
    }
    cout << endl;

    return 0;
}
```

Output:

```
stock: ACC LIMITED    price: 531.5
stock: GRASIM         price: 1326.2
stock: RANBAXY        price: 209
stock: TATA STEEL     price: 182
```

Heterogeneous Containers

One can store different types of objects in the same container. These are called heterogeneous containers.
Heterogeneous containers are not a part of the standard STL library.

Thank You