Chapter 9

Objective

- Templates (Parameterized Types)
- Parameterizing Stand Alone Function
- Parameterizing User Defined Data Type
- Static Data Members in Template Class
- Parameterizing a Class
- Introduction to the Standard Template Library (STL)

Unit 1

Templates (Parameterized Types)

- Templates define generic family of classes or stand alone functions.
- The code written using template has the highest potential reusability.
- The parameter allows the creation of unique **data_type** that is use as a parameter to a class or a stand alone function.

- Compiler substitutes the characteristics for specific type by using template parameter.
- Template type parameter may be used for any user define functions.
- The template is written using a keyword “typename” or “class” for some general type.
- Once typename is declared such as "T" which holds all the properties of actual datatype that the object of specific type requires.

Unit 1

Parameterizing Stand Alone Function

```
1. #include <iostream>                    // Example 9-1
2. using namespace std;
3.
4. template <typename B>
5. B square(const B number)
6. {
7.     return(number * number);
8. }
9.
10. int main()
11. {
12.   const int ix = 5;
14.   const float fx = 5.0;
15.   // always have decimal point for float
16.   cout.setf(ios::showpoint);
17.   cout << "INTEGER SQUARE: " << square(ix) << "\n";
18.   cout << "FLOAT SQUARE: " << square(fx) << "\n";
19. }
```

Output:
INTEGER SQUARE: 25
FLOAT SQUARE: 25.0000

Unit 2

Parameterizing User Defined Data Type

- User defined data type can be parameterized for the templates
- Parameterized class takes the **data type** as its parameter.
- A class template definition looks like a regular class definition, except it is prefixed by the keyword template.
- While implementing class template member functions, the definitions are prefixed by the keyword template.

Rules To Use Templates

1. One parameterized class cannot be nested in another
2. Static members within a parameterized class are created multiple times for each instance of a data type.

Unit 2

Parameterizing a Class

```
1  #include <iostream>                    // Example 9-2
2  using namespace std;
3
4  template <typename Type>
5  class Array {
6      public:
7            Array(int sz = 0);
8           ~Array() { delete [] m_AnyArray;}
9            void Init(Type nData, int nIndex);
10            int GetSize() { return m_Size;}
11           void Show();
12     private:
13           Type *m_AnyArray;
14           int m_Size;
15  };
```

Unit 2

Parameterizing a Class

```
16  // Parameterized class constructor body
17  template <typename Type>
18  Array<Type>::Array(int sz)
19  {
20     m_AnyArray = new Type[sz];
21     m_Size = sz;
22  }
23
24  template <typename Type>
25  void Array<Type>::Init(Type nData, int nIndex)
26  {
27     m_AnyArray[nIndex] = nData;
28  }
```

Unit 2

```
29  template <typename Type>
30  void Array<Type>::Show()
31  {
32     for (int i=0; i < m_Size; i++) cout << m_AnyArray[i] << " ";
33     cout << "\n";
34  }
35  int main()
36  {
37     Array<int> intArray(30);
38     for (int i=0; i < intArray.GetSize(); i++) intArray.Init(i, i);
39     intArray.Show();
40
41     Array<char> charArray(26);
42     for (int Alpha=0; Alpha < charArray.GetSize(); Alpha++)
43        charArray.Init('A' + Alpha, Alpha);
44     charArray.Show();
45  }
```

Output:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Unit 2

Parameterizing a Class

```
1  #include <iostream>              // Example 9-3
2  using namespace std;
3  template <class T>
4  class Stack  {
5      public:
6           Stack(int = 10) ;
7           ~Stack() { delete [] stackPtr ; }
8           int push(const T&);
9           int pop(T &item); // pop an element off the stack
10          int isEmpty()const { return top == -1 ; }
11          int isFull() const { return top == size - 1; }
12     private:
13          int size ; // number of elements on Stack.
14          int top ;
15          T* stackPtr ;
16 } ;
```

Unit 2

Parameterizing a Class

```
17  //constructor with the default size 10
18  template <class T>
19  Stack<T>::Stack(int s)
20  {
21      size = s > 0 && s < 1000 ? s : 10 ;
22      top = -1 ; // initialize stack
23      stackPtr = new T[size] ;
24  }
25
26  template <class T>
27  int Stack<T>::push(const T& item) // push an element onto the Stack
28  {
29     if (!isFull()) {
30       stackPtr[++top] = item ;
31       return 1 ; // push successful
32     }
33     return 0 ; // push unsuccessful
34  }
```

Unit 2

Parameterizing a Class

```
35  template <class T>
36  int Stack<T>::pop(T& popValue) // pop an element off the Stack
37  {
38     if (!isEmpty()) {
39        popValue = stackPtr[top--] ;
40        return 1 ; // pop successful
41     }
42     return 0 ; // pop unsuccessful
43  }
```

Unit 2

```
44  int main()
45  {
47     Stack<float> fs(5) ;
48     cout << "Pushing elements onto fs" << endl ;
49     float f = 1.1 ;
50     while (fs.push(f)) {
51         cout << f << ' ' ;
52         f += 1.1 ;
53     }
54     cout << endl << "Stack Full." << endl << "Pop from fs" << endl ;
56     while (fs.pop(f)) cout << f << ' ' ;
57     cout << endl << "Stack Empty" << endl << endl ;
59     Stack<int> is ;
60     cout << "Pushing elements onto is" << endl ;
61     for (int i = 1;  (is.push(i)); i++) cout << i << ' ' ;
54     cout << endl << "Stack Full." << endl << "Pop from is" << endl ;
64     int i = 0;
65     while (is.pop(i)) cout << i << ' ' ;
66    cout << endl << "Stack Empty" << endl ;
67  }
```

Unit 2

OutPut:

```
Pushing elements onto fs
1.1 2.2 3.3 4.4 5.5
Stack Full.

Popping elements from fs
5.5 4.4 3.3 2.2 1.1
Stack Empty

Pushing elements onto is
1 2 3 4 5 6 7 8 9 10
Stack Full

Popping elements from is
10 9 8 7 6 5 4 3 2 1
Stack Empty
```

Unit 3

Static Data Members in Template Class

- Static members within a parameterized class are created multiple times for each instance of a data type

```
1  #include <iostream>                    // Example 9-4
2  using namespace std;
3  template <typename DataType>
4  class CHouse {
5     public:
6           CHouse() { }
7           static void SetRoomCount(DataType n);
8           int ShowColorCount() const { return m_nTotalRooms; }
9
10    //private:
11           static DataType m_nTotalRooms;
12           static DataType m_nColor;
13 };
14
15 template <typename DataType>
16 void CHouse<DataType>::SetRoomCount(DataType n)
17 {
18     m_nTotalRooms = n;
19 }
```

Unit 3

Static Data Members in Template Class

```
20  template<> int CHouse<int>::m_nColor = 0;
21  template<> int CHouse<int>::m_nTotalRooms = 10;
22
23  int main()
24  {
25     CHouse<int>::SetRoomCount(4);
26     CHouse<int> intObj;
27     cout << "Total Rooms: " << intObj.ShowColorCount();
28     cout << "\n";
29     const CHouse<int> intConstObj;
30     cout << "Total Rooms using const object: ";
31     cout << intObj.ShowColorCount() << "\n";
32  }
```

Output:

Total Rooms: 4

Total Rooms using const object: 4

Unit 4

Parameterize Class Using Operator [] Function

```
1  #include <iostream>                    // Example 9-5
2  using namespace std;
3  // prefix template <typename ALL> declares a class template
4  // and use Vector as a class name in the declaration.
5  // Vector is a parameterized class with the type ALL as its parameter.
6  template <typename ALL> class Vector {
7       public:
8            Vector(int n) { data = new ALL[n];  size = n; }
9           ~Vector() { delete [] data; }
10           ALL& operator[](int i) {return data[i]; }
11      private:
12            ALL *data;
13            int size;
14  };
15  class info {
16         public:
17              char alpha;
18              int num;
19  };
```

Unit 4

Parameterize Class Using Operator [] Function

```
20  int main()
21  {
22     Vector<int> x_int(8); // integer array
23     for (int i = 0; i < 8; i++) x_int[i] = i * 3;
24     for (int i = 0; i < 8; i++)  cout << x_int[i] << ' ';
25     cout << '\n';
26     Vector<float> x_float(8);    // float array
27
28     cout.setf(ios::showpoint); // always have decimal point for float
29     for (int i = 0; i < 8; i++)   x_float[i] = i * 3.1;
30     for (int i = 0; i < 8; i++)  cout << x_float[i] << ' ';
31     cout << '\n';
32     Vector<info> x_info(5);
33     x_info[0].alpha  =  'A';
34     x_info[1].alpha  =  'B';
35     x_info[2].alpha  =  'C';
36     x_info[3].alpha  =  'D';
37     x_info[4].alpha  =  'E';
```

Unit 4

Parameterize Class Using Operator [] Function

```
38    x_info[0].num  = 4;
39    x_info[1].num  = 6;
40    x_info[2].num  = 2;
41    x_info[3].num  = 1;
42    x_info[4].num  = 8;
43    for (int i = 0; i < 5; ++i) {
44       cout << x_info[i].alpha << '=';
45       cout << x_info[i].num << ' ';
46    }
47
48    cout << '\n';
49  }
```

```
Output:
0 3 6 9 12 15 18 21
0.00000 3.10000 6.20000 9.30000 12.4000 15.5000 18.6000 21.7000
A=4 B=6 C=2 D=1 E=8
```

Unit 4

Parameterize Class Using Operator [] Function

```
1  #include <iostream> // Example 9-6
2  using namespace std;
3
4  // prefix template <typename DataType> declares a class template and
5  // use CHouse as a class name in the declaration. CHouse is a
6  // parameterized class with the type DataType as its parameter.
7  template <typename DataType>
8  class CHouse {
9      public:
10         CHouse(int n);
11         ~CHouse() { delete [] data; }
12         DataType& operator[](int i) {return data[i]; }
13     private:
14         DataType *data;
15         int size;
16   };
```

Unit 4

```
17  template <typename DataType> CHouse<DataType>::CHouse(int n)
18  {
19     data = new DataType[n];
20     size = n;
21  }
22  class CFurniture {
23      public:
24            char m_Chair;
25            int m_nTable;
26  };
27  class ManageHouseInfo {
28      public:
29            ManageHouseInfo(CHouse<CFurniture> &x);  // Copy constructor
30            void Show(CHouse<CFurniture> &x);
31  };
32  void ManageHouseInfo::Show(CHouse<CFurniture> &x)
33  {
34    for (int i = 0; i < 5; i++)
35       cout << x[i].m_Chair << '=' << x[i].m_nTable << ' ';
36    cout << '\n';
37  }
```

Unit 4

```
38  ManageHouseInfo::ManageHouseInfo(CHouse<CFurniture> &x)
39  {
40     x[0].m_Chair = 'A';
41     x[1].m_Chair = 'B';
42     x[2].m_Chair = 'C';
43     x[3].m_Chair = 'D';
44     x[4].m_Chair = 'E';
45     x[0].m_nTable  = 4;
46     x[1].m_nTable  = 6;
47     x[2].m_nTable  = 2;
48     x[3].m_nTable  = 1;
49     x[4].m_nTable  = 8;
50  }
51
52  int main()
53  {
54     CHouse<CFurniture> x(5);
55     ManageHouseInfo obj(x);
56     obj.Show(x);
57  }
```

Output:
A=4 B=6 C=2 D=1 E=8

Unit 5

Introduction to the Standard Template Library (STL)

- STL is a fundamental part of the C++ Standard.
- It provides comprehensive set of tools and facilities that can be used for most types of applications.
- STL introduces the idea of a generic type of container
- Regardless of container type the operations and element are identical

- **Containers and Iterators**
-
- Containers are implemented via template class definitions.
- A container is an object that represents a group of elements of a certain type.
- It is stored in a way that depends on the type of container (i.e., array, linked list, etc.).
- An iterator is a pointer-like object (that is, an object that supports pointer operations) that is able to "point" to a specific element in the container.
- The iterator class definition is inside the container class
- The iterator's operations depend on what type of container is used and therefore it is container-specific.
- Iterators provide a way of specifying a position in a container.
- An iterator can be incremented or dereference, and two iterators can be compared.

Unit 5

Most Frequently Used Containers

The std namespace provides the definition of following containers and their iterators if appropriate header file is included.

- **vector** Dynamic array of variables or objects.
- **list** Link list of variables or objects
- **deque** Array-like structure, with efficient insertion and removal at both ends
- **set** Set of unique elements (Collection of ordered data in a balanced binary tree)
- **map** Associative key-value pair held in balanced binary tree structure.
- **stack** LIFO (last in, first out) structure
- **queue** FIFO (first-in, first-out) structure
- **iterator** Is a declared to be associated with a single container class type.

Unit 5

Using Specific Containers

```
#include <vector>
#include <stack>
#include <list>

using namespace std;

vector<int> values;
stack<int> back_orders;
list<char> undo_list;
```

The examples of the above containers are provided in the next few slides, lets go over each example in detail.

Unit 5

Example of vector Containers Using Integer Type

```
1  #include <iostream>  // Example 9-7
2  #include <vector> // stl vector header
3  using namespace std; // saves us typing std:: before vector
4  int main()
5  {
6        // create an array of integers
7        vector<int> arNumbers;
8
9        // add elements to array
10       for (int i = 0; i < 5; i++) arNumbers.push_back(i * 12);
11
12       // display the total number of elements in the array
13       cout << "Total Elements: " <<  int(arNumbers.size()) << endl;
14
15       // display the array's contents on the screen
16       for(int i = 0; i < int(arNumbers.size()); i++)
17       cout << "Value at position" << i << " is " << arNumbers[i] << endl;
18  }
```

Unit 5

Example of vector Containers Using Integer Type

```
1  #include <iostream> // Example 9-8
2  #include <vector> // stl vector header
3  using namespace std; // saves us typing std:: before vector
4
5  int main()
6  {
7     // create an array of integers
8     vector<int> arNumbers;
9
10   // add elements to our array
11   for (int i = 0; i < 5; i++) arNumbers.push_back(i * 12);
12
13   // display the total number of elements of the array
14   cout << "Total Elements: " <<  int(arNumbers.size()) << endl;
```

Unit 5

Example of vector Containers (Continue)

```
15    // create a vector<int>::iterator and set the position to which
16    // it points to the beginning of the vector array in memory
17    vector<int>::iterator itNum = arNumbers.begin();
18
19    // Now, we iterate through the array until the iterator exceeds
20    // the end of the array. You will notice that in this for loop
21   // there is no initialisation section, because it is done before for
22   // loop.
23    for(; itNum < arNumbers.end(); itNum++) {
24     cout << "Value at position=> " << int((itNum - arNumbers.begin()));
25     cout << " is " <<  *itNum << endl;
26   } // end for loop
27  }
```

Unit 5

Example of vector Containers Using User Defined Data Type

```
1  #include <iostream> // Example 9-9
2  #include <vector>
3  using namespace std;
4  struct Furniture {
5        char m_Chair;
6        int  m_nTable;
7        Furniture(char c, int t) : m_Chair(c), m_nTable(t) { }
8  };
9
10 int main()
11 {
12    vector<Furniture> vObj;
13    for (int i = 0; i < 10; i++) vObj.push_back(Furniture('A'+i,i));
14
15   for (int i = 0; i < vObj.size(); i++ ) {
16     cout << vObj[i].m_Chair << " " << vObj[i].m_nTable << endl;
17   }
18 }
```

Unit 5

Example of stack Containers Using Character Type

```
1  #include <iostream> // Example 9-10
2  #include <stack>
3  using namespace std;
4
5  int main()
6  {
7    stack <char> alphabet;
8    for (int i = 0; i < 10; i++) alphabet.push('A'+i);
9
10   cout << "There are " << alphabet.size();
11   cout << " alphabet in the stack " << endl;
12   cout << "The alphabet on the top of stack is ";
13   cout  << alphabet.top() << endl;
14
15   alphabet.pop();
16   cout << "Now top alphabet is " << alphabet.top() << endl;
17   cout << alphabet.size();
18  }
```

Unit 5

Example of list Containers Using Character Type

```
1  #include <iostream> // Example 9-11
2  #include <list>
3  using namespace std;
4
5  int main()
6  {
7     list <char> alphabet;
8     for (int i = 0; i < 10; i++) alphabet.push_front('A' + i);
9
10    alphabet.reverse();
11    alphabet.insert(alphabet.end(), 'Z');
12    alphabet.push_back('P');
13    alphabet.push_front('K');
14    for(list<char>::iterator list_iter = alphabet.begin();
15    list_iter != alphabet.end(); list_iter++)
16    cout << *list_iter << " "; // use iterator to access the values
```

Unit 5

Example of list Containers (Continue)

```
17    cout << endl;
18
19    while (!alphabet.empty()) {
20       cout << alphabet.front() << " ";
21       alphabet.pop_front();
22    }
23    cout << endl;
24
25    cout << "There are " << alphabet.size();
26    cout << " alphabet in the list " << endl;
27 }
```