Unit 6: Template and Generic Programming

6.1 Template Functions and Classes

- Template in C++ is an interesting feature that is used for generic programming.
- Generic Programming is an approach of programming where generic types are used as parameters in algorithms such that the same block of code works for a variety of data types.
- In other words, using templates allows us to create a single function or a single class to work with different data types without the same code being rewritten for different data types.
- Template in C++ can be defined as a blueprint or formula for creating a generic function or a class.
- The keyword "template" in C++ is used for the template's syntax with parameter specification(s) using the keyword "class" or "typename" inside angled brackets which defines the data type itself as a variable, followed by the class or function definition.
- Templates in C++ works in such a way that it gets expanded at compile-time, just like macros [e.g. #define area(r) 3.1416*r*r].

Syntax:

```
template <class T>
function/class definition ...
```

OR

```
template <typename T>
function/class definition ...
```

- In C++, templates are of two types:
  a) *Function template*
     A generic function created by parameterizing the data types of variables used inside the function.

  b) *Class template*
     A generic class created by parameterizing the data types of the data members.

Function Templates

Example 1: In the following example, GetMax is designed as a generic function which takes two arguments of any similar data type that can be compared using the '>' operator and returns the maximum value among the two.

```
#include <iostream>
using namespace std;

template<class T>
T GetMax(T a, T b)
{
    T m;
    if(a > b)
        m = a;
    else
        m = b;
    return m;
}

int main()
{
    int   i = 7,    j = 5;
    char  a = 'P',  b = 'Q';
    float x = 15.5, y = 17.9;

    cout << endl << GetMax(x, y);
    cout << endl << GetMax(i, j);
    cout << endl << GetMax(a, b);

    return 0;
}
```

Example 2: In the following example, Swap is designed as a generic function which takes two arguments of any similar data type and exchange their values (swap) with each other.

```
#include <iostream>
using namespace std;

template<class U>
void Swap(U & a, U & b)
{
    U m;
    m = a;
    a = b;
    b = m;
}

int main()
{
    float x = 5.5, y = 7.9;
    int   i = 7,   j = 5;
    char  c = 'P', d = 'Q';

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

    cout << endl << "x = " << x << " and y = " << y;
    cout << endl << "i = " << i << " and j = " << j;
    cout << endl << "c = " << c << " and d = " << d;

    return 0;
}
```

Class Templates

Example 1: A generic class to store the x and y coordinates of a two dimensional point where the data types of the ordinates (x and y) can be any compatible numeric type.

```
#include <iostream>
using namespace std;

template<class T>
class Point
{
  private:
    T x, y;

  public:
    Point(T a = 0, T b = 0)
    {
        x = a;
        y = b;
    }

    void Show()
    {
        cout << endl << "(" << x << ", " << y << ")";
    }
};

int main()
{
    Point<int> p1(3, 4);
    p1.Show();

    Point<float> p2(5.5, 6.4);
    p2.Show();

    return 0;
}
```

Example 2: Construct a template class to hold the Name, age and salary of an employee where name is stored as a string and age and salary can be any of the following data types: int, float, or double. Finally, write a program to show its usage.

```
#include <iostream>
using namespace std;

template<class T, class U>
class Employee{
  private:
    string Name;
    T Age;
    U Salary;

  public:
    Employee(string name, T age, U salary)
    {
        Name   = name;
        Age    = age;
        Salary = salary;
    }

    void Display()
    {
        cout << endl << "------------------------------";
        cout << endl << "Name   : " << Name;
        cout << endl << "Age    : " << Age;
        cout << endl << "Salary : " << Salary;
        cout << endl << "------------------------------";
    }
};
```

```
int main()
{
    // Employee with Age in float and Salary in int
    Employee<float, int> e1("Rajesh", 32.5, 45000);
    e1.Display();

    // Employee with Age in int and Salary in float
    Employee<int, float> e2("Mahesh", 45, 54000.50);
    e2.Display();

    return 0;
}
```

6.2 Container Class and the Standard Template Library (STL)

Standard Template Library

The standard template library (STL) in C++ is a powerful set of templates which provides general-purpose template classes and functions that implement many popular and commonly used data structures and algorithms. STL allows programmers to store the data effectively, and allows efficient manipulation of stored data. Some of the most popular data structures are: vector, linked list, stack, queue, etc.

Components of STL

The core of C++ Standard Template Library is composed of following well-structured components:

- Containers ➔ implemented as generic classes
- Algorithms ➔ implemented as generic functions
- Iterators ➔ special classes/objects

1) Containers:

A container is a holder object that stores a collection of other objects (its elements) of a certain kind. They are implemented as class templates, which allows a 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 special objects called iterators.

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need depends on the functionality offered by the container as well as its efficiency in performing the functionality. This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them. To use a specific container, relevant header file must be included. All containers are defined under the namespace std.

Types of Containers:

a) Sequence containers:

- ***vector*** (growable/dynamically sized array)
- ***deque*** (double ended queue, easy to add/remove elements at both ends)
- ***list*** (linked list, easy to add/remove elements anywhere)

b) Associative Containers

- ***set*** (data sorted in tree structure, unique)
- ***map*** (Key-value pair, unique)
- ***multiset*** (data sorted in tree structure, duplicates allowed)
- ***multimap*** (key-value pair, duplicates allowed)

c) Container Adaptors (derived containers)

- ***stack*** (Last in first out)
- ***queue*** (First in first out)
- ***priority_queue*** (high priority element taken out first, heap)

2) Algorithms:

Algorithms act on containers with some predefined functions. They provide the means to perform operations like initialization, sorting, searching, reversing, transforming, etc., of the contents of containers. The functionalities are implemented as separate function templates than member functions such that the same algorithm can be used for different containers. To use any algorithm the header file <algorithm> needs to be included.

3) Iterators are the "smart pointers" that point to the data in the container and help in traversing over (stepping through) complex data structures, which simple pointers cannot do efficiently. There are five different kinds of iterators:

- Input iterators
- Output iterators
- Forward iterators
- Bi-directional iterators
- Random-access iterators

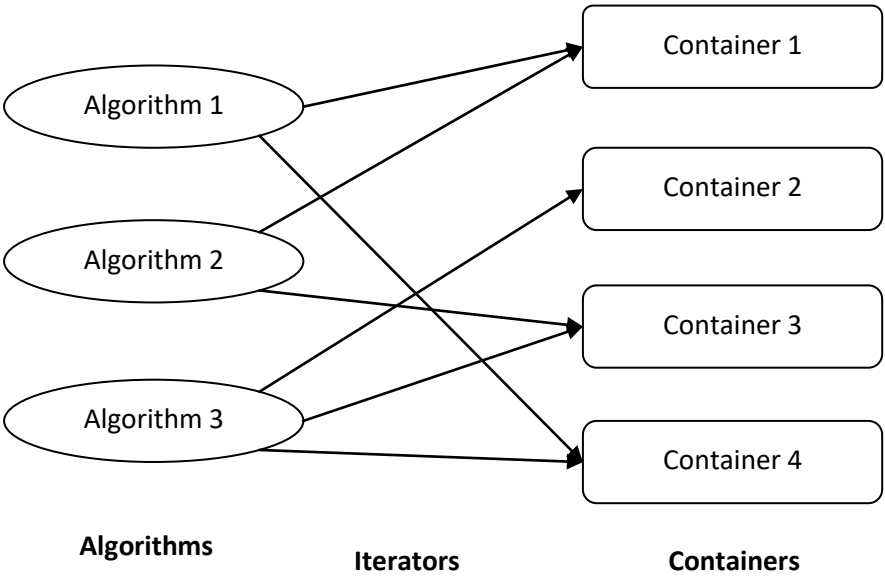


Figure: Relation between Containers, Iterators, and Algorithms

Example 1: Demonstration of Stack

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> iStack;

    iStack.push(111);
    iStack.push(222);
    iStack.push(333);

    if(iStack.empty())
        cout << endl << "Stack is empty!";
    else
        cout << endl << "Stack is not empty!";

    cout << endl << iStack.top();
    iStack.pop();

    cout << endl << iStack.top();
    iStack.pop();

    cout << endl << iStack.top();
    iStack.pop();

    if(iStack.empty())
        cout << endl << "Stack is empty!";
    else
        cout << endl << "Stack is not empty!";

    return 0;
}
```

Example 2: Demonstration of Vector

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v1;

    // Push some data at the back
    v1.push_back(123);
    v1.push_back(623);
    v1.push_back(555);
    v1.push_back(432);
    v1.push_back(456);

    // Accessing the vector elements like array
    cout << endl << "Contents of v1:";
    for(int i=0; i < v1.size(); i++)
        cout << endl << v1[i];

    // Use of algorithms to sort and reverse
    sort   (v1.begin(), v1.end());
    reverse(v1.begin(), v1.end());

    cout << endl << "After Sorting and reversing ...";

    // Accessing the vector elements using iterator
    vector<int>::iterator p;
    p = v1.begin();
    while(p != v1.end())
    {
        cout << endl << *p;
        p++;
    }

    return 0;
}
```

6.3 Exception Handling

Exception:

An exception is a run-time problem that arises due to an erroneous condition during the execution of a program.

Examples:

- attempt to find the square root of a negative number
- attempt to divide by zero
- accessing an array element which is beyond the bounds
- trying to pop and item from an empty stack

Exception handling:

- It is a mechanism to respond to such exceptional circumstance while a program is being executed.
- Exception handling provides a way to transfer control from one part of a program which generates an exception to another block where necessary corrective actions are performed.
- It isolates the main block of code from error handling routine.
- In C++, the exception handling mechanism is built upon three keywords (constructs): **try**, **throw**, and **catch**.
- When we place a function call or a block of code inside a try block with relevant exception handler(s) in the catch block(s) for exception handling, it is also known as protected code.

try:

The function or block of code which might generate or throw an exception is placed in try block.

syntax:

```
try{
        //call a function which throws and exception
        or
        //throw an exception directly if a particular
        //errorneous condition is encountered
}
```

throw:

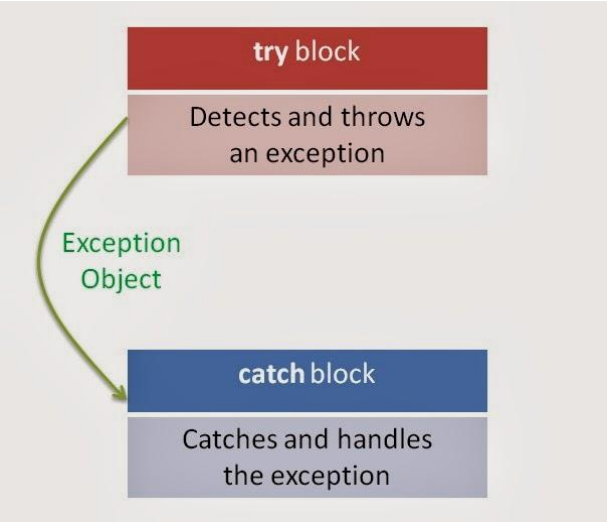
A function or a block of code throws an exception when a problem is encountered. This is done using the keyword "throw". Generally, an object belonging to a standard exception class or its derivative is thrown. However, any kind of value belonging to any data type can be thrown.

syntax:

```
throw object;
```

catch:

The code for handling the exception is placed in a catch block. There can be multiple catch blocks depending upon the types of objects thrown for a single try block. The catch blocks must follow immediately after the try block.



Example: The division function throws an exception when the denominator (b) equals zero. Hence, it is called from inside the try block in main( ). Whenever, the denominator is zero, instead of doing any calculation, an exception is thrown which is caught from the catch block where necessary action is taken.

```
#include <iostream>
using namespace std;

float division(float a, float b)
{
    if(b==0)
        throw 0;
    else
        return a/b;
}

int main()
{
    float a, b, c;
    cout << "Enter a and b:";
    cin >> a >> b;

    try
    {
        c = division(a, b);
        cout << "c = " << c;
    }
    catch(int e)
    {
        cout << "Exception! denominator=" << e;
    }

    return 0;
}
```