Templates

Function Templates

T -> Type parameter

```
template <class T>
T max(T x, T y) {
   if (x > y) return x;
   else return y;
}
```

The compiler generates code for us when it comes across statements such as:

```
i = max(j, k);
```

For each call, the compiler generates the complete function, replacing the type parameter with the type or class to which the arguments belong.

Function Templates

It also works for user defined types:

```
Date d1,d2,d3;
d3 = max(d1,d2);
```

Provided that we have an operator> for the Date class.

Function templates can themselves be overloaded:

```
template <class T, class S>
T max(T x, S y) {
  if (x > y) return x;
  else return y;
}
```

Class Templates

- Member functions of a class template are themselves function templates with the same template header as their class.

```
template<class T>
class X{
   T square(T  t) { return t*t; }
};
```

Class Templates

Consider the class declaration:

```
X<short>  x;
```

It is instantiated by the compiler, replacing the template parameter T with the type passed to it.

Therefore, the compiler generates the following class and object

```
Class   X_short{
  short  square(short  t) {return t*t;}
};
X_short  x;
```

Class Stack

```
#include <iostream>
using namespace std;
template <class T>
class Stack {
private:
    T* array;
    int top;
    int capacity;

public:
    Stack(int size = 20) {
        array = new T[size];
        capacity = size;
        top = -1;
    }
   ~Stack() {
        delete[] array;
    }
    void push(const T& x) {
        if (top == capacity - 1) {
            cout << "Stack
Overflow\n";
            return;
        }
        array[++top] = x;
    }

    void pop() {
        if (top == -1) {
            cout << "Stack
Underflow\n";
            return;
        }
        top--;
    }
    void display() {
        if (top == -1)
            cout << "Stack is
Empty\n";
        else{
        for (int i = top; i >= 0; i--)
            cout << array[i] << " ";
        cout << endl;
        }
    }
};
int main() {
    Stack<int> s(10);
   s.push(10); s.push(20);
s.push(30); s.display();
    s.pop();s.display();
    return 0;
}
```

Class template with multiple type of parameters

- Example:

```
template <class T, class U>
Class KeyValuePair{
    T key;
    U value;


}
```

Non-type parameters

- Like function templates, a class template may have several type parameters. Moreover, some of them may be ordinary non-type parameters.


```
template <class T, int n, class U, class V>
```

- Templates are instantiated at compile time, and so, values passed to non-type parameters must be constants.

Non-type parameters

```
template<class T, int n>
class X {};

int main()
{
  X<float, 22> x1;  //OK
  const int n=44;
  X<char, n> x2;    //OK
  int m=66;
  X<short, m> x3;  //error! – m  must be a constant
}
```

Class Stack with Non-type parameter

```
#include <iostream>
using namespace std;
template <class T, int size>
class Stack {
private:
    T array[size];
    int top;

public:
    Stack() {
        top = -1;
    }

   ~Stack() {
      //Destructor can be skipped
since memory has not been
allocatd dynamically
    }
    void push(const T& x) {
        if (top == size - 1) {
            cout << "Stack
Overflow\n";
            return;
        }
        array[++top] = x;
    }

    void pop() {
        if (top == -1) {
            cout << "Stack
Underflow\n";
            return;
        }
        top--;
    }
    void display() {
        if (top == -1)
            cout << "Stack is
Empty\n";
        else{
        for (int i = top; i >= 0; i--)
            cout << array[i] << " ";
        cout << endl;
        }
    }
};
int main() {
    Stack<int,10> s;
   s.push(10); s.push(20);
s.push(30); s.display();
    s.pop();s.display();
    return 0;
}
```

Template Specialization

To customize the behavior of a template for a specific type or set of types

Two types of template specialization:

- **Full specialization**: Specializes the entire template for a specific type.
- **Partial specialization**: Specializes part of the template for certain types or conditions (only for class templates).

Template Specialization (Function)

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

template <typename T>
T square(T value) {
    return value * value;
}

// Specialization for string
template <>
string square<string>(string s) {
    string result;
    int length = s.length();
    for (int i = 0; i < length; ++i)
        result += s;
    return result;
}
```

```
int main() {
    int n = 5;
    cout << "Square of " << n << " is: " << square(n) << endl;

    double d = 3.5;
    cout << "Square of " << d << " is: " << square(d) << endl;

    string str = "Him";
    cout << "Square of string "<< str << " is: " << square(str);

    return 0;
}
```

Full Specialization (Class)

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

template <typename T>
class Calculator {
public:
    T add(T a, T b) {
        return a + b;
    }
};

// Full specialization for char type
template <>
class Calculator<char> {
public:
    char add(char a, char b) {
        cout << "Specialized for char-";
        return a;  // Return first character
    }
};

int main() {
    Calculator<int> intCalc;
    cout << "Sum of ints: " << intCalc.add(5, 3) <<
endl;
    Calculator<char> charCalc;
    cout << "Sum of chars: " << charCalc.add('A', '!')
<< endl;
    return 0;
}
```

Partial Specialization (Class)

```
#include <iostream>
#include <string>
using namespace std;
template <typename K, typename V>
class KeyValuePair {
public:
    K key;
    V value;
    KeyValuePair(K k, V v) : key(k), value(v) {}

    void display() {
        cout << "Key: " << key << ", Value: " << value << endl;
    }
};

// Partial specialization when the key is of type int
template <typename V>
class KeyValuePair<int, V> {
public:
    int key;
    V value;
    KeyValuePair(int k, V v) : key(k), value(v) {}

    void display() {
        cout << "Key (int): " << key << ", Value: " << value << endl;
    }
};

// Partial specialization when the key is a pointer type
template <typename K, typename V>
class KeyValuePair<K*, V> {
public:
    K* key;
    V value;

    KeyValuePair(K* k, V v) : key(k), value(v) {}

    void display() {
        cout << "Key (pointer): " << *key << ", Value: " << value <<
endl;
    }
};
```

```
int main() {
    // Using the generic template
    KeyValuePair<string, string> pair1("Name", "John");
    pair1.display();

    // Using the partial specialization for int key
    KeyValuePair<int, string> pair2(101, "Alice");
    pair2.display();

    // Using the partial specialization for pointer key
    int x = 55;
    KeyValuePair<int*, string> pair3(&x, "Pointer Key");
    pair3.display();

    return 0;
}
```

Vector

- a dynamic array that can resize itself when elements are inserted or removed
- Can be randomly accessed by [] or at()
- allocates memory automatically
- Can be used with iterators
- Properties
  - Size
  - Capacity

Constructor

- vector<int> v1; // Creates an empty vector of integers
- vector<int> v2(10); // Creates a vector of size 10 with default values (0 for int)

Access

- int x = v1[0]; // first element
- int y = v1.at(1); // second element, with bounds checking

```
v.push_back(10);
v.pop_back();
v.clear()
```

Using Iterators

```
for (vector<int>::iterator it = v1.begin(); it != v1.end(); ++it) {
    cout << *it << " ";
}
```

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

int main() {
    // Declare a vector of integers and add some elements
    vector<int> vec = {10, 20, 30, 40, 50};

    // Create an iterator to traverse the vector
    vector<int>::iterator it;

    // Display the elements using the iterator
    cout << "Vector elements: ";
    for (it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";   // Dereferencing the iterator to
access the element
    }
    cout << endl;

    // Modify vector elements using iterator
    for (it = vec.begin(); it != vec.end(); ++it) {
        *it += 5;   // Adding 5 to each element
    }

    // Display modified vector elements
    cout << "Modified vector elements: ";
    for (it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```