C++ templates

What is template?

- Template is a C++ language construct that support **generic programming**
  - Type-independent patterns that can work with **multiple data types.**
- Templates can be used to create a family of classes or functions.
- Template may be considered similar to macros.
  - When an object is defined with a specific data type, the template definition for that class is substituted with the required data type.
- It also allows the compiler to generate multiple versions of a function by allowing parameterized data types.

C++ templates

- Function Templates
  - These define logic behind the algorithms that work for multiple data types.
- Class Templates
  - These define generic class patterns into which specific data types can be plugged in to produce new classes.

Function templates

- Function to swap two integers

```
void swap(int &x, int &y) {
      int temp = x;
      x = y;
      y = temp;
}
```

- Function to swap two doubles

```
void swap(double &x, double &y) {
      double temp = x;
      x = y;
      y = temp;
}
```

- Function to swap two Complex

```
void swap(Comple &x, Complex &y) {
      Complex temp = x;
      x = y;
      y = temp;
}
```

Function template(2)

- It is observed from the previous slide that the function logic remains the same.
- Only thing that differs is the data type.
- We can define a function template that could be used for any data type.

Format for function template

keyword

Defines the generic class

```
template <class T>
returntype function_name (argument of type T){

    //function body
    // use type T wherever required in your function
    ...
}
```

Example

```
// include namespace std not required here
template <class T>
void swap(T &x, T &y) {
      T temp;
      temp = x;
      x = y;
      y = temp;
}

int main(){
      int a=20,b=10;
      swap(a,b);
      std::cout << “ a = “ << a << “ b = “ << b << std::endl;

      float m= 9.8, n = 6.7;
      swap(m,n);
      std::cout << “ m = “ << m << “ n = “ << n << std::endl;

      Complex c1(5,7), c2(9,11);
      swap(c1,c2);
      std::cout << c1 << c2;
}
```

Another example

```
template <class T>
void swp(T &x, T &y) {
      T temp;
      temp = x;
      x = y;
      y = temp;
}

template <class T>
void bubble(T arr[], int n){
      for(int i=0; i<n-1 ;i++)
          for(int j=n-1; j>=i ; j--)
              if(arr[j] < arr[j-1])
                  swp(arr[j],arr[j-1]);
}
```

```
int main(){
      int x[5] ={ 7, 5, 3, 4, 6 };
      float y[5] = { 1.2, 7.8, 5.6, 3.4, 6.7 };

      bubble(x,5);
      bubble(y,5);

      ...
}
```

Function template with multiple parameters

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

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

int main(){
        int x = 10;
        char y ='c';
        cout << " max is " << max(x,y) << endl;

        double z = 7.5;
        cout << " max is " <<  max(x,z);
}
```

Class template

- Class template allows us to define a class with the data type as a parameter and to use it later to create a class with any specific data type.
  - A **Generic class**
- Generic format

```
template <class T>
class  class_name{

     //use the type T
     //wherever required in class members
     ...
}
```

Example

```
template <class T>
class Vector{
    T* vec;
    int size;
public:
    Vector (int m){
        vec = new T[size=m];
        for(int i=0;i<size; i++)
            vec[i] = 0;
    }
    Vector (T *a, int m){
        vec = new T[size=m];
        for(int i=0;i<size; i++)
            vec[i] = a[i];
    }
    T sum(){
        T sum = 0;
        for(int i=0;i<size; i++)
            sum += vec[i];
        return sum;
    }
};
```

Type to be provided here

```
int main(){
    Vector <int> v0(10);
    int intar[]={4,5,6,7,8};
    Vector <int> v1(intar,5);
    float arr[]={1.2,3.5,5.6,8.7, 9.8};
    Vector <float> v2(arr,5);
    std::cout << " Sum black int vec= "
<< v0.sum() << std::endl;
    std::cout << " Sum int vec= " <<
v1.sum() << std::endl;
    std::cout << " Sum float vec = " <<
v2.sum() << std::endl;
}
```

Using Default Arguments with Template Classes

- A template class can have a default argument associated with a generic type.
  - template <class X=int> class myclass {   //...   };
- The default value is used when no explicit value is specified when the class object is instantiated.
- Default arguments for non-type parameters are specified using the same syntax as default arguments for function parameters.

Example

```
template <class T=int>
class Vector{
    T* vec;
    int size;
public:
    Vector (int m){
        vec = new T[size=m];
        for(int i=0;i<size; i++)
            vec[i] = 0;
    }
    Vector (T *a, int m){
        vec = new T[size=m];
        for(int i=0;i<size; i++)
            vec[i] = a[i];
    }
    T sum(){
        T sum = 0;
        for(int i=0;i<size; i++)
            sum += vec[i];
        return sum;
    }
};
```

No type provided, default type assigned

```
int main(){
    Vector <int> v0(10);
    int intar[]={4,5,6,7,8};
    Vector v1(intar,5);
    float arr[]={1.2,3.5,5.6,8.7, 9.8};
    Vector <float> v2(arr,5);
    std::cout << " Sum black int vec= "
<< v0.sum() << std::endl;
    std::cout << " Sum int vec= " <<
v1.sum() << std::endl;
    std::cout << " Sum float vec = " <<
v2.sum() << std::endl;
}
```

End of template