Chapter 8: Templates

BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

Introduction

- Template supports generic programming, which allows developing reusable software components such as functions, classes supporting different data types in a single frame work.
  - For example , class vector can be used for int, float or double vector type. Function ‘ addition’ can be used for addition of any type of data
- When object or variable of specific type is defined for actual use, the template definition for class or function is substituted with required data type.
- Types of templates :
  - Function template
  - Class template

Function template

- A function template specifies how an individual function can be constructed.
- Syntax :

```
template <class / typename typeT >
ret-type func_ name ( param List of typeT )
{
/ / function body
}
```

  - Where, typeT is a placeholder

```cpp
template <class T>
void swaping (T & x, T & y)
{   T z;
    z=x;
    x=y;
    y=z;
}

main( )
{
char ch1,ch2;
cout<<"Enter two characters:"<<endl;
cin>>ch1>>ch2;
swaping(ch1,ch2);
cout<<ch1<<ends<<ch2<<endl;
int a,b;
cout<<"Enter two integers:"<<endl;
cin>>a>>b;
swaping(a,b);
cout<<a<<ends<<b<<endl;

float p,q;
cout<<"Enter two real
numbers:"<<endl;
cin>>p>>q;
swaping(p,q);
cout<<p<<ends<<q<<endl;
}
```

Output:

```
Enter two characters:
r t
t r
Enter two integers:
7 9
9 7
Enter two real numbers:
5.6 8.2
8.2 5.6
```

Function template with multiple parameters

```
template <class T1, class T2>
void sum (T1 x,T2 y)
{
    cout<<"Sum="<<x+y<<endl;
}

main( )
{
sum(2, 3);
sum(9.8, 6);
sum(3, 4.5);
}
```

Output:

Sum=5

Sum=15.8

Sum=7.5

Overloading function template

- Function template can also be overloaded
  - 1. Overloading with function
  - 2. Overloading with function template

1. Overloading with function

```
template <class T>
void display(const T & a)
{
cout << a << ends;
}

void display(int & a,  int n) //overloaded display()
{
int ctr;
for(ctr=0;ctr<n;ctr++)
cout << a << ends;
}
```

```
main()
{
char c = 'a';
display(c);
cout<<endl;
display(100,3);
cout<<endl;
display(10.85);
}
```

Output:

a

100 100 100

10.85

2. Overloading with function template

```
template <class T>
void print( T a)
{         cout<<a<<endl;
}

template <class T>
void print( T a, int n)
{
int i;
cout<<endl;
for (i=0; i<n; i ++)
cout<<a<<ends;
}
```

```
main()
{
print(1);
print(3.4);
print(455,5);
print("hello",3);
}
```

Output:

1

3.4

455 455 455 455 45

hello hello hello

Class template

- similar to functions, classes can also be declared to operate on different data types. Such classes are class templates.
- These classes model a generic class which support similar operations for different data types.
- **Syntax :**

```
template <class T>
class classnm
{           T member1;
            T member2;
public:
            T fun();          };
```

- **Objects for class template is created like:**

```
classnm <datatype> obj;
obj.memberfun();
```

Member function as template function

- If the member functions are defined within the template class body, then they are defined as normal functions
- If the member functions are defined outside the template class body, they should always be defined with the full template definition.
- Syntax:

```
 template <class typeT >
ret-type classname< typeT>  :: func_ name ( param List)
{
/ / function body
}
```

```
template <class T>
class Add
{
T a, b;
public:
void getdata()
{
cout<<"Enter 2 nos : ";
cin>>a>>b;
}
void display();
};

template <class T>
void Add <T>::display( )
{
cout<<"sum="<<a+b<<endl;
}
```

```
main()
{
Add <int> ob1;
Add <float> ob2;
cout<<"For integer type"<<endl;
ob1.getdata( );
ob1.display( );
cout<<"For float type"<<endl;
ob2.getdata( );
ob2.display( );
}
```

Output:

```
For integer type
Enter 2 nos : 5 7
sum=12
For float type
Enter 2 nos : 7.4 2.1
sum=9.5
```

Class template with multiple parameter

```
template <class Type1, class Type2>
class myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b)
{
        i = a; j = b;
}
void show()
{
cout << i << ' ' << j << '\n';
}
};
```

```
main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('A', "Electrical Engineering");
ob1.show(); // show int, double
ob2.show(); // show char, char *
}
```

Output:

```
10 0.23
A Electrical Engineering
```

Non- template type argument

```
template <class T, int N>
class Array
{
T a[N];
public:
void setdata(T value, int i)
{
a[ i ]=value;
}
T display(int i)
{
return a[i];
}
};
```

```
 main()
{          Array <float ,5> ob1;
           Array <int ,10> ob2;
cout<<"For float type"<<endl;
           ob1.setdata( 1.5, 1);
           ob1.setdata( 2.5, 2);
           ob1.setdata( 3.5, 3);
           cout<<ob1.display(2)<<endl;
cout<<"For int type"<<endl;
           int num;
           for(int i=0; i<10; i++)
           {          cin>>num;
                      ob2.setdata(num, i );
                      }
for(int i=0; i<10; i++)
           cout<<ob2.display( i)<<ends;
}
```

Output:

For float type

2.5

For int type

34

46

346

67

78

90

06

32

56

89

34 46 346 67 78 90 6 32 56 89

Default argument with class template

```
template <class T=float, int n=5>
class Array
{        T a[n];
public:
void setdata()
{        for (int i=0; i<n; i++)
          cin>>a[i];
}
void display();
};
```

```
 template <class T, int n>
void Array<T, n>::display()
{
   T sum=0;
   for (int i=0; i<n; i++)
      sum+=a[i];

cout<<"Sum="<<sum<<endl;
}
```

```
main()
{
cout<<"For float type"<<endl;
      Array < > ob1;
     ob1.setdata( );

     ob1.display();


cout<<"For int type"<<endl;
      Array <int,3> ob2;
     ob2.setdata();

     ob2.display();
}
```

Derived class template

- Three cases of derived class templates:
  - 1. Deriving template from a template
  - 2. Deriving non-template from template
  - 3. Deriving template from non template

1. Deriving template from a template

```
template <class T>
class base
{
T a;
public:
base(T x):a(x){}
void display()
{
    cout<<"a="<<a;
}
};
```

```
template <class T,class T1>
class derived: public base<T>
{    T1 b;
public:
    derived(T x, T1 y): base<T>(x),
                          b(y){ }
    void display()
{
    base<T>::display();
    cout<<"b="<<b;
}
};
```

```
main()
{
derived<int,float> d1(20, 30.5);
d1.display();
}
```

2. Deriving non-template from template

```
template <class T>
class base
{
T a;
public:
base(T x):a(x){}
void display()
{
    cout<<"a="<<a;
}
};
```

```
class derived: public base<int>
{
    int b;
public:
    derived(int x, int y): base<int>(x),
                                        b(y){ }
    void display()
{
    base<int>::display();
    cout<<"b="<<b;
}
};
```

```
 main()
{
derived d1(20, 30);
d1.display();
}
```

3. Deriving template from non template

```
class base
{
int a;
public:
base(int x):a(x){}
void display()
{
    cout<<"a="<<a;
}
};
```

```
template <class T>
class derived: public base
{    T b;
public:
    derived(int x, T y): base(x),
                                  b(y){ }
    void display()
{
    base::display();
    cout<<"b="<<b;
}
};
```

```
 main()
{
derived<int> d1(20, 30);
d1.display();
}
```