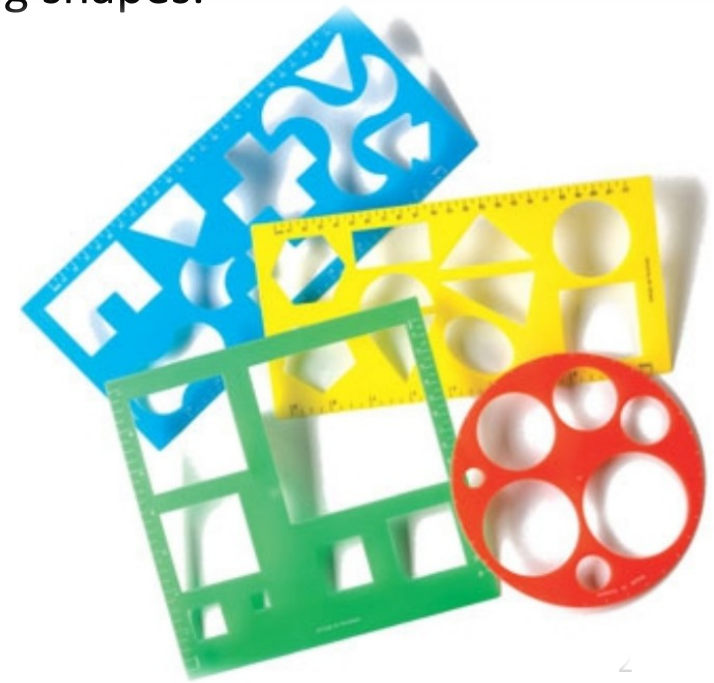
Function Templates

CS(217) Object Oriented Programming

Abeeda Akram

What is a Template?

- A **template** is a model or mold that can be used as a guide to create similar things.
- For example template is like a stencil ruler by using that we can draw same shape with different colors.
  - Once the stencil is created, it can be used many times for drawing shapes.



Templates in C++

- The template is one of C++'s most sophisticated and high-powered features that is used for generic programming.
  - It is a mechanism for automatic code generation, and allows for substantial improvements in programming efficiency.
- Using templates, we can create.
  1. Generic functions
  2. Generic classes
  - In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.
  - We can use one function or class with several different types of data without explicitly recode specific versions for each data type.

Function Templates and Function Overloading

- **Function templates** are special functions that serve as a framework or mold for creating other similar functions
  - without explicitly recoding specific versions for each data type.
- In function overloading we need to write different functions for handling different datatypes, but with similar operations or code.
- A function template can be used
  - To remove the overhead of function overloading for different datatypes with similar implementation of code.
- Function templates cannot be used
  - When Overloaded functions have different code or number of parameters.

Function Templates and Function Overloading

- In function overloading we need to write different functions for handling different datatypes, but with similar code.
- Example, find maximum of two values, we need to write four different functions with same code to handle different datatypes.
- We can replace all four functions with **single template function**.

```
// 1 int
int maximum(int x, int y){
    if (x>y)
        return x;
    else
        return y;
}
// 2 float
float maximum(float x, float y){
    if (x>y)
        return x;
    else
        return y;
}
```

```
// 3 double
double maximum(double x, double y){
    if (x>y)
        return x;
    else
        return y;
}
// 4 char
char maximum(char x, char y){
    if (x>y)
        return x;
    else
        return y;
}
```

Function Templates **Template header**

- First write keyword **`template`**
- Followed by List of template type parameters in angle brackets (**<** and **>**)
- Each parameter is preceded by keyword **`class`** or **`typename`**

```
template < class Type >
template < typename Type >
template < typename Type1, typename Type2>
```

- The labels **`Type, Type1, Type2`** are called a *template type parameters*.
- **Type parameter** is simply a placeholder or label that is replaced by an actual datatype, when the function is invoked.
- Type parameters can be used as
  1. Arguments to function
  2. Return type of function
  3. Local variables within function

Function Templates **Definition**

1. Add template header before function.
2. Define function with generic code, use type parameter in place of actual datatype.

Template function definition to find maximum of two values.

```
// Template function
// Type parameter is used here as function arguments and return type
template < typename Type >
 Type maximum (Type x, Type y){
  if (x>y)
        return x;
        else
        return y;
  }
  //No code should be written between template header and function definition
```

Function Templates **Call with basic datatypes**

- At compile time, when compiler finds a call to template function,
  - It generates the complete copy of template function by replacing the type parameters with the datatypes to which the calling arguments belong.
  - This is called ***implicit specialization*** or *function template instance*.
- If template function is never called, then no copy of template function is created by compiler.
- Compiler will generate four copies of template function maximum for **int**, **float**, **double** and **char**.

```
// Template function
    template < typename Type >
    Type maximum(Type x, Type y){
    if (x>y)
          return x;
    else
          return y;
    }
```

```
void main(){
       cout << maximum(55,88);     // int
       cout << maximum('A', 'x'); // char

    float f1= 3.9, f2=5.5555;
       cout << maximum(f1,f2);     // float

    double d1= 3.9, d2=5.5555;

       cout << maximum(d1,d2);     //

       double

}
```

Function Templates **Call with class objects**

- Compiler can also generate copy of template function by replacing the type parameters with the user defined **class objects**.
- Any operator or function call that is used with types must be defined in classes, otherwise compile time error will occur.
- The operator functions (>) and (<<) should be overloaded in Point class.
- Compiler will generate a copy of template function maximum for **Point class objects**.

```
template < typename Type >
Type maximum(Type x, Type y){
 if (x>y)
        return x;
 else
        return y;
 }
```

```
void main(){

        Point p1(3, 9), p2(11, 10); //
        Point
        cout << maximum( p1, p2);
}
```

Function Templates in C++

- Function templates cannot be used when
  - Overloaded functions have different code or number of parameters.
  - We **cannot** replace following functions with single template function.

```
// 1 int two parameters
int maximum(int x, int y){
    if (x>y)
        return x;
    else
        return y;
}
```

```
// 2 int three parameters
int maximum(int x, int y, int z){
    if (x>y && x>z)
        return x;
        else if (y>x && y>z)
        return y;
        else
          return z;

}
```

Function Templates **Example Swap**

- Type parameter can be used as placeholder for **references**.
- Template function definition to **swap** two values of any datatype.

```
template < typename T >
void swap (T & x, T & y){// Type parameter used as function arguments
 T temp = x;  // Type parameter used as local variable
 x = y;
 y = temp;
 }

 void main(){
    float f1= 3.9, f2=5.5555;
       swap (f1,f2);    // Compiler will generate one copy for float
    int i1= 3559, i2=587;
       swap (i1,i2);    // Compiler will generate one copy for int

 }
```

Function Templates **Example findMin**

- Type parameter can be used as placeholder for **pointers**.
- Template function can also take normal parameters along with Type parameters.
- Template function definition to **find minimum value from array** of any datatype.
  - size of array is always integer value irrespective of data type.

```
template < typename T >
 T findMin (T *arr, int size){  // Type parameter as arguments and return type
 T min = arr[0];      // Type parameter as local variable
 for(int i =1; i< size; i++)
 if (arr[i]< min)
        min = arr[i];
 return min;
 }

 void main(){
    float arr[5] = {3.5, 6.7, 10.4, 11.455, 1.44};
      cout<< findMin (arr , 5);    // Compiler will generate one copy for float
 }
```

Function Templates **More than one Generic Types**

- Template function can be designed with more than one template type parameters.
- Template function to print data of different or same variable types.

```
template < typename T1, typename T2 >
void printData (T1 a, T2 b){  // Type parameter as arguments
 cout << "First is : " << a << endl;
 cout << "Second is: " << b << endl;
}

void main(){
      printData(10 , 'D');// one copy for int and char
      printData("I Like Programming" ,10.5);// one copy for char* and float
      printData(Point(4, 5) ,10); // one copy for Point and int
      printData(555 ,10); // one copy for int and int
}
```

Function Templates **Examples**

- Generic Template functions can be designed for following
  - Find Maximum value from array of any datatype
  - Calculate average value of data in arrays
  - Sort arrays in ascending or descending order
  - Find common elements in two arrays
  - Merge two arrays.
  - Print data of 1-D or 2-D arrays.
  - Compact arrays by removing some data.