Function Templates

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 in C++ and function overloading

- 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 in C++ 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 in C++: 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
  - Arguments to function
  - Return type of function
  - Local variables within function

Function Templates in C++: Template header

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 in C++: call

- 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 in C++: call with class objects

- Compiler can also generate copy of template function by replacing the type parameters with the user defined **class objects**.
- Any operators or function calls that are 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 in C++

- 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 in C++

- 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 in C++:More than one Generic Types

- Template function can be designed with more than one template typ 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 in C++:More than one Generic Types

Write a template function that returns the average of all the elements of an array. The arguments to the function should be the array name and the size of the array (type int). In main(), exercise the function with arrays of type int, long, double, and char.

Function Templates overloading

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

// Overloaded template function with three
parameters
     template < typename T >
     T maximum(T x, T y, T z){
        if (x>y && x>z)
          return x;
        else if (y>x && y>z)
          return y;
        else
         return z;

   }
```

```
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


// overloaded int called
      cout << maximum(55,88,39);
// overloaded float called
      cout << maximum(5.7, 9.88, 3.9);

}
```

Function Templates overloading

```
// Template function with two Parameter
     template < typename T >
     T maximum(T x, T y){
        if (x>y)
                return x;
        else
                return y;
      }
// Overloaded template function with three
parameters
     template < typename T >
     T maximum(T x, T y, T z){
        if (x>y && x>z)
          return x;
        else if (y>x && y>z)
          return y;
        else
         return z;

   }
```

```
// Overloaded template function with array
  template < typename T >
  T maximum (T * arr , int size){
        T max = arr[0];
        for(int i =1; i< size; i++)
                if (arr[i]> max)
                        max = arr[i];
        return max;
  }
  void main(){
        int arr[5] = {1, 5, 3, 9, 7};
  // overloaded int array called
        cout<< maximum (arr , 5);
  // int called with two parameters
        cout<< maximum (arr[0], arr[2]);
  // overloaded int called with three parameters
        cout<< maximum (arr[3], arr[1], arr[4]);
  }
```

Function Templates

- Function templates cannot work well in some situations when
  - Some functions need different code for specific datatypes but number of parameters remain same.
  - We **cannot** overload template functions to resolve this issue.

```
// Template function with two
Parameter
    template < typename T >
    T maximum(T x, T 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

      char arr[5] = "sdsd";
      char arr2[5] = "sfgf";
      cout << maximum(arr, arr2); // char *
// Wrong comparison for character arrays as
compare first character only
}
```

Function Templates specialization

```
// Add Specialized Template function with two Parameters for char * data type
     template <>
     char* maximum <char *> (char* x, char* y){
        if (strcmp(x , y) == 1)
                return x;
        else
                return y;
      }

    void main(){
        char arr[5] = "sdsd";
        char arr2[5] = "sfgf";
        cout << maximum(arr, arr2); // char *
    // Now specialized function is called and work properly for char *

        cout << maximum("abcd","axyz");// const char *
    //It will not work for constant character arrays
    }
```

Function Templates specialization

```
template <>
const char* maximum <const char *> (const char* x, const char* y)
{
   if (strcmp(x , y) == 1)
          return x;
   else
          return y;
 }

 void main(){
   cout << maximum("abcd","axyz");// const char *
   cout << maximum("axyz","abcd");// const char *
 // Now the specialized function is called and work properly for const char *
}
```