Function Template Specialization

CS217 Object Oriented Programming

Function Templates in C++

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

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

```
// 2 int
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 **Overloading**

- Function templates can be overloaded to handle this issue.
  - Function name and return type remain same.
  - Change number of parameters in template function.
  - Change implementation of code accordingly.

- Example:
  - We have designed the template function to find maximum of two values.
  - Overload template function to find maximum of three values
  - Overload template function to find maximum from an array of any size.

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]);
  }
```

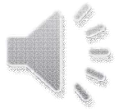
Function Templates in C++

- 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**

- Function templates cannot work well in some situations when
  - Some functions need different code for specific datatypes but number of parameters remain same.
  - Template specialization is to design an explicitly specialized function for a particular datatype along with existing template function.

1. Add empty template header before function

```
template <>
```

2. Add datatype name for specialization after function name <>

```
return type functionname < datatypename > (parameter list){
        // implementation of function
}
```

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**

```
// Add another Specialized Template function with two Parameters for const char * data
type
     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 *
    }
```

Function Templates **Specialization**

```
// Add another Specialized Template function with three Parameters for char * data type
    template <>
    char* maximum <char *> (char* x, char* y, char* z )
    {
       if (strcmp(x, y) > 0 && strcmp(x, z) > 0)
              return x;
       else if (strcmp(y, x) > 0 && strcmp(y, z) > 0)
              return y;
       else
              return z;
     }

    void main(){
       char arr[] = “abc";
       char arr2[] = “def";
       char arr2[] = “fgh";
       cout << maximum(arr, arr2, arr3); // char * three parameters
    }
```


Function Templates **Specialization**

```
// Add another Specialized Template function with three Parameters for const char * data type
    template <>
    const char* maximum <const char *> (const char* x, const char* y, const char* z )
    {
       if (strcmp(x, y) > 0 && strcmp(x, z) > 0)
              return x;
       else if (strcmp(y, x) > 0 && strcmp(y, z) > 0)
              return y;
       else
              return z;
     }

    void main(){
       cout << maximum("abcd","axyz");// const char * two parameters
       cout << maximum("abc", "def", "fgh"); // const char * three parameters
    }
```

Function Templates **Specialization or Overloading**

```
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};
        cout<< maximum (arr , 5); // const int*

        char arr[5] = "abcd";
        cout << maximum(arr, 5); // const char*


        char arr2[5][4] = {"abc", "def", "fgh", "ljk", "lmn"};
        cout << maximum(arr2, 5);
```

Template function will not work for arrays of strings
What to do in this case Specialization or Overloading?

```
}
```

Function Templates **Specialization or Overloading**

```
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(){

        char** ptr = new char* [5];
        for (int i = 0; i < 5; i++)
                ptr[i] = new char[4];

        strcpy(ptr[0], "abc"); strcpy(ptr[1], "def"); strcpy(ptr[2], "ghi");
        strcpy(ptr[3], "jkl"); strcpy(ptr[4], "lmn");
        cout << maximum(ptr, 5);

        for (int i = 0; i < 5; i++)
                delete ptr[i];
        delete ptr;
```

Template function will not work for dynamic arrays of strings
What to do in this case Specialization or Overloading?

```
}
```