Lecture 16.2

Topics:
1. C++ Template Mechanism – Function
2. Class Templates

---

1. C++ Template Mechanism — Function Templates

What is template? In C++, template mechanism allows class names and parameters to be passed to functions and classes.

Why is template useful?

Consider the following group of overloaded functions for `swap()`.

```
void swap(char &, char &);
void swap(int &, int &);
void swap(long &, long &);
void swap(float &, float &);
void swap(double &, double &);
```

The basic task is to swap two values specified on the heading. The implementation may be chosen so that the operations inside the functions are the same except for the types of data. One possible implementation is as follows:

```
void swap(double& d1, double& d2) {
  double dTemp;
  dTemp = d1;
  d1 = d2;
  d2 = dTemp;
  return;
}
```

Clearly, by replacing all `double`'s with either `char`'s, `int`'s then other overloaded versions can be obtained. What if a template is available for replacing these types? Certainly, this helps the user to simplify his or her code.

1.1 Function Templates

If each of the overloaded versions of a function uses the same logic and operations, regardless of the types of its arguments and local objects, then a generic function can be used. This generic function does not depend on any specific type through the template mechanism. The general form of template declaration is as follows:

```
template< class identifier >
FunctionDefinition
```

where

- **`template`** is the keyword to invoke the template declaration.
- **`identifier`** is the actual type to be used in function instantiation.
- **`class`** is the required keyword.
- **`FunctionDefinition`** provides the structure of function.

The keyword **`class`** is somewhat a misnomer. This is because `identifier` can be of any types that may be built-in or derived. However, to avoid using a new keyword for this purpose the keyword **`class`** has been used.

In the above form, `FunctionDefinition` is a function construct. It has a function heading and body where identifiers are used in places of actual type and data that will be supplied at the time of instantiation.

Questions on the use of function template come from the procedure that compiler uses to instantiate (generate, define) the requested function (an instantiation). More on this will be given shortly (after class template).

1.2 Example

Example 1

```
/**
 *Program Name:  cis25L1621.cpp
 *Discussion:    Function Template
 */
#include <iostream>
using namespace std;

template<class T>
void swapT(T& v1, T& v2) {
  T temp;
  temp = v1;
  v1 = v2;
  v2 = temp;
  return;
}

int main() {
  int i1 = 10;
  int i2 = 20;

  swapT(i1, i2);
  cout << "\ni1 = " << i1 << " ; i2 = " << i2 << endl;

  double d3 = 30.5;
  double d4 = 40.5;

  swapT(d3, d4);
  cout << "\nd3 = " << d3 << " ; d4 = " << d4 << endl;

  return 0;
}
```

OUTPUT

```
i1 = 20 ; i2 = 10

d3 = 40.5 ; d4 = 30.5
```

It is easy to generate and use function template as the above example shows.

Notice that `T` is just a name and it can be replaced with any other name. Clearly, two overloaded functions are generated to handle `int` and `float` data separately.

In a way, this is where the user should be aware of what is going on within the compiler. This will help to decide when or if the use of template is needed. A frequent implementation of function template is found in class template.

2. Class Templates

Similar rationale for using function template can be made for class template. For example, in defining `class Stack,` data can be either of `int` type or any other types. Certainly, different stack classes can be created for all of these cases.

However, class template would be a great tool to use in this situation. It is preferable to declare a generic class template in which a type-parameter provides the type of the data member.

2.1 Definition/Syntax

The general form of class template is as follows:

```
template < class identifier >
ClassDefinition
```

where

- **`template`** is the keyword to invoke the template declaration.
- **`identifier`** is the actual type (value) to be used in class instantiation.
- **`class`** is the required keyword.
- **`ClassDefinition`** provides the structure of class.

Despite that class template is great, it has been argued about the use of the template construct. Class template, as well as function template, provides a generic structure (definition) for creating classes and functions. So why is the hesitation? The answer lies partially within the mechanism that a compiler is used to handle template.

When a class template is used, compiler must locate the template definition first. Then it constructs the layout (specific definition) for the instantiated class. If templates use other templates or nested then all of these templates need to be instantiated. Note that by providing parameter(s) to class template generic class is formed. It is generic in the sense that parameters are merely placeholders for specific data types; these parameters are not data types.

When a function template is used, the compiler basically follows the same steps as in class template. The different here is that object code for the instantiated function is generated and placed in the code of the calling module.

The final thought is on the additional tasks that the compiler has to do to prevent multiple copies of the same instantiation. These steps require overhead and may slow down the process.

2.2 Example

Example 2

```
/**
 *Program Name:  cis25L1622.cpp
 *Discussion:    Class Template
 */
#include <iostream>
using namespace std;

template<class T>
void swapT(T& v1, T& v2) {
  T temp;
  temp = v1;
  v1 = v2;
  v2 = temp;
  return;
}
```

```
template<class T>
class Value {
  T tValue;
public:
  Value() {
    cout << "\nConstructor -- Creating object!\n";
    tValue = 0;
  }
  ~Value() {
    cout << "\nDestructor -- Removing Object!\n";
  }

  void setValue(T old) {
    tValue = old;
  }

  T getValue();
};

template<class T>
T Value<T>::getValue() {
  return tValue;
}

int main() {
  int i1 = 10, i2 = 20;
  swapT(i1, i2);
  cout << "\ni1 = " << i1 << " ; i2 = " << i2;
  float f3 = 30.5, f4 = 40.5;
  swapT(f3, f4);
  cout << "\nf3 = " << f3 << " ; f4 = " << f4 << endl;

  Value<int> vObj;
  vObj.setValue(5);
  cout << "\nValue stored in vObj : tValue : "
       << vObj.getValue();

  cout << "\n\n";
  return 0;
}
```

OUTPUT

```
i1 = 20 ; i2 = 10
f3 = 40.5 ; f4 = 30.5

Constructor -- Creating object!

Value stored in vObj : tValue : 5


Destructor -- Removing Object!
```