Class Templates

In this lesson, we'll learn about class templates.

WE'LL COVER THE FOLLOWING

A **class template** defines a family of classes.

Syntax

```
template < parameter-list >
class-declaration
```

Explanation

parameter-list: A non-empty comma-separated list of the template parameters, each of which is either a non-type parameter, a type parameter, a template parameter, or a mixture of any of those.

class-declaration: A class declaration. The class name declared becomes a template name.

Instantiation

The process of substituting the template parameters with the template arguments is called `instantiation`.

> In contrast to a function template, a **class template** is not capable of automatically deriving the template parameters. Each template argument must be **explicitly** specified. This restriction no longer exists with C++17.

Let’s have a look at the declaration of function and class templates:

Function Template Declaration

```
template <typename T>
void xchg(T& x, T&y){
...
}

int a, b;
xchg(a, b);
```

Class Template Declaration

```
template <typename T, int N>
class Array{
...
};

Array<double, 10> doubleArray;
Array<Account, 1000> accountArray;
```

Method Templates

Method templates are function templates used in a class or class template.

> Method templates can be defined inside or outside the class. When we define the method template outside the class, the syntax is quite complicated because we must repeat the class template declaration and the method template declaration.

Let’s have a look at the declaration of the method template inside the class and its definition outside the class:

```
template <class T, int N> class Array{
public:
  template <class T2>
  Array<T, N>& operator = (const Array<T2, N>& a); ...
};

template<class T, int N>
```

```
template<class T2>
  Array<T, N>& Array<T, N>::operator = (const Array<T2, N>& a{

  ...
  }
```

> The destructor and copy constructor cannot be templates.

Inheritance

Classes and class templates can inherit from each other in arbitrary combinations.

- If a class or a class template inherits from a class template, the methods of the base class or base class template are **not** automatically available in the derived class.

```
template <typename T>
struct Base{
  void func(){ ...
};

template <typename T> struct Derived: Base<T>{
  void func2(){
  func();       // ERROR
  }
}
```

There are three ways to make a method from the derived class template available.

3 Solutions:

- Qualification via this pointer: `this->func()`
- Introducing the name `using Base<T>::func`
- Full qualified access `Base<T>::func()`

Templates: Alias Templates

Alias templates, *aka* template typedefs, allow us to give a name to partially bound templates. An example of partial specialization from templates is given below:

```
template <typename T, int Line, int Col> class Matrix{
...
};


template <typename T, int Line>
using Square = Matrix<T, Line, Line>;

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>;

Matrix<int, 5, 3> ma;
Square<double, 4> sq;
Vector<char, 5> vec;
```

To learn more about class templates, click here.

In the next lesson, we’ll learn about some examples of class templates.