Class Templates

In this lesson, we will learn about class templates in embedded programming.

WE'LL COVER THE FOLLOWING

A class template will be defined by placing the keyword `template` in front of the class template followed by type or non-type parameters.

- The parameters are declared by `class` or `typename`.
- The parameter can be used in the class body
- You can define the methods of the class template inside or outside the class.

```
template <typename T, int N>
   class Array{
     T elem[N];
                  ...
```

Instantiation

The process of substituting the template parameter by the template arguments is called **instantiation**. In contrast to a function template, a class template is not capable of automatically deriving the template parameter.

> Each template argument must be explicitly specified.

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 you define the method template outside the class, the syntax is complicated since you must repeat the class template declaration and the method template declaration. Let's take a look at the declaration of the method template. Note: The destructor and copy constructor cannot be templates.

Let's have a look at the declaration of the method template:

```
template <class T, int N> class Array{
public:
template <class T2>
Array<T, N>& operator = ( ...


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

Inheritance

Class and class template can inherit from each other in arbitrary combination.

> 🔑 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.

3 Solutions:

- Qualification via this pointer:

```
this->func()
```

- Introducing the name

```
Base<T>::func
```

- Full qualified access

```
Base<T>::func()
```

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

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

Templates: Alias Templates

Alias templates, also known as template typedefs, allow you to give a name to partially bound templates. An example of partial specialization from templates is offered below:

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

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

Let’s take a look at a few examples of class templates in the next lesson.