Template Parameter

In this lesson, we will discuss template parameters.

WE'LL COVER THE FOLLOWING

Template Parameter

Every template is parametrized by one or more template parameters, indicated in the parameter-list of the template.

C++ supports three different kinds of template parameter:

1. Type parameter

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

2. Non-type parameter

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

3. Template-template Parameter

```
template <typename T, template <typename, typename> class Cont>
class Matrix{ ...
Matrix<int, std::vector> myIntVec;
```

Types

Type parameters are of class types and fundamental types.

```
class Account;

template <typename T>
class ClassTemplate{};

ClassTemplate<int> clTempInt;
ClassTemplate<double> clTempDouble;
ClassTemplate<Account> clTempAccount;

ClassTemplate<std::string> clTempString;
```

Non-Types

Non-types are template parameters that can be evaluated at compile time.

The following types are possible:

- Integers and enumerations
- Pointer on objects, functions, and on the attributes of a class
- References to objects and functions
- `std::nullptr_t` constant

> Float point numbers and strings cannot be used as non-type parameters.

Dependent Names

Firstly, what is a dependent name? A dependent name depends on a template parameter. Let’s break that down further.

```
template<typename T>
struct X : B<T> // "B<T>" is dependent on T
{
    typename T::A* pa; // "T::A" is dependent on T
    void f(B<T>* pb) {
        static int i = B<T>::i; // "B<T>::i" is dependent on T
        pb->j++; // "pb->j" is dependent on T
    }
};
```

Now, the fun starts. A dependent name can be a type, a non-type, or a template template parameter. The name lookup is the first difference between non-dependent and dependent names.

- **Non-dependent names** are looked up at the point of the template definition.
- **Dependent names** are looked up at the point of template instantiation.

If you use a dependent name in a template declaration or template definition, the compiler cannot determine if this name refers to a type, a non-type, or a template template parameter. In this case, the compiler assumes that the dependent name refers to a non-type, which may be wrong. This is when you need the help of the compiler.

> If a dependent name could be a type, a non-type, or a template, you will have to give the compiler a hint.

Use `typename` if the Dependent Name is a Type

```
template <typename T>
void test(){
    std::vector<T>::const_iterator* p1;          // (1)
    typename std::vector<T>::const_iterator* p2; // (2)
}
```

Without the `typename` keyword in line 4, the name `std::vector<T>::const_iterator` would be interpreted as a non-type. Consequently, the `*` stands for multiplication and not for a pointer declaration. This occurs in line 3.

Similarly, if your dependent name is a template, you must give the compiler a hint.

Use `.template` if the Dependent Name is a Template

```
template<typename T>
struct S{
    template <typename U> void func(){}
}
template<typename T>
void func2(){

    S<T> s;
    s.func<T>();              // (1)
    s.template func<T>();     // (2)
}
```

This is similar to what we've already outlined. Compare line 8 and line 9. When the compiler reads the name `s.func`, it interprets it as non-type, meaning that the `<` sign stands for the comparison operator but not opening square bracket of the template argument of the generic method `func`. In this case, you must specify that s.func is a template such as in (2): `s.template func`. When you have a dependent name, use `typename` to specify that it is a type or `.template` in order to specify that it is a template.

> When you have a dependent name, use `typename` to specify that it is a type or `.template` to specify that it is a template.

Let's look at a few examples of template parameters in the next lesson.