Templates: Specialization

In this lesson, we will learn about template specialization.

WE'LL COVER THE FOLLOWING

Templates define the behavior of a family of classes and functions.

- Special types, non-types, or templates as arguments must be treated special
- You can fully specialize templates. Class templates can even be partially specialized.
- The methods and attributes of specialization don’t have to be identical.
- General or Primary templates can coexist with partially or fully specialized templates.

> The compiler prefers fully specialized to partially specialized templates and partially specialized templates to primary templates.

Primary Template

The primary template has to be declared before the partially or fully specialized templates.

- If the primary template is not needed, a declaration is sufficient.

```
template <typename T, int Line, int Column> class Matrix;
template <typename T>
class Matrix<T, 3, 3>{};

template <>class Matrix<int, 3, 3>{};
```

Partial Specialization

The partial specialization of a template

- is only supported for class templates
- has template arguments and template parameters

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

template <typename T>
class Matrix<T, 3, 3>{};

template <int Line, int Column>
class Matrix<double, Line, Column>{};

Matrix<int, 3, 3> m1; // class Matrix<T, 3, 3>
Matrix<double, 10, 10> m2; // class Matrix<double, Line, Column>
Matrix<std::string, 4, 3> m3;  // class Matrix<T, Line, Column>
```

Rules for Partial Specializations:

1. The compiler uses a partial specialization if the parameters of the class are a subset of the template arguments.
2. The template parameters that are not specified must be given as template arguments, as seen in lines 3 and 4 in the code snippet.
3. The number and sequence of the template arguments must match with the number and sequence of the template parameters of the primary template.
4. If you use defaults for template parameters, you don't have to provide the template arguments. Only the primary template accepts defaults.

Rules for Right Specialization:

There are three rules for obtaining the right specialization:

1. The compiler finds only one specialization, and it uses this specialization.

2. The compiler finds more than one specialization. The compiler uses the most specialized template. If the compiler doesn't find the most specialized one, it throws an error.

3. The compiler finds no specialization. It uses the primary template.

If template `A` is more specialized than template `B`:

- `B` can accept all arguments that `A` can accept.
- `B` can accept arguments that `A` cannot accept.

Full Specialization

For a fully specialized template, you must provide all template arguments for the template parameters.

- The number of template parameters is reduced to an empty list.

```
template <typename T> struct Type{
std::string getName() const {
    return "unknown";
    };
};
template <>
struct Type<Account>{
    std::string getName() const {
    return "Account";
};
```

If you define the methods of a class template outside the class, must specify the template arguments after the name of the class in angle brackets. Define the method of fully specialized class templates outside the class body without the empty template parameter list: `template <>`.

```
template <typename T, int Line, int Column>
struct Matrix;

template <>
struct Matrix<int, 3, 3>{
   int numberOfElements() const;
};

// template <>
int Matrix<int, 3, 3>::numberOfElements() const {
```

```
    return 3 * 3;
};
```

Let’s take a look at some examples of template specialization in the next lesson.