# Specialization

Let's learn about template specialization in this lesson.

WE'LL COVER THE FOLLOWING



## Specialization #

Template specialization addresses the need to have different code for different template argument types. Templates define the behavior of families of classes and functions.

- Often it is necessary that special types, non-types, or templates as arguments are treated as 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, just a declaration will suffice.

```
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 and it 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 which are not specified must be given as template arguments, for example, 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:

Three rules for getting the right specialization:

1. If the compiler finds only one specialization, it uses that specialization.
2. If the compiler finds more than one specialization, it uses the most specialized one. If the compiler can't find the most specialized one, it throws an error.
3. If the compiler finds no specialization, it simply uses the primary template.

Template `A` is more specialized than template `B` if:

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

To learn more about partial specialization, click here.

## Full Specialization

For a fully specialized template, you have to 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 of the class, you have to specify the template arguments in angle brackets after the name of the class. Define the method of a fully specialized class template 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;
};
```

To learn more about full specialization, click here.

---

In the next lesson, we'll look at a few examples of template specialization.