C++ Templates Explained

Template in C++ explain with examples

In C++, templates are a powerful feature that allows you to write generic code that can work with different data types. They provide a way to create functions, classes, or structures that can operate with any data type without being explicitly defined for each type.

1. Function Templates

Function templates allow you to define a function that can work with any data type. The compiler generates the correct function based on the type of argument passed.

Syntax:

```cpp
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

Here, `T` is a placeholder for the data type. You can call this function with any type that supports the `+` operator (e.g., `int`, `double`, etc.).

Example:

```cpp
#include <iostream>
using namespace std;

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(3, 4) << endl;       // Outputs: 7
    cout << add(3.5, 4.5) << endl;   // Outputs: 8
    return 0;
}
```

2. Class Templates

Class templates allow you to create a class that can handle any data type.

Syntax:

```cpp
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T getValue() {
        return value;
    }
};
```

Here, `T` is a placeholder for the data type of the `Box` class.

Example:

```
cpp
```

```
#include <iostream>
using namespace std;

template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T getValue() {
        return value;
    }
};

int main() {
    Box<int> intBox(10);
    Box<double> doubleBox(3.14);

    cout << intBox.getValue() << endl;       // Outputs: 10
    cout << doubleBox.getValue() << endl;    // Outputs: 3.14

    return 0;
}
```

3. Template Specialization

Template specialization allows you to define a different implementation for a specific data type.

Example:

cpp

```cpp
#include <iostream>
using namespace std;

template <typename T>
void print(T value) {
    cout << "Generic: " << value << endl;
}

// Specialization for int type
template <>
void print<int>(int value) {
    cout << "Integer: " << value << endl;
}

int main() {
    print(10);      // Outputs: Integer: 10
    print(3.14);    // Outputs: Generic: 3.14
    return 0;
}
```

4. Template with Multiple Parameters

You can also create templates that accept multiple data types.

Example:

cpp

```cpp
#include <iostream>
using namespace std;

template <typename T, typename U>
void print(T t, U u) {
    cout << t << " and " << u << endl;
}

int main() {
    print(5, "Hello");       // Outputs: 5 and Hello
    print(3.14, 10);         // Outputs: 3.14 and 10
    return 0;
}
```

Summary

Templates are useful in situations where you want to create generic functions or classes that can work with any data type. By using `template <typename T>`, you can write flexible and reusable code. Templates also allow specialization and multiple parameters to customize their behavior based on the types provided.

2/2

Why you write here typename

In C++ templates, the keyword `typename` is used to define a placeholder for a data type. It signals to the compiler that the symbol following it is a **type parameter** in a template definition. This keyword is interchangeable with the keyword `class` in most contexts when declaring a type parameter in templates.

Why `typename` is used

1. **To define type parameters in templates:**
   `typename` tells the compiler that `T` (or any name you give) is a type that will be provided when the template is instantiated.

Example:

```cpp
template <typename T> // or template <class T>
T add(T a, T b) {
    return a + b;
}
```

Here, `T` is a placeholder for a type like `int`, `double`, etc., provided when you call `add`.

2. **Clarifying dependent types in template specializations:** In some advanced use cases, `typename` is required to clarify that a nested name in a template depends on a template parameter and refers to a type.

Example:

```cpp
template <typename T>
class Container {
public:
    typename T::value_type data; // 'typename' is required to indicate
'value_type' is a type
};
```

Without `typename`, the compiler might assume `T::value_type` is a variable or member function, leading to a compilation error.

Difference between `typename` and `class`

In the context of templates, there is **no functional difference** between `typename` and `class` when declaring type parameters. You can use either.

Example:

```cpp
template <typename T>
T multiply(T a, T b) {
    return a * b;
}

// Equivalent to
template <class T>
T multiply(T a, T b) {
    return a * b;
}
```

However:

- **`typename`** is more descriptive when used to clarify dependent types in nested templates (e.g., `typename T::value_type`).
- **`class`** is often used in older code or when templates represent generic classes.

When to prefer `typename`

- Use `typename` when defining nested types or when it better expresses your intent.
- Use either `typename` or `class` in simple template definitions as per coding style preferences.

Summary

The keyword `typename` in templates is used to define a type placeholder, just like `class`. While interchangeable in most cases, it is required in certain contexts to clarify that a nested name depends on a template type and refers to a type. This helps the compiler disambiguate between types and values.