Template Arguments

In this lesson, we'll learn about template arguments.

WE'LL COVER THE FOLLOWING

Template Arguments

Template arguments can, in general, automatically be deduced for function templates. The compiler deduces the template arguments for the function arguments. From the user's perspective, function templates feel like functions.

Conversion:

- The compiler uses simple conversions for deducing the template arguments from the function arguments.
- The compiler removes `const` or `volatile` from the function arguments and converts C-arrays and functions to pointers.

Template argument deduction for function templates:

```
template <typename T>
void func(ParamType param);
```

Two datatypes were deduced:

- `T`
- `ParamType`

`ParamType` can be

- Reference or pointer
- Universal reference(`&&`)
- Value (copy)

1. The parameter type is a reference or a pointer

```
template <typename T>
void func(T& param);
// void func(T* param);
func(expr);
```

- `T` ignores reference or pointer
- Pattern matching on `expr` for `T&` or `T`

2. The parameter type is a universal reference (`&&`)

```
template <typename T>
void func(T&& param);
func(expr);
```

- expr is an lvalue: `T` and `ParamType` become lvalue references
- `expr` is an rvalue: `T` is deduced such as the `ParamType` is a reference (case 1)

3. Parameter type is a value (copy)

```
template <typename T>
void func(T param);
func(expr);
```

1. `expr` is a reference: the reference (pointer) of the argument is ignored

2. `expr` is `const` or `volatile`: `const` or `volatile` is ignored

Template Arguments (C++17)

The constructor can deduce its template arguments from its function arguments.

Template Argument deduction for a constructor is available since C++17, but for function templates since C++98.

```
std::pair<int, double> myPair(2011, 1.23);
std::pair myPair(2011, 1.23);
```

Many of the `make_` functions such as `std::make_pair` are not necessary any more:

```
auto myPair = std::make_pair(2011, 1.23);
```

Argument Deduction

The types of function arguments have to be exact, otherwise, no conversion takes place.

```
template <typename T>
bool isSmaller(T fir, T sec){
  return fir < sec;
}

isSmaller(1, 5LL); // ERROR int != long long int
```

Providing a second template parameter makes this example work.

```
template <typename T, typename U>
bool isSmaller(T fir, U sec){
    return fir < sec;
}
isSmaller(1, 5LL);   // OK
```

Explicit Template Arguments

Unlike in line 5 in the previous example, sometimes the template argument types need to be explicitly specified. This is necessary in the following cases:

Explicit Template Arguments

- if the template argument cannot be deduced from the function argument.
- if a specific instance of a function template is needed.

```
template <typename R, typename T, typename U>
R add(T fir, U sec){
   return fir * sec;
}
add<long long int>(1000000, 1000000LL);
```

> Missing template arguments are automatically derived from the function arguments.

Default Template Arguments

The default for template parameters can be specified for class templates and function templates. If a template parameter has a default parameter, all subsequent template parameters also need a default argument.

```
template <typename T, typename Pred = std::less<T>>
bool isSmaller(T fir, T sec, Pred pred = Pred()){
  return pred(fir, sec);
}
```

To learn more about template arguments, click here.

In the next lesson, we'll take a look at the examples of template arguments.