if constexpr

The lesson explains why 'if constexpr' should be used, and how it is different from simple if statements.

WE'LL COVER THE FOLLOWING

- Why Compile Time If?

This is a big one! The compile-time `if` for C++!

The feature allows you to discard branches of an `if` statement at compile-time based on a constant expression condition.

```
if constexpr (cond)
        statement1; // Discarded if cond is false
    else
        statement2; // Discarded if cond is true
```

For Example:

```
template <typename T>
auto get_value(T t)
{
  if constexpr (std::is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

`if constexpr` has the potential to simplify a lot of template code - especially when tag dispatching, SFINAE or preprocessor techniques are used.

Why Compile Time If?

At first, you may ask, why do we need `if constexpr` and those complex templated expressions... wouldn't normal `if` work? Here's a test code:

templated expressions... wouldn't normal `if` work? Here's a test code:

```
template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
  if (is_constructible_v<Concrete, Ts...>) // normal `if`
    return make_unique<Concrete>(forward<Ts>(params)...);
  else
    return nullptr;
}
```

The above routine is an "updated" version of `make_unique`: it returns `std::unique_ptr` when the parameters allow it to construct the wrapped objects, or it returns `nullptr`.

Below there's a simple code that tests `constructArgs`:

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

template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
  if (is_constructible_v<Concrete, Ts...>) // normal `if`
    return make_unique<Concrete>(forward<Ts>(params)...);
  else
    return nullptr;
}

class Test
{
  public:
  Test(int, int) { }
};
int main()
{
  auto p = constructArgs<Test>(10, 10, 10); // 3 args!
}
```

The code tries to build `Test` out of three parameters, but notice that `Test` has a constructor that takes **two** `int` arguments. *This will produce a similar error:*

```
In instantiation of 'typename std::_MakeUniq<_Tp>::__single_object
std::make_unique(_Args&& ...) [with _Tp = Test; _Args = {int, int, int};
typename std::_MakeUniq<_Tp>::__single_object
 = std::unique_ptr<Test, std::default_delete<Test> >]':
```

```
main.cpp:8:40: required from 'std::unique_ptr<_Tp>
    constructArgs(Ts&& ...) [with Concrete = Test; Ts = {int, int, int}]'
```

The reason for the error message is that the compiler after template deduction compiles the following code:

```
if (std::is_constructible_v<Concrete, 10, 10, 10>)
  return std::make_unique<Concrete>(10, 10, 10);
else
  return nullptr;
```

During runtime, the `if` branch won't be executed - as `is_constructible_v` returns `false`, yet the code in the branch must compile.

That's why we need `if constexpr`, to "*discard*" code and compile only the matching statement. To fix the code you have to add `constexpr`:

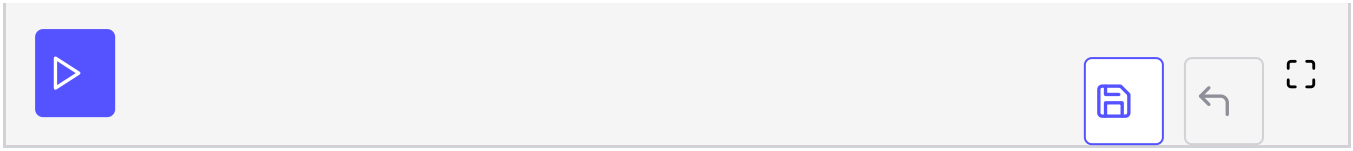
```
#include <iostream>
#include <memory>
using namespace std;

template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
  std::cout << is_constructible_v  <Concrete, Ts...> << "\n";
  std::cout << __func__ << ": ";
        ((std::cout << params << ", "), ...);
        std::cout << "\n";
  if constexpr (is_constructible_v <Concrete, Ts...>) // fixed!
  {
    std::cout << __func__ << ": ";
          ((std::cout << params << ", "), ...);
          std::cout << "\n";
    return make_unique<Concrete>(forward<Ts>(params)...);
  }
  else
  {
   return NULL;
  }
}

class Test
{
  public:
  Test(int, int) {}
};

int main()
{
  auto p = constructArgs<Test>(10, 10, 10); // 3 args!
```

```
}
```



Now, the compiler evaluates the `if constexpr` condition at compile time and for the expression

```
auto p = constructArgs<Test>(10, 10, 10);
```

the whole `if` branch will be "*removed*" from the second step of the compilation process.

To be precise, the code in the discarded branch is not entirely removed from the compilation phase. Only expressions that are dependent on the template parameter used in the condition are not instantiated. The syntax must always be valid.

For example:

```
template <typename T>
void Calculate(T t){
if constexpr (is_integral_v<T>){
  // ...
  static_assert(sizeof(int) == 100);
  }
else{
  execute(t);
  strange syntax
  }
}
```

In the above artificial code, if the type `T` is `int`, then the `else` branch is discarded, which means `execute(t)` won't be instantiated. But the line `strange syntax` will still be compiled (as it's not dependent on `T`) and that's why you'll get a compile error about that. What's more, another compilation error will come from `static_assert`, the expression is also not dependent on `T`, and that's why it will always be evaluated.

Head over to the next lesson to see updates used for Templates Simplification.