Type Comparisons and Modifications

Sometimes we need to manipulate or compare different types. We can use the type traits library for that!

WE'LL COVER THE FOLLOWING

Type Comparisons

The library supports three kinds of type comparisons.

Function	Description

`template class Base, class Derived>`
 	
`struct is_base_of`
	Checks if `Derived` is derived from `Base`.

`template <class From, class To>`
 	
`struct is_convertible`
	Checks if `From` can be converted to `To`.

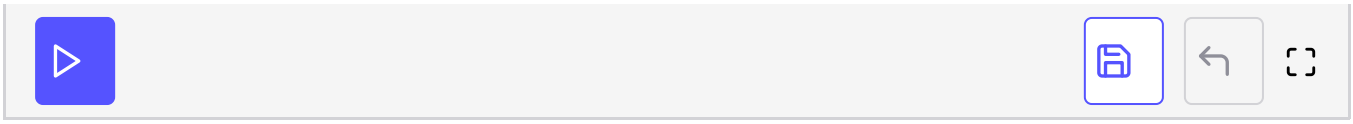
`template <class T, class U>`
 	
`struct is_same`
	Checks if the types `T` and `U` are the same.

Type comparison

Type Modifications

The type traits library enables you to modify types during compile time. So you can modify the constness of a type:

```
// typeTraitsModifications.cpp
#include <iostream>
#include <type_traits>
using namespace std;

//output 0 if the function returns false and 1 if the function returns true
int main(){
  cout << is_const<int>::value << "\n";                      // 0
  cout << is_const<const int>::value << "\n";                // 1
  cout << is_const<add_const<int>::type>::value << "\n";     // 1

  typedef add_const<int>::type myConstInt;
  cout << is_const<myConstInt>::value << "\n";               //1

  typedef const int myConstInt2;
  cout << is_same<myConstInt, myConstInt2>::value << "\n";  // 1

  cout << is_same<int, remove_const<add_const<int>::type>::type>::value << "\n";    // 1
  cout << is_same<const int, add_const<int>::type>::value << "\n"; // 1

  return 0;
}
```



Type modifications

The function `std::add_const` adds the constness to a type, while `std::remove_const` removes it.

There are a lot more functions available in the type traits library. So you can modify the const-volatile properties of a type.

```
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;

template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;
```

You can change at compile time the sign,

```
template <class T> struct make_signed;
template <class T> struct make_unsigned;
```

or the reference or pointer properties of a type.

```
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

The three following functions are especially valuable for the writing of generic libraries.

```
template <class B> struct enable_if;
template <class B, class T, class F> struct conditional;
template <class... T> common_type;
```

You can conditionally hide with `std::enable_if` a function overload or template specialization from overload resolution. `std::conditional` provides you with the ternary operator at compile time and `std::common_type` gives you the type, to which all type parameter can be implicitly converted to. `std::common_type` is a variadic template, therefore the number of type parameters can be arbitrary.

> 🗝 **C++14 has a shorthand for ::type**
> If you want to get a `const int` from an `int` you have to ask for the type: `std::add_const<int>::type`. With the C++14 standard use simply `std::add_const_t<int>` instead of the verbose form: `std::add_const<int>::type`. This rule works for all type traits functions.

In the next lesson, we will move on to another utility in C++ – the time library.