1 Fibonacci

In the seminars there was an example of how we can calculate Fibonacci numbers (`https://en.wikipedia.org/wiki/Fibonacci_number`) with template parameters and functions. It looked like this:

```
template <int N>
int fibonacci()
{
  return fibonacci<N-1>() + fibonacci<N-2>();
}

template <>
int fibonacci<0>()
{
  return 0;
}

template <>
int fibonacci<1>()
{
  return 1;
}
```

There is one drawback of this; for this calculation to actually be performed during compile-time we have to trust that the compiler is capable of removing all function calls during its optimization phase.

For calculating compile-time values, it would be a better guarantee if we could do it in the type system directly instead. There is a way to do this; by embedding the values into types instead of function calls.

In this assignment you are to create a class template called `Fibonacci` that takes one template parameter; an `int` called `N`.

`Fibonacci` should contain a static, constant variable called `value` that corresponds to the `N`th fibonacci number.

You should specialize `Fibonacci` for `N == 0` and for `N == 1`, such that their values are `0` and `1` respectively.

The nth fibonacci number is calculated with this relationship:

$$\text{fib}(n) = \begin{cases} \text{fib}(n-1) + \text{fib}(n-2), \text{if } n > 1, \\ 1, \text{if } n = 1 \\ 0, \text{if } n = 0 \end{cases}$$

2 Type traits

Sometimes when working with templates we want to do different things depending on what the deduced type is. Specializations allow us to override behaviour for different types or patterns, but this isn't always enough.

Sometimes we want to check if a type fulfills a certain condition. For example, suppose we have an array that must contain immutable (constant) values:

```
template <typename T, int N>
class Const_Array
{
  static_assert(/* check if T is const */,
                "T must be const");
  // ...
};
```

Then we want to be able to check that `T` is a constant value and report an error if it is not.

In this exercise you are going to create some simple *type trait classes*. Type trait classes are used to solve the problem specified above.

A type trait is an empty class template (i.e. it doesn't have any non-static data members) that is used to check conditions on the given type. Usually it contains a static constant variable of type `bool` called `value` that denotes whether a condition is met or not.

The use case for such a class would be this:

```
template <typename T, int N>
class Const_Array
{
  static_assert(is_const<T>::value,
                "T must be const");
  // ...
};
```

As an example during the seminar we implemented a special type trait called `is_same` that took two template type parameters and set `value` to `true` if the two given types were the same. It looked like this:

```
// primary template
template <typename T1, typename T2>
struct is_same
{
  static bool const value {false};
};

// partial specialization if the two parameters are the same
template <typename T>
```

```
struct is_same<T, T>
{
  static bool const value {true};
};
```

Here we create a primary template for which `value` is `false` and then a specialization where `value` is `true` instead. This is usually how type traits are implemented.

Use this technique to create these type traits:

- `is_const`, checks if the given type `T` is `const` (`T const`)
- `is_pointer`, checks if the given type `T` is a normal pointer (`T*`)
- `is_array`, checks if the given type is an array or arbitrary type and arbitrary size
- `is_const_pointer`, checks if the given type is a constant pointer (`T* const`)

Note: You should not have to include *anything* to make this work.

In the given file `type_traits.cc` there are a few testcases for these type traits.

Hint: you might want to refactor out shared code between the different type traits into base classes.