1 Remove reference & pointers

This exercise is divided into two parts, this is just to make it easier to read. Both parts are part of the same exercise. There are testcases for both parts given in `remove_traits.cc`.

1.1 `remove_reference`

A thing about type traits that we haven't covered yet is that they can be used to modify types. Suppose we have the following function:

```
template <typename T>
T copy(T value)
{
  return value;
}
```

This function is supposed to copy the parameter that was passed. This will work fine most of the time. But what happens if `T` is a reference? Then this function will take an object as a reference, and return it as a reference, meaning we never make a copy.

It can actually happen in some complicated situations that the compiler must deduce `T` to be some kind of reference rather than a plain type, and if that happens `copy` will be worthless.

Of course, we want our function to work in these cases as well, so we want to make sure that whatever we return from the function is **not** a reference. This can be done with a type trait class template, lets call it `remove_reference`, that takes one template type parameter `T` and contains an *inner type alias* (a `using`-declaration) called `type` which is the type of `T` with any eventual references removed. I.e. it should look something like this:

```
template <typename T>
struct remove_reference
{
  using type = T;
};
```

With appropriate specializations for when `T` is a reference type.

Your job is to create these specializations for `remove_reference` (`type` cannot be an rvalue-reference nor an lvalue-reference, even if `T` is a reference type).

If you do this correctly, then the `copy` function should always work if we implement it as such:

```
template <typename T>
typename remove_reference<T>::type copy(T value)
{
  return value;
}
```

1.2 `remove_pointers`

You should also implement a type trait class template called `remove_pointers` that removes any pointers from the type. Note that there can be pointers to pointers, and these should all be removed leaving only a non-pointer type left.

Example: `remove_pointers<int**>::type` should be `int`.

Note that this should work for any number of nested pointers, even for pointers that are constant (for example `remove_pointers<int* const*>::type`, should be `int`).

You should use the same technique as you did with `remove_reference`.

Hint: Think recursively, with the base case being that there are no pointers left in the type.

1.3 Simplify (Optional)

In C++17 we got access to alias templates. This means that we can simplify how we use our type traits. We can go from this:

```
remove_reference<int&&>::type
```

to

```
remove_reference_t<int&&>
```

In this final part you should add two alias templates, one for each type traits you have created; `remove_reference_t` and `remove_pointers_t`. With these you should then simplify the testcases to not use `::type` anywhere. Note that a corresponding simplification exists for `std::is_same` so you can update the testcases if you want to by changing:

```
std::is_same<int, int>::value
```

into

```
std::is_same_v<int, int>
```

2 Allocators

The concept known as *Allocators* were briefly mentioned during the seminar. In this exercise we will familiarize ourselves with this concept.

An allocator is a class template that specifies how dynamic memory is allocated (hence the name). This concept is often used whenever we have a class that needs to manage data in a general way because this allows the users to create their own custom allocator. Thus allowing users to customize the behaviour (in this case the behaviour is how to retrieve data) without changing your code.

In `allocator.cc` there is an implementation of a stack given. Your job is to make this into a class template that takes two template parameters; a type template parameter `T` that contains the type of the values stored in the stack, and a template-template parameter `Allocator` that represents the allocator that this instantiation should use to allocate memory.

Note: `Allocator` *cannot* be a type template parameter since we want to allocate `Node`-objects. `Node` is private inside `Stack` meaning the user cannot instantiate the allocator class themselves. We must instantiate the allocator ourselves inside `Stack`. I.e. we want the user to write:

```
Stack<std::string, My_Allocator> st{};
```

when they create a stack.

You should create two class templates; `New_Allocator` and `Tracker_Allocator`, both of which takes one template parameter; the data type that can be allocated.

If no allocator is specified by the user, then `Allocator` should default to `My_Allocator`. That is:

```
Stack<std::string> st{};
```

should be equivalent to:

```
Stack<std::string, New_Allocator> st{};
```

The following `static` functions should be defined in an allocator:

- `create`: Takes an arbitrary amount of parameters; these parameters should be passed to the constructor of the object. The parameters should be taken as *forwarding references* and then forwarded to the constructor. Finally it returns a pointer to the allocated object.
- `destroy`: should take a pointer to an object and deallocate that memory.

Both `New_Allocator` and `Tracker_Allocator` should use `new` and `delete` to create and destroy objects, respectively. However, `Tracker_Allocator` should print a message every time an object is allocated and/or deallocated, allowing the programmer to see when objects are allocated and deallocated. The message should include the address to the object in question.

Make sure to extend the tests in `allocators.cc` so you test both allocators with different types.

3 Add pointers

In this exercise you are going to use `std::enable_if` to implement a type trait called `add_pointers`. This type trait would most likely not have any real world use, but it is good for learning purposes.

The type trait `add_pointer` will take two types, and "add" all pointers into one single type. **Example:**

- `int + int = int`
- `int* + int* = int**`,
- `int + int*** = int***`,
- `int** + int** = int****`

and so on.

In `add_pointers.cc` there is a class template `add_pointers` given and some testcases. For `add_pointers` to compile properly we need to implement the function template `details::add_pointers_helper`: this is your job!

`details::add_pointers_helper` is function template that is callable with no function parameters, and *at least* two template parameters; `T` and `U`. There should be two versions of `details::add_pointers_helper`, one for whenever `U` is a pointer, and one for whenever `U` is *not* a pointer.

The idea is simple: we want to, one by one, move a `*` from `U` and add it to `T`, in a recursive manner, where the base case is when we have no other `*` in `U`. To implement this we need to do the following:

For the version when `U` is not a pointer, the return type should be `T` since we have no pointer to move from `U` to `T`. When `U` is a pointer, the return type should be the same as the return type of the instantiation `details::add_pointers_helper<T*, V>()` where `V*` is the same type as `U` (in other words; the second template parameter is `U` where we have removed a pointer from it).

You can use `std::is_pointer<U>::value` defined in `<type_traits>` to check whether `U` is a pointer or not, and you can use `std::remove_pointer<U>::type` to remove a pointer from `U`.