13.15 — Alias templates

In lesson 10.7 -- Typedefs and type aliases, we discussed how type aliases let us define an alias for an existing type.

Creating a type alias for a class template where all template arguments are explicitly specified works just like a normal type alias:

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

template <typename T>
void print(const Pair<T>& p)
{
    std::cout << p.first << ' ' << p.second << '\n';
}

int main()
{
    using Point = Pair<int>; // create normal type alias
    Point p { 1, 2 };        // compiler replaces this with Pair<int>

    print(p);

    return 0;
}
```

Such aliases can defined locally (e.g. inside a function) or globally.

Alias templates

In other cases, we might want a type alias for a template class where not all of the template arguments are defined as part of the alias (and will instead be provided by the user of the type alias). To do this, we can define an **alias template**, which is a template that can be used to instantiate type aliases. Just like type aliases do not define distinct types, alias templates do not define distinct types.

Here's an example of how this works:

```
#include <iostream>

template <typename T>
struct Pair
{
    T first{};
    T second{};
};

// Here's our alias template
// Alias templates must be defined in global scope
template <typename T>
using Coord = Pair<T>; // Coord is an alias for Pair<T>

// Our print function template needs to know that Coord's template parameter T is a
type template parameter
template <typename T>
void print(const Coord<T>& c)
{
    std::cout << c.first << ' ' << c.second << '\n';
}

int main()
{
    Coord<int> p1 { 1, 2 }; // Pre C++-20: We must explicitly specify all type
template argument
    Coord p2 { 1, 2 };      // In C++20, we can use alias template deduction to
deduce the template arguments in cases where CTAD works

    std::cout << p1.first << ' ' << p1.second << '\n';
    print(p2);

    return 0;
}
```

In this example, we're defining an alias template named `Coord` as an alias for `Pair<T>`, where type template parameter `T` will be defined by the user of the Coord alias. `Coord` is the alias template, and `Coord<T>` is the instantiated type alias for `Pair<T>`. Once defined, we can use `Coord` where we would use `Pair`, and `Coord<T>` where we would use `Pair<T>`.

There are a couple of things worth noting about this example.

First, unlike normal type aliases (which can be defined inside a block), alias templates must be defined in the global scope (as all templates must).

Second, prior to C++20, we must explicitly specify the template arguments when we instantiate an object using an alias template. As of C++20, we can use **alias template deduction**, which will deduce the type of the template arguments from an initializer in cases where the aliased type would work with CTAD.

Third, because CTAD doesn't work on function parameters, when we use an alias template as a function parameter, we must explicitly define the template arguments used by the alias template. In other words, we do this:

```
template <typename T>
void print(const Coord<T>& c)
{
    std::cout << c.first << ' ' << c.second << '\n';
}
```

Not this:

```
void print(const Coord& c) // won't work, missing template arguments
{
    std::cout << c.first << ' ' << c.second << '\n';
}
```

This is no different than if we'd used `Pair` or `Pair<T>` instead of `Coord` or `Coord<T>`.