Visiting Multiple Variants

This lessons discusses visiting multiple variants. Let's take a look at it in detail!

WE'LL COVER THE FOLLOWING

Variants And Overloading

`std::visit` allows you not only to visit one variant but many in the same call. However, it's essential to know that with multiple variants, you have to implement function overloads taking as many arguments as the number of input variants. And you have to provide all the possible combination of types.

For example, for the following:

```
std::variant<int, float, char> v1 { 's' };
std::variant<int, float, char> v2 { 10 };
```

You have to provide 9 function overloads if you call `std::visit` on the two variants:

```
std::visit(overload
{
  [](int a, int b) { },
  [](int a, float b) { },
  [](int a, char b) { },
  [](float a, int b) { },
  [](float a, float b) { },
  [](float a, char b) { },
  [](char a, int b) { },
  [](char a, float b) { },
  [](char a, char b) { }
}, v1, v2);
```

If you skip one overload, then the compiler will report an error.

A Look at an Example

Have a look at the following example, where each variant represents an ingredient and we want to compose two of them together:

```
#include <iostream>
#include <variant>
template<class... Ts> struct overload : Ts... {
    using Ts::operator()...;
};
template<class... Ts> overload(Ts...) -> overload<Ts...>;
struct Pizza { };
struct Chocolate { };
struct Salami { };
struct IceCream { };

int main() {
    std::variant<Pizza, Chocolate, Salami, IceCream> firstIngredient { IceCream() };
    std::variant<Pizza, Chocolate, Salami, IceCream> secondIngredient { Chocolate()};
    std::visit(overload{
        [](const Pizza& p, const Salami& s) {
            std::cout << "here you have, Pizza with Salami!\n";
        },
        [](const Salami& s, const Pizza& p) {
            std::cout << "here you have, Pizza with Salami!\n";
        },
        [](const Chocolate& c, const IceCream& i) {
            std::cout << "Chocolate with IceCream!\n";
        },
        [](const IceCream& i, const Chocolate& c) {
            std::cout << "IceCream with a bit of Chocolate!\n";
        },
        [](auto a, auto b) {
            std::cout << "invalid composition...\n";
        },
    }, firstIngredient, secondIngredient);
    return 0;
}
```

The above code uses `overload` and uses multiple lambdas rather than a separate `struct` with overloads for `operator()`.

> What's interesting is that the example provides implementation only for "valid" ingredient compositions, while the "rest" is handled by generic lambdas (from C++14).

A generic lambda `[](const auto& a, const auto& b) { }` is equivalent to the following callable type:

```
class UnnamedUniqueClass { // << compiler specific name...
public:
    template<typename T, typename U>
    auto operator () (const T& a, const T& b) const {  }
};
```

The generic lambda used in the example will provide all the remaining function overloads for the ingredient types. Since it's a template, it will always fall behind the concrete overload (lambda with concrete types) when the best viable function is determined.

Other `std::variant` Operations

Just for the sake of completeness:

- You can **compare** two variants of the same type:
  - if they contain the same active alternative, then the corresponding comparison operator is called.
  - If one variant has an "earlier" alternative then it's "less than" the variant with the next active alternative.
- Variant is a value type, so you can **move it**.
- `std::hash` on a variant is specialised if `std::hash` is available for all type alternatives. The hash value might be different than a hash for an active type as this enables to distinguish between variants that have duplicated types like `std::variant<int, int, float>`.

Now that you've learned in detail about overloading and visitors, it's time to look at exceptions arising during the creation of an alternative in a variant. The next lesson discusses this in detail.