Overload

In this part we'll take a look at overloading and C++ 17 overload uses.

WE'LL COVER THE FOLLOWING

With this utility you can write all lambdas for all matching types in one place:

```
std::variant<int, float, std::string> myVariant;
std::visit
(
  overload (
    [](const int& i) { std::cout << "int: " << i; },
    [](const std::string& s) { std::cout << "string: " << s; },
    [](const float& f) { std::cout << "float: " << f; }
  ),
  myVariant;
);
```

Currently, this helper is not a part of the Standard Library (it might be added into with C++20). Let's see how you can implement it with the code given below.

Implementation

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

The code creates a `struct` that inherits from **lambdas** and uses their `Ts::operator()`. The whole structure can now be passed to `std::visit` - it will then select the proper overload.

`overload` Uses Three C++17 Features

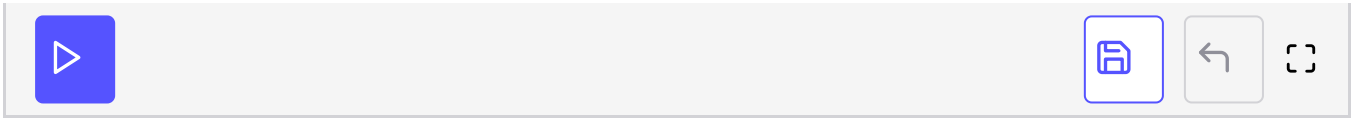
`overload` Uses Three C++17 Features

- Pack expansions in `using` declarations - short and compact syntax with variadic templates.
- Custom template argument deduction rules - this allows the compiler to deduce types of lambdas that are the base classes for the pattern. Without it, we'd have to define a "make" function.
- Extension to aggregate Initialisation - the overload pattern uses aggregate initialisation to init base classes. Before C++17, it was not possible.

How to Use Overload: An Example

```
// variant_overload.cpp
// example for "C++17 In Detail"
// by Bartlomiej Filipek
// 2018/2019

#include <iostream>
#include <variant>

struct MultiplyVisitor {
    float mFactor;

    MultiplyVisitor(float factor) : mFactor(factor) { }

    void operator()(int& i) const {
        i *= static_cast<int>(mFactor);
    }

    void operator()(float& f) const {
        f *= mFactor;
    }

    void operator()(std::string& ) const {
        // nothing to do here...
    }
};

template<class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main() {
    auto PrintVisitor = [](const auto& t) { std::cout << t << '\n'; };
    auto TwiceMoreVisitor = [](auto& t) { t*= 2; };

    std::variant<int, float, std::string> intFloatString { "Hello" };
    std::visit(PrintVisitor, intFloatString);

    std::variant<int, float> intFloat { 20.4f };
    std::visit(PrintVisitor, intFloat);
    std::visit(TwiceMoreVisitor, intFloat);
    std::visit(PrintVisitor, intFloat);

    std::visit(MultiplyVisitor(0.5f), intFloat);
```

```
    std::visit(PrintVisitor, intFloat);

    std::variant<int, float, std::string> intFloatString2 { "Hello" };

    std::visit(overloaded{
        [](int& i) { i*= 2; },
        [](float& f) { f*= 2.0f; },
        [](std::string& s) { s = s + s; }
    }, intFloatString2);
    std::visit(PrintVisitor, intFloatString2);

    return 0;
}
```



> Here's the paper for the proposal of `std::overload`: P0051 - C++ generic overload function.

And you can read more about the mechanics of the overload pattern in this blog post at bfilipek.com: 2 Lines Of Code and 3 C++17 Features - The overload Pattern

---

The next lesson touches upon the concept of visiting multiple variants. Read on to find out more.