Visitors for std::variant

This lesson explains std::visit, it''s declaration and use.

WE'LL COVER THE FOLLOWING

With the introduction of `std::variant`, we also got a handy STL function called `std::visit`. It can call a given **“visitor”** on all passed variants.

Visitor Declaration

```
template <class Visitor, class... Variants>
constexpr visit(Visitor&& vis, Variants&&... vars);
```

And it will call `vis` on the currently active type of variants.

If you pass only one variant, then you have to have overloads for the types from that variant. If you give two variants, then you have to have overloads for all possible pairs of the types from the variants.

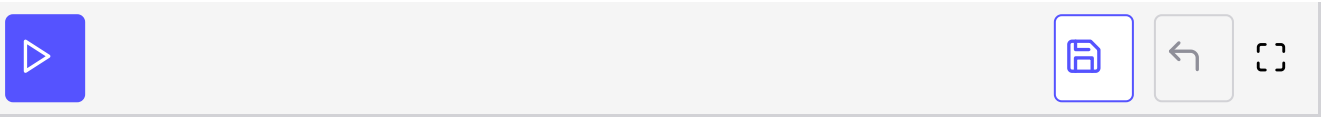
A visitor is **“a Callable that accepts every possible alternative from every variant “**.

Examples

Let’s see some examples:

```
#include <iostream>
#include <variant>
using namespace std;

int main() {
// a generic lambda:
    auto PrintVisitor = [](const auto& t) {
```

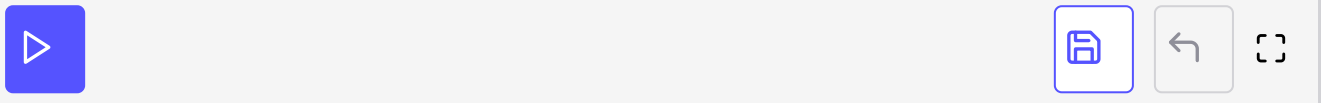
```
        std::cout << t << '\n';
    };
    std::variant<int, float, std::string> intFloatString { "Hello" };
    std::visit(PrintVisitor, intFloatString);
}
```



In the above example, a generic lambda is used to generate all possible overloads. Since all of the types in the variant support `<<` (stream output operator), then we can print them.

In another example we can use a visitor to change the value:

```
#include <iostream>
#include <string>
#include <variant>
using namespace std;

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

auto TwiceMoreVisitor = [](auto& t) {
    t*= 2;
};

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

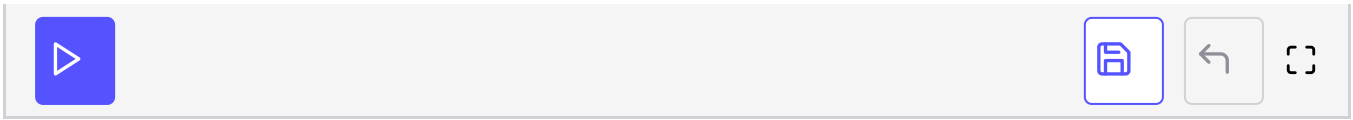


Generic lambdas can work if our types share the same **“interface”**, but in most cases, we’d like to perform different actions based on an active type.

That’s why we can define a structure with several overloads for the `operator()`:

```
#include <iostream>
#include <variant>
#include <string>
using namespace std;

struct MultiplyVisitor
```

```
{
    float mFactor;
    MultiplyVisitor(float factor) : mFactor(factor) { }
    void operator()(int& i) const {
        i *= static_cast<int>(mFactor);
        cout<< "i is: "<< i<<endl;
    }
    void operator()(float& f) const {
        f *= mFactor;
        cout<< "f is: "<< f<<endl;
    }
    void operator()(std::string& ) const {
        cout<< "nothing to do here..."<<endl;
    }
};

auto PrintVisitor = [](const auto& t) {
    std::cout << "Answer from Print Visitor is:" <<t << '\n';
};

int main() {
    std::variant<int, float> intFloat { 20.4f };
    std::visit(MultiplyVisitor(0.5f), intFloat);
    std::visit(PrintVisitor, intFloat);

    std::variant<int, float> intFloat2 { 20};
    std::visit(MultiplyVisitor(2), intFloat2);
    std::visit(PrintVisitor, intFloat2);
}
```



In the example, you might notice that `MultiplyVisitor` uses a state to hold the desired scaling factor value. That gives a lot of options for visitation.

With lambdas, we got used to declaring things just next to its usage. And when you need to write a separate structure, you need to go out of that local scope. That's why it might be handy to use `overload` construction which is what the next lesson discusses.