std::variant demo

Let's take a look at the basic example of std::variant and it's explanation!

WE'LL COVER THE FOLLOWING

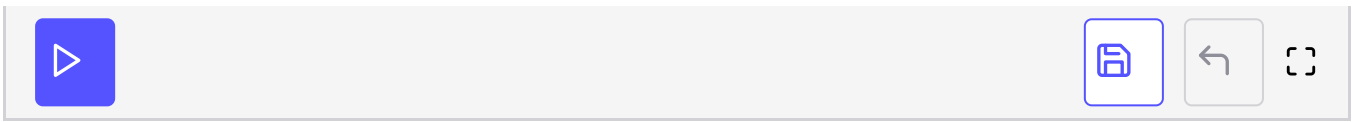
Before C++17, if you wanted a type-safe union, you could use `boost::variant` or another third-party library. But now you have `std::variant`.

Here's a demo of what you can do with this new type:

std::variant Basic Example

```
#include <iostream>
#include <variant>

void* operator new(std::size_t count) {
    std::cout << "allocating " << count << " bytes" << std::endl;
    return malloc(count);
}

struct SampleVisitor {
    void operator()(int i) const { std::cout << "int: " << i << '\n'; }
    void operator()(float f) const { std::cout << "float: " << f << '\n'; }
    void operator()(const std::string& s) const { std::cout << "string: " << s << '\n'; }
};

int main() {
    std::variant<int, float, std::string> intFloatString;
    static_assert(std::variant_size_v<decltype(intFloatString)> == 3);

    //default initialized to the first alternative, should be 0
    std::visit(SampleVisitor{}, intFloatString);
    // visit(vis,vars)  - Applies the visitor vis to the variants vars
    // where vis    -   a Callable that accepts every possible alternative from every variant
    // vars   - list of variants to pass to the visitor

    std::cout << "index will show the currently used 'type'" << std::endl;
    std::cout << "- when intFloatString is default initialized to the first alterantive ie. i
    std::cout << "index = " << intFloatString.index() << std::endl;
```

```
    std::cout << "- when intFloatString = 100.0f " << std::endl;
    intFloatString = 100.0f;
    std::cout << "index = " << intFloatString.index() << std::endl;

    std::cout << "- when intFloatString = hello super world " << std::endl;
    intFloatString = "hello super world";
    std::cout << "index = " << intFloatString.index() << std::endl;

    // check currently used type with holds_alternative
    if (std::holds_alternative<int>(intFloatString))
            std::cout << "the variant holds an int!\n";
        else if (std::holds_alternative<float>(intFloatString))
            std::cout << "the variant holds a float\n";
        else if (std::holds_alternative<std::string>(intFloatString))
            std::cout << "the variant holds a string\n";

    // try with get_if:
    // get_if: obtains a pointer to the value of a pointed-to variant given the index or the
    if (const auto intPtr (std::get_if<int>(&intFloatString)); intPtr) {
        std::cout << "int: " << *intPtr << '\n';}
    else if (const auto floatPtr (std::get_if<float>(&intFloatString)); floatPtr) {
        std::cout << "float: " << *floatPtr << '\n';}


    // try/catch and bad_variant_access
    try {
        auto f = std::get<float>(intFloatString);
        std::cout << "float! " << f << '\n';
    }
    catch (std::bad_variant_access&) {
        std::cout << "our variant doesn't hold float at this moment...\n";
    }

    // visit:
    std::visit(SampleVisitor{}, intFloatString);
    intFloatString = 10;
    std::visit(SampleVisitor{}, intFloatString);
    intFloatString = 10.0f;
    std::visit(SampleVisitor{}, intFloatString);

    return 0;
}
```



Elaboration of the Example

• **Line 16, 20**: If you don't initialize a variant with a value, then the variant is initialized with the first type. In that case, the first alternative type must have a default constructor. **Line 20** will print the value **0**.

• **Line: 27, 31, 35, 38, 40, 42**: You can check what the currently used type is via `index()` or `holds_alternative`.

• **Line 47, 49**: You can access the value by using `get_if` (it returns null pointer

when the type is not active).

• **Line 55, 58**: You can access the value by using `get` (the compiler might throw the `bad_variant_access` exception).

• **Type Safety** - the variant doesn't allow you to get a value of the type that's not active.

• **No** extra heap allocation occurs.

• **Line 9, 20, 63, 65, 67**: You can use a visitor to invoke an action on a currently active type. The example uses `SampleVisitor` to print the currently active value. It's a simple structure with overloads for `operator()`. The visitor is then passed to `std::visit` which performs the visitation.

• The variant class calls destructors and constructors of non-trivial types, so in the example, the `string` object is cleaned up before we switch to new variants.

Now that you've learned the basics of `std::variant`, the next lesson will elaborate on its uses.