std::variant

The last part of this section deals with std::variant which allows us to create a variable from any of the types specified in the std::variant container.

std::variant is a type-safe union. An instance of std::variant has a value from one of its types. The type must not be a reference, array or void. A std::variant can have a type more than once. A default-initialised std::variant is initialised with its first type; therefore, its first type must have a default constructor. By using var.index you get the zero-based index of the alternative held by the std::variant var. var.valueless_by_exception returns false if the variant holds a value. By using var.emplace you can create a new value in-place. There are a few global functions used to access a std:variant. The function template var.holds_alternative lets you check if the std::variant holds a specified alternative. You can use std::get with an index and with a type as argument. By using an index, you will get the value. If you invoke std::get with a type, you only will get the value if it is unique. If you use an invalid index or a non-unique type, you will get a std::bad_variant_access exception. In contrast to std::get which eventually returns an exception, std::get_if returns a null pointer in the case of an error.

The following code snippet shows you the usage of a std::variant.

```
#include <variant>
#include <string>
#include <cassert>

using namespace std::literals;

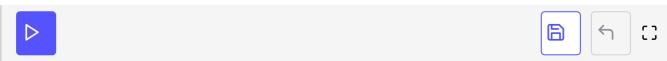
int main()
{
    std::variant<int, float> v, w;
    v = 12; // v contains int
    int i = std::get<int>(v);
    w = std::get<int>(v);
    w = std::get<0>(v); // same effect as the previous line
    w = v; // same effect as the previous line
```

```
// std::get<aouble(v); // error: no double in [int, float]
// std::get<3>(v);  // error: valid index values are 0 and 1

try {
    std::get<float>(w); // w contains int, not float: will throw
}
    catch (const std::bad_variant_access&) {}

std::variant<std::string> x("abc"); // converting constructors work when unambiguous
x = "def"; // converting assignment also works when unambiguous

std::variant<std::string, bool> y("abc"); // casts to bool when passed a char const *
assert(std::holds_alternative<bool>(y)); // succeeds
y = "xyz"s;
assert(std::holds_alternative<std::string>(y)); //succeeds
}
```



std::variant

v and w are two variants. Both can have an int and a float value. Their
default value is 0. v becomes 12 and the following call std::get<int>(v)
returns the value. The next three lines show three possibilities to assign the
variant v to w, but you have to keep a few rules in mind. You can ask for the
value of a variant by type std::get<double>(v) or by index: std::get<3>(v).
The type must be unique and the index valid. The variant w holds an int
value; therefore, I get a std::bad_variant_access exception if I ask for a float
type. If the constructor call or assignment call is unambiguous, a conversion
can take place. This is the reason that it's possible to construct a
std::variant<std::string> from a C-string or assign a new C-string to the
variant.

std::variant has a interesting non-member function std::visit that allows you the execute a callable on a list of variants. A callable is something which you can invoke. Typically this can be a function, a function object, or lambda expression. For simplicity reasons, I use a lambda function in this example.

```
vecvarianc - {3, 2, 3.4, 10011, 20111, 3.31, 2017,
 for (auto& v: vecVariant){
   std::visit([](auto&& arg){std::cout << arg << " ";}, v);</pre>
                                 // 5 2 5.4 100 2011 3.5 2017
 }
 cout<<std::endl;</pre>
 // display each type
 for (auto& v: vecVariant){
   std::visit([](auto&& arg){std::cout << typeid(arg).name() << " ";}, v);</pre>
                                // i c d x l f i (these letters refer to int char double ]
  }
 cout<<endl;</pre>
 // get the sum
  std::common_type<char, long, float, int, double, long long>::type res{};
 for (auto& v: vecVariant){
   std::visit([&res](auto&& arg){res+= arg;}, v);
 std::cout << "res: " << res << std::endl;</pre>
                                                  // res: 4191.9
 // double each value
 for (auto& v: vecVariant){
   std::visit([&res](auto&& arg){arg *= 2;}, v);
   std::visit([](auto&& arg){std::cout << arg << " ";}, v);</pre>
                               // 10 d 10.8 200 4022 7 4034
 }
  return 0;
}
```



std::visit

Each variant in this example can hold a char, long, float, int, double, or long long. The first visitor [](auto&& arg){std::cout << arg << " ";} will output the various variants. The second visitor std::cout << typeid(arg).name() << " ";} will display its types.

Now I want to sum up the elements of the variants. First I need the right result type at compile time. std::common_type from the type traits library will provide it. std::common_type gives the type to which all types char, long, float, int, double, and long long can implicitly be converted to. The final {} in res{} causes it to be initialised to 0.0. res is of type double. The visitor [&res](auto&& arg){arg *= 2;} calculates the sum and the following line displays it.