Chapter 24. Boost.Variant

Boost.Variant provides a class called `boost::variant` that resembles `union`. You can store values of different types in a `boost::variant` variable. At any point only one value can be stored. When a new value is assigned, the old value is overwritten. However, the new value may have a different type from the old value. The only requirement is that the types must have been passed as template parameters to `boost::variant` so they are known to the `boost::variant` variable.

`boost::variant` supports any type. For example, it is possible to store a `std::string` in a `boost::variant` variable – something that wasn't possible with `union` before C++11. With C++11, the requirements for `union` were relaxed. Now a `union` can contain a `std::string`. Because a `std::string` must be initialized with placement new and has to be destroyed by an explicit call to the destructor, it can still make sense to use `boost::variant`, even in a C++11 development environment.

Example 24.1. Using `boost::variant`

```
#include <boost/variant.hpp>
#include <string>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  v = 'A';
  v = "Boost";
}
```

`boost::variant` is defined in `boost/variant.hpp`. Because `boost::variant` is a template, at least one parameter must be specified. One or more template parameters specify the supported types. In Example 24.1, **v** can store values of type `double`, `char`, or `std::string`. However, if you tried to assign a value of type `int` to **v**, the resulting code would not compile.

Example 24.2. Accessing values in `boost::variant` with `boost::get()`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << boost::get<double>(v) << '\n';
  v = 'A';
  std::cout << boost::get<char>(v) << '\n';
  v = "Boost";
  std::cout << boost::get<std::string>(v) << '\n';
}
```

To display the stored values of **v**, use the free-standing function `boost::get()` (see Example 24.2).

`boost::get()` expects one of the valid types for the corresponding variable as a template parameter. Specifying an invalid type will result in a run-time error because validation of types does not take place at compile time.

Variables of type `boost::variant` can be written to streams such as the standard output stream, bypassing the hazard of run-time errors (see Example 24.3).

Example 24.3. Direct output of `boost::variant` on a stream

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << v << '\n';
  v = 'A';
  std::cout << v << '\n';
  v = "Boost";
  std::cout << v << '\n';
}
```

For type-safe access, Boost.Variant provides a function called `boost::apply_visitor()`.

Example 24.4. Using a visitor for `boost::variant`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  void operator()(double d) const { std::cout << d << '\n'; }
  void operator()(char c) const { std::cout << c << '\n'; }
  void operator()(std::string s) const { std::cout << s << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

As its first parameter, `boost::apply_visitor()` expects an object of a class derived from `boost::static_visitor`. This class must overload `operator()` for every type used by the `boost::variant` variable it acts on. Consequently, the operator is overloaded three times in Example 24.4 because **v** supports the types `double`, `char`, and `std::string`.

`boost::static_visitor` is a template. The type of the return value of `operator()` must be specified as a template parameter. If the operator does not have a return value, a template parameter is not required, as seen in the example.

The second parameter passed to `boost::apply_visitor()` is a `boost::variant` variable.

`boost::apply_visitor()` automatically calls the `operator()` for the first parameter that matches the type of the value currently stored in the second parameter. This means that the sample program uses different overloaded operators every time `boost::apply_visitor()` is invoked – first the one for `double`, followed by the one for `char`, and finally the one for `std::string`.

The advantage of `boost::apply_visitor()` is not only that the correct operator is called automatically. In addition, `boost::apply_visitor()` ensures that overloaded operators have been provided for every type supported by `boost::variant` variables. If one of the three overloaded operators had not been defined, the code could not be compiled.

If overloaded operators are equivalent in functionality, the code can be simplified by using a template (see Example 24.5).

Example 24.5. Using a visitor with a function template for `boost::variant`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const { std::cout << t << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

Because `boost::apply_visitor()` ensures code correctness at compile time, it should be preferred over `boost::get()`.