Tuples and More

- C++11 / Boost Tuple
- Boost Optional
- Boost Any
- Boost Variant

C++11: Tuple

Tuples are kind of ad-hoc structures and are mainly used to temporarily combine several unrelated values to be handled as unit, e.g. in a function result.

The similarity of `std::tuple` to `std::pair` for the above use case is obvious and if tuples had been part of C++98 they would have probably been used in some places were currently `std::pair` is used.

Because the number of elements in a tuple is unlimited* they cannot be names (like `first` and `second` in a pair) and a different technique must be applied to access the individual parts of a tuple.

*: Interestingly, the other border case are not tuples with a single element, but completely empty ones. These not only may have some use in very generic, templated data structures but can also help to write compile time algorithms to process all elements of a tuple. This can be done by recursive calls of variadic templates, terminated with a specialization for the (degenerate) empty tuple.

Definition and Initialization of `std::tuple`

When defining a tuple the type of its elements may be explicitly defined:

```
std::tuple<int, double, const char *> t;
```

This may – of course – be combined with an initialization:*

```
std::tuple<int, double, const char *> t{42, std::sqrt(2), "hello"};
```

Types can be omitted by using `auto`:

```
auto t = std::make_tuple(42, std::sqrt(2), "hello");
// or:
auto t(std::make_tuple(42, std::sqrt<float>(2), "hello"));
```

But: As a subtlety of "uniform initialization" the following creates a tuple too, though quite a different one:

```
auto t{std::make_tuple(42, std::sqrt(2), "hello")};
```

*: And of course any classic initialization syntax may be used here in place of the curly braces style.

Element Access of `std::tuple`

To access individual elements a global getter-function must be used:

```
… std::get<0>(t) … // the int
… std::get<1>(t) … // the double
… std::get<2>(t) … // the const char *
```

[!] It should be understood that the "element index" for `std::get` has to be a compile time constant.

Therefore the following will not work:[*]

```
for (int i = 0; i < 3; ++i) std::cout << std::get<i>(t) << '\n';
```

[*]: There are ways to print all the elements of a tuple – but it must be done with a loop *unrolling itself completely at compile time*, as for each element a different overload of `operator<<` might have to be used.

Unpacking an `std::tuple`

There is a different technique that unpacks all elements of a tuple at once:

```cpp
int count;
double value;
const char *name;
std::tie(count, value, name) = t;
```

Obviously this has its greatest advantage if all values have to go into a variable of their own, or at least most values:

```cpp
int count;
std::string name;
std::tie(count, std::ignore, name) = t;
```

Compared to element access with `std::get` it is said that a small overhead may be imposed by `std::tie`, but surely far from substantial.[*]

[*]: Vague language is used here to indicate that details will vary between implementations and future compilers may apply code-generation techniques to improve chances for optimization, putting `std::tie` at par with direct member assignment.

Modifying `std::tuple`

As `std::get` returns a reference, it is straight forward to modify individual elements of a tuple:

```
std::get<0>(t) = 12;
std::get<1>(t) *= 2;
```

Tuples can be also modified as a whole, only given each of their elements are assignment compatible:

```
std::tuple<int, double> t1{3}, t2{12, std::sqrt(2)};
auto t3 = std::make_tuple(true, 0u);
t1 = t2; t2 = t3; t3 = t1;
```

Assigning all elements at once fails if some or all are `const`-qualified:

```
std::tuple<const int, double> t4{-1, 0.0};
const std::tuple <int, double> t5{0, 0.0};
t1 = t4; // OK (of course can assign FROM const)
t4 = t1; // FAILS because first member is const
t5 = t1; // FAILS because all members are const
std::get<1>(t4) = std::get<1>(t1); // OK
```

Boost: Tuple

C++11 `std::tuple` emerged from `boost::tuple`.

There are few differences most of which are related to lifted restrictions because C++11 supports true variadic templates.

- With `std::tuple` there is no upper limit to the number of contained elements.
- For `boost::tuple` the maximum is limited – usually to 10 or 20, depending on the configuration step (optionally) run as first thing during the Boost installation.

Another difference is that `std::tuple` can **only** be accessed with the global getter while `boost::tuple` also has a member for that purpose:*

```
… std::get<2>(t) …  // C++11 and Boost
… t.get<2>() …      // Boost only
```

*: Of course, here `t` is assumed to be a tuple with at least three elements.

Boost: Optional

The use of `boost::optional` may be considered as an alternative to using pointers and having the `nullptr` represent the *does-not-exist* case.

- Differently from the pointer approach with `boost::optional` no heap allocation will occur.
  - Instead `boost::optional` reserves space for its payload data type **plus** a flag (e.g. on the stack in case of a local variable) …
  - … uses Placement `new` when the payload space is eventually to be initialized …
  - … and an Explicit Destructor Call when it is to be invalidated.

```
boost::optional<int> x; // default initialized …
assert(!x);             // ... it's not yet valid ...
x = 42;                 // ... now gets assigned ...
assert(x);              // ... tested here ...
assert(x.get() == 42);  // ... retrieved here
```

Boost: Any

The use of `boost::any` may be considered as an alternative to using untyped pointers (`void *`) to achieve "runtime polymorphism" for types that cannot be related through a common base class.

- Internally `boost::any` refers to the assigned data via a `void*` member …
- … while with an additional member it tracks the type assigned last …
- … for which it can be queried in a convenient syntax.

```
boost::any x;                 // default initialized to nullptr
if (…)                        // depending on a runtime condition ...
    x = true;                 // ... may actually hold a boolean ...
else if (…)                   // ... or in a different case ...
    x = std::sqrt(2.0);       // ... may actually hold a double ...
else if (…)                   // ... or yet differently ...
    x = std::string("hi!"); // ... an std::string ...
else                          // ... or ...
    x = …;                    // ... (who knows?)
```

Boost: Any (cont.)

To access a `boost::any` its content type must always be tested for.

There is a rather systematic way to do so:*

```
if (bool *p = boost::any_cast<bool>(&x)) {
    … *p … // access member if of type bool
}
else if (double *p = boost::any_cast<double>(&x)) {
    … *p … // access member if of type double
}
else if (std::string *p = boost::any_cast<std::string>(&x)) {
    … *p … // access member if of type std::string
}
```

The pointers `p` can also be `auto` or `auto *`-typed.

That way there is **only one** type to adapt when a code block gets copy-pasted to implement a new type case branch.

*: Here a lesser known feature from C++98 is at work, allowing to define a variable inside an `if`-condition so that it gets local scope, limited to the statement or block following, much like it is often used for counting variables in `for` loops.

Boost: Variant

The type `boost::variant` is similar to `boost::any` with the important exception that the set of types must be known at compile time:*

```
boost::variant<bool,
               std::string,
               double> x;  // from a (default initialized) bool
…                          // (as bool is the first on the list)
x = std::sqrt(2.0);        // can be set to any other type from
…                          // the list, either exactly matching
x = std::string("hello"); // but also applying conversions, if
…                          // an unambiguous choice can be made,
x = "world";               // like const char * to std::string
x = nullptr;               // or nullptr_t to bool (-> false)
```

Therefore it can be used as replacement for a classic C-style `union` augmented with a type tracking member automatically set on initialization and assignment.

*: Effectively this means that there is a closer coupling to the client code as for `boost::any`, where the set of types involved of course is also fixed in the source code, but more remotely, i.e. only in the context of initialization, assignment, and value access.

Boost: Variant (cont.)

A variant can be accessed in several ways:

- Specifying the expected type directly, e.g. with `boost::get<bool>(x)`, with an exception thrown when the content is different.
- By trying a type with `auto *p = boost::get<bool>(&x)`, followed by an explicit or implicit test against the `nullptr`.[*]
- With a compile time variant of the visitor pattern.

```
struct process : boost::static_visitor<> {
    void operator()(bool b) const { … } // process content if bool
    void operator()(double d) const { … }         // ... if double
    void operator()(const std::string &s) const { … }  // ... etc.
};
…
boost::apply_visitor(process(), x);
```

Note that applying a visitor detects missing cases at compile time!

[*]: In so far it is similar to `boost::any` but with slightly different syntax.

Boost: Variant (cont.)

In vistor-style there is also the option to fold cases with common source code into a template, while still handling special cases special:*

```
struct print : boost::static_visitor<> {
    std::ostream &os;
    print(std::ostream &os_) : os(os_) {
        os.setf(ios::boolalpha);
    }
    template<typename T>
    void operator()(const T &v) const {
        os << v;
    }
    void operator()(const std::string &v) const {
        os << '"' << v << '"';
    }
};
…
boost::apply_visitor(print(std::cout), x);
```

Missing type cases now might not have the required implementation.

*: An overloaded `operator<<` is already defined for `boost::variant` but of course without any special-casing.