Check Point®
SOFTWARE TECHNOLOGIES LTD

THE STD::TUPLE CONTAINER

Why, When, and How

Noam Weiss

Agenda

- Historical Overview
- Case Study
- Main features of std::tuple
- Limitations of std::tuple (or why don’t we use it more)
- Honorable mentions

Brief History

- The std::tuple contain zero or more elements of potentially different type
  - Uses veridic templates
  - Can be thought of as an extension of std::pair

- It was introduced, along with helpers, in C++11

- It was slightly enhanced in C++14, and more so in C++17

Case study

Suppose I have a function called `query(…)` and after invoking her I want to know three things:

1. What answer did I receive
2. From whom did I receive the answer
3. And statistics such as how long it took, attempts made, and so on.

The problem is that a function can only return one element, so we need to find a way to turn one into three.

Solutions

There are three major categories of solutions to this problem:

1. Output parameters.
2. Side effects.
3. Wrapping several elements in a container.

The main advantage of the third option is that in the case of failure there are no question as to the state of the output parameters or side effects.

Solution template

```
std::tuple<answer_t, source_t, stats_t>
quary(…)
{
    answer_t answer;
    source_t source;
    stats_t stats;
    …
    return std::tuple<answer_t, source_t, stats_t>(answer, source, stats);
}
```

Solution template

```
std::tuple<answer_t, source_t, stats_t>
quary(…)
{
    answer_t answer;
    source_t source;
    stats_t stats;
    …
    return std::make_tuple(answer, source, stats);
}
```

Solution template

```
std::tuple<answer_t, source_t, stats_t> res = query(…);

…

/* Check if the answer is from a reliable source */
if (std::get<1>(res).is_reliability_at_least(…)) {
    …
}

…
```

Solution template

```
std::tuple<answer_t, source_t, stats_t> res = query(…);
source_t &source = std::get<1>(res);
…

/* Check if the answer is from a reliable source */
if (source.is_reliability_at_least(…)) {
    …
}


…
```

Solution template

```
answer_t answer;
source_t source;
stats_t stats;
std::tie(answer, source, stats) = query(…);
…

/* Check if the answer is from a reliable source */
if (source.is_reliability_at_least(…)) {
    …
}

…
```

Solution template

```
answer_t answer;
source_t source;
stats_t stats;

try {
    std::tie(answer, source, stats) = query(…);
} catch(…) {
    answer = …;
    source = …;
    stats = …;
}
```

Solution template – C++17

```
auto [answer, source, stats] = query(…);
…

/* Check if the answer is from a reliable source */
if (source.is_reliability_at_least(…)) {
    …
}

…
```

std::tuple VS. struct

```
std::tuple<
    answer_t,
    source_t,
    stats_t
>
```

```
struct {
    answer_t first;
    source_t second;
    stats_t third;
}
```

1. Semantics.
2. Standardization.
3. Strict ordering.
4. Memory consumption.

The std::tuple

- Constructor
- Assignment
  - Since C++20 returns `constexpr`
- swap
  - Also exists as an external function
  - Since C++20 returns `constexpr`

Basic support functions

- std::make_tuple
- std::tie
- std::get
- Comparison operators

The std::get

The std::get function is templated on a number `I` and tuple types `Types`, and returns a reference to the element in the `I`-th position in the tuple.

Since C++14 there is a version that is templated on a specific type `T` and tuple types `Types`. And returns a reference to the only element of that type in the tuple.

```
std::get<answer_t>(res);
```

The std::tuple_size

- Given a tuple, it gives its size
- Inherits from std:: integral _ constant
  - Members
    - `value` - static
  - Methods
    - `operator std::size_t`
    - `operator()` – since C++14
  - Types
    - `value_type`
    - `type`
- Since C++17 there is a helper definition
    - `temlpate <class T> inline constexpr std::size_t tuple_size_v;`

Working with tuples

Lets say that we want to print the content of a tuple for debugging purposes, and we assume that all the types are printable.

We would like to write some thing like this:

```
for (auto &elem : my_tuple) {
    std::cout << elem << std::endl;
}
```

Won’t work, since std::tuple doesn’t have iterators!

Working with tuples

```
size_t my_tuple_size =
    tuple_size<decltype(my_tuple)>::value;

for (size_t i = 0; i < my_tuple_size; ++i) {
    cout << get<i>(my_tuple) << endl;
}
```

What type is this?

Template parameter must be known at compile time!

We can't Iterate over tuples due to the different types of the elements!

Turning loop into recursion

```
template <class Tuple, size_t N>
struct tuple_forward_loop
{
    template <typename Callable>
    static void invoke(Tuple &t, Callable &func)
    {
        tuple_forward_loop<Tuple, N-1>::invoke(func);
        func(get<N-1>(t));
    }
};
```

Stop condition

```
template <class Tuple>
struct tuple_forward_loop<Tuple, 0>
{
    template <typename Callable>
    static void invoke(Tuple &, Callable &) {}
};
```

Activation

```
template <typename Callable, class Tuple>
void iterate_forward(Callable &func, Tuple &t)
{
    tuple_forward_loop<
        tuple_size<Tuple>::value,
        Tuple
     >::invoke(t, func);
}
```

Syntactic sugar

```
template <typename Callable, class Tuple>
void iterate_forward(Tuple &t)
{
    Callable func;
    iterate_forward(func, t);
}
```

Putting it all together

```
struct printer
{
    temlpate <typename Printable>
    void operator()(const Printable &val) { cout << val << endl; }
}

tuple<int, float, string> my_tuple{17, 3.14, “my sharona”};
iterate_forward<printer>(my_tuple);
```

```
17
3.14
my sharona
```

Pretty print

```
struct pretty_printer
{
    pretty_printer() { cout << “(“; }
    ~pretty_printer() { cout << “)” << endl }
    temlpate <typename Printable>
    void operator()(const Printable &val)
    {
        if (!first) cout << “, “;
        cout << val;
        first = false;
    }

    bool first = true;
};
```

```
(17, 3.14, my sharona)
```

Additional std::tuple related elements

Classes

- std::tuple_element
- std::uses_allocator

Functions

- std::forward_as_tuple
- std::tuple_cat

Constants

- std::ignore

Additional C++17 elements

- Improve deduction rules
- Use tuple as a set of parameters for function invocation
  - std::apply
  - std::make_from_tuple

Summary

- std::tuple is useful in replacing “trivial” structures
- There is still work to be done to make it more useful

Check Point®
SOFTWARE TECHNOLOGIES LTD

THANK YOU