Tuples: The Good, The Bad and the Ugly

Timothy C. Wright
Ann Arbor/Detroit C++ Group
February 28th, 2018

```
// Normally use POD to store information

struct Animal
{
    std::string name;
    int numberOfLegs;
    float weight;
};
```

```
// What if we want to get values from a C function?

// We might have many structures and many functions
extern “C” {
void getAnimal(char name[10], int *legs, float *weight);
}
```

```
Animal a;
getAnimal(&a.name, &a.numberOfLegs, &a.weight); //ERROR
// Can’t assign a.name from the char*
```

```
//Compiles!

char name[10];
getAnimal(name, &a.numberOfLegs, &a.weight);
```

```
a.name = name;

assert(a.name == std::string(“cat”))
```

What if we have many C functions?

Can we make the code simpler?

getAnimal

getPlant

getBuilding

getGeography

```
// Can we use a tuple instead?

// Kind of like a POD struct
std::tuple<std::string, int, float> anAnimalTuple;
```

```
std::tuple<std::string, int, float> anAnimalTuple;

// Interface to tuple more ceremony, different
std::string theAnimalName = std::get<0>(anAnimalTuple);
int numberOfLegs = std::get<1>(anAnimalTuple);
float weight = std::get<2>(anAnimalTuple);
```

```
//Use unscoped enum

enum AnimalAttributes {Name, Legs, Weight};

std::string theAnimalName = std::get<Name>(anAnimalTuple);
int numberOfLegs = std::get<Legs>(anAnimalTuple);
float weight = std::get<Weight>(anAnimalTuple);
```

```
// But unscoped enums are not type safe
enum PlantAttributes {Name, Height, Width};
std::tuple<std::string, float, float> aPlantTuple;
float width = std::get<Weight>(aPlantTuple); // Compiles
```

```cpp
template <typename... TArgs>
class Animal
{
public:
    enum Names { Name, Legs, Weight };

    template<Names N>
    auto get() const {
        return std::get<N>(m_t);
    }

    template<Names N, typename T>
    auto set(T t)
    {
        std::get<N>(m_t) = t;
    }

private:
    std::tuple<TArgs...> m_t;

};
```

```
Animal<std::string, int, float> animal;
animal.set<Animal::Name>(5);
auto value = animal.get<Animal::Name>();
```

```
auto value = animal.get<0>();  // fail
```

Back to the C function:

```
char name[10];
getAnimal(name, &a.numberOfLegs, &a.weight);
```

```
callFunc(getAnimal, anAnimalTuple);
```

```
// Start with C++ Function

void getAnimal2(std::string &name, int &legs, float &weight) {
    name = "dog";
    legs = 4;
    weight = 51;
}
callFunc(getAnimal2, anAnimalTuple);

assert(std::get<0>(anAnimalTuple) == std::string("dog"));
```

```
template<typename F, typename Tuple>
void callFunc(F f, Tuple &t)
{
    using seq_type = std::make_index_sequence<std::tuple_size<Tuple>{}>;
    callFunc(f, t, seq_type{});
};
```

```
template<typename F, typename Tuple, size_t...Is>
void callFunc(F f, Tuple &t, std::index_sequence<Is...>)
{
    f(std::get<Is>(t)...);
};
```

What about our C function?

```
extern "C" {
void getAnimal(char name[10], int *legs, float *weight) {
    strncpy(name, "cat", 3);
    *legs = 4;
    *weight = 15;
}
}

AnAnimal animal;   //??
callFunc(getAnimal, animal);
```

```
template<size_t N>
struct a_string
{
    size_t length = N;
    char value[N]={};
};

using AnAnimal = std::tuple< a_string<10>, int, float>;
```

```
template<typename F, typename... Args>
auto callFunc(F f, std::tuple<Args...> &t)
{
  static const std::size_t args_count = sizeof...(Args);
  dispatchToFunc(f, t, std::make_index_sequence<args_count>());
  return t;
}
```

```
template <typename F, typename Tuple, std::size_t... I>
void dispatchToFunc(F f, Tuple &t, std::index_sequence<I...>)
{
    f(getModifiableValue(std::get<I>(t))...);
}
```

```
template<typename T>
T* getModifiableValue(T &t)
{
    return &t;
}
```

```
template<size_t N>
auto getModifiableValue(a_string<N> &od)
{
    return od.value;
}
```

```
template<size_t N>
struct a_string
{
    size_t length = N;
    char value[N]={};
};
```

```
extern "C" {
void getC_Animal(char name[10], int *legs, float *weight) {
    strncpy(name, "cat", 3);
    *legs = 4;
    *weight = 15;
}
}
```

```
AnAnimal animal;
callFunc(getAnimal, animal);
assert(std::strncmp(std::get<0>(animal).value, "cat", 3)==0);
```

```
// Have
using AnAnimal = std::tuple< a_string<10>, int, float>;

//Want
using AnAnimal = std::tuple<std::string, int, float>;
```

```
callFunc(getC_Animal, animal);
auto converted = convertValues(animal);
// converted is tuple<std::string, int, float>
```

```
template<typename... Args>
auto convertValues(std::tuple<Args...> &t)
{
    static constexpr std::size_t args_count = sizeof...(Args);
    return convertValues_impl(t, std::make_index_sequence<args_count>());
}
```

```
template<typename OldTuple, std::size_t... I>
auto convertValues_impl(OldTuple &t, std::index_sequence<I...>)
{
    auto out = std::make_tuple(valueConverter(std::get<I>(t))...);
    return out;
}
```

```cpp
template<typename T>
T valueConverter(T &t)
{
    return t;
}

template<size_t N>
auto valueConverter(a_string<N> &od)
{
    std::string v(od.value);
    return v;
}
```

```
// Remove types from a tuple

using namespace std::string_literals;

auto a_bunch_of_types = std::make_tuple("s1"s, "s2"s, 3, 4, "s3"s);


auto reduced_bunch = remove_type<int>(a_bunch_of_types);
assert(std::tuple_size<decltype(reduced_bunch)>::value == 3)
```

```
template<typename ToRemove, typename Tuple>
auto remove_type(Tuple &t)
{
    using seq_type = std::make_index_sequence<std::tuple_size<Tuple>{}>;
    return remove_type_impl<ToRemove, Tuple>(t, seq_type());
}
```

```
template<typename ToRemove, typename Tuple, std::size_t... Is>
auto remove_type_impl(Tuple &t, std::index_sequence<Is...>)
{
    return std::tuple_cat(remove_type(std::get<Is>(t),
                typename std::is_same<ToRemove,
                typename std::tuple_element<Is, Tuple>::type>::type())...);
};
```

```
// Choose a function based on std::is_same<T, U>

template<typename U>
auto remove_type(U &u, std::integral_constant<bool, true>)
{
    return std::tuple<>();
};

template<typename U>
auto remove_type(U &u, std::integral_constant<bool, false>)
{
    return std::tuple<U>(u);
};
```

Conclusion

- Accessing a member of a tuple type safe
- Using tuples to get values from a function call
- Convert types in a tuple
- Remove types from a tuple

Things I did not cover

- Converting

std::tuple<std::vector<int>> to std::vector<std::tuple<int>>

- tuples of reference variables

- Boost libraries like Fusion and Hana

    - Give you Macros to convert a struct to a form for type manipulation

    - More tuple support