How to add C++ structured binding support to your own types

devblogs.microsoft.com/oldnewthing/20201015-00

October 15, 2020

Raymond Chen

Last time, we took a quick look at C++ structured binding. This time, we'll see how to add structured binding support to your own types.

For concreteness, let's say that we want to make this class available to structured binding:

```
class Person
{
public:
  std::string name;
  int age;
};
```

Now, technically, you don't have to do anything to make this available to structured binding because there are special rules that automatically enable structured binding for simple structures. But let's do it manually, just so we can see how it's done.

Step 1: Include `<utility>`.

Step 2: Specialize the `std::tuple_size` so that its `value` is a `std::size_t` integral constant that says how many pieces there are.

In our case, we have two pieces, so the value is 2.

```
namespace std
{
  template<>
  struct tuple_size<::Person>
  {
    static constexpr size_t value = 2;
  };
}
```

If you have included `<type_traits>`, then you can use the predefined `integral_constant` template class to do the work of declaring the `value`.

```
namespace std
{
  template<>
  struct tuple_size<::Person>
      : integral_constant<size_t, 2> {};
}
```

Step 3: Specialize the `std::tuple_element` so that it identifies the type of each piece. You need as many specializations as you have pieces you declared in Step 2. The indices start at zero.

```
namespace std
{
  template<>
  struct tuple_element<0, ::Person>
  {
    using type = std::string;
  };

  template<>
  struct tuple_element<1, ::Person>
  {
    using type = int;
  };
}
```

If you have only two parts, you can simplify this by taking advantage of `std::conditional`.

```
namespace std
{
  template<size_t Index>
  struct tuple_element<Index, ::Person>
    : conditional<Index == 0, std::string, int>
  {
    static_assert(Index < 2,
      "Index out of bounds for Person");
  };
}
```

If you have more than two pieces, I guess you could chain the `conditional`s:

```
namespace std
{
  template<size_t Index>
  struct tuple_element<Index, ::Whatever>
    : conditional<Index == 0, std::string,
        conditional<Index == 1, int, whatever>>
  {
    static_assert(Index < 3,
      "Index out of bounds for Whatever");
  };
}
```

but this gets unwieldy really fast. I would repurpose `std::tuple`, which we saw some time ago was a handy way to store a bunch of types.

```
namespace std
{
  template<size_t Index>
  struct tuple_element<Index, ::Whatever>
    : tuple_element<Index, tuple<std::string, int, whatever>>
  {
  };
}
```

We even rely on `tuple_element` to do the bounds checking for us!

Step 4: Provide all of the `get` functions.

You have quite a few choices here. One option is to make the `get` functions members of your original class.

```
class Person
{
public:
  std::string name;
  int age;

  template<std::size_t Index>
  std::tuple_element_t<Index, Person>& get()
  {
    if constexpr (Index == 0) return name;
    if constexpr (Index == 1) return age;
  }
};
```

Or you can add them as free functions.

```
template<std::size_t Index>
std::tuple_element_t<Index, Person>& get(Person& person)
{
  if constexpr (Index == 0) return person.name;
  if constexpr (Index == 1) return person.age;
}
```

Adding them as free functions is convenient if you are trying to retrofit structured binding onto an existing class that you cannot change.

Another decision you have to make is which types of bindings you want to support, and what they will produce. The examples I gave above support mutable lvalue references and produce mutable lvalue references. This means that future assignments into the deconstructed values wll propagate into the original, assuming it too was captured by reference.

```
Person p;

auto&& [name, age] = p;
name = "Fred";
age = 42;
```

The `auto&&` captures `p` by universal reference, so the `get` calls are made on the original object `p`. Those `get`s return references, so the modifications to `name` and `age` are modifications into the original object `p`.

But here's another case:

```
Person p;

auto [name, age] = p;
name = "Fred";
age = 42;
```

In this case, the `auto` captured `p` by value into the hidden variable. When the `get` calls are made on the hidden variable, they are made on a copy, which means that the modifications to `name` and `age` are modifications of the copy, not the original object `p`.

And then we have this:

```
const Person p;

auto&& [name, age] = p;
```

This fails to compile because the `get` methods do not support `const Person`. You probably want those to return const references to the `name` and `age`.

This means that in practice, you need const and non-const overloads of the `get` method. And while you're at it, you may as well complete the set with the const and non-const rvalue overloads.

```
class Person
{
public:
  std::string name;
  int age;

  template<std::size_t Index>
  std::tuple_element_t<Index, Person>& get() &
  {
    if constexpr (Index == 0) return name;
    if constexpr (Index == 1) return age;
  }

  template<std::size_t Index>
  std::tuple_element_t<Index, Person> const& get() const&
  {
    if constexpr (Index == 0) return name;
    if constexpr (Index == 1) return age;
  }

  template<std::size_t Index>
  std::tuple_element_t<Index, Person>& get() &&
  {
    if constexpr (Index == 0) return std::move(name);
    if constexpr (Index == 1) return std::move(age);
  }

  template<std::size_t Index>
  std::tuple_element_t<Index, Person> const& get() const&&
  {
    if constexpr (Index == 0) return std::move(name);
    if constexpr (Index == 1) return std::move(age);
  }
};
```

Fortunately, you can consolidate a lot of this with a helper method that infers the necessary boilerplate.

```
class Person
{
public:
  std::string name;
  int age;

  template<std::size_t Index>
  auto&& get()       &  { return get_helper<Index>(*this); }

  template<std::size_t Index>
  auto&& get()       && { return get_helper<Index>(*this); }

  template<std::size_t Index>
  auto&& get() const &  { return get_helper<Index>(*this); }

  template<std::size_t Index>
  auto&& get() const && { return get_helper<Index>(*this); }

private:
  template<std::size_t Index, typename T>
  auto&& get_helper(T&& t)
  {
    static_assert(Index < 2,
      "Index out of bounds for Custom::Person");
    if constexpr (Index == 0) return std::forward<T>(t).name;
    if constexpr (Index == 1) return std::forward<T>(t).age;
  }
};
```

Note that we had to restore the static assertion because we are no longer relying on `tuple_element` to do the bounds checking.

It is more common to use free functions instead of member functions, in which case you would have something like this:

```
template<std::size_t Index, typename T>
auto&& Person_get_helper(T&& p)
{
  static_assert(Index < 2,
    "Index out of bounds for Custom::Person");
  if constexpr (Index == 0) return std::forward<T>(t).name;
  if constexpr (Index == 1) return std::forward<T>(t).age;
}

template<std::size_t Index>
auto&& get(Person& p)
{
  return Person_get_helper<Index>(p);
}

template<std::size_t Index>
auto&& get(Person const& p)
{
  return Person_get_helper<Index>(p);
}

template<std::size_t Index>
auto&& get(Person&& p)
{
  return Person_get_helper<Index>(std::move(p));
}

template<std::size_t Index>
auto&& get(Person const&& p)
{
  return Person_get_helper<Index>(move(p));
}
```

Now, there's no requirement that the `get` methods return references. You can have them return values, and the structured binding will simply capture values rather than references. This is handy if the underlying object doesn't have access to references.

```
class Person
{
public:
  std::string CalculateName() const;
  int CalculateAge() const;

  template<std::size_t Index>
  auto get() const
  {
    static_assert(Index < 2,
      "Index out of bounds for Custom::Person");
    if constexpr (Index == 0) return CalculateName();
    if constexpr (Index == 1) return CalculateAge();
  }
};
```

We'll see an application of this trick next time.

Bonus chatter: Since the structured binding transformation is purely syntactic, there's no rule that prevents you from having the `get` functions return things that are unrelated to the source of the binding. It's probably not a great idea, though, since nobody will be expecting that.

Raymond Chen

Follow