Consistent Metafunction Aliases

Document #:	N3887
Date:	2013-12-26
Project:	Programming Language C++ Library Evolution Group

Reply-to:	Michael Park <mcypark@gmail.com>

1 Introduction

This paper is inspired by [N3655], section 3 of which was adopted at WG21's 2013 Bristol meeting. That section provided (as did its predecessor [N3546]) template aliases for the result of *metafunctions* in **<type_traits>**. However, the broader question of analogous template aliases throughout the remainder of the standard library was not addressed.

This paper recommends a systematic guideline to steer future WG21 decisions in deciding when a *metafunction-name_**t*** template alias should accompany a standard library *metafunction*. After applying this recommended guideline to the entire C++14 standard library, we conclude that **tuple_element_t** is the only missing alias. We then propose wording (a) to remedy this lack and (b) to take advantage of the proposed remedy. Finally, we also present an alternative guideline and its implications, and provide justifications for favoring the recommended guideline.

2 Motivation and Scope

[N3655] provided the motivation for the existence of the **_t** template aliases. The goal of this paper is to introduce a guideline for WG21 to systematically decide when a *metafunction* in the standard library should be accompanied by a *metafunction-name_**t*** template alias. Specifically, we want to ensure that future standard library *metafunctions* stay consistent with those we currently have.

The proposed guideline is the following:

> A class template should be accompanied by a *metafunction-name_**t*** template alias if it provides a public member type named **type** and no other accessible members.

An exhaustive search through the text of [N3797] reveals that the only class templates that meet the above guideline are the following:

- *TransformationTraits* from section 3 of [N3655].
- **tuple_element**

Since **tuple_element_t** is the only missing template alias, we propose to add it.

3 Impact on the Standard

This proposal is a pure library extension. It does not require any new language features, it is merely an extension of an existing practice adopted from [N3655].

4 Proposed Wording

Add to **<tuple>** synopsis of [N3797]:

```
// 20.4.2.5: tuple helper classes:
+ template<size_t I, class T>
+ using tuple_element_t = typename tuple_element<I, T>::type;
```

The definition of **get** functions in **§20.3.4**, and **§20.4.2.6** of [N3797] can be simplified as follows:[1]

```
// 20.3.4: tuple-like access to pair:
template<size_t I, class T1, class T2>
-  constexpr typename tuple_element<I, std::pair<T1, T2> >::type&
+  constexpr tuple_element_t<I, pair<T1, T2> >&
-    get(std::pair<T1, T2>&) noexcept;
+    get(pair<T1, T2>&) noexcept;

template<size_t I, class T1, class T2>
-  constexpr typename tuple_element<I, std::pair<T1, T2> >::type&&
+  constexpr tuple_element_t<I, pair<T1, T2> > &&
-     get(std::pair<T1, T2>&&) noexcept;
+     get(pair<T1, T2>&&) noexcept;

template<size_t I, class T1, class T2>
-   constexpr const typename tuple_element<I, std::pair<T1, T2> >::type&
+   constexpr const tuple_element_t<I, pair<T1, T2> >&
-     get(const std::pair<T1, T2>&) noexcept;
+     get(const pair<T1, T2>&) noexcept;

// 20.4.2.6, element access:
template <size_t I, class... Types>
-  constexpr typename tuple_element<I, tuple<Types...> >::type&
+  constexpr tuple_element_t<I, tuple<Types...> >&
    get(tuple<Types...>&) noexcept;

template <size_t I, class... Types>
-  constexpr typename tuple_element<I, tuple<Types...> >::type&&
+  constexpr tuple_element_t<I, tuple<Types...> >&&
    get(tuple<Types...>&&) noexcept;

template <size_t I, class... Types>
-  constexpr typename tuple_element<I, tuple<Types...> >::type const&
+  constexpr const tuple_element_t<I, tuple<Types...> >&
    get(const tuple<Types...>&) noexcept;
```

[1] Note the removal of explicit **std::** namespace qualifiers and a standardized placement of **const** as a drive-by-fix.

5 Design Decisions

We considered an alternative guideline:

> A class template should be accompanied by a *metafunction-name_**t*** template alias if it provides a public member type named **type** and possibly other accessible members.

An exhaustive search through the text of [N3797] reveals that the class templates that meet this alternative guideline are the following:

- **integral_constant**
  - *TransformationTraits* from section 4 of [N3655]
  - **is_bind_expression**
  - **is_error_code_enum**
  - **is_error_condition_enum**
  - **is_placeholder**
  - **ratio_equal**
  - **ratio_greater**
  - **ratio_greater_equal**
  - **ratio_less**
  - **ratio_less_equal**
  - **ratio_not_equal**
  - **tuple_size**
  - **uses_allocator**
- **ratio**
- **ratio_add**
- **ratio_divide**
- **ratio_multiply**
- **ratio_subtract**
- **reference_wrapper**

This guideline is overly accepting, because we are rarely interested in the **type** member of these class templates. Consider **integral_constant** along with all of the class templates listed under it which inherit from **integral_constant**. The class member we are far more interested in for these class templates is the **value** member. For **ratio**, and **ratio**_*operations*, the **type** member is simply a type alias for itself, and **reference_wrapper**'s **type** member is nothing but an identity type alias for T.

Also consider their intended use cases. All of **is_*property***, **ratio_*compare***, and **uses_allocator** are intended for compile-time boolean tests. **tuple_size** is intended to acquire the size of a **tuple**. **integral_constant** is intended to capture a compile-time value and is sometimes used for tagged dispatch. **ratio_*operations*** are used for compile-time rational arithmetic. **reference_wrapper** is simply intended for containing references in our containers. The intended use cases make it clear that these are not *metafunctions* that yield a type as a result.

On the contrary, the proposed guideline captures the correct intended use of the class templates. If the only member provided for a class template is **type**, it is most likely a *metafunction* which yields a type as a result. This categorization holds true for all of *TransformationTraits* from section 3 of [N3655] and **tuple_element**. For example, **add_const** is a *metafunction* which yields a const-qualified version of the type as a result, and **tuple_element** is a *metafunction* which yields the type that lies at a specified index within a list of types as a result. Therefore providing a template alias for such class templates will be useful for convenient access to the **type** member.

6 Acknowledgements

Thanks to Walter E. Brown for encouraging me to write this paper.

7 References

[N3546] Walter E. Brown, *TransformationTraits Redux*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3546.pdf

[N3655] Walter E. Brown, *TransformationTraits Redux, v2*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3655.pdf

[N3797] Stefanus Du Toit, *Working Draft, Standard for Programming Language C++*
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf