MARCH 30, 2018MARCH 30, 2018 ⁄ JEFFAMSTUTZ

Building a C++ SIMD Abstraction (4/N) – Type Traits Are Your Friend

In this post I want to share with you how type traits can be an asset to C++ template code. Given that tsimd is based on a core template over a collection of values stored in a SIMD register, there's plenty of motivation for finding ways to keep the interface and implementation *easy to use* and *difficult to misuse*. Obviously these are goals for better code and not a hard set of rules, but I think type traits can definitely aid template code moving toward these goals.

If you haven't already, I suggest getting caught up on earlier posts in this series: **motivation for simd (https://jeffamstutz.io/2017/12/07/building-a-c-simd-abstraction-1-n-motivation/)**, **a look at SIMD programming options (https://jeffamstutz.io/2017/12/14/building-a-c-simd-abstraction-2-n-status-quo-my-perspective/)**, and **introducing tsimd (https://jeffamstutz.io/2018/01/02/building-a-c-simd-abstraction-3-n-a-tour-of-tsimd/)**.

So what are type traits?

Type traits are templates which let you infer compile-time information about a give type or instance. C++11 introduced a number of type traits that come in the standard library (found in the "type_traits" header, **reference here (http://en.cppreference.com/w/cpp/types)**), but type traits are also tools which library makers can write themselves to solve specific problems found in library development.

In my experience, type traits come in two flavors: those which generate a value based on given type(s) and those which generate an output type from given type(s). By convention, those types which generate some value result from their input types define a static member called 'value'. *These values could be technically anything, but I almost always compute a simple boolean condition with them*. Also by convention, types which generate new types from an input type define a type alias member called 'type'. I use these conventions in my own type traits to be consistent with the standard library and I highly recommend that you do the same…we don't want to remember more than we have to!

Why would I use type traits?

Given the above flavors of type traits, I find myself using them in the following situations.

Type inference

In implementing tsimd I found myself needing to map multiple input types into another set of types. Here, given a pack, where 'T' is the element type of the pack and 'W' is the width, I could easily use a type trait which maps the pair to a particular output type. Here's a concrete example:

```cpp
/* Dummy placeholder type when  doesn't make sense
   for a particular CPU ISA */
template <typename T, int W>
struct simd_undefined_type {};

/* Core type trait */
template <typename T, int W>
struct simd_type
{
  using type = simd_undefined_type<T, W>;
};

/* Specialize for AVX */
#ifdef __AVX__
template <>
struct simd_type<float, 8>
{
  using type = __m256;
};

template <>
struct simd_type<int, 8>
{
  using type = __m256i;
};
#endif

/* Continue on with other ISA specializations... */
```

In this example, I use a type trait to infer which intrinsic type should represent the pack. It's OK if you don't know what a '__m256' actually is because that is exactly what tsimd is trying to hide from you! For the sake of some context of how this gets used, here's a stripped down version of the pack class definition which highlights just part of the simd_type usage.

```cpp
#include <tsimd_traits.h>

template <typename T, int W>
struct pack
{
  using intrinsic_t = typename simd_type<T, W>::type;

  /* ... */

  /* Construct pack with existing intrinsic register */
  pack(intrinsic_t value);

  /* ... */

  /* Data stored in this pack, multiple representations */
  union
  {
    intrinsic_t v;
    std::array<T, W> arr;
  };
};
```

Thus you can see that with one type trait to infer what the underlying intrinsic type should be, only one declaration is required for the different contexts which a pack may interact with an intrinsic type.

There are a few lurking advantages here too, one of which I'll point out: because intrinsic_t turns into a dummy type in situations where there the intrinsic type doesn't make sense, some interfaces intentionally become invalid. In the case of the constructor from intrinsic_t to pack (as declared above), when compiling for AVX you can't construct a pack with a single __m128 (an SSE register), even though it is a valid type for pack. Furthermore, if you use a pack, the union storing the various representations of the data will silently turn the 'v' member into nothing. Therefore (for the most part) all of this gets implemented exactly once: no need to duplicate definitions of pack for different ISAs.

Type validation

Another case for using type traits is to validate a template parameter against a particular condition or set of conditions. Here's a simple example:

```cpp
template <typename T1, typename T2>
struct same_size
{
  static constexpr bool value = (sizeof(T1) == sizeof(T2));
};

/* example usage... */
static_assert(same_size<float, int32_t>::value, "should be true");
static_assert(!same_size<float, char>::value, "should be false");
```

Simply put, same_size takes in two types and defines a compile-time boolean whether the two types are the same size in bytes.

Another example is detecting if a template type (such as pack) is an instantiation of that template, without having to infer each of the template arguments. One way to solve this is to use inheritance and the std::is_base_of type trait. In tsimd, pack does this with the following (simplified, of course).

```cpp
#include <type_traits>

struct pack_base {}; // only for compile-time decection

template <typename T, int W>
struct pack : public pack_base
{
  /* pack members, blah blah blah */
};

/* type trait to detect if a given type is a pack or not */
typename <typename TYPE_IN_QUESTION>
struct is_pack
{
  static constexpr bool value =
    std::is_base_of<pack_base, TYPE_IN_QUESTION>::value;
};

/* example usage */
using vfloat8 = pack<float, 8>;
static_assert(is_pack<vfloat8>::value, "should be true");
static_assert(!is_pack<float>::value, "should be false");
```

I've found the use of an empty base class as a way to detect general usage of a template to be useful in a number of places, include both tsimd and OSPRay.

SFINAE (Substituion Failure Is Not An Error)

The last case I find myself using type traits is to make SFINAE a more bearable thing. SFINAE is about culling certain overloads from being candidates during overload resolution. To some of you that may sound confusing, but hopefully an example will help. Consider the following plus operator between a pack and scalar type:

```
using vfloat8 = pack<float, 8>;

/* assume the following pack + pack operator already exists */
vfloat8 operator+(const vfloat8 &p1, const vfloat8 &p2);

/* template in question... */
template <typename OTHER_T>
vfloat8 operator+(const vfloat8 &p, OTHER_T o)
{
  return p + vfloat8(o);// add two packs
}
```

The problem with the above example is that OTHER_T needs to be convertible to float (the element type in vfloat8) in order for operator+() to be considered between vfloat8 and OTHER_T. I won't go into tons of detail about how many ways you can use SFINAE (lookup std::enable_if examples if you want to see how ugly it can get), but by far the most readable way of using SFINAE is to use an extra default argument at the end of the template parameter list which uses a single type trait which encapsulates everything that OTHER_T *can* be. The way this will look is as follows.

```
using vfloat8 = pack<float, 8>;

template <typename OTHER_T, typename = /*insert type trait here*/>
vfloat8 operator+(const vfloat8 &p, OTHER_T o);
```

Thus the goal is to define a type trait which OTHER_T is 1) not a pack and 2) is convertible to the given pack element type. Let's start with defining the trait template.

```
template <typename PACK_T, typename OTHER_T>
struct valid_pack_scalar_operator
{
  // assume we have is_pack, as described earlier
  static_assert(is_pack<PACK_T>::value, "PACK_T must be a pack!");

  // assume pack defines an alias for the element type
  using element_t = PACK_T::element_t;

  // the core value we are trying to compute
  static constexpr bool value =
    !is_pack<OTHER_T>::value &&
    std::is_convertible<element_t, OTHER_T>::value;
};

/* shortcut for using above value inside std::enable_if */
template <typename PACK_T, typename OTHER_T>
using valid_pack_scalar_operator_t =
  std::enable_if_t<
    valid_pack_scalar_operator<PACK_T, OTHER_T>::value
  >;
```

Note that the enable_if alias (valid_pack_scalar_operator_t) is a standard pattern in the STL, where a given trait will give you a "{trait_name}_t" version to use for this very purpose. So now that we have the above valid_pack_scalar_operator_t type defined, we can constrain the operator+() template as follows.

```
using vfloat8 = pack<float, 8>;

template <typename OTHER_T,
          typename = valid_pack_scalar_operator_t<vfloat8, OTHER_T>>
vfloat8 operator+(const vfloat8 &p, OTHER_T o);

/* example usage */
vfloat8 a = //...
vfloat8 b = //...
std::vector<float> v = //...

auto c = a + b;// works
auto d = a + v;// doesn't work, unless pack<> + std::vector<> defined somewher
```

Thus the overall goal is achieved: a template which can take *anything* gets appropriately culled down to only the types which it expects without the need to explicitly write out every version of the template manually. Furthermore, other overloads could also be introduced without this version colliding with it. In other words, if someone else wanted to define operator+() which takes any container type, one could use a similar strategy, but with a type trait which SFINAEs out OTHER_T which isn't a container….etc, etc.

SFINAE is a complicated topic in itself, but if you build up type traits which directly express (by giving it a name) a particular predicate of interest (i.e. are the given type(s) I am interested in OK for this template or not?) will help you keep things straight and *somewhat* readable. Yes, **I know that C++ concepts will improve this**, but that is not yet a mainstream enough feature implement in enough compilers that I can use it right now (it's not even standardized yet!)…so as of today, this is what I've got.

Conclusion

Type traits come in two flavors: one which define types and others which define a compile-time values. With these you can do useful things to improve your C++ template code like checking to make sure template types are well defined according to your template's assumptions. I hope that you find type traits useful in making your template code *easier to use* and ***difficult to misue***.

As tsimd continues to mature, there will be more to talk about…so the series is definitely not over. The hiatus from writing was certainly not intentional, I'm just busy!

Anyway, until next time…happy coding!
Posted in C++, Clear Code, Reducing Complexity, Uncategorized/Leave a comment