Ruminations
Mulling over topics I find interesting

A simple introduction to type traits

Posted on 11/23/2011 by Aaron Ballman

Type traits are a slightly more advanced topic in C++ because it they are heavily used in template metaprogramming. However, it is not an impenetrable concept, and it comes with some great benefits if you like to write generic, reusable code. I'd like to give you a simple introduction to the concept of type traits, and show some of the more beneficial use cases for them.

When you use templates in your source code, you are creating the ability to perform truly generic actions. This is a blessing for code reusability because it allows you to write your code once, and use it for multiple situations. A great example of this would be the entire Standard Template Library (STL) that ships with C++ compilers. You have access to classes like vector and map which allow you to create arbitrary lists and tables of data, regardless of what that data is. However, there are situations where you need more control over the template types in order to write efficient, or correct, code.

For instance, let's say you are writing a function that byte swaps values. It doesn't really matter if you get a 16-bit integer, a 32-bit integer, a 64-bit integer, so you try to write a generic function to do it. A naive implementation might look something like this:

```
template <typename T>
T byte_swap( T value ) {
  unsigned char *bytes = reinterpret_cast< unsigned char * >( &value );
  for (size_t i = 0; i < sizeof( T ); i += 2) {
    // Take the value on the left and switch it
    // with the value on the right
    unsigned char v = bytes[ i ];
    bytes[ i ] = bytes[ i + 1 ];
    bytes[ i + 1 ] = v;
  }
  return value;
}
```

If you pass in the 32-bit value 0x11223344, you will get back the value 0x22114433; or if you pass in the 16-bit value 0x1122 you will get back 0x2211. We're done, right? Well, not exactly.

Now we come to the crux of the problem with template programming — in its current form, you have no control over what type T gets passed in. So you could also pass a double to this function, and it will happily scramble it. Or worse yet, you could pass in a char, and it will scramble it with a byte of random memory you don't own, possibly causing a crash!

One possible way to solve this is with partial specialization of the template. You can create versions of the function that are specialized for the types you want to have special behavior for. For instance, we could add the following:

```
template <>
double byte_swap( double value ) {
  assert( false && "Illegal to swap doubles" );
```

```
  return value;
}

template <>
char byte_swap( char value ) {
  assert( false && "Illegal to swap chars" );
  return value;
}
```

Now, if the caller attempts to pass in a double or a char, they will get the versions of the function which assert and harmlessly return the input value. However, what happens if the caller passes in a float? Or an unsigned char? Or a pointer? You can quickly see how such a simple function would balloon out into many specializations, all to protect the caller. As it stands with our example currently, we've already written the same number of functions as it would take to just byte swap the three integer types separately!

This is where type traits come in handy. A type trait is a way for you to get information about the types passed in as template arguments, at compile time, so you can make more intelligent decisions. The basics behind a type trait are:

- You use a templated structure, usually named with the type trait you are after. Eg) is_integer, is_pointer, is_void
- The structure contains a static const bool named *value* which defaults to a sensible state
- You make specializations of the structure representing the traits you want to expose, and have those set their bool value to a sensible state
- You use a type trait by querying its value, like: my_type_trait<T>::value

For instance, let's make a type trait that decides whether the value is swappable. In this case, the only valid datatypes we want to consider are signed or unsigned shorts, longs or long longs. Our type trait could look something like:

```
template <typename T>
struct is_swapable {
  static const bool value = false;
};

template <>
struct is_swapable<unsigned short> {
  static const bool value = true;
};

template <>
struct is_swapable<short> {
  static const bool value = true;
};

template <>
struct is_swapable<unsigned long> {
  static const bool value = true;
};

template <>
struct is_swapable<long> {
  static const bool value = true;
};

template <>
```

```
struct is_swapable<unsigned long long> {
  static const bool value = true;
};

template <>
struct is_swapable<long long> {
  static const bool value = true;
};
```

Now we can modify our swap function to make use of this type trait by adding:

```
assert( is_swapable<T>::value && "Cannot swap this type" );
```

That is the basics of type traits in a nutshell! Let's recap the important pieces. There is a structure named is_swapable, and by default nothing is swapable so its value is set to false. Then we have specializations for all of the non-char, non-bool, integer datatypes which set the value to true. When we want to use the type trait, we ask the structure for its value and the compiler figures out which structure to pull the value from via template specialization. This is the general pattern you will see for almost all type traits.

Now you might be saying, "but now there's a ton more code, and there's still no type safety because you can still pass in whatever you want!", and you are correct. However, there are a few saving graces. In C++11, there is now a standard STL header called type_traits, and it contains traits for almost everything you can think of. There are traits to tell you the basic datatypes, whether something is a pointer, whether something is an array, whether something is const, etc. There are even traits to remove traits, like turn a pointer type into a value type, or a const type into a non-const type. Most of the time you are able to make use of these built-in type traits for your own code. Also, as part of C++11 is the new static_assert functionality, which uses compile-time constant expressions as a way to generate errors. So, for instance, our byte_swap assert code could look like this in C++11:

```
static_assert( std::is_integral< T >::value && sizeof( T ) >= 2, "Cannot swap values of this
```

The is_integral type trait is part of the C++11 standard, but it also includes char and bool as integral types. So we check to make sure the size of the type passed in is at least two bytes. If either of these tests fail, the assert is fired at compile time, generating the error message "Cannot swap values of this type."

If you can't use C++11 yet, all hope is not lost. The boost library comes with many type traits, as well as its own implementation of static assserts. Or, if all else fails, you can create your own type traits, as we've seen above.

Regardless, type traits are an incredibly powerful tool to use when doing template programming. It allows you to have more fine-tuned control over your template functionality, ensuring that no one accidentally misuses it with silent errors.

Another thing to keep in mind is that type traits can make use of other type traits. So, if you wanted to create an is_swapable type trait for use in multiple functions, you could do so like this:

```
template <typename T>
struct is_swapable {
  static const bool value = std::is_integral< T >::value && sizeof( T ) >= 2;
};
```

Now you should have a basic understanding of type traits and why they're important. The next step is for you to determine whether they're appropriate for you to use in your source base. There's a fine line between using type traits and using template specializations. Generally, if there is a sensible, harmless fallback for template functionality, I will use specialization to perform the fallback. The same is true when I wish to optimize a function for a particular type. If there is no harmless fallback, or the type passed would be a programming error, I use static_assert in conjunction with type traits. However, it is not a one-size-fits-all feature. Regardless, you now have another powerful tool to use with template code!

This entry was posted in C/C++ and tagged C++0x, datatypes, templates. Bookmark the permalink.

8 Responses to *A simple introduction to type traits*

k *says:*
12/12/2013 at 1:15 pm

Thanks, very good explanation. Source code looks a bit funny.

Trevor Hickey *says:*
12/19/2013 at 12:45 pm

Great tutorial. Spot on. I find a lot of blogs get too technical and confusing, but you explained an advanced topic perfectly. Good examples also. I'm glad you talked about their practical use, and not just "look how complicated templates can get in order to do generic stuff."

found a typo:

```
template
struct is_swapable {
static const bool value = std::is_integral && sizeof( T ) >= 2;
};
```

should be

```
template
struct is_swapable {
static const bool value = std::is_integral::value && sizeof( T ) >= 2;
};
```

*you forgot the value.

Thanks, I'm going to RSS your blog one day, whenever I get an RSS feeder set up.

Aaron Ballman *says:*
01/03/2014 at 1:00 pm

@Trevor — I'm glad you found the post helpful! And thank you for pointing out the typo, I've corrected the example.

Neil Gatenby *says:*
02/24/2015 at 12:53 pm

Really informative ; thanks a lot

Chris Chiesa *says:*
12/09/2015 at 2:34 pm

I've been using C++ since early 1998, and in all that time this is pretty much the first article I've seen that is both short enough to get all the way through, and comprehensible throughout. Congratulations and thanks!

Eugene *says:*
12/28/2015 at 3:33 am

First time I understood it in less than 10 minutes – I have not seen anybody else trying to explain template traits that is nearly this good.

Thanks. Great job.

Luis Pinto *says:*
10/11/2016 at 10:46 am

Very good job explaining this, at least the purpose of traits is much more clear here than in any other web tutorials I've been looking at!

Congrats!

---

captainwong *says:*
07/20/2017 at 12:00 am

FYI, it cannot be used for bit field of structure.

```
namespace test5 {

template
struct is_swapable {
static const bool value = std::is_integral::value && sizeof(T) >= 2;
};

template
T byte_swap(T value) {
std::assert(is_swapable::value && "Cannot swap values of this type");
unsigned char *bytes = reinterpret_cast(&value);
for (size_t i = 0; i < sizeof(T); i += 2) {
// Take the value on the left and switch it
// with the value on the right
unsigned char v = bytes[i];
bytes[i] = bytes[i + 1];
bytes[i + 1] = v;
}
```

```
return value;
}

void test()
{
std::cout << "——————————test5——————————" << std::endl;
std::cout << "is_swapable char " << is_swapable::value << std::endl;
std::cout << "is_swapable short " << is_swapable::value << std::endl;
std::cout << "is_swapable unsigned char " << is_swapable::value << std::endl;
std::cout << "is_swapable unsigned short " << is_swapable::value << std::endl;
std::cout << "is_swapable double " << is_swapable::value << std::endl;
std::cout << "is_swapable long " << is_swapable::value << std::endl;
std::cout << "is_swapable float " << is_swapable::value << std::endl;
std::cout << "is_swapable double " << is_swapable::value << std::endl;

struct s {
int hi : 24;
int lo : 8;
};

s s;
s.hi = 65535;
s.lo = 255;

std::cout << "is_swapable s.hi " << std::boolalpha << is_swapable::value << std::endl;
std::cout << "is_swapable s.lo " << std::boolalpha << is_swapable::value << std::endl;
}
```

and the output is:

```
——————————test5——————————
is_swapable char false
is_swapable short true
is_swapable unsigned char false
is_swapable unsigned short true
is_swapable double false
is_swapable long true
is_swapable float false
is_swapable double false
is_swapable s.hi true
is_swapable s.lo true
```
