Variadic CRTP

Curious, but Recurring

The Curiously-Recurring Template Pattern (CRTP) is a pleasant and common C++ coding idiom. Basically it's a way for a class to inherit a self-customized capability. For the traditional first example of CRTP, consider an object counter:

```
template <typename T>
class Ctr {
public:
    Ctr() { ++ctr_; }
    Ctr(Ctr const &) { ++ctr_; }
    ~Ctr() { --ctr_; }
    static size_t get() { return ctr_; }
private:
    static size_t ctr_;
};

template <typename T> size_t Ctr<T>::ctr_ = 0;
```

To inherit this object-counting capability we specialize `Ctr` with ourselves and then derive from the specialization:

```
class Thing1 : public Ctr<Thing1> { ~~~ };
class Thing2 : public Ctr<Thing2> { ~~~ };
```

`Thing1` and `Thing2` each inherit a properly self-customized object counting capability.

The Barton-Nackman Trick is another traditional example:

```
template <typename T>
class Eq {
    friend bool operator ==(T const &a, T const &b)
         { return a.compare(b) == 0; }
    friend bool operator !=(T const &a, T const &b)
         { return a.compare(b) != 0; }
};

template <typename T>
class Rel {
    friend bool operator <(T const &a, T const &b)
         { return a.compare(b) < 0; }
    friend bool operator <=(T const &a, T const &b)
         { return a.compare(b) <= 0; }
    friend bool operator >(T const &a, T const &b)
         { return a.compare(b) > 0; }
    friend bool operator >=(T const &a, T const &b)
```

```
        { return a.compare(b) >= 0; }
};

class Thing3 : public Eq<Thing3>, public Rel<Thing3> {
public:
    int compare(Thing3 const &rhs) const;
    ~~~
};
```

Thing3 inherits customized equality and relational operators.

More Flexible CRTP

One problem with this traditional application of CRTP is that it's inflexible. For example, Thing1 objects share an object counter. What if we'd like a version of Thing1 that can instead be compared with equality operators? We can get this level of flexibility by specifying the CRTP capability as a template template parameter.

```
template<template <typename> class CRTP>
class Thing1: public CRTP<Thing1<CRTP>> { ~~~ };
```

Now we can specify versions of Thing1 that have a counter or some other inherited capability.

```
Thing1<Ctr> a; // gets an object counter
Thing1<Eq> b;  // gets equality operators
```

We can get additional flexibility with a variadic template template parameter. For example, here's a counter capability similar to our original counter that accepts two template arguments:

```
template <typename T, typename I = size_t>
class ObjectCounter {
public:
    ObjectCounter() { ++ctr_; }
    ObjectCounter() { ++ctr_; }
    ~ ObjectCounter() { --ctr_; }
    static I get() { return ctr_; }
private:
    static I ctr_;
};

template<template <typename...> class CRTP>
class Thing1: public CRTP<Thing1<CRTP>> { ~~~ };

Thing1<Ctr> a;            // gets an object counter
Thing1<ObjectCounter> c; // gets a fancy counter
```

However, we still don't have the flexibility to take an arbitrary number of CRTP capabilities. For example, we might want to have an object counter (or two), relational operators, and equality operators. We can accomplish this with template template parameter packs or variadic template template parameter packs.

```
template<template <typename...> class... CRTPs>
class Thing3 : public CRTPs<Thing3<CRTPs...>>... { ~~~ };

Thing3<Ctr, Eq> d;            // counted, equality comparable
Thing3<Ctr, Eq, Ctr2, Rel> e; // the works!
```

Restricted Numeric Types

As an example of how this approach might be of practical use, let's create a numeric type toolkit that allows specification of restricted numeric types. Fedor Picus implemented a similar but much more sophisticated facility circa 2008 (he called them either "armored built-in types" or "restricted value types" depending on the context), but at that time he didn't have variadic templates at his disposal.

Let's define a number as a very basic numeric type that holds a value and inherits a set of properties expressed as CRTP base classes.

```
template <typename N, template <typename...> class... CRTPs>
class Number : public CRTPs<Number<N, CRTPs...>>... {
public:
   using S = decay_t<underlying_arithmetic_type_t<N>>;
   constexpr Number() // note: intentionally uninitialized
        {}
   constexpr Number(S value)
        : value_(value) {}
   constexpr S value() const
        { return value_; }
   constexpr void set_value(S a)
        { value_ = a; }
private:
   N value_;
};
```

We can then define sets of interesting numeric capabilities. For example, we might want a particular type to be output-streamable

```
template <typename T>
class Stream_i {
    friend std::ostream &operator <<(std::ostream &a, T b)
        { return a << b.value(); }
};
```

or shiftable

```
template <typename T>
class Shift_i {
    friend T operator <<(T a, T b)
        { return T(a.value() << b.value()); }
    friend T operator >>(T a, T b)
        { return T(a.value() >> b.value()); }
};
```

or equality-comparable

```
template <typename T>
class Eq_i {
    friend constexpr bool operator ==(T a, T b)
        { return a.value() == b.value(); }
    friend constexpr bool operator !=(T a, T b)
        { return a.value() != b.value(); }
};
```

and so on. A restricted numeric type is generated by selecting an underlying type and specifying what operations are available on the type.

```
using restricted_int = Number<int, Eq_i, Rel_i, Add_i, Stream_i>;
```

This restricted int can be compared with relational and equality operators, added, and streamed. By design, it doesn't support other operations like subtraction, multiplication, or increment.

Hardware Registers

Some of the orneriest numeric types you're likely to find are those used to represent memory-mapped registers. They want to be aligned just right, have a particular number of bytes, support only some operations, either do or don't want to be read or written, may change value without warning, and so on. Let's start by creating a "read/write register" type that is restricted to an expected set of operations.

```
using hw_register = unsigned volatile;
using rw_register = Number<hw_register, Eq_i, Bit_i, Shift_i>;
```

This register type is a read/write unsigned type that can be compared for equality, used with bitwise operators, and shifted. What if we need read only or write only registers?

Dan Saks came up with a useful facility for expressing read only, write only, and unused data types with template type wrappers. For example, here's a write only wrapper:

```
template <typename T>
class write_only {
public:
    write_only() {}
    write_only(T const &v) : m_(v) {}
    void operator =(T const &v) { m_ = v; }
    write_only(write_only const &) = delete;
    write_only &operator =(write_only const &) = delete;
private:
    T m_;
};
```

We can use these wrappers to augment the meaning of our register type that we earlier provided through CRTP.

```
using wo_register = write_only<rw_register>;
using ro_register = read_only<rw_register>;
using un_register = unused<rw_register>;
```

Now it's fairly straightforward to lay out a well-behaved memory-mapped device:

```
struct UART {
   un_register ULCON;  // line control
   un_register UCON;   // control
   ro_register USTAT;  // status
   wo_register UTXBUF; // transmit
   ro_register URXBUF; // receive
   rw_register UBRDIV; // baud rate divisor
};
```

We can be more restrictive if we want. For instance, it probably doesn't make sense to be able to shift a status register, and perhaps we're not interested in the value of the register as a whole, but only with the values of individual sets of bits within the register.

```
using status_register = Number<hw_register, Bit_i>;
using ro_status_register = read_only<status_register>;
~~~
struct UART {
   un_register ULCON;          // line control
   un_register UCON;           // control
   ro_status_register USTAT;   // status
   wo_register UTXBUF;         // transmit
   ro_register URXBUF;         // receive
   rw_register UBRDIV;         // baud rate divisor
};
```

Never Ass|u|me

As always when working with hardware, it makes a lot of sense to verify one's assumptions. While we're at it, we may as well verify our assumptions about all specializations of `Number`.

```
template <typename N, template <typename...> class... CRTPs>
class Number : public CRTPs<Number<N, CRTPs...>>... {
public:
```

```
    constexpr Number() // note: intentionally uninitialized
         { check(); }
    constexpr Number(S value)
         : value_(value) { check(); }
    ~~~
  private:
    static constexpr void check() {
         constexpr auto is_arith
           = is_arithmetic<underlying_arithmetic_type_t<N>>::value;
         constexpr auto is_sl = is_standard_layout<Number>::value;
         static_assert(sizeof(Number) == sizeof(N),
              "problem with size of Number");
         static_assert(alignof(Number) == alignof(N),
              "problem with alignment of Number");
         static_assert(is_arith,
              "underlying type for Number must be arithmetic");
         static_assert(is_sl, "Number is not standard layout");
    }
    N value_;
  };
```

Unfortunately, not all compilers used to compile this code pass the test. Some C++ compilers have trouble applying the Empty Base Class Optimization (EBO) under multiple inheritance, causing the first static assertion in `check` to fail if there is more than a single CRTP base class. Others fail one or more of the remaining assertions for one reason or another, depending on the size and alignment of the `value_` data member.

In next weak's installment, we'll see how to mitigate these issues by having `Number` ask pointed personal questions of the compiler and customize its implementation based on the answers received.