Emulating Mixins in C++

- Mixin
  - Idea: Mixin is a prefabricated set of definitions that is dumped into a scope
  - Example: Every random-access iterator must define 5 types and 20 operators, providing similar functionality in various ways. It would be advantageous to have syntax like this:

```
class my_iterator includes generic_random_access_iterator</* some arguments */>
{ // some definitions
};
```

    - There is no such syntax in C++ yet (and will not be in foreseeable future)
  - Important: Functions inside the mixin must be able to see the other definitions in the scope where the mixin is used, as if they were located there

```
mixin mixin1 { void function1() { ++var1; } };
mixin mixin2 { void function2() { function1(); } };
class final_class includes mixin1, mixin2 { int var1; };
```

Mixin

- Emulating mixins by inheritance:

```
class my_iterator : public generic_random_access_iterator</* some arguments */>
{ // some definitions
};
```

- Problems
  - A part of the required interface references the final class:

```
my_iterator & operator+=( std::ptrdiff_t b) { /*...*/ return *this; }
```

    - How can we access my_iterator inside generic_random_access_iterator?
  - The required interface contains non-member functions:

```
my_iterator operator+( std::ptrdiff_t a, const my_iterator & b);
```

    - How can we implement this inside the mixin?
  - The requirements include a conversion between related iterators:
    - Either via conversion constructor in my_const_iterator – but constructors are not inherited

```
my_const_iterator(const my_iterator & b);
```

    - Or via conversion operator in my_iterator

```
operator my_const_iterator() const;
```

    - This is an additional method present in only one of the two iterator classes

Mixin

- Referencing the final class in a mixin
  - The mixin must have a parameter

```
template< typename final_class, /*...*/>
class generic_random_access_iterator {
  final_class & operator+=( std::ptrdiff_t b)
  { /*...*/
    return *static_cast<final_class*>(this);
  }
};
```

  - The mixin is used like this:

```
class my_iterator : public generic_random_access_iterator< my_iterator, /*...*/>
{ /*...*/ };
```

- This approach is ugly and dangerous:

```
class my_second_iterator : public generic_random_access_iterator< my_iterator, /*...*/>
{ /*...*/ };
```

Mixin

- Global functions as a mixin

```
template< typename final_class, /*...*/>
final_class operator+( std::ptrdiff_t a,
    const generic_random_access_iterator<final_class, /*...*/> & b)
{ /*...*/ }
```

  - This is an operator on the mixin class, not on the final_class
    - It could have unwanted effects

- The conversion operator
  - We need to know the other final class too

```
template< typename final_class, typename final_const_class, /*...*/>
class generic_random_access_iterator {
  operator final_const_class() const
  { /*...*/ }
};
```

  - But we don't want the conversion the other way – we need two mixin classes!

Mixins and policy classes

- Instead of writing this...

```
template< typename final_class, typename final_const_class, /*...*/>
class generic_random_access_iterator {
  operator final_const_class() const
  { /*...*/ }
};
```

- ...policy classes allow shorter template parameter lists...

```
template< typename policy>
class generic_random_access_iterator {
  operator typename policy::final_const_class() const { /*...*/ }
};
```

  - ...at the cost of declaring a policy class

```
class my_iterator; class my_const_iterator;
struct my_policy {
  using final_class = my_iterator;
  using final_const_iterator = my_const_iterator;
  /* ... */
};
class my_iterator : public generic_random_access_iterator< my_policy>
{ /*...*/ };
```

Mixins and policy classes

- With a policy class, things may be also implemented the other way round:

```
struct my_const_policy {
  using const_policy = my_const_policy;
  /* ... */
};
struct my_policy {
  using const_policy = my_const_policy;
  /* ... */
};
using my_iterator = generic_random_access_iterator< my_policy>;
using my_const_iterator = generic_random_access_iterator< my_const_policy>;
```

- This is probably the only approach which can really work in C++
  - It is limited to one final generic class, not really a mixin
  - It may correctly support non-member functions like:

```
template< typename P>
inline generic_random_access_iterator< P> operator+(
  std::ptrdiff_t a, generic_random_access_iterator< P> b);
```