Mixin and CRTP in C++98/11

Zoltán Porkoláb, PhD.
gsd@elte.hu
http://gsd.web.elte.hu

C++ specific patterns

- Mixin
  - Liskov substitutional principle
  - C++11 mixins
- Curiously Recurring Template Pattern (CRTP)
  - Operator generation
  - Counting
  - Polymorphic chain
  - Static polymorphism

Mixins

- Class inheriting from its own template parameter
- Not to mix with other mixins (e.g. Scala)
- Reversing the inheritance relationship:
  - One can define the Derived class before Base class
  - Policy/Strategy can be injected

```
template <class Base>
class Mixin : public Base  { ... };

class RealBase { ... };
Mixin<RealBase> rbm;

class Strategy { ... };
Mixin<Strategy> mws;
```

Liskov substitutional principle

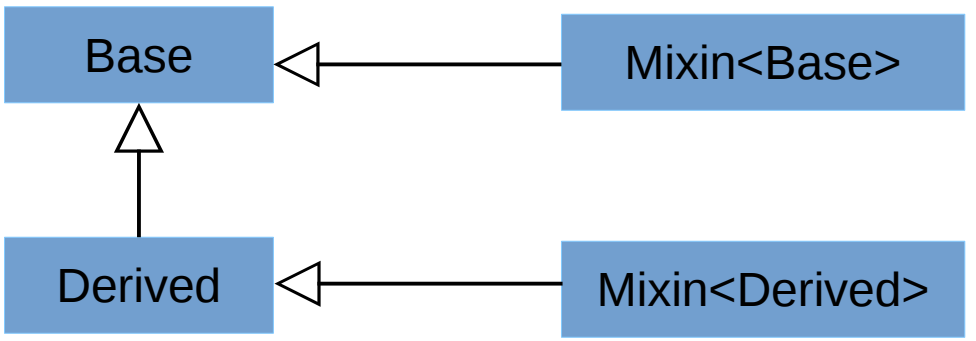
- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

```
class Base { ... };
class Derived : public base { ... };

template <class T> class Mixin : public T { ... };

Base         b;
Derived      d;

Mixin<Base>      mb;
Mixin<Derived>   md;

b  = d       // OK
mb = md;     // Error!
```

Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

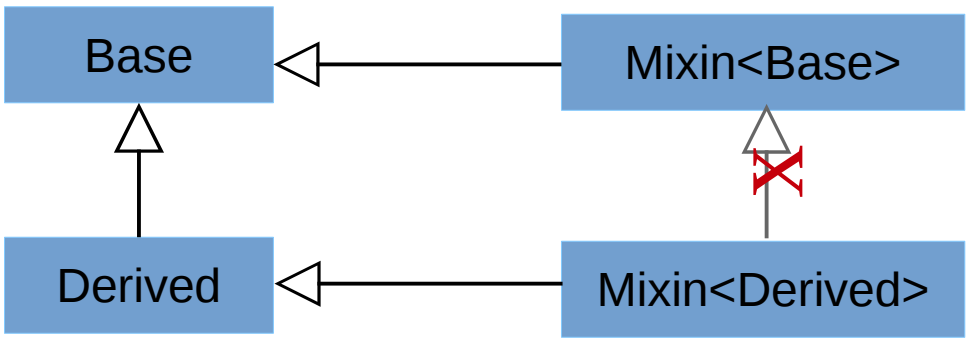
```
class Base { ... };
class Derived : public Base { ... };

template <class T> class Mixin : public T { ... };

Base         b;
Derived      d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d       // OK
mb = md;     // Error!
```



Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- Mixin<Derived> is not inherited from Mixin<Base>

```
class Base { ... };
class Derived : public Base { ... };

template <class T> class Mixin : public T { ... };

Base         b;
Derived      d;

Mixin<Base>     mb;
Mixin<Derived>  md;

b  = d       // OK
mb = md;     // Error!
```



Constraints

- No concept checking in C++ (yet)

```
class Sortable { void sort(); };

//  be careful with lazy instantiation
template <class Sortable>
class WontWork : public Sortable
{
public:
    void sort()
    {
        Sortable::srot();  // !!misspelled
    }
};
void client()
{
  WontWork<HasSortNotSrot> w;  // still compiles
  w.sort()   // syntax error only here!
}
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
T sum(T first, Args... args)            // function parameter pack
{
  return first + sum(args...);
}

int main()
{
  double lsum = sum(1, 2, 3.14, 8L, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
  return v;
}
template<typename T, typename... Args>  // template parameter pack
std::common_type<T,Args...>::type sum(T first, Args... args)
{
  return first + sum(args...);
}

int main()
{
  double lsum = sum(1, 2, 3.14, 8L, 7);

  std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
  std::string ssum = sum(s1, s2, s3, s4);
}
```

Mixin reloded (C++11)

- Variadic templates make us possible to define variadic set of base

```
struct A {};
struct B {};
struct C {};
struct D {};

template<class... Mixins>
class X : public Mixins...
{
public:
    X(const Mixins&... mixins) : Mixins(mixins)... { }
};

int main()
{
  A a; B b; C c; D d;

  X<A,B,C,D> xx(a,b,c,d);
}
```

Curiously Recurring Template Pattern (CRTP)

- James Coplien 1995
- F-bounded polymorphism 1980's
- Operator generation, Enable_shared_from_this, Static polymorphism

```
template <typename T>
struct Base
{
    // ...
};

struct Derived : Base<Derived>
{
    // ...
};
```

Operator generator

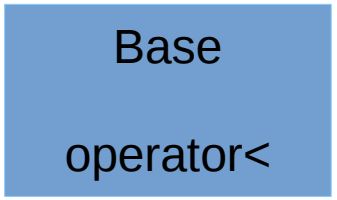
```
class C
{
  // ...
};

bool operator<(const C& l, const C& r)
{
  // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l==r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

Operator generator

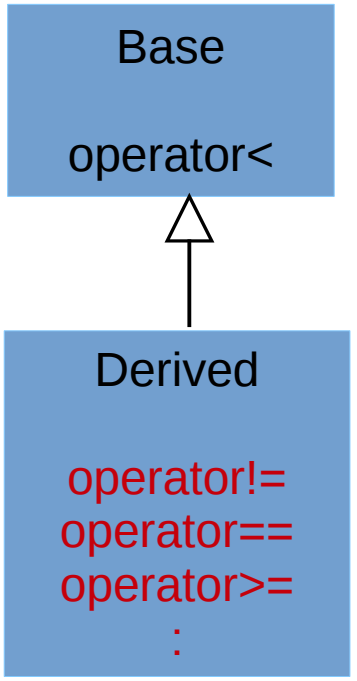
```
class C
{
  // ...
};

bool operator<(const C& l, const C& r)
{
  // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l==r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

- We want automatically generate the operators from operator<

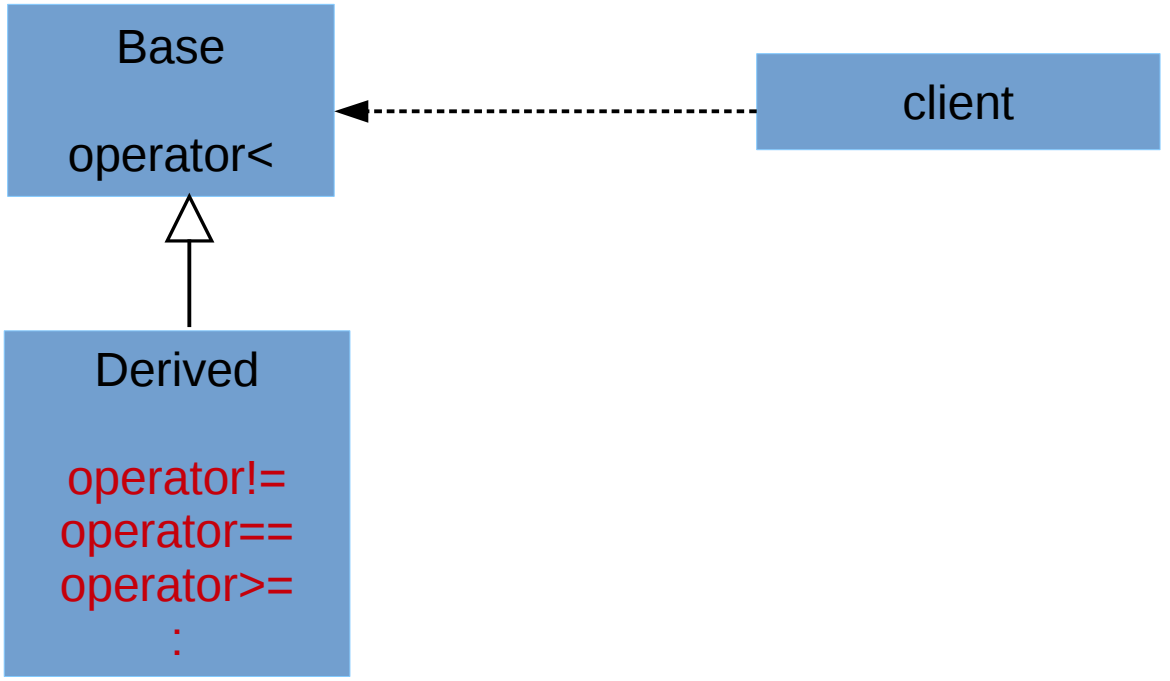
Operator generator in Derived



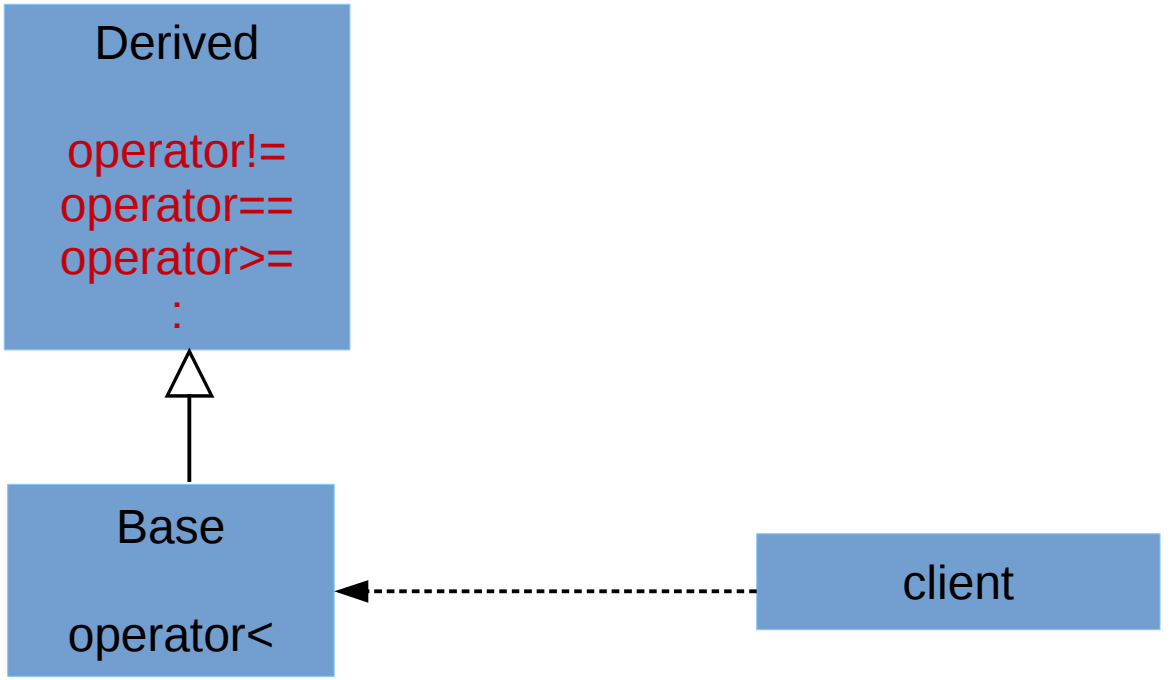
Operator generator in Derived



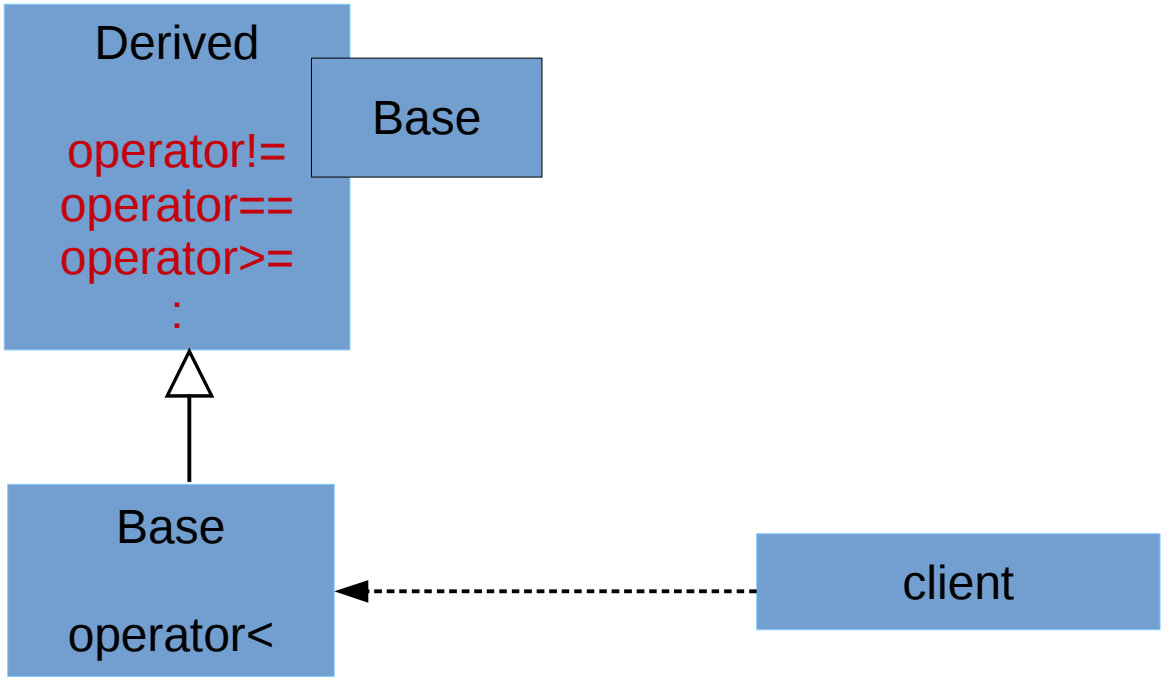
Operator generator in Derived



Operator generator in Base



Operator generator in Base



Enable shared from this

```
#include <memory>
#include <cassert>

class Y : public std::enable_shared_from_this<Y>
{
public:

    std::shared_ptr<Y> f()
    {
        return shared_from_this();
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p)); // p and q must share ownership
}
```

Object counter

```
template <typename T>
struct counter {
    static int objects_created;
    static int objects_alive;
    counter() {
        ++objects_created;
        ++objects_alive;
    }
    counter(const counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~counter() // objects should never be removed through pointers of this type
    {
        --objects_alive;
    }
};
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X> { ... };
class Y : counter<Y> { ... };
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};
```

```
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!


}
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};
class ColorPrinter : public Printer
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!


}
```

Polymorphic chaining

```
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  Printer& print(T&& t) { m_stream << t; return *this; }

  template <typename T>
  Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
  ostream& m_stream;
};
class ColorPrinter : public Printer
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!
  // compile error              v here we have Printer, not ColorPrinter
  ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

Polymorphic chaining

```
template <typename ConcretePrinter>
class Printer
{
public:
  Printer(ostream& pstream) : m_stream(pstream) {}

  template <typename T>
  ConcretePrinter& print(T&& t) { m_stream << t;
                          return static_cast<ConcretePrinter&>(*this); }
  template <typename T>
  ConcretePrinter& println(T&& t) { m_stream << t << endl;
                          return static_cast<ConcretePrinter&>(*this); }
 private:
  ostream& m_stream;
};
class ColorPrinter : public Printer<ColorPrinter>
{
public:
  ColorPrinter() : Printer(cout) {}
  ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};
void f()
{
  Printer{myStream}.println("hello").println(500);  // works!
  ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

Static polymorphism

- When we separate interface and implementation
- But no run-time variation between objects

```
template <class Derived>
struct Base
{
  void interface()
  {
    // ...
    static_cast<Derived*>(this)->implementation();
    // ...
  }
  static void static_funcion()
  {
    Derived::static_sub_funcion();
  }
};
struct Derived : Base<Derived>
{
  void implementation();
  static void static_sub_function();
};
```

Thank you!

gsd@elte.hu
http://gsd.web.elte.hu