TDDD38/726G82 - Advanced programming in C++

Template design patterns

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1. Static Polymorphism
2. Mixins
3. Traits
4. Algorithm Specialization

LINKÖPING UNIVERSITY

1 Static Polymorphism
2 Mixins
3 Traits
4 Algorithm Specialization

Static Polymorphism

Dynamic Polymorphism

```
class Shape
{
public:
  virtual ~Shape() = default;
  virtual void   draw() const = 0;
  virtual double area() const = 0;
};
```

```
class Circle : public Shape
{
public:
  void   draw() const override;
  double area() const override;
};

class Rectangle : public Shape
{
public:
  void   draw() const override;
  double area() const override;
};
```

Static Polymorphism

Dynamic Polymorphism

```
void draw(std::vector<Shape*> const& shapes)
{
  for (Shape* s : shapes)
    s->draw();
}

double area(std::vector<Shape*> const& shapes)
{
  double total{0.0};
  for (Shape* s : shapes)
    total += s->area();
  return total;
}
```

Static Polymorphism

Dynamic Polymorphism

```
int main()
{
  std::vector<Shape*> shapes {
    new Circle   {/* ... */},
    new Rectangle{/* ... */},
    // ...
  };
  draw(shapes);
  cout << "Total area: " << area(shapes)
       << endl;
}
```

Static Polymorphism

Dynamic Polymorphism

- Dynamic polymorphism is implemented with inheritance and virtual functions
- A very powerful tool for solving a lot of problems
- Unfortunately it comes with a very steep cost
- The virtual table leads to extra indirection
- Makes CPU cache sad :(

Static Polymorphism

Let's try to make it cheaper

```
class Circle
{
public:
  void   draw() const;
  double area() const;
};

class Rectangle
{
public:
  void   draw() const;
  double area() const;
};
```

Static Polymorphism

Using these classes

```
template <typename Shape>
void draw(std::vector<Shape> const& shapes)
{
  for (Shape const& s : shapes)
    s.draw();
}

template <typename Shape>
double area(std::vector<Shape> const& shapes)
{
  double total{0.0};
  for (Shape const& s : shapes)
    total += s.area();
  return total;
}
```

Static Polymorphism

Static Polymorphism

- Its hard to store a collection of arbitrary types in containers
- Therefore we must have a separate vector for each type
- (We could of course also use `std::variant`)

Static Polymorphism

But... What about the vectors?

```
int main()
{
  std::vector<Circle>    circles    { /* ... */ };
  std::vector<Rectangle> rectangles { /* ... */ };
  draw(circles);
  draw(rectangles);
  cout << "Total area: "
       << area(circles) + area(rectangles)
       << endl;
}
```

Static Polymorphism

Advantages and Disadvantages

- The polymorphism is done at compile-time rather than runtime
- Works exactly like normal classes and functions, compilers don’t need to treat this separately
- More easily optimized by the compiler
- Can’t store similar types as a common type

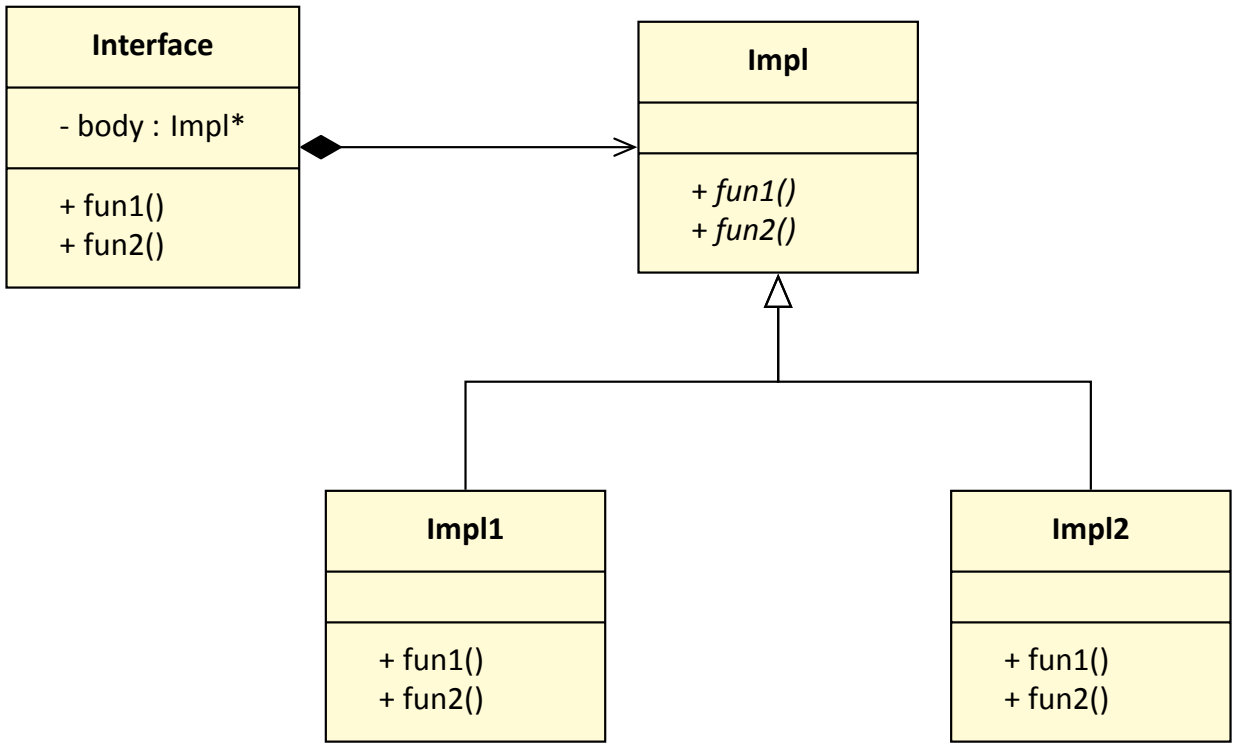
Static Polymorphism

Design Patterns

- Most design patters relies on polymorphism
- however, they are usually implemented with dynamic polymorphism
- but many of them can be implemented more efficiently with static polymorphism
- Let’s look at an example

Static Polymorphism

Bridge Pattern



Static Polymorphism

Bridge Pattern with Dynamic Polymorphism

```
class Interface
{
public:

  void fun1()
  {
    body->fun1();
  }

  void fun2()
  {
    body->fun2();
  }

private:

  Impl* body;
};
```

```
class Impl
{
public:
  virtual void fun1() = 0;
  virtual void fun2() = 0;
};
class Impl1 : public Impl
{
public:
  void fun1() override;
  void fun2() override;
};
class Impl2 : public Impl
{
public:
  void fun1() override;
  void fun2() override;
};
```

Static Polymorphism

Bridge Pattern with Static Polymorphism

```
template <typename Impl>
class Interface
{
public:
  void fun1()
  {
    body.fun1();
  }
  void fun2()
  {
    body.fun2();
  }
private:
  Impl body;
};
```

```
class Impl1
{
public:
  void fun1();
  void fun2();

};

class Impl2
{
public:
  void fun1();
  void fun2();

};
```

Static Polymorphism

Generic Programming

- When we are using design patterns with static polymorphism we lose the runtime aspect
- However we gain something new; we create abstractions that generalizes algorithms and data structures
- A good example is the STL, where most components can be customized with custom types through design patterns
- This concept is called *generic programming*

Static Polymorphism

Generic Programming

- Generic programming is used to allow the user to customize a general purpose algorithm or data structure for their own needs
- this way we can generalize our code without any runtime cost
- an example of a design pattern with generic programming is the *iterator*

Static Polymorphism

Generic Programming

```
template <typename It>
It max_element(It start, It last)
{
  It max{start};
  for (; start != last; ++start)
  {
    if (*start > *max)
      max = start;
  }
  return max;
}
```

Static Polymorphism

Generic Programming

- We as implementers of `max_element` don't need to know anything about the container
- all we need to care about is the interface of the iterator
- this will allow the user to do whatever they want inside the iterator as long as they supply the correct interface

Static Polymorphism

Policy Pattern

- One of the most general design patterns used in C++ is the *policy pattern*
- the policy patterns involves factoring out parts of the algorithm/data structure so that the user can customize certain behaviours
- the user declares how certain aspects of the algorithm should behave thus allowing for general implementations of common algorithmic patterns

Static Polymorphism

Policy Pattern

```
template <typename It, typename Policy>
auto sum(It start, It last, Policy p = {})
{
  auto result { p.initial(start, last) };
  for(; !p.done(start, last); p.next(start, last))
  {
    p.combine(result, *start);
  }
  return result;
}
```

Static Polymorphism

Policy Pattern

```
template <typename It>
class SumPolicy
{
public:
  using T = typename It::value_type;

  T initial() { return {}; }

  bool done(It start, It last)
  { return start == last; }

  void combine(T& result,
               T const& val)
  { result += val; }

  void next(It& start, It)
  { ++start; }
};
```

```
template <typename It>
class EveryOtherPolicy
{
public:
  using T = typename It::value_type;

  // ...

  void next(It& start, It last)
  {
    if (++start != last)
    {
      ++start;
    }
  }

  // ...
};
```

Static Polymorphism

An example in STL

```
template <typename T>
struct my_allocator
{
  using value_type = T;

  my_allocator() = default;
  T* allocate(std::size_t n)
  {
    int bytes{n*sizeof(T)};
    cout << "allocated " << n << " elements" << endl;
    return static_cast<T*>(std::malloc(bytes));
  }

  void deallocate(T* p, std::size_t n)
  {
    cout << "dellocated " << n << " elements" << endl;
    std::free(p);
  }
};
```

Static Polymorphism

An example in STL

```
int main()
{
  std::vector<int, my_allocator<int>> v{};
  for (int i{0}; i < 100; ++i)
    v.push_back(i);
}
```

Static Polymorphism

An example in STL

```
allocated 1 elements
allocated 2 elements
deallocated 1 elements
allocated 4 elements
deallocated 2 elements
allocated 8 elements
deallocated 4 elements
allocated 16 elements
deallocated 8 elements
allocated 32 elements
deallocated 16 elements
allocated 64 elements
deallocated 32 elements
allocated 128 elements
deallocated 64 elements
deallocated 128 elements
```

Static Polymorphism

STL + Policy = ♥

- STL uses the policy pattern quite extensively
- because of this it is a good idea to actually use the policy pattern for your own code so it is more compatible with the STL
- every algorithm in STL that takes a function is technically using the policy pattern

1 Static Polymorphism
2 Mixins
3 Traits
4 Algorithm Specialization

LINKÖPING UNIVERSITY

Mixins

Mixin classes

```
struct Point
{
  double x;
  double y;
};

class Polygon
{
private:
  vector<Point> points;
};
```

Mixins

Mixin classes

- `Point` is a pre-defined class that represents a point on the plane
- `Polygon` keeps a set of points
- This is however not as general as we would like
- The user have no control over what information is actually encoded in the `Point` class

Mixins

Mixin classes

```
template <typename P>
class Polygon
{
private:
  vector<P> points;
};

struct Color_Point : public Point
{ std::string color; };

struct Label_Point : public Point
{ std::string label; };
```

Mixins

Mixin classes

```
int main()
{
  Polygon<Point> polygon;
  Polygon<Color_Point> color_polygon;
  Polygon<Label_Point> color_polygon;
}
```

Mixins

Mixin classes

- Let the user decide which class to use inside the `Polygon` class by making it a class template
- This sometimes known as a *parametrized type*
- Then we can extend `Point` through inheritance
- This will ensure that the interface of the template parameter is correct

Mixins

Multiple Inheritance

- Of course, if we want a `Point` that both have a label and a color it would be bad practice to reimplement those features again
- in C++ we are allowed to inherit from multiple classes at once
- This works like normal inheritance
- So we can just create a `Label_Color_Point` to get a point with both features by inheriting from `Label_Point` and `Color_Point`

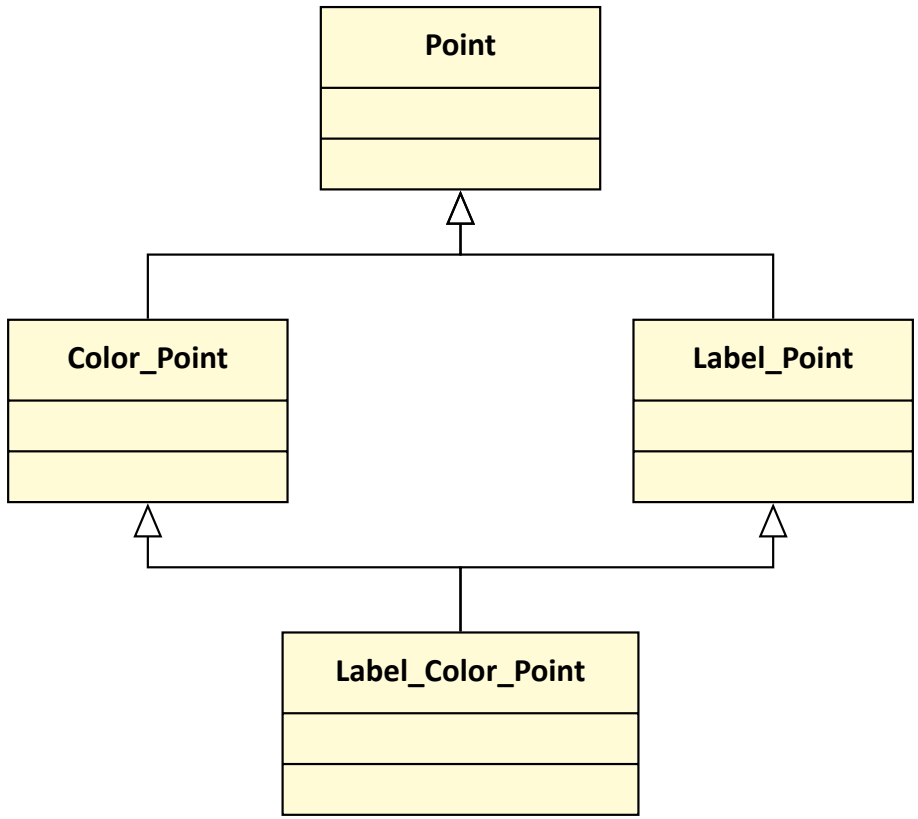
Mixins

Multiple Inheritance

```
struct Label_Color_Point
  : public Color_Point, public Label_Point
{ };
```

Mixins

Multiple Inheritance



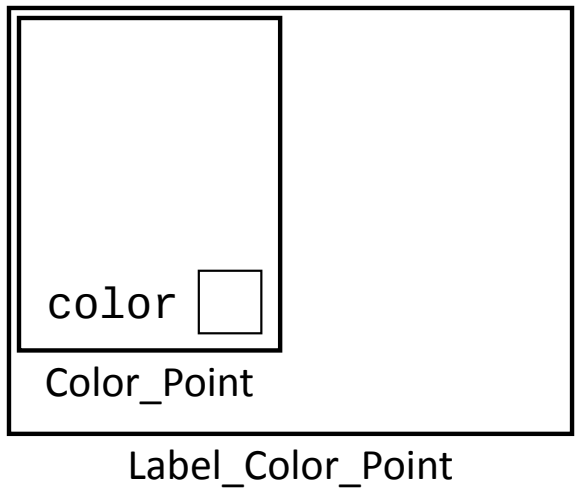
Mixins

Multiple Inheritance

Label_Color_Point

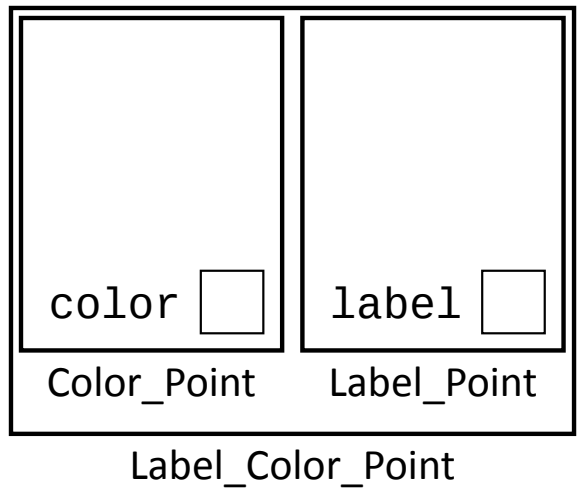
Mixins

Multiple Inheritance



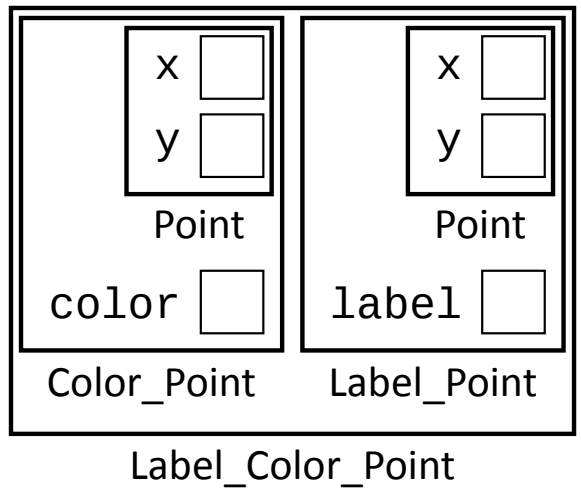
Mixins

Multiple Inheritance



Mixins

Multiple Inheritance



Mixins

Diamond Problem

```
Label_Color_Point p{};
p.x = 5; // which x ?!
```

Mixins

Diamond Problem

```
Label_Color_Point p{};
p.x = 5; // which x ?!
```

Ambiguous!

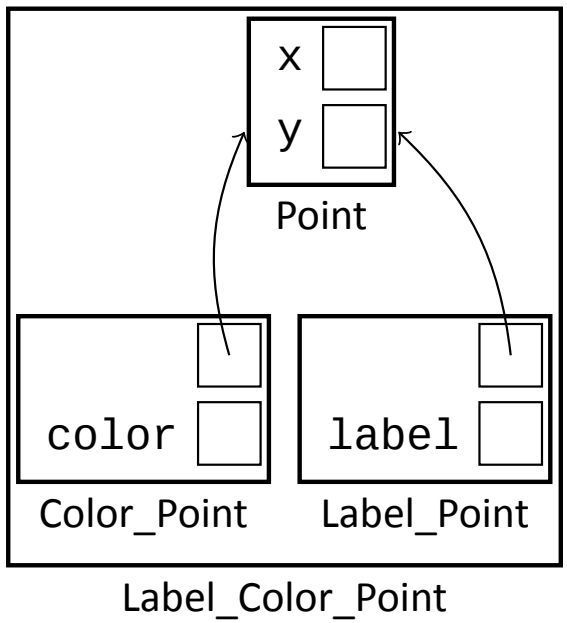
Mixins

A possible solution

```
struct Color_Point : virtual public Point
{ std::string color; };

struct Label_Point : virtual public Point
{ std::string label; };

struct Color_Label_Point
  : public Color_Point, public Label_Point
{ };
```

Mixins

Multiple Inheritance



Mixins

Problems with `virtual` inheritance

- This is not a nice solution
- The solution is not local to our class `Label_Color_Point`
- Instead we have to modify the base classes which breaks encapsulation
- It also introduces dynamic polymorphism which is slow
- The classes cannot assume that their base class is invariant, other classes can change
- This will make sure that the optimizer can’t do its job

Mixins

A better solution

- A better way to make this work is to not have shared base classes
- Unfortunately this defeats our original purpose since we inherited from `Point` to get the correct interface
- This is where *Mixins* come in
- We can see mixins as inheritance upside down

Mixins

Mixin classes

```
template <typename... Mixins>
struct Point : public Mixins...
{
  Point() : Mixins()..., x{}, y{} { }
  double x;
  double y;
};
```

Mixins

Mixin classes

```
struct Color
{ std::string color; };

struct Label
{ std::string label; };

int main()
{
  using My_Point = Point<Label, Color>;
  Polygon<My_Point> polygon;
}
```

Mixins

Mixin classes

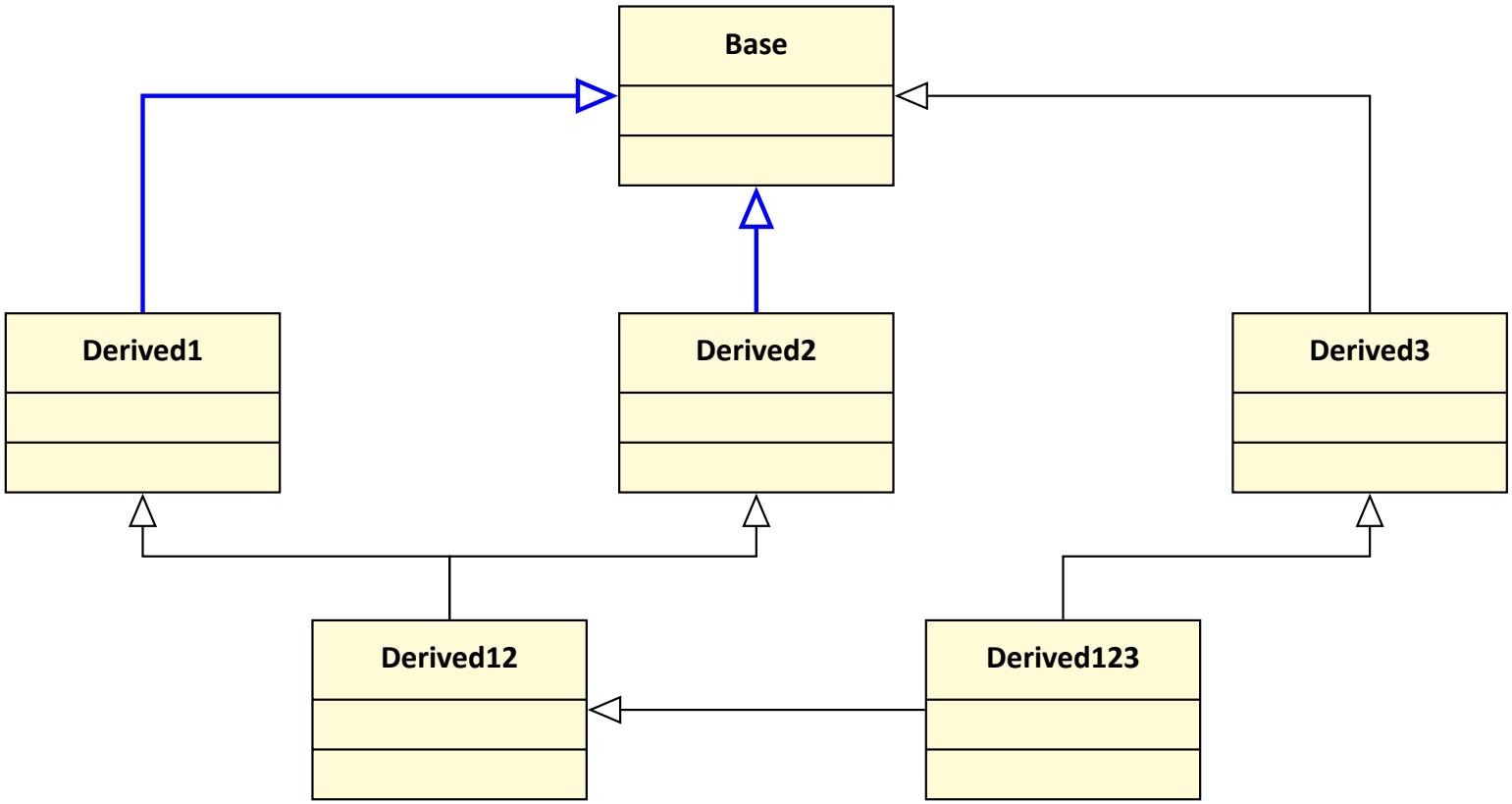
- Here the user don’t need to know how `Point` work
- Instead the user can create mixin classes and hand them to `Point`, thus forcing it to inherit from the users code
- That way, each mixin class can be implemented entierly on its own
- We are *mixing in* our own base classes into the `Point` class to customize the interface and data of `Point`

Static Polymorphism

What will be printed?

```
struct Base
{
    Base() { cout << "B"; }
};

struct Derived1 : virtual Base { };

struct Derived2 : virtual Base { };

struct Derived3 : Base { };

struct Derived12 : Derived1, Derived2 { };

struct Derived123 : Derived12, Derived3 { };

int main()
{
    Derived123 d123 {};
}
```

Static Polymorphism



Mixins

What will be printed?

- Since `Derived1` and `Derived2` inherit `Base` `virtual` this means that `Derived12` will only contain one instance of `Base`.
- `Derived123` inherits from `Derived12` and `Derived3`.
- `Derived3` inherits from Base, but not `virtual` so `Derived3` will always contain one instance of `Base`.
- This gives us a total of 2 `Base` instances in `Derived123`.
- Therefore it will print BB.

Traits

Consider the following

```
template <typename T1, typename T2>
auto& max(T1 const& a, T2 const& b)
{
  if (a > b)
  {
    return a;
  }
  return b;
}
```

```
int main()
{
  // will print 2
  cout << max(1,2) << endl;
  // will print 2.5
  cout << max(1.1, 2.5) << endl;
  // will not compile :(
  cout << max(1, 1.9) << endl;
}
```

Traits

Possible fix

```
#include <type_traits>
template <typename T1, typename T2>
std::common_type_t<T1, T2>& max(T1 const& a,
                                T2 const& b)
{
  if (a > b)
  {
    return a;
  }
  return b;
}
```

Traits

Possible fix

- `std::common_type<T1, T2>::value` is an alias for the smallest type that both `T1` and `T2` can be represented as
- is available in `<type_traits>`
- `std::common_type_t<T1, T2>` is an alias for `std::common_type<T1, T2>::value`

Traits

common_type

```
template <typename T1, typename T2>
struct common_type;

// specializations
template <typename T>
struct common_type<T, T>
{ using value = T; };

template <>
struct common_type<int, double>
{ using value = double; };

// etc...
```

Traits

Traits

- `std::common_type` is a so called *Trait class*
- Trait classes are types that defines specific properties about type
- It does *not* specify behaviour, that is the job of policy classes
- traits classes are used to signal specific properties of a type inside a generic context

Traits

Query Traits

```
template <typename T,
          typename Container>
void add(Container& c, T const& t)
{
  c.push_back(t);
}

template <typename T,
          typename Container>
void add(Container& c, T&& t)
{
  c.push_back(std::move(t));
}
```

```
int main()
{
  int const y{};
  int x{};
  std::vector<int> v{};

  // will call second version
  add(v, 5);
  // will call second version (???)
  add(v, x);
  // will call first version
  add(v, y);
}
```

Traits

Query Traits

- The `add` function works
- however; it will move every value except those that are `const`
- that is not what we intended...
- forwarding references work against us here

Traits

A solution

```
template <typename T, typename Container>
auto add(Container& c, T&& t)
  -> std::enable_if_t<
       std::is_rvalue_reference_v<decltype(t)>>
{
  c.push_back(std::move(t));
}
```

Traits

`std::is_rvalue_reference`

- `std::is_rvalue_reference<T>::value` is `true` if `T` is an rvalue reference, and `false` otherwise
- `std::is_rvalue_reference_v<T>` is an alias for `std::is_rvalue_reference<T>::value`
- this is a *query trait class* which checks a condition on the specified type
- *query trait classes* can also return specific values, it doesn't have to be a `bool`

Traits

`std::is_rvalue_reference`

```
template <typename T>
struct is_rvalue_reference
{
  static constexpr bool value {false};
};
template <typename T>
struct is_rvalue_reference<T&&>
{
  static constexpr bool value {true};
};
template <typename T>
constexpr bool is_rvalue_reference_v{
  is_rvalue_reference<T>::value};
```

Static Polymorphism

STL + Traits = ♥

- As you might have guessed; traits is another common pattern in STL
- it is very useful for compile-time constructs that depend on types
- fixes many problems related to templates
- A tip is to look at the `<type_traits>` header

Traits

`iterator_traits`

- Recall that there are different categories of iterators
- in a lot of cases we actually would like to know which category our current iterator belongs to
- in the same vein it would be nice to have a way to access the inner type of the iterator (i.e. the type returned from `*it`)
- due to this, each iterator must define type aliases in their public interface that represent various traits about the iterator

Traits

iterator_traits

```
template <typename T>
class std::vector<T>::iterator
{
// ...
public:
  // type returned from (it2 - it1)
  using difference_type = std::ptrdiff_t;
  // type returned from *it
  using value_type = T;
  using pointer = T*;
  using reference = T&;
  // which category this iterator belong to
  using iterator_category = std::random_access_iterator_tag;
// ...
};
```

Traits

`iterator_traits`

- Since these traits are defined inside the iterator we can use them to retrieve crucial information
- however, recall that pointers also are iterators
- but pointers cannot have inner aliases
- due to this we must take special action when wanting to retrieve traits related to iterators (in case they are pointers)

Traits

`iterator_traits`

```
template <typename It>
typename It::value_type get(It it)
{
  return *it;
}
int main()
{
  vector<int> v {1, 2, 3};
  int a[] {1, 2, 3};
  cout << get(begin(v)) << endl; // OK
  cout << get(begin(a)) << endl; // not OK
}
```

Traits

iterator_traits

```
template <typename It>
typename std::iterator_traits<It>::value_type
get(It it)
{
  return *it;
}
int main()
{
  vector<int> v {1, 2, 3};
  int a[] {1, 2, 3};
  cout << get(begin(v)) << endl; // OK
  cout << get(begin(a)) << endl; // OK
}
```

Traits

`iterator_traits`

- Because of pointers there is a general way to retrieve relevant information about an iterator even if it is of non-class type
- the trait pattern is perfect for this
- there is a trait class known as `std::iterator_traits` in STL
- it only contains type aliases, nothing else

1 Static Polymorphism
2 Mixins
3 Traits
4 Algorithm Specialization

Algorithm Specialization

Getting a value from a container

```
template <typename It>
auto at(It start, It end, size_t index)
{
  if (start + index < end)
    return *(start + index);
  throw /* ... */;
}
```

Algorithm Specialization

Getting a value from a container

```
int main()
{
  std::vector<int> v{5,8,11};
  std::map<string, int> m { {"a", 1} };
  // print 11
  cout << at(begin(v), end(v), 2) << endl;
  // doesn't compile
  cout << at(begin(begin(m), end(m), 0).first
       << endl;
}
```

Algorithm Specialization

Why doesn't it compile?

- the `at` function assumes that `start + index` works
- however that expression is only valid if `start` is a *RandomAccessIterator*
- but `std::map` only have *BidirectionIterator*s
- so we would want to write more general code

Algorithm Specialization

More general version

```
template <typename It>
auto at(It start, It end, size_t index)
{
  while (index-- > 0 && start != end)
    ++start;
  if (start != end)
    return *start;
  throw /* ... */;
}
```

Algorithm Specialization

A new problem

- This version works for all input iterators
- but... it is needlessly slow for random access iterators
- optimally we would want this function to have two overloads, one for random access and one for everything else
- let us try to solve this!

Algorithm Specialization

Tag dispatching!

```
template <typename It>
auto at_helper(It start, It end, size_t index,
               std::random_access_iterator_tag)
{
  if (start + index < end)
    return *(start + index);
  throw /* ... */;
}

template <typename It>
auto at_helper(It start, It end, size_t index,
               std::input_iterator_tag)
{
  while (index-- > 0 && start != end)
    ++start;
  if (start != end)
    return *start;
  throw /* ... */;
}
```

Algorithm Specialization

Tag dispatching!

```
template <typename It>
auto at(It start, It end, size_t index)
{
  using traits = std::iterator_traits<it>;
  return at_helper(start, end, index,
            typename traits::iterator_category{});
}
```

Algorithm Specialization

A different approach to tag dispatching

```
template <typename It>
constexpr bool is_random_access_iterator {
  std::is_convertible_v<
    typename std::iterator_traits<It>::iterator_category,
    std::random_access_iterator_tag>
};

template <typename It,
          typename = std::enable_if_t<
            is_random_access_iterator<It>
          >>
auto at(It start, It end, size_t index);
```

Algorithm Specialization

A different approach to tag dispatching

```
template <typename It>
constexpr bool is_input_iterator {
  std::is_convertible_v<
    typename std::iterator_traits<It>::iterator_category,
    std::input_iterator_tag>
};

template <typename It,
          typename = std::enable_if_t<
            is_input_iterator<It> && !is_random_access_iterator<It>
          >>
auto at(It start, It end, size_t index);
```

Algorithm Specialization

A different approach to tag dispatching

- Quite horrible syntax
- However `std::enable_if` allows us to check the tag, and other conditions as well (if we so please)
- it is important to make sure that all the conditions are mutually exclusive
- otherwise we get ambigious calls (in this case, *RandomAccessIterator* is a subset of *InputIterator*)

Algorithm Specialization

Enter C++17 and `constexpr if`

```
template <typename It>
auto at(It start, It end, size_t index)
{
  if constexpr (is_random_access_iterator<It>)
  {
    if (start + index < end)
      return *(start + index);
  }
  else if constexpr (is_input_iterator<It>)
  {
    while (index-- > 0 && start != end)
      ++start;
    if (start != end)
      return *start;
  }
  throw /* ... */;
}
```

Algorithm Specialization

`constexpr if`

- Like a normal `if`-statement
- but it is evaluated during compile-time
- every path not taken is discarded
- the condition must be evaluated during compile-time

www.liu.se

LINKÖPING UNIVERSITY