Intensive C++

Implementing Binary Operators

Angelika Langer
Trainer/Consultant

http://www.AngelikaLanger.com

Objective

- Learn about the challenges of implementing binary operators.
  - It's simple for a single class and quite a challenge for a hierarchy value types.

Which type of classes ?

We will consider a class hierarchy of classes with the following properties:

- value semantics
  - a common case in C++
  - object "owns" its data members
- composition by inheritance
  - as opposed to "by delegation"
  - base class is a concrete (non-abstract) class
  - derived classes add data members

Which type of functions ?

We will consider functions with the following properties:

- redefined in the class hierarchy
  - implemented in the base class
  - redefined by every derived class
- self-referential binary functions
  - work on two objects of the same type
  - examples: copying, comparing, ...

Agenda

- Assignment Operator
- Comparison Operator

A Class Hierachy

```
class Point2D
{
 public: ...
 Point2D& operator=(const Point2D& rhs);        1
};
class Point3D : public Point2D
{
 public: ..
 Point3D& operator=(const Point3D& rhs);        2
};
class ColoredPoint : public Point2D
{
 public: ...
 ColoredPoint& operator=(const ColoredPoint& 3 s);
};
```

Problem

- Invocation through references leads to mixed-type assignment and object slicing.

```
void someFunction(Point2D& lhs, Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes `Point2D::operator=()`

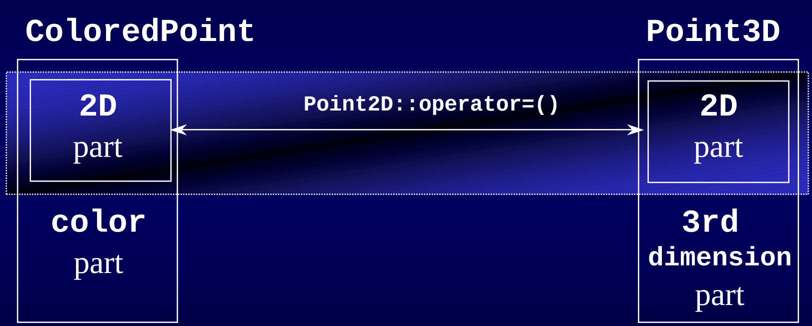
```
ColoredPoint red, blue;
someFunction(red,blue);
```

assigns 2D part of colored points ?

```
ColoredPoint red;
Point3D origin;
someFunction(red, origin);
```

assigns 2D part of colored and 3D point ?

Object Slicing



Other cases …

- Invocation through references to base and derived type:

```
void someFunction(Point2D& center, ColoredPoint& loc)
{ ...
  center = loc;
  loc = center;
  ...
}
```

invokes `Point2D::operator=()`

does not compile

Other cases …

- Invocation through references to derived type:

```
void someFunction(Point3D& rhs, Point3D& lhs)
{ ...
  rhs = lhs;
  ...
}
```

invokes `Point3D::operator=()`

Non-Virtual Assignment

rhs static type	rhs dynamic type \ lhs	Point2D& Point2D	Point2D& Point3D	Point2D& Colored Point	Point3D& Point3D
Point2D& (1)	Point2D	OK	slice	slice	slice
	Point3D	slice	slice	slice	slice
	ColoredPoint	slice	slice	slice	slice
Point3D & (2)	Point3D	-	-	-	OK

What's the problem ... ?

- no pass-by-value, yet object slicing - what's wrong ... ?

```
class Point2D
{
 public: ...
  Point2D& operator=(const Point2D& rhs);
};
class Point3D : public Point2D
{
 public: ..
  Point3D& operator=(const Point3D& rhs);
};
class ColoredPoint : public Point2D
{
 public: ...
  ColoredPoint& operator=(const ColoredPoint& rhs);
};
```

non-virtual functions

Another Class Hierachy

```
class Point2D
{
 public: ...
  virtual Point2D& operator=(const Point2D& rhs);     A
};
class Point3D : public Point2D
{
 public: ..
  virtual Point3D& operator=(const Point2D& rhs);     B
};
class ColoredPoint : public Point2D
{
 public: ...
  virtual ColoredPoint& operator=(const Point2D& rhs  C
};
```


Problem

- Invocation through references leads to mixed-type assignment and potential crashes.

```
void someFunction(Point2D& lhs, Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes virtual function

```
Point3D red, blue;
someFunction(red,blue);
```

invokes `Point3D::operator=()`

```
Point3D origin;
ColoredPoint red;
someFunction(origin,red);
```

invokes `Point3D::operator=()`
assigns Point3D part of ColoredPoint ???

Problem

- We still get slices ...

```
void someFunction(Point2D& lhs, Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes virtual function

```
Point2D origin2D;
Point3D origin3D;

someFunction(origin2D,origin3D);

someFunction(origin3D, origin2D);
```

invokes `Point2D::operator=()` assigns Point2D part of Point3D ?

invokes `Point3D::operator=()` might crash

Virtual Assignment

rhs static type	rhs dynamic type \ lhs static type / dynamic type	Point2D& – Point2D	Point2D& – Point3D	Point2D& – Colored Point	Point3D& – Point3D
Point2D&	Point2D (A)	OK	slice	slice	slice
Point2D&	Point3D (B)	crash	OK	crash	OK
Point2D&	ColoredPoint (C)	crash	crash	OK	crash
Point3D&	Point3D (B)	crash	OK	crash	OK

Crash

- Why would certain invocations lead to a crash?
- Behavior depends on the implementation:
  - assignment takes base class references
  - must do a type check somehow

```
class Point3D : public Point2D
{
 public: ...
  virtual Point3D& operator=(const Point2D& rhs)
  {
    ... is Point2D a Point3D ? ...
  }
};
```


Crash

- Worst case implementation:
  - blind downcast => program crash
- Friendly implementation
  - uses RTTI (dynamic cast or typeid)
  - if type check indicates
    - same type
      - perform assignment
    - alien type
      - thrown an exception ?
      - perform slice comparison ?

Dynamic Cast

- What does a type check via dynamic cast (as opposed to a check via typeid) mean? Is it correct?

```
class Point3D : public Point2D
{
 public: ...
  virtual Point3D& operator=(const Point2D& rhs)
  { ...
    Point3D& tmp = dynamic_cast<Point3D&>(rhs);
    // throws bad_cast exception in case of failure
    ...
  }
};
... same for Point2D and ColoredPoint ...
```


Virtual Assignment

rhs static type	rhs dynamic type \ lhs static type / dynamic type	Point2D& / Point2D	Point2D& / Point3D	Point2D& / Colored Point	Point3D& / Point3D
Point2D&	Point2D (A)	OK	slice	slice	slice
Point2D&	Point3D (B)	exc	OK	exc	OK
Point2D&	ColoredPoint (C)	exc	exc	OK	exc
Point3D&	Point3D (B)	exc	OK	exc	OK

Typeid

- Check for type match and allow assignment only for objects of the same type.

```
class Point2D
{
 public: ...
  virtual Point2D& operator=(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point2D part ...
  }
};
```


Type Check

```
class Point3D : public Point2D
{
 public: ...
  virtual Point3D& operator=(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point3D part ...
  }
};
... same for ColoredPoint ...
```

Invocation

- Type check leads to runtime failure in form of an exception.

```
void someFunction(Point2D& lhs, Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes virtual function

```
Point3D red, blue;
someFunction(red,blue);
```

invokes `Point3D::operator=()`

```
Point3D origin;
ColoredPoint red;
someFunction(origin,red);
```

invokes `Point3D::operator=()` throws exception

Virtual Assignment

rhs static type	rhs dynamic type \ lhs static type / dynamic type	Point2D& Point2D	Point2D& Point3D	Point2D& Colored Point	Point3D& Point3D
Point2D&	Point2D (A)	OK	exc	exc	exc
	Point3D (B)	exc	OK	exc	OK
	ColoredPoint (C)	exc	exc	OK	exc
Point3D&	Point3D (B)	exc	OK	exc	OK

Evaluation

- What have we achieved ?
- Symmetric behavior
  - A can be assigned to B if and only if B can be assigned to A
- All combinations compile
  - same-type comparison works
  - mixed-type comparison fails (at runtime with an exception)

What if ...

- ... we wanted to permit mixed-type assignment?
- Mixed-type assignment need not be rejected per se.
  - all Point2Ds have something in common
  - assignment of incompatible Point2Ds could be interpreted as assignment of common part
- Goal:
  - no slicing for same-type assignment
  - symmetric slice comparison for mixed-type assignment

Goal

rhs static type	rhs dynamic type	lhs static type: Point2D&, dynamic type: Point2D	Point2D&: Point3D	Point2D&: ColoredPoint	Point3D&: Point3D
Point2D&	Point2D (A)	OK	slice	slice	slice
Point2D&	Point3D (B)	slice	OK	slice	OK
Point2D&	ColoredPoint (C)	slice	slice	OK	slice
Point3D&	Point3D (B)	slice	OK	slice	OK

How can it be implemented ... ?

Two choices:

- table solution
  - key: typeid of right- and left-hand side
  - value: function pointer to assignment functionality
- double dispatch
  - uses virtual function table dispatch

Dispatch Table

```
class Point2D
{
 private:
  class DispatchTable
  {public:
    typedef Point2D&(*fptrType)(Point2D&, const Point2D&);
    DispatchTable();
    fptrType getFunction(const type_info& lhs,
                         const type_info& rhs);
   private:
    map<typeidPair,fptrType> tab;
  };
  static DispatchTable dispatchTable;
};
```


Dispatch Table

```
DispatchTable()
{
    const type_info& typPoint2D      = typeid(Point2D);
    const type_info& typPoint3D      = typeid(Point3D);
    const type_info& typColoredPoint  = typeid(ColoredPoint);
    tab[typeidPair(typPoint2D     ,typPoint2D)]      = &Point2D::assign;
    tab[typeidPair(typPoint2D     ,typPoint3D)]      = &Point2D::assign;
    tab[typeidPair(typPoint3D     ,typPoint2D)]      = &Point2D::assign;
    tab[typeidPair(typPoint3D     ,typPoint3D)]      = &Point3D::assign;
    tab[typeidPair(typPoint2D     ,typColoredPoint)] = &Point2D::assign;
    tab[typeidPair(typColoredPoint,typPoint2D)]      = &Point2D::assign;
    tab[typeidPair(typColoredPoint,typColoredPoint)] = &ColoredPoint::assign;
    tab[typeidPair(typPoint3D     ,typColoredPoint)] = &Point2D::assign;
    tab[typeidPair(typColoredPoint,typPoint3D)]      = &Point2D::assign;
}
fptrType getFunction(const type_info& lhs,const type_info& rhs)
{
    return tab[typeidPair(lhs, rhs)];
}
```

Point2D

```
class Point2D
{public:
  Point2D& operator=(const Point2D& rhs)
  { DispatchTable::fptrType fptr
    = dispatchTable.getFunction(typeid(*this), typeid(rhs));
    return fptr(*this,rhs);
  }
 private:
  static Point2D& assign(Point2D& lhs, const Point2D& rhs)
  {  ... perform Point2D assignment ...
     return *lhs;
  }
};
```


Point3D

```
class Point3D : public Point2D
{public:
  // operator=(const Point2D& rhs) inherited from class Point2D



 private:
  static Point2D& assign(Point2D& lhs, const Point2D& rhs)
  {  ... perform Point3D assignment ...
     return lhs;
  }
};
```

Type Info Pair

- `type_info` objects cannot be copied
  - must be passed by reference
- `pair` does not permit reference members
  - must wrap `type_info` objects into a wrapper type
- `type_info` does not have an `operator<` defined
  - must use `type_info.before()`

```
class typeidPair
{public:
    const type_info& first;
    const type_info& second;
    typeidPair(const type_info& a1, const type_info& a2)
      : first(a1), second(a2) {}
};
inline bool operator<(const typeidPair& x, const typeidPair& y)
{   return x.first.before(y.first) ||
         (!(y.first.before(x.first)) && x.second.before(y.second));
}
```


Double Dispatch

- Double Dispatch uses the *vtable* as the dispatch table.
  - *vtable* dispatch uses left-hand side's type
- Idea: dispatch twice
  - dispatch according to left-hand side's type
  - switch roles of left- and right-hand side
  - dispatch again (according to right-hand side's type)

Double Dispatch

```
class Point2D
{public:
  Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
 private:
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point2D&)*this); }
  virtual Point2D& assignHelper(const Point2D& rhs) const
  {  return ((Point2D&)rhs).assignPoint2D(*this);  }
  virtual Point2D& assignHelper(const Point3D& rhs) const
  {  return ((Point2D&)rhs).assignPoint2D(*this);  }
  virtual Point2D& assignHelper(const ColoredPoint& rhs) const
  {  return ((Point2D&)rhs).assignPoint2D(*this);  }
  Point2D& assignPoint2D(const Point2D& rhs)
  { ... perform Point2D assignment ...
    return *this;
  }
};
```


Double Dispatch

```
class Point3D : public Point2D
{public:
  // Point2D& operator=(const Point2D&) inherited from base class
 private:
  virtual Point2D& assign(const Point2D& rhs)
  {  return rhs.assignHelper((Point3D&)*this);  }
  virtual Point2D& assignHelper(const Point2D& rhs) const
  {  return ((Point2D&)rhs).assignPoint2D(*this); }
  virtual Point3D & assignHelper(const Point3D & rhs) const
  {  return ((Point3D &)rhs).assignPoint3D (*this);  }
  virtual Point2D& assignHelper(const ColoredPoint& rhs) const
  {  return ((Point2D&)rhs).assignPoint2D(*this);  }
  Point3D& assignPoint3D(const Point3D& rhs)
  { Point2D::assignPoint2D(rhs);
    ... perform Point3D assignment ...
    return *this;
  }
};
```

Double Dispatch

```
Point2D& a1 = Point2D();
Point2D& a2 = Point2D();
a1 = a2;
```

```
class Point2D
{ Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point2D&)*this); }
  virtual Point2D& assignHelper(const Point2D& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
  virtual Point2D& assignHelper(const Point3D& rhs);
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point2D& assignPoint2D(const Point2D& rhs);
};
```


Double Dispatch

```
Point2D& a1 = Point2D(); Point2D& a2 = Point3D();  a1 = a2;
```

```
class Point2D
{ Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point2D&)*this); }
  virtual Point2D& assignHelper(const Point2D& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
  virtual Point2D& assignHelper(const Point3D& rhs);
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point2D& assignPoint2D(const Point2D& rhs);
};
class Point3D : public Point2D
{
  virtual Point2D& assign(const Point2D& rhs);
  virtual Point2D& assignHelper(const Point2D& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
  virtual Point3D & assignHelper(const Point3D & rhs);
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point3D& assignPoint3D(const Point3D& rhs);
};
```

Double Dispatch

```
Point2D& a1 = Point3D(); Point2D& a2 = Point3D();  a1 = a2;
```

```
class Point2D
{ Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
  virtual Point2D& assign(const Point2D& rhs);
  virtual Point2D& assignHelper(const Point2D& rhs);
  virtual Point2D& assignHelper(const Point3D& rhs);
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point2D& assignPoint2D(const Point2D& rhs);
};
class Point3D : public Point2D
{
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point3D&)*this); }
  virtual Point2D& assignHelper(const Point2D& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
  virtual Point3D & assignHelper(const Point3D& rhs)
  { return ((Point3D &)rhs).assignPoint3D (*this); }
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point3D& assignPoint3D(const Point3D& rhs);
};
```


Double Dispatch

```
Point3D& m1 = Point3D();
Point3D& m2 = Point3D();
m1 = m2;
```

```
class Point3D : public Point2D
{ // inherited operator from base class
  Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point3D&)*this);  }
  virtual Point2D& assignHelper(const Point2D& rhs);
  virtual Point3D & assignHelper(const Point3D & rhs)
  { return ((Point3D &)rhs).assignPoint3D (*this); }
  virtual Point2D& assignHelper(const ColoredPoint& rhs);
  Point3D& assignPoint3D(const Point3D& rhs);
};
```

Double Dispatch

```
Point2D& a1 = Point3D(); Point2D& a2 = ColoredPoint();  a1 = a2;
```

1

3

2

```
class Point2D
{ Point2D& operator=(const Point2D& rhs)
  { return assign(rhs); }
  ...
  Point2D& assignPoint2D(const Point2D& rhs);
};
class Point3D : public Point2D
{
  virtual Point2D& assign(const Point2D& rhs)
  { return rhs.assignHelper((Point3D&)*this); }
  ...
  virtual Point2D& assignHelper(const ColoredPoint& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
};
class ColoredPoint : public Point2D
{ Point2D& operator=(const Point2D& rhs);
  ...
  virtual Point2D& assignHelper(const Point3D& rhs)
  { return ((Point2D&)rhs).assignPoint2D(*this); }
};
```


Invocation

- Invocation through references leads to mixed-type assignment and (intended) object slicing.

```
void someFunction(Point2D& lhs, Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

triggers double dispatch

```
Point3D red, blue;
someFunction(red,blue);
```

invokes `Point3D::assignPoint3D()`

```
Point3D origin;
ColoredPoint red;
someFunction(origin,red);
```

invokes `Point2D::assignPoint2D()`

Assignment With (Double / Table) Dispatch

rhs static type	rhs dynamic type \ lhs	Point2D& Point2D	Point2D& Point3D	Point2D& Colored Point	Point3D& Point3D
Point2D&	Point2D A	OK	slice	slice	slice
Point2D&	Point3D B	slice	OK	slice	OK
Point2D&	ColoredPoint C	slice	slice	OK	slice
Point3D&	Point3D B	slice	OK	slice	OK

Evaluation

- Double Dispatch is the classic solution.
  - does not need RTTI
  - less maintainable
    - because dispatch logic is spread over all classes in the hierarchy
- Dispatch Table is more maintenance-friendly.
  - one central point
    - that must be modified when hierarchy grows

Assignment Via Values

- Whole discussion only concerns invocation of assignment operator through references.

```
void someFunction(Point2D lhs, Point2D rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes
`Point2D::operator=()`

Invokes base class assignment even if assignment operator is virtual.

Virtual vs. Synthetic Assignment

- Definition of virtual assignment does not prevent generation of synthetic assignment.

```
class Point2D
{
 public: ...
  virtual Point2D& operator=(const Point2D& rhs);
};
class Point3D : public Point2D
{
 public: ...
  virtual Point3D& operator=(const Point2D& rhs);
};
class ColoredPoint : public Point2D
{
 public: ...
  virtual ColoredPoint& operator=(const Point2D& rhs);
};
```

plus synthetic
`operator=(const Point3D&)`

plus synthetic
`operator=(const ColoredPoint&)`

Consistency

- The synthetic assignment should be consistent with the virtual assignment.

```
void someFunction(Point3D lhs,Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes virtual
`Point3D::operator=(const Point2D& rhs)`

should have the same effect

```
void someFunction(Point3D lhs,Point3D rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes synthetic
`Point3D::operator=(const Point3D& rhs)`

Ensuring Consistency (i)

- Explicitly define the "synthetic" assignment.
  - implement by delegation to actual assignment

```
class Point2D
{public: ...
  virtual Point2D& operator=(const Point2D& rhs);
};
class Point3D : public Point2D
{public: ...
  virtual Point3D& operator=(const Point2D& rhs);

  Point3D& operator=(const Point3D& rhs)
  { return operator=(static_cast<Point2D&>(rhs)); }
};
class ColoredPoint : public Point2D
{public: ...
  virtual ColoredPoint& operator=(const Point2D& rhs);

  ColoredPoint& operator=(const ColoredPoint& rhs)
  { return operator=(static_cast<Point2D&>(rhs)); }
};
```

Ensuring Consistency (ii)

- Use virtual helper function instead of declaring assignment itself as virtual.

```
class Point2D
{public: ...
  Point2D& operator=(const Point2D& rhs)
  { return doAssign(rhs); }
 protected:
  virtual Point2D& doAssign(const Point2D& rhs);
};
class Point3D : public Point2D
{protected:
  virtual Point3D& doAssign(const Point2D& rhs);
};
class ColoredPoint : public Point2D
{protected:
  virtual ColoredPoint& doAssign(const Point2D& rhs);
};
```


Non-Virtual vs. Synthetic Assignment

- Synthetic assignment hides inherited assignment.

```
class Point2D
{
 public: ...
  Point2D& operator=(const Point2D& rhs);
};
class Point3D : public Point2D
{
 public: ...

};
class ColoredPoint : public Point2D
{
 public: ...

};
```

plus synthetic
`operator=(const Point3D&)`

plus synthetic
`operator=(const ColoredPoint&)`

Consistency

- The synthetic assignment should be consistent with the virtual assignment.

```
void someFunction(Point3D lhs,Point2D& rhs)
{ ...
  lhs = rhs;
  ...
}
```

does not compile

explicit assignment is never called

```
void someFunction(Point3D lhs,Point3D rhs)
{ ...
  lhs = rhs;
  ...
}
```

invokes synthetic `Point3D::operator=(const Point3D& rhs)`

Ensuring Consistency

- Avoid hiding of base class `operator=`.
  - insert using directive in derived classes

```
class Point2D
{public: ...
  Point2D& operator=(const Point2D& rhs);
};
class Point3D : public Point2D
{public: ...
  using Point2D::operator=;
};
class ColoredPoint : public Point2D
{public: ...
  using Point2D::operator=;
};
```

Conclusion

- non-virtual assignment
  - Leads to radical slicing in all cases.
  - Even derived objects are sliced to their base class parts.
  - Usually undesired.
- virtual assignment with typeid check
  - Eliminates all slicing.
  - Mixed-type assignment results in an exception.
- virtual assignment with double/table dispatch
  - Allows slicing in all cases.
  - Mixed-type assignments lead to base class slicing.

Agenda

- Assignment Operator
- Comparison Operator

Comparison

- Comparison for equality (i.e. `operator==()`) has similar issues.
  - ... and additional ones ...

- Keep in mind the following natural requirements to an equality comparison:
  - Reflexivity: `x == x` yields `true`
  - Symmetry: if `x == y` then `y == x`
  - Transitivity: if `x == y` and `y == z` then `x == z`

A Class Hierachy

- Consider the usual hierarchy of value types:

```
class Point2D
{
 friend
 bool operator==(const Point2D& lhs, const Point2D& rhs);
};
class Point3D : public Point2D
{
 friend
 bool operator==(const Point3D& lhs, const Point3D& rhs);
};
class ColoredPoint : public Point2D
{
 friend
 bool operator==(const ColoredPoint& lhs,
                 const ColoredPoint& rhs);
};
```

Invocation

```
bool compare(const Point2D& lhs, const Point2D& rhs)
{ return (lhs == rhs); }
```

```
Point3D origin(0,0,0);
Point3D center(0,0,100);
... compare(origin,center) ...
... origin == center ...
```

invokes `operator==(Point2D&,Point2D&)` i.e. compares only coordinates

invokes `operator==(Point3D&, Point3D&)`

```
Point3D origin(0,0,0);
ColoredPoint here(0,0,RED);
... origin == here ...
```

invokes `operator==(Point2D&,Point2D&)` i.e. compares only coordinates

Comparison

rhs static type	rhs dynamic type	lhs static type: Point2D& — dynamic type: Point2D	Point2D& — Point3D	Point2D& — Colored Point	Point3D& — Point3D
Point2D&	Point2D	OK	slice	slice	slice
Point2D&	Point3D	slice	slice	slice	slice
Point2D&	ColoredPoint	slice	slice	slice	slice
Point3D&	Point3D	slice	slice	slice	OK

Solution 1

- Same slicing problem as before with assignment.
- Comparison is symmetric.
  - different from assignment
  - comparison is not a member function
- Solve the slicing problem by prohibiting mixed-type comparison.
  - as before with assignment
  - perform type check and throw an exception

Type Check

```
class Point2D
{
 friend
   bool operator==(const Point2D& lhs, const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
     ... compare Point2D part ...
   }
};

bool operator==(const Point2D& lhs, const Point2D& rhs)
{ return lhs.equals(rhs);  }
```

Type Check

```
class Point3D : public Point2D
{
 friend
 bool operator==(const Point2D& lhs, const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
     ... compare Point3D part ...
   }
};

... same for ColoredPoint ...
```


Invocation

```
bool compare(Point2D& lhs, Point2D& rhs)
{ return (lhs == rhs); }
```

```
Point3D origin(0,0,0);
Point3D center(0,0,100);
... compare(origin,center) ...
... origin == center ...
```

invokes
`Point3D::equals(Point3D&)`

```
Point3D origin(0,0,0);
ColoredPoint here(0,0,RED);
... origin == here ...
```

invokes
`Point3D::equals(Point3D&)`
i.e. type check fails

Same-Type Comparison

rhs static type	rhs dynamic type \ lhs static type / dynamic type	Point2D&			Point3D&
		Point2D	Point3D	Colored Point	Point3D
Point2D&	Point2D	OK	exc	exc	exc
	Point3D	exc	OK	exc	OK
	ColoredPoint	exc	exc	OK	exc
Point3D&	Point3D	exc	OK	exc	OK

Solution 2

- The type check solves the problem.
  - what if we want to allow mixed-type comparison ?
- Try dispatch solution (using table or double dispatch).
  - it worked for the assignment
  - why shouldn't it work for comparison as well ?

Double Dispatch

```
class Point2D
{friend bool operator==(const Point2D& lhs,const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { return other.equalsHelper((Point2D&)*this); }
   virtual bool equalsHelper(const Point2D& other) const
   {   return equalToPoint2D(other);   }
   virtual bool equalsHelper(const ColoredPoint& other) const
   {   return equalToPoint2D((Point2D&)other);   }
   virtual bool equalsHelper(const Point3D& other) const
   {   return equalToPoint2D((Point2D&)other);   }
   bool equalToPoint2D(const Point2D& other) const
   {... compare Point2D part ...}
};

bool operator==(const Point2D& lhs, const Point2D& rhs)
{ return lhs.equals(rhs); }
```


Double Dispatch

```
class Point3D : public Point2D
{friend bool operator==(const Point2D& lhs,const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { return other.equalsHelper((Point3D&)*this); }
   virtual bool equalsHelper(const Point2D& other) const
   {  return equalToPoint2D(other);   }
   virtual bool equalsHelper(const ColoredPoint& other) const
   {  return equalToPoint2D((Point2D&)other);    }
   virtual bool equalsHelper(const Point3D& other) const
   {  return equalToPoint3D(other);   }
   bool equalToPoint3D(const Point3D& other) const
   {... compare Point3D part ...}
};

... same for ColoredPoint ...
```

Invocation

```
bool compare(Point2D& lhs, Point2D& rhs)
{ return (lhs == rhs); }
```

```
Point3D origin(0,0,0);
Point3D center(0,0,100);
... compare(origin,center) ...
... origin == center ...
```

invokes `Point3D::equalsToPoint3D(Point3D&)`

```
Point3D origin(0,0,0);
ColoredPoint here(0,0,RED);
... origin == here ...
```

invokes `Point2D::equalsToPoint2D(Point2D&)`
i.e. compares 2D coordinates

Mixed-Type Comparison

rhs static type	rhs dynamic type	lhs static type Point2D&: Point2D	Point2D&: Point3D	Point2D&: Colored Point	Point3D&: Point3D
Point2D&	Point2D	OK	slice	slice	slice
Point2D&	Point3D	slice	OK	slice	OK
Point2D&	ColoredPoint	slice	slice	OK	slice
Point3D&	Point3D	slice	OK	slice	OK

Transitivity

```
Point2D origin(0,0);
ColoredPoint start(0,0,WHITE);
ColoredPoint goal(0,0,RED);

if (start == origin  && origin == goal)

// ... it should follow that start == goal ...
assert(start == goal);
```

```
         (0,0)
        /     \
   WHITE       RED
   (0,0)       (0,0)
```

- Conceptual problem:
  - Slice comparison will always lead to intransitive, incorrect comparison if different "slices" are involved.

Transitivity

- Mixed-type comparison is non-transitive.
- How about mixed-type assignment ?

```
Point2D origin(0,0);
ColoredPoint start(0,0,WHITE);
ColoredPoint goal(0,0,RED);

start = origin = goal;

// ... it should follow that start == goal ...
assert(start == goal);
```

```
         (0,0)
        /     \
   WHITE       RED
   (0,0)       (0,0)
```

... exactly the same conceptual problem !!!

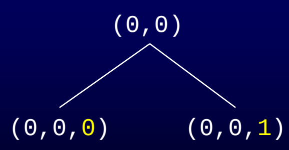
What's the crux?

- The underlying problem lies in the semantics of our mixed-type operations.
  - Correct semantics require a projection.
- Example:
  - A `Point3D` is comparable to a `Point2D` if and only if the 3rd coordinate is `0`.

```
Point2D origin(0,0);
Point3D start(0,0,0);
Point3D goal(0,0,1);

if (start == origin  && origin == goal)
```

`true` `false`



Projection

- Projections are debatable.
- Example:
  - A `ColoredPoint` is comparable to a `Point2D` if and only if the color is `BLACK`. Or `WHITE` ? Or `RED` ?
  - It follows that a `Point3D` is comparable to a `ColoredPoint` if the 3rd coordinate is `0` and the color is `BLACK`

Misconception

- We are using different notions of comparison:
  - comparison of derived types includes derived-specific parts
  - base class comparison ignores derived-specific parts
  - mixed-type comparison does yet another thing

- Since we use the same name (e.g. `operator==`) for *all* notions we expect transitivity *across* different notions of comparison.
  - that's only doable with projections

Many Distinct Operations

- Instead of unifying different notions under one umbrella we could keep the different notions distinct.

Benefit:

- transitivity within *one* notion of comparison is more natural
- leads to a notion of comparison for each class in the hierarchy
- no overriding or polymorphic behavior
- must use different function names with different signatures for different notions
  - compare2DPart, compare3DPart, ...
  - assignPoint2DPart, assignPoint3DPart, ...
- no implicit slicing (you explicitly say what you want)

Type-Specific Operations

```
class Point2D
{
 friend bool compare2DPart(const Point2D& lhs, const Point2D& rhs);
};
class Point3D : public Point2D
{
 friend bool compare3DPart(const Point3D& lhs, const Point3D& rhs);
};
...
```

```
Point2D origin(0,0);
ColoredPoint start(0,0,WHITE);
ColoredPoint goal(0,0,RED);

if (compare2DPart(start,origin)  && compare2DPart(origin == goal))

   // ... it should follow that start == goal ...
   assert(compare2DPart(start,goal));
```


2DPoint Comparison

rhs \ lhs: static type	lhs: dynamic type	Point2D&			Point3D&
rhs: static type	**rhs: dynamic type**	Point2D	Point3D	Colored Point	Point3D
Point2D&	Point2D	slice	slice	slice	slice
Point2D&	Point3D	slice	slice	slice	slice
Point2D&	ColoredPoint	slice	slice	slice	slice
Point3D&	Point3D	slice	slice	slice	slice

3DPoint Comparison

rhs static type	rhs dynamic type \ lhs	Point2D&			Point3D&
		Point2D	Point3D	Colored Point	Point3D
Point2D&	Point2D	-	-	-	-
	Point3D	-	-	-	-
	ColoredPoint	-	-	-	-
Point3D&	Point3D	-	-	-	slice

Many Distinct Operations

Downside:

- there is no `operator==` any longer
  - there is not just *one* notion of comparison for *all* classes in the hierarchy
- operators such as `operator=`, `operator==`, `operator<`, etc. may be required by other components
  - e.g. non-assignable types cannot be element types in STL containers
  - not a problem for homogenous collections such as STL containers
    - we cannot instantiate STL containers on reference types anyway
  - might be problematic in other context

Recap (i)

- non-virtual binary operation
  - Leads to radical slicing in all cases.
  - Even derived objects are sliced to their base class parts.
  - Asymmetric, non-transitive.
  - Usually undesired.
- virtual binary operation with typeid check
  - Eliminates all slicing.
  - Mixed-type assignment results in an exception.
  - Unifies different notions.
  - Symmetric, transitive.
  - Recommended.

Recap (ii)

- virtual binary operation with double/table dispatch
  - Allows slicing in all cases.
  - Mixed-type assignments lead to base class slicing.
  - Slicing is non-transitive or has debatable semantics (projection).
  - Rarely a good idea.
- no assignment
  - Makes slicing explicit.
  - No unification of different notions.
  - No polymorphic behavior.
  - Symmetric, transitive.
  - May or may not be the right approach.

Conclusion

- It's more a design issue than an implementation issue.
- The trouble starts with class hierarchies
  - where operations can be applied to objects of different types
  - through base class references
- Is inheritance the right design choice in the first place?
  - *is* a ColoredPoint a Point? or is a ColoredPoint an abstraction that consists of a Color and a Point?
  - is a Point3D an Point2D? are there Point2Ds? is Point2D concrete or an abstraction?
  - is a Student a Person? or is "Student" a role of a Person?

Use inheritance judiciously

- Avoid hierarchies of value types.
  - Without class hierarchies there is no inadvertant mixed-type operations.
  - Use composition instead of inheritance of data.
- Hierarchies of value types create lots of issues regarding base - derived class relationships.
  - Affects all operations that involve two objects from the hierarchy.
    - Assignment
    - Copying
    - Comparison
    - ...

Semantics of binary operations

- Carefully figure out which semantics a binary operation should have.
  - Critical cases are operations performed on objects of different types.
  - Invocation cannot be prevented because of the base-derived relationship.
- Define sensible semantics for the mixed-type case:
- Recommended:
  - Perform type check and reject mixed-type operation.

Recommended Assignment

```
class Point2D
{public: ...
  virtual Point2D& operator=(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point2D part ...
  }
};
```

```
class Point3D : public Point2D
{ public: ...
  virtual Point3D& operator=(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point3D part ...
  }
  Point3D& operator=(const Point3D& rhs)
  { return operator=(static_cast<Point2D&>(rhs)); }
};
... same for ColoredPoint ...
```

Recommended Assignment

```
class Point2D
{public: ...
  Point2D& operator=(const Point2D& rhs)
  { return doAssign(rhs); }
 protected:
  virtual Point2D& doAssign(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point2D part ...
  }
};
```

```
class Point3D : public Point2D
{public: ...
  using Point2D&::operator=;
 protected:
  virtual Point3D& doAssign(const Point2D& rhs)
  { ...
    if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
    ... assign Point3D part ...
  }
};
... same for ColoredPoint ...
```


Recommended Comparison

```
class Point2D
{friend bool operator==(const Point2D& lhs, const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
     ... compare Point2D part ...
   }
};
bool operator==(const Point2D& lhs, const Point2D& rhs)
{ return lhs.equals(rhs);  }
```

```
class Point3D : public Point2D
{friend bool operator==(const Point2D& lhs, const Point2D& rhs);
 private:
   virtual bool equals(const Point2D& other) const
   { if (typeid(*this) != typeid(rhs))
       throw TypeMismatchException();
     ... compare Point3D part ...
   }
};
... same for ColoredPoint ...
```

Contact

Angelika Langer
Training & Mentoring
Object-Oriented Software Development in C++ & Java
Munich, Germany
Email: **info@AngelikaLanger.com**
http: **//www.AngelikaLanger.com**