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 Hierarchy

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);  
};
### Problem

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

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

Assigns 2D part of colored and 3D point.

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

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

Assigns 2D part of colored points.

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

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

Assigns 2D part of colored and 3D point.

### Object Slicing

- **ColoredPoint**
  - 2D part
  - Color part

- **Point3D**
  - 2D part
  - 3rd dimension part

Invokes `Point2D::operator=`.

Other cases ...

• Invocation through references to base and derived type:

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

Other cases ...

• Invocation through references to derived type:

```cpp
void someFunction(Point3D& rhs, Point3D& lhs)
{
    rhs = lhs;
    ...
}
```
### Non-Virtual Assignment

<table>
<thead>
<tr>
<th>rhs</th>
<th>static type</th>
<th>dynamic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point2D</td>
<td>Point2D &amp;</td>
<td>OK</td>
</tr>
<tr>
<td>Point2D</td>
<td>Point2D &amp;</td>
<td>slice</td>
</tr>
<tr>
<td>Point2D</td>
<td>ColoredPoint</td>
<td>slice</td>
</tr>
<tr>
<td>Point2D</td>
<td>Point3D</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td>Point3D &amp;</td>
<td>slice</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td>Point3D &amp;</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td>Point3D &amp;</td>
<td>slice</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td>Point3D</td>
<td>slice</td>
</tr>
</tbody>
</table>

What's the problem ... ?

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

```cpp
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);
};
```
Another Class Hierarchy

```cpp
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);
};
```

Problem

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

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

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

- Invokes virtual function
- Assigns Point3D part of ColoredPoint ??
Problem

- We still get slices ...

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

```cpp
Point2D origin2D;
Point3D origin3D;
someFunction(origin2D, origin3D);
someFunction(origin3D, origin2D);
```

invokes `Point2D::operator=()`
invokes `virtual function` 
invokes `Point3D::operator=()`
might crash

Virtual Assignment

<table>
<thead>
<tr>
<th>Lhs</th>
<th>static type</th>
<th>dynamic type</th>
<th>Points</th>
<th>Point2D</th>
<th>Point3D</th>
<th>Colored Point</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>static type</td>
<td>dynamic type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point2D</td>
<td></td>
<td></td>
<td>A</td>
<td>OK</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td></td>
<td></td>
<td>B</td>
<td>crash</td>
<td>OK</td>
<td>crash</td>
<td>OK</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td></td>
<td></td>
<td>C</td>
<td>crash</td>
<td>crash</td>
<td>OK</td>
<td>crash</td>
</tr>
<tr>
<td>Point3D</td>
<td></td>
<td></td>
<td>B</td>
<td>crash</td>
<td>OK</td>
<td>crash</td>
<td>OK</td>
</tr>
<tr>
<td>Point3D</td>
<td></td>
<td></td>
<td>B</td>
<td>crash</td>
<td>crash</td>
<td>OK</td>
<td>crash</td>
</tr>
</tbody>
</table>
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

```cpp
class Point3D : public Point2D
{
    public: ...
    virtual Point3D& operator=(const Point2D & rhs)
    {
        // Check if Point2D is a Point3D
        if (isPoint2D & Point3D)
        {
            // Perform assignment
        }
    }
};
```

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?

```cpp
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

<table>
<thead>
<tr>
<th>Lhs</th>
<th>static type</th>
<th>dynamic type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>ColoredPoint</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>static type</td>
<td>point 2D&lt;BR&gt;Point 3D&lt;BR&gt;ColoredPoint</td>
<td>dynamic type</td>
<td>Point2D</td>
<td>Point3D</td>
<td>ColoredPoint</td>
<td>Point3D</td>
</tr>
<tr>
<td>Point 2D&lt;BR&gt;Point 3D&lt;BR&gt;ColoredPoint</td>
<td>OK&lt;BR&gt;exc&lt;BR&gt;exc</td>
<td>slice&lt;BR&gt;OK&lt;BR&gt;OK</td>
<td>slice&lt;BR&gt;exc&lt;BR&gt;OK</td>
<td>slice&lt;BR&gt;exc&lt;BR&gt;exc</td>
<td>OK&lt;BR&gt;exc&lt;BR&gt;exc</td>
<td>slice&lt;BR&gt;exc&lt;BR&gt;exc</td>
</tr>
</tbody>
</table>
**TypeId**

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

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

**Type Check**

```cpp
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.

Virtual Assignment

<table>
<thead>
<tr>
<th></th>
<th>Lhs</th>
<th>static type</th>
<th>dynamic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>static type</td>
<td>Point 2D</td>
<td>OK</td>
<td>exc</td>
</tr>
<tr>
<td>dynamic type</td>
<td>Point 3D</td>
<td>exc</td>
<td>OK</td>
</tr>
<tr>
<td>static type</td>
<td>ColoredPoint</td>
<td>exc</td>
<td>exc</td>
</tr>
<tr>
<td>dynamic type</td>
<td>Point 3D</td>
<td>exc</td>
<td>OK</td>
</tr>
</tbody>
</table>
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
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
class Point2D
{
    private:
    class DispatchTable
    {public:
        typedef Point2D(*fptrType)(Point2D&, const Point2D&);
        fptrType getFunction(const type_info& lhs,
                              const type_info& rhs);
    private:
        map<typeidPair, fptrType> tab;
    };
    static DispatchTable dispatchTable;
};
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;
    }
};

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()`

```cpp
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

```cpp
class Point2D
{public:
    Point2D& operator=(const Point2D& rhs)
    { return assign(rhs); }
private:
    virtual Point2D& assign(const Point2D& rhs)
    { return 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

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

```cpp
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);
};

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

Double Dispatch

```cpp
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 Point2D& assignHelper(const Point3D& rhs);
    virtual Point2D& assignHelper(const ColoredPoint& rhs);
    Point3D& assignPoint3D(const Point2D& rhs);
};

Point2D a1 = Point2D();
Point2D a2 = Point3D();
a1 = a2;
```
Double Dispatch

```cpp
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 Point3D & assignHelper(const Point2D & rhs);
    Point2D & assignPoint2D(const Point2D & rhs);
};

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);
    Point2D & assignPoint2D(const Point2D & rhs);
};
```

```cpp
Point3D & m1 = Point3D();
Point3D & m2 = Point3D();
m1 = m2;
```
**Double Dispatch**

```cpp
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
{
    virtual Point2D& assignHelper(const Point3D& rhs)
    {
        return ((Point2D&)rhs).assignPoint2D(*this);
    }
};
```

1. `Point2D a1 = Point3D(); Point2D a2 = ColoredPoint(); a1 = a2;`
2. `void someFunction(Point2D& lhs, Point2D& rhs)
   {
       lhs = rhs;
   }`
3. `Invocation through references leads to mixed-type assignment and (intended) object slicing.
   - `Point3D red, blue; someFunction(red, blue);`
   - `Point3D origin; ColoredPoint red; someFunction(origin, red);`
   - `Point3D red, blue; someFunction(red, blue);`
   - `Point3D origin; ColoredPoint red, someFunction(origin, red);`
   - `Point3D origin; ColoredPoint red; someFunction(origin, red);`
   - `void someFunction(Point2D& lhs, Point2D& rhs)
   {
       ...
       lhs = rhs;
   }`
   - `triggers double dispatch`

**Invocation**

- Invocation through references leads to mixed-type assignment and (intended) object slicing.
### Assignment With (Double/Table) Dispatch

<table>
<thead>
<tr>
<th></th>
<th>static type</th>
<th>dynamic type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>Colored Point</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>Point2D A</td>
<td></td>
<td>OK</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td></td>
<td>Point3D B</td>
<td></td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>ColoredPoint C</td>
<td></td>
<td>slice</td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
</tr>
<tr>
<td>lhs</td>
<td>Point2D A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Point3D B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ColoredPoint C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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.

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

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.

```c++
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);
};
```
Consistency

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

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

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

should have the same effect

```cpp
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

```cpp
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 ColoredPoint& 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.

```cpp
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.

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

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

```cpp
void someFunction(Point3D lhs, Point2D rhs)
{
    lhs = rhs;
    ...
}
```
does not compile

explicit assignment is never called

```cpp
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

```cpp
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. \texttt{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 Hierarchy

• Consider the usual hierarchy of value types:

```cpp
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

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

- Invokes `operator==(Point2D, Point2D)` i.e. compares only coordinates

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

- Invokes `operator==(Point3D, Point3D)` i.e. compares only coordinates

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

### Comparison

<table>
<thead>
<tr>
<th>rhs</th>
<th>static type</th>
<th>dynamic type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>ColoredPoint</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>OK</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>Point2D</td>
<td></td>
<td></td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td></td>
<td></td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td></td>
<td></td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>static type</th>
<th>dynamic type</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>OK</td>
</tr>
<tr>
<td>Point3D</td>
<td></td>
<td></td>
<td>slice</td>
</tr>
</tbody>
</table>

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

```cpp
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

```cpp
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

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

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

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

 invokes `Point3D::equals(Point3D)`

 invokes `Point3D::equals(Point3D)` 
 i.e. type check fails
### Same-Type Comparison

<table>
<thead>
<tr>
<th>static type</th>
<th>dynamic type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>ColoredPoint</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point2D</td>
<td>OK</td>
<td>exc</td>
<td>exc</td>
<td>exc</td>
<td></td>
</tr>
<tr>
<td>Point3D</td>
<td>exc</td>
<td>OK</td>
<td>exc</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>ColoredPoint</td>
<td>exc</td>
<td>exc</td>
<td>OK</td>
<td>exc</td>
<td></td>
</tr>
<tr>
<td>Point3D</td>
<td>exc</td>
<td>OK</td>
<td>exc</td>
<td>OK</td>
<td></td>
</tr>
</tbody>
</table>

### 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

```cpp
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); }
};
```

... same for ColoredPoint ...

```cpp
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 equalToPoint3D((Point2D&)other); }
        virtual bool equalsHelper(const Point3D& other) const
        { return equalToPoint3D((Point2D&)other); }
        bool equalToPoint3D(const Point3D& other) const
        { ... compare Point3D part ... }
    }

    ... same for ColoredPoint ...
```
## Binary Operators

### Invocation

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

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

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

**Invokes** `Point3D::equalsToPoint3D(Point3D&)`

**i.e. compares 2D coordinates**

### Mixed-Type Comparison

<table>
<thead>
<tr>
<th></th>
<th>static type</th>
<th>dynamic type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>ColoredPoint</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static type</td>
<td>Point2D</td>
<td>OK</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>dynamic type</td>
<td>Point3D</td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
</tr>
<tr>
<td>static type</td>
<td>ColoredPoint</td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>dynamic type</td>
<td>Point3D</td>
<td>slice</td>
<td>OK</td>
<td>slice</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
Transitivity

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

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

... 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.

```plaintext
Point2D origin(0,0);
Point3D start(0,0,0);
Point3D goal(0,0,1);
if (start == origin && origin == goal)
  true
else 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

```cpp
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

<table>
<thead>
<tr>
<th></th>
<th>lhs</th>
<th>static type</th>
<th>dynamic type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Point2D</td>
<td>Point3D</td>
</tr>
<tr>
<td>Point2D</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>lhs</th>
<th>static type</th>
<th>dynamic type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Point2D</td>
<td>Point3D</td>
</tr>
<tr>
<td>Point2D</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>Point3D</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
<tr>
<td>ColoredPoint</td>
<td>slice</td>
<td>slice</td>
<td>slice</td>
</tr>
</tbody>
</table>
### 3DPoint Comparison

<table>
<thead>
<tr>
<th></th>
<th>static type</th>
<th>Point2D</th>
<th>Point3D</th>
<th>ColoredPoint</th>
<th>Point3D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Point2D</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Point3D</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>ColoredPoint</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Point3D</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>slice</td>
</tr>
</tbody>
</table>

### 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

```cpp
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**

```cpp
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**

```cpp
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