CS170#11.1

Inheritance

Vadim Surov

Outline

- Introduction
- Example: Point3D

Introduction

- Given two classes, they can have 3 possible relationships:
    - They could be *independent*
    - They could be related by *aggregation*
        - *containment*
        - *composition*
    - They could be related by *inheritance*

Introduction

- Aggregation describes a "has-a" relationship
  - One class contains (has) another class
  - E.g., a Weapon "has-a" string as a name, a Player "has-a" Weapon (or has many Weapons), etc.
- Inheritance describes an "is-a-kind-of" relationship
  - One class shares attributes with another
  - E.g., a dog "is-a-kind-of" animal, a Ranged Weapon "is-a-kind-of" Weapon

Example: Point3D

- This is a simple example to illustrate how inheritance works
- Let's define a **struct** that describes a point in 2D space:

```
struct Point2D
{
  double x;
  double y;
};
```

Example: Point3D

- We can define another **struct** for a point in 3D space:

```
struct Point3D
{
  double x;
  double y;
  double z;
};
```

- Or we can use composition:

```
struct Point3D_composite
{
  Point2D xy; // struct contains a Point2D object
  double z;
};
```

Example: Point3D

- Accessing the members:

```
void PrintXY(const Point2D& pt){
  std::cout << pt.x << ", " << pt.y;
}
void PrintXYZ(const Point3D& pt){
  std::cout << pt.x << ", " << pt.y
            << ", " << pt.z;
}
void PrintXYZ(const Point3D_composite& pt){
  std::cout << pt.xy.x << ", " << pt.xy.y;
  std::cout << ", " << pt.z;
}
```

Example: Point3D

- We could rewrite PrintXYZ() for Point3D_composite to reuse PrintXY():

```
void PrintXYZ(const Point3D_composite& pt)
{
  PrintXY(pt.xy); // Delegate for x and y
  std::cout << ", " << pt.z;
}
```

Example: Point3D

- Another way is to use *inheritance*:

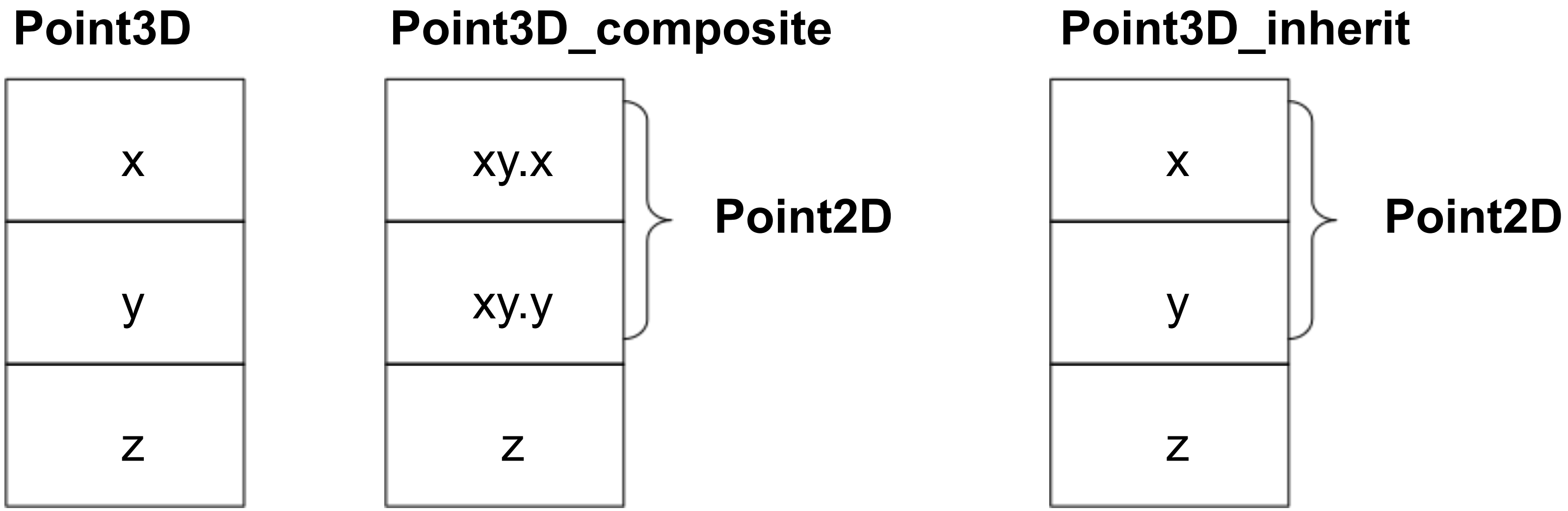
```
struct Point3D_inherit : public Point2D
{
  double z;
};
```

- Its corresponding PrintXYZ() function:

```
void PrintXYZ(const Point3D_inherit& pt)
{
  std::cout << pt.x << ", " << pt.y
            << ", " << pt.z;
}
```

Example: Point3D

- All 3 structures are organized in the same way in memory:

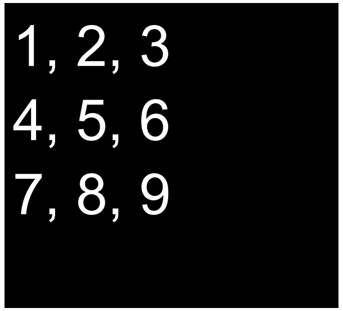


Example: Point3D

- Sample usage:

```
Point3D pt3; // "flat" version
pt3.x = 1; pt3.y = 2; pt3.z = 3;
PrintXYZ(pt3);
std::cout << std::endl;
Point3D_composite ptc; // composite version
ptc.xy.x = 4; ptc.xy.y = 5; ptc.z = 6;
PrintXYZ(ptc);
std::cout << std::endl;
Point3D_inherit pti; // inheritance version
pti.x = 7; pti.y = 8; pti.z = 9;
PrintXYZ(pti);
std::cout << std::endl;
```

- Output:



```
1, 2, 3
4, 5, 6
7, 8, 9
```

Example: Point3D

- Syntax:

```
struct Point3D_inherit: public Point2D
```

- Point2D is the **base class** for Point3D_inherit
- Point3D_inherit is the **derived class**
- The `public` keyword indicates that public methods for the base class remains public for the derived class
  - This is called *public inheritance*

Example: Point3D

- Let's supply member functions:

```
struct Point2D
{
  double x;
  double y;

  void print(void)
  {
    std::cout << x <<
        ", " << y;
  }
};
```

```
struct Point3D
{
  double x;
  double y;
  double z;

  void print(void)
  {
    std::cout << x << ", "
         << y << ", " << z;
  }
};
```

Example: Point3D

- Since all the data members in Point2D are public, we can access them directly
- For Point3D_composite:

```
struct Point3D_composite {
  Point2D xy;
  double z;

  void print(void){
    // Point2D members are public
    std::cout << xy.x << ", " << xy.y;
    std::cout << ", " << z;
  }
};
```

Example: Point3D

- For Point3D_inherit:

```
struct Point3D_inherit : public Point2D
{
  double z;
  void print(void)
  {
    // Point2D members are public
    std::cout << x << ", " << y;
    std::cout << ", " << z;
  }
};
```

Example: Point3D

- Using the methods:

```
Point3D pt3; // "flat" version
pt3.x = 1; pt3.y = 2; pt3.z = 3;
pt3.print();
std::cout << std::endl;
Point3D_composite ptc; // composite version
ptc.xy.x = 4; ptc.xy.y = 5; ptc.z = 6;
ptc.print();
std::cout << std::endl;
Point3D_inherit pti; // inheritance version
pti.x = 7; pti.y = 8; pti.z = 9;
pti.print(); // Is this legal? Ambiguous?
std::cout << std::endl;
```

Example: Point3D

- Now let's go "C++ style"
- Our new **class** Point2D:

```
class Point2D {
public:
  // Constructor
  Point2D(double x, double y) : x(x), y(y) { };
  void print(void) {
    std::cout << x << ", " << y;
  }
private:
  double x;
  double y;
};
```

Example: Point3D

- Definition for **class** Point3D:

```
class Point3D {
public:
  Point3D(double x, double y, double z)
    : x(x), y(y), z(z) { };
  void print(void) {
    std::cout << x << ", " << y << ", " << z;
  }
private:
  double x;
  double y;
  double z;
};
```

Example: Point3D

- With composition (aggregation), we must initialize the contained Point2D object using the member initialization list:

```
class Point3D_composite {
public:
  Point3D_composite(double x, double y, double z)
    : xy(x, y), z(z) { };
  void print(void){
    xy.print(); // Point2D members are private
    std::cout << ", " << z;
  }
private:
  Point2D xy
  double z;   };
```

Example: Point3D

- With inheritance, we must initialize the contained Point2D *base class object* using the member initialization list:

```
class Point3D_inherit : public Point2D {
public:
  Point3D_inherit(double x, double y, double z)
    : Point2D(x, y), z(z) { };
  void print(void) {
    Point2D::print(); //Point2D members are private
    std::cout << ", " << z;
  }
private:
  double z;
};
```

Example: Point3D

- Sample usage:

```
Point3D pt3(1, 2, 3); // "flat" version
pt3.print();
std::cout << std::endl;
Point3D_composite ptc(4, 5, 6); // composite
                                    version
ptc.print();
std::cout << std::endl;
Point3D_inherit pti(7, 8, 9); // inheritance
                                   version
pti.print();
std::cout << std::endl;
```