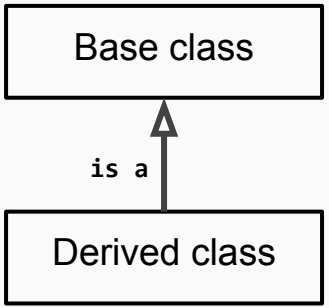
Inheritance

UPC - Videogame Design & Development - Programming II

Inheritance

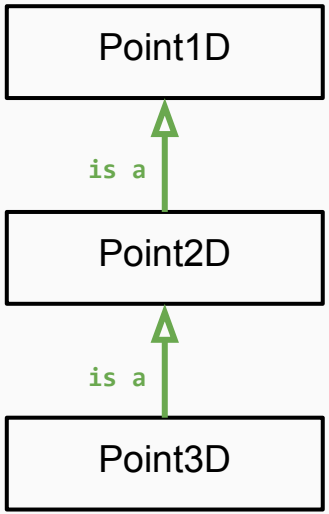
Inheritance is a mechanism from which a class inherits some properties from a parent class



Inheritance allows creating new classes that **inherit attributes (data members) and methods (member functions) from a parent class**.

In the example, the **Derived class** will have all attributes and methods from **Base class** plus its own ones.

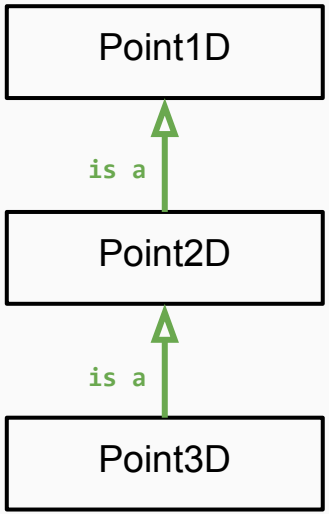
Inheritance in C++




```
class Point1D {
  public:
    int x;
    void setX(int X) {x = X;}
};

class Point2D : public Point1D {
  public:
    int y;
    void setY(int Y) {y = Y;}
};

class Point3D : public Point2D {
  public:
    int z;
    void setZ(int Z) {z = Z;}
};
```

```
int main ()
{
  // Declaring a 1D point
  Point1D p1;
  p1.setX(1);
  cout << p1.x << endl;

  // Declaring a 2D point
  Point2D p2;
  p2.setX(2); p2.setY(2);
  cout << p2.x << " " << p2.y << endl;

  // Declaring a 3D point
  Point3D p3;
  p3.setX(3); p3.setY(3); p3.setZ(3);
  cout << p3.x << " " << p3.y << " " << p3.z << endl;

  return 0;
}
```

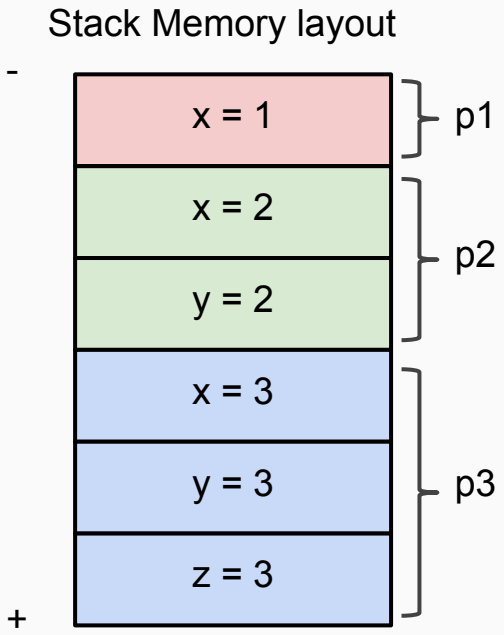
Inheritance in C++




```
class Point1D {
  public:
    int x;
    void setX(int X) {x = X;}
};

class Point2D : public Point1D {
  public:
    int y;
    void setY(int Y) {y = Y;}
};

class Point3D : public Point2D {
  public:
    int z;
    void setZ(int Z) {z = Z;}
};
```

```
int main ()
{
  // Declaring a 1D point
  Point1D p1;
  p1.setX(1);
  cout << p1.x << endl;

  // Declaring a 2D point
  Point2D p2;
  p2.setX(2); p2.setY(2);
  cout << p2.x << " " << p2.y << endl;

  // Declaring a 3D point
  Point3D p3;
  p3.setX(3); p3.setY(3); p3.setZ(3);
  cout << p3.x << " " << p3.y << " " << p3.z << endl;

  return 0;
}
```

Inheritance in C++

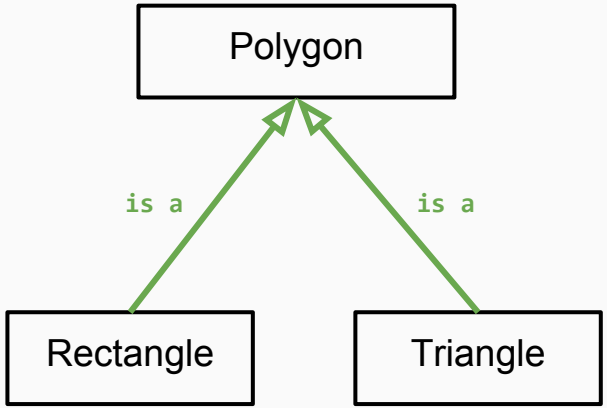



```
int main ()
{
  // Declaring a 1D point
  Point1D p1;
  p1.setX(1);
  cout << p1.x << endl;

  // Declaring a 2D point
  Point2D p2;
  p2.setX(2); p2.setY(2);
  cout << p2.x << " " << p2.y << endl;

  // Declaring a 3D point
  Point3D p3;
  p3.setX(3); p3.setY(3); p3.setZ(3);
  cout << p3.x << " " << p3.y << " " << p3.z << endl;

  return 0;
}
```

Inheritance in C++

```
class Polygon
{
  protected:
    int width, height;

  public:
    void set_size(int w, int h)
    {
      width = w;
      height = h;
    }
};
```

```
class Rectangle : public Polygon
{
  public:
    int area () const
    {
      return width * height;
    }
};

class Triangle : public Polygon
{
  public:
    int area () const
    {
      return width * height / 2;
    }
};
```



Inheritance in C++

```
class Polygon
{
  protected:
    int width, height;

  public:
    void set_size(int w, int h)
    {
      width = w;
      height = h;
    }
};
```

```
class Rectangle : public Polygon
{
  public:
    int area () const
    {
      return width * height;
    }
};

class Triangle : public Polygon
{
  public:
    int area () const
    {
      return width * height / 2;
    }
};
```

```
int main ()
{
  // Declaring a rectangle
  Rectangle rect;
  rect.set_size(4,5);
  cout << rect.area() << endl;
  // Will print 20

  // Declaring a triangle
  Triangle trgl;
  trgl.set_size(4,5);
  cout << trgl.area() << endl;
  // Will print 10

  return 0;
}
```

Access modifiers

Access from	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside	yes	no	no

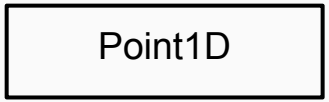
The access to members of a class (either data members or functions) can be limited with access modifiers (**public**, **protected**, and **private**).

Access modifiers specify the level of accessibility to the members of a class from different locations:

- Same class
- Derived classes
- External code (outside of the class hierarchy)

Access modifiers

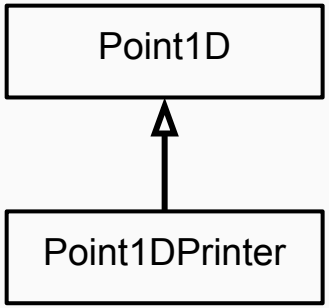
Access from	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside	yes	no	no




```
class Point1D {
  private:
    const char *name;
  protected:
    int x;
    const char *getName() const {
      return name; // Ok
    }
  public:
    void setX(int x_) {
      x = x_; // Ok
    }
    int getX() const {
      return x; // Ok
    }
};
```

Access modifiers

Access from	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside	yes	no	no

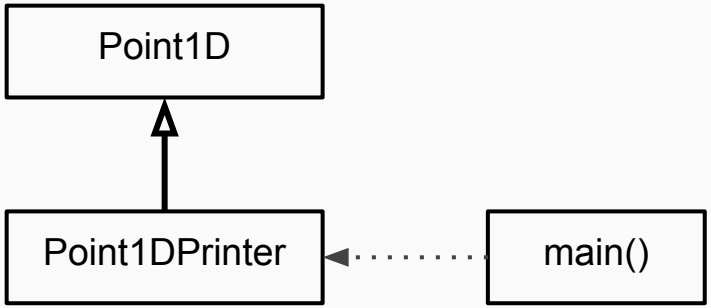



```
class Point1D {
  private:
    const char *name;
  protected:
    int x;
    const char *getName() const {
      return name; // Ok
    }
  public:
    void setX(int x_) {
      x = x_; // Ok
    }
    int getX() const {
      return x; // Ok
    }
};
```

```
class Point1DPrinter
            : public Point1D {
  public:
    void printName() const
    {
      // Error
      cout << name << endl;
      // Ok
      cout << getName() << endl;
    }
    void printX() const {
      // Ok
      cout << x << endl;
    }
};
```

Access modifiers

Access from	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside	yes	no	no




```
class Point1D {
  private:
    const char *name;
  protected:
    int x;
    const char *getName() const;
  public:
    void setX(int X);
    int getX() const;
};

class Point1DPrinter
                : public Point1D {
  public:
    void printName() const;
    void printX() const;
};
```

```
int main ()
{
  // Declaring a 1D point printer
  Point1DPrinter p1;

  p1.x = 1; // Error
  p1.setX(1); // Ok

  cout << p1.x << endl; // Error
  cout << p1.getX(); // Error
  p1.printX(); // Ok

  cout << p1.name << end; // Error
  cout << p1.getName(); // Error
  p1.printName(); // Ok

  return 0;
}
```

Order of construction

```
class Point1D {
  public:
    int x;
    Point1D() : x(0) {
      cout << “Ctor Point1D” << endl;
    }
};

class Point2D : public Point1D {
  public:
    int y;
    Point2D() : y(0) {
      cout << “Ctor Point2D” << endl;
    }
};
```

If not otherwise specified, the default constructor of the base class is automatically called by the derived one.

The base class constructor is called first, then the derived class one.

Try this code:

```
int main()
{
      Point2D p;
      system(“pause”);
      return 0;

}
```

Order of destruction

```
class Point1D {
  public:
    int x;
    Point1D() : x(0) {
      cout << “Ctor Point1D” << endl;
    }
};

class Point2D : public Point1D {
  public:
    int y;
    Point2D() : y(0) {
      cout << “Ctor Point2D” << endl;
    }
};
```

The destructors are called in inverse order. The constructor of the derived class is called first, and then the constructor of the base class.

Add destructors printing “Dtor Point1D” and “Dtor Point2D” and try the code again:

```
int main()
{
      Point2D p;
      system(“pause”);
      return 0;

}
```

Specialized constructors

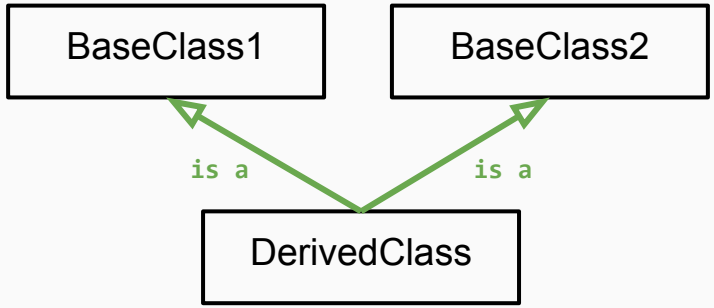
```
class Point1D {
  public:
    int x;
    Point1D(int x_) : x(x_) {
      cout << “Ctor Point1D” << endl;
    }
};

class Point2D : public Point1D {
  public:
    int y;
    Point2D(int x_, int y_) :
      Point1D(x_), y(y_) {
      cout << “Ctor Point2D” << endl;
    }
};
```

If other constructor must be called instead of the default constructor, then it must be specified in the derived class constructor.

```
int main()
{
      Point2D p(3,5);
      system(“pause”);
      return 0;

}
```

Avoid multiple inheritance

```
class BaseClass1 {
  ...
};

class BaseClass2 {
  ...
};

class DerivedClass :
        public BaseClass1,
        public BaseClass2
{
  ...
};
```



C++ allows inheritance from more than a parent class simultaneously.

Beware of multiple inheritance, try not to use them. It is not usually a good idea.

Multiple inheritance