Class Inheritance

- inheritance: private, protected, public
- constructors
- member functions
- virtual functions: virtual and pure virtual

Inheritance

- Now we've introduced the concepts of OOP, it's time to move onto the *power* of OOP.
- **Inheritance** allows us to recycle code, and build a hierarchy of objects.
- We do this by defining a **Base** class. The **Derived** class inherits the properties of the Base class, and *adds to them*.
- Let's revisit the `Point` class from before (we've added a `moveTo` member), and use it as the new Base class:

```
#ifndef __POINT_HH   // Point1.hh
#define __POINT_HH
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rMoveTo(const Point&);
  void moveTo(const Point&);
private:
  int m_x, m_y;
};
#endif // __POINT_HH
```

```
#include <math.h>   // Point1.cc
#include "Point1.hh"

Point::Point(int initX, int initY) {
  m_x = initX;
  m_y = initY;
}
void Point::print() { cout << "(" << m_x << ", " << m_y << ")"; }
int Point::r() { return (int)sqrt( m_x*m_x + m_y*m_y ); }
void Point::moveTo(const Point& p) {
  m_x = p.m_x;
  m_y = p.m_y;
}
void Point::rMoveTo(const Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```

We'll define some shapes:

- circle
- square

that inherit from `Point`.

After all, every shape has coordinates, *plus* other features, such as radius, side length, etc.

```
#ifndef __SHAPE_HH   // Shape1.hh
#define __SHAPE_HH
#include "Point1.hh"

class Circle : public Point {
public:
  Circle(int initX=0, int initY=0, int initR=0);
  void print();
private:
  int m_r; };

class Rectangle : public Point {
public:
  Rectangle(int initX=0, int initY=0, int initLX=0, int initLY=0);
  void print();
private:
  int m_lx, m_ly; };
#endif // __SHAPE_HH
```

```
#include "Shape1.hh"   // Shape1.cc

Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY) { m_r = initR;  }

void Circle::print() {
  cout << "I am a circle at: ";
  Point::print();
  cout << ".  radius = " << m_r;  }

Rectangle::Rectangle(int initX, int initY, int initLX, int initLY)
  : Point(initX, initY) { m_lx = initLX; m_ly = initLY;  }

void Rectangle::print() {
  cout << "I am a rectangle at: ";
  Point::print();
  cout << ".  sides = " << m_lx << ", " << m_ly;  }
```

```
#include "Shape1.hh"   // Inheritance-1.cc

int main() {
  Circle c(3,4,6);
  c.print();
  cout << endl;
  Rectangle r(6,7, 3, 2);
  r.print();
  cout << endl;
  return 0;
}
```

Points to Note:

1. The Base class, `Point` doesn't know anything about the derived classes

2. The derived class uses the syntax:
   `class Derived : public Base`
   to say it inherits from `Base`. The derived class has *all* the member data and functions of the base class, *plus* any others that it defines.

3. The rules for which members are available to the derived class(es) are logical, but sometimes confusing:

   - members in the Base class declared `public:` are accessible to derived classes
   - members in the Base class declared `private:` are *not* accessible to derived classes. Not that in this example, `Circle` does not need `m_x`, since we can rely on `Point`'s member functions.

- often we want Base class members to be available to derived classes, but not *other* classes. Then we declare them `protected:`

4. The rules for which members are available to *descendants* of the derived classes are logical, but sometimes confusing:
   - Inheritance marked `public` enables descendants of the derived class to continue the inheritance.
   - Inheritance marked `private` stops further inheritance.
5. We give each derived class its own `print` function (a circle is different from a point).
6. In the derived class constructor, we pass parameters to the Base class constructor – it is constructed *before* the derived class.
7. The Base and Derived classes each have a `print()` member – specify which one we want to use with the scope resolution operator, `::`

Style

Initializing member data in the constructor

We have learned how to initialize a *built-in* type with: `int i(7);` This is really calling the constructor for a type `int`.

C++ is symmetric between *built-in* types and *user-defined* types or classes.

Since we can use:

```
Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY)
```

to call the base constructor for class `Point` we can also use:

```
Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY), m_r(initR) {}
```

to call the constructor for type `int`.

We will do this from now on – not only in derived classes, but in *all* classes.

Everything else stays the same. It is just more *elegant*.

Our code now becomes:

```
#include <math.h>   // Point2.cc
#include "Point2.hh"

Point::Point(int initX, int initY) : m_x(initX), m_y(initY) {}

void Point::print() { cout << "(" << m_x << ", " << m_y << ")"; }

int Point::r() { return (int)sqrt( m_x*m_x + m_y*m_y ); }

void Point::moveTo(const Point& p) {
  m_x = p.m_x;
  m_y = p.m_y;
}
void Point::rMoveTo(const Point& p) {
  m_x += p.m_x;
  m_y += p.m_y;
}
```

```
#include "Shape2.hh"   // Shape2.cc

Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY), m_r(initR) {}

void Circle::print() {
  cout << "I am a circle at: ";
  Point::print();
  cout << ".  radius = " << m_r;
}
Rectangle::Rectangle(int initX, int initY, int initLX, int initLY)
  : Point(initX, initY), m_lx(initLX), m_ly(initLY) {}

void Rectangle::print() {
  cout << "I am a rectangle at: ";
  Point::print();
  cout << ".  sides = " << m_lx << ", " << m_ly; }
```

Now we can use the power of inheritance.

`Point` already has access functions and other utility functions, so we can just use them.

- Inheritance is an example of *code reuse*
- As far as possible, inherit `private` (rather than protected) data members
- if necessary, using `protected` member functions. Why?
  - If we change the Base class `protected` data, we will break all descendant classes.

```
#include "Shape1.hh"   // Inheritance-3.cc

int main() {
  Circle c(3,4,6);
  c.print();
  cout << endl;
  cout << "the distance of my center from the origin is: "
       << c.r() << endl;
  Rectangle r(6,7, 3, 2);
  r.print();
  cout << endl;
  r.rMoveTo(Point(10,11));
  cout << "I have just moved to (" << r.x()
       << "," << r.y() << ")" << endl;
  return 0;
}
```

- The Inheritance can be continued indefinitely: parents can have children, and those children become parents.
- Suppose we want to make a `Square` class using `rectangle` as a base class.
- (In this example, we would probably make `Square` inherit directly from `Point`.)
- We just do the "obvious":

```
#ifndef __SHAPE_HH   // Shape3a.hh
#define __SHAPE_HH
#include "Point3a.hh"
class Circle : public Point {
public:
  Circle(int initX=0, int initY=0, int initR=0);
  void print();
private:
  int m_r;  };
class Rectangle : public Point {
public:
  Rectangle(int initX=0, int initY=0, int initLX=0, int initLY=0);
  void print();
private:
  int m_lx;   int m_ly;  };
class Square : public Rectangle {
public:
  Square(int initX=0, int initY=0, int initL=0);  };
#endif // __SHAPE_HH
```

```
#include "Shape3a.hh"   // Shape3a.cc

Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY), m_r(initR) {}
void Circle::print() {
  cout << "I am a circle at: ";
  Point::print();
  cout << ".  radius = " << m_r;
}
Rectangle::Rectangle(int initX, int initY, int initLX, int initLY)
  : Point(initX, initY), m_lx(initLX), m_ly(initLY) {}
void Rectangle::print() {
  cout << "I am a rectangle at: ";
  Point::print();
  cout << ".  sides = " << m_lx << ", " << m_ly;
}
Square::Square(int initX, int initY, int initL)
  : Rectangle(initX, initY, initL, initL) {}
```

Virtual Functions

We now come to one of the more powerful (and difficult) features of C++.

- In everything so far, the compiler has known from the signature, or explicit scoping, *which* function to call, and can link accordingly.
- Let's add a `show()` method, which could be a graphics drawing routine, (for simplicity it can be just like `print()`)
- Further, let's make the `moveTo` and `rmoveTo` methods "`show`" themselves after the object has been moved.

Here's the question:

- There is only one `moveTo` (in class `Point`).
- How does it know *which* `show` method to invoke?

If we just modify the files in the "obvious" way:

```
#include <math.h>   // Point4.cc
#include "Point4.hh"

Point::Point(int initX, int initY)
  : m_x(initX), m_y(initY) {}
void Point::print() {
  cout << "(" << m_x << ", " << m_y << ")";  }
int Point::r() {
  return (int)sqrt( m_x*m_x + m_y*m_y );  }
void Point::moveTo(const Point& p) {
  m_x = p.m_x;   m_y = p.m_y;
  show();  }
void Point::rMoveTo(const Point& p) {
  m_x += p.m_x;  m_y += p.m_y;
  show();  }
void Point::show() {
  cout << "I am a Point at (" << x() << "," << y() << ")" << endl; }
```

then `moveTo` and `rmoveTo` think that `show` is `Point`'s `show()`.

- This is *not* what we want.
- We can fix this by declaring `Point::show()` to be *virtual*.

```
#ifndef __POINT_HH   // Point5.hh
#define __POINT_HH
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rMoveTo(const Point&);
  void moveTo(const Point&);
  virtual void show();
private:
  int m_x, m_y;
};
#endif // __POINT_HH
```

- A **virtual** function means that the choice of *which* function to use is deferred until run time.

- It is done by building a **virtual function table**. The (small) price is at run time, there is an extra lookup to invoke the right function.

- If we *don't* supply a function `show()` in a derived class, we default back to the `show()` in the Base class.

- We can also use an **Abstract Base Class**. Here the virtual function is not implemented *at all*, but only used for derived classes to inherit from. We then *force* the implementation in the derived class by using a **Pure Virtual Function**.

This is achieved with the declaration:

```
virtual void show()=0;
```

The `=0` says: "don't implement this function in this class". If it's not implemented *anywhere* we'll get a linker error.

```
#ifndef __POINT_HH   // Point6.hh
#define __POINT_HH
#include <iostream.h>

class Point {
public:
  Point(int initX=0, int initY=0);
  void print();
  int x() { return m_x; }
  int y() { return m_y; }
  int r();
  void rMoveTo(const Point&);
  void moveTo(const Point&);
  virtual void show()=0;   // pure virtual function
private:
  int m_x, m_y;
};
#endif // __POINT_HH
```

```
#include "Shape6.hh"   // Shape6.cc

Circle::Circle(int initX, int initY, int initR)
  : Point(initX, initY), m_r(initR) {}
void Circle::print() {
  cout << "I am a circle at: ";
  Point::print();   cout << ".  radius = " << m_r; }
void Circle::show() {
  cout<<"I am a Circle at ("<<x()<<","<<y()<<")"<<endl; }
Rectangle::Rectangle(int initX, int initY, int initLX, int initLY)
  : Point(initX, initY), m_lx(initLX), m_ly(initLY) {}
void Rectangle::print() {
  cout << "I am a rectangle at: ";
  Point::print(); cout<< ".  sides = "<< m_lx<<", "<<m_ly; }
void Rectangle::show() {
  cout<<"I am a Rectangle at ("<<x()<<","<<y()<<")"<<endl; }
Square::Square(int initX, int initY, int initL)
  : Rectangle(initX, initY, initL, initL) {}
```

```
#include "Shape6.hh"   // Inheritance-6.cc

int main() {
  Circle c(3,4,6);
  c.print();
  cout << endl;
  c.moveTo(Point(7,8));
  Rectangle r(6,7, 3, 2);
  r.print();
  cout << endl;
  r.rMoveTo(Point(10,11));
  Square s(9, 3, 1);
  s.print();
  cout << endl;
  s.rMoveTo(Point(1,1));
  return 0;
}
```

- Virtual functions become even more powerful (and useful) when we use pointers.
- This feature of OOP is called *Polymorphism*