

**• Inheritance**

- Derived object “is a” base object

**• Common member function**

- Present in both base and derived

**• Which function to invoke?**

- If derived object is accessed as if it were a base-class object (upcast)

---

---

---

---

---

---

---

---

**Upcast Example**

Base function overridden in derived

```

class Sensor
{ public:
  void Calibrate ();
  . . .
};
class TempSensor : public Sensor
{ public:
  void Calibrate ();
  . . .
};
  
```

---

---

---

---

---

---

---

---

**Upcast Example**

```

void main()
{
  TempSensor ts1;
  ts1.Calibrate ();
  Sensor* p1 = &ts1;
  p1->Calibrate ();
}
  
```

Annotations:

- Declare derived object (points to `TempSensor ts1;`)
- Invoke function; calls derived (points to `ts1.Calibrate ();`)
- Reference with base pointer (points to `Sensor* p1 = &ts1;`)
- Invoke function; calls base! (points to `p1->Calibrate ();`)

---

---

---

---

---

---

---

---

Does This Seem Right?



- Only one object
- If referenced as derived
  - Calls derived version of function
- If as base (through pointer or reference)
  - Calls base version of function
- Inconsistent behavior??

---

---

---

---

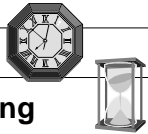
---

---

---

---

Early/Late Binding



- The problem - early binding
  - Function version linked with class
  - Function call bound at compile time
- The solution - late binding
  - Choose function at run time
  - Actual object type (dynamic type?)
    - Even if accessed through a base-class pointer or reference!

---

---

---

---

---

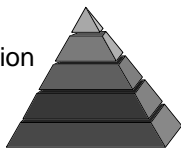
---

---

---

Virtual Functions

- Function declared in base
  - With `virtual` keyword
- Override function in derived
  - No `virtual` needed (but ok)
- Function call
  - Invokes most-derived version




---

---

---

---

---

---

---

---

Virtual Functions

Virtual function in base class

```

class Sensor
{
public:
    virtual void Calibrate ();
};
class TempSensor : public Sensor
{
public:
    void Calibrate (); ← Overridden in
                        derived class
};

```

---

---

---

---

---

---

---

---

Virtual Example

```

void main ()
{
    TempSensor ts1;
    ts1.Calibrate ();
    Sensor* p1 = &ts1;
    p1->Calibrate ();
}

```

This time, both function calls invoke the derived version of the function.

Polymorphism!

---

---

---

---

---

---

---

---

Overriding Functions

- **Behavior identical** (in all derivations)?
- **Yes: Invariant over specialization**
  - Nonvirtual function in base
  - Not in derived
    - Don't override nonvirtual functions!
- **No: Variant over specialization**
  - Virtual function in base
  - Override in derived (or use base version)

---

---

---

---

---

---

---

---

Pure Virtual Function

- **If:**
  - Common behavior (function)
  - But all derived variations unique
  - No point in implementing base function
- **Then:**
  - Declare common interface in base
  - Virtual syntax plus “= 0”
    - virtual void Calibrate()=0;

---

---

---

---

---

---

---

---

Abstract Base Classes

Page 754

- **Really a generic sensor?**
  - No, can't buy one!
- **Really only defining an interface**
  - Meant to describe common details
- **Want to prevent instantiation**
  - So no one can create object of class
  - Define pure virtual function
    - One or more (at least one)

---

---

---

---

---

---

---

---

Object Slicing Revisited

```
void main ()
{
    TempSensor ts1;
    ts1.Calibrate ();
    Sensor s2;
    s2 = ts1;
    s2.Calibrate ();
}
```

Assignment to base class copies only base part!

Result is base function call. (Also with CC pass/return by value.)

Solution: use pointer or reference

---

---

---

---

---

---

---

---

Virtual Destructors

- **Destructor clean-up for class**
  - E.g., free dynamic memory
- **What if base class pointer?**
  - Calls only base destructor
    - Not derived
- **Solution: virtual destructor!**




---

---

---

---

---

---

---

---

Virtual Destructors

Page 754

- **All destructors are called**
  - From most-derived to original base
- **Declare if any virtual function**
  - Otherwise derived destructors may not be called
    - If delete through base pointer

---

---

---

---

---

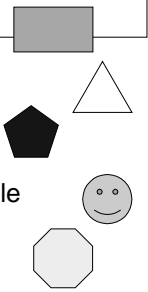
---

---

---

Another Example

- **Graphics "paint" program**
- **Base class: shape**
- **Derived classes:**
  - Line, fancy line, circle, rectangle
- **Common behavior**
  - E.g., draw on screen
- **Shape container (list<Shape\*>)**
  - Why not list<Shape>?




---

---

---

---

---

---

---

---

```
// File: shape.h
// Copyright 1997 by Mark J. Sebern, Milwaukee School of Engineering
// Definitions for "shape" classes.
// This set of classes demonstrates run-time polymorphism and virtual
// functions.
//
// Revision history:
//
// Version 1.1
// 30 April 1999
// Mark Sebern
// Updated insertion operators to current practice, avoiding "friend"
// declarations.

#ifndef SHAPE_H
#define SHAPE_H

#include <string>
using namespace std;

// Class "Point" represents a point in 2-D Cartesian space.
// It has the obvious data members and accessors/mutators.
class Point
{
public:
    Point ();
    Point (double xval, double yval);
    Point (const Point& pt);
    double GetX () const;
    double GetY () const;
    void SetX (double xval);
    void SetY (double yval);
    Point& operator= (const Point& pt);
    void Insert (ostream& os) const;
private:
    double x;
    double y;
};

// Auxiliary stream insertion function.
ostream& operator<< (ostream& os, const Point& pt);

// Class "Shape" is an abstract base class, and cannot itself be
// instantiated. It serves to provide a common interface for derived
// classes that represent different graphic shapes.
// To assist in understanding what is going on, each object derived from
// "Shape" gets a unique serial number that is generated from a static
// data member.
class Shape
{
public:
    virtual void Draw () const = 0;
    virtual Shape* MakeCopy () const = 0;
    virtual ~Shape ();
    int Num () const;
    // SetOut and Out are used to provide an output stream
    // that can be used for writing out messages from inside
    // member functions in classes derived from "Shape". This
    // is not really part of the "Shape" functionality, but
    // allows the logging of messages to screen or file so that
    // we can see what is going on inside the objects.
    // "SetOut" is used to store a pointer to an ostream and
    // "Out" retrieves a reference to that stream when it is
    // needed for output.
    static void SetOut(ostream& os);
    static ostream& Out();
protected:
    // For use by derived classes only:
    Shape ();
    Shape& operator= (const Shape&);
private:
    int shape_num;
    static int shape_count;
    static ostream* outp;
```

```
// Copy of Shapes not permitted, so make copy constructor private.
Shape (const Shape& s);
};

// The "Line" class represents a line segment. It is derived from the
// base class "Shape" and overrides the "Draw" and "MakeCopy" virtual
// functions.
class Line: public Shape
{
public:
    Line ();
    Line (const Point& p1, const Point& p2, double wid = 0.0);
    Line (const Line& ln);
    Line& operator= (const Line&);
    void SetP1 (const Point& p1);
    void SetP2 (const Point& p2);
    Point GetP1 () const;
    Point GetP2 () const;
    void SetWidth (double w);
    double GetWidth () const;
    void Draw () const;
    Shape* MakeCopy () const;
    ~Line ();
private:
    Point  pt1;
    Point  pt2;
    double width;
};

// Class "FancyLine" derives from "Line", which in turn derives from
// "Shape".
class FancyLine: public Line
{
public:
    enum LineType { SOLID, DASHED, DOTTED };
    FancyLine ();
    FancyLine (const Point& p1, const Point& p2,
               LineType lt, double wid = 0.0);
    FancyLine (const FancyLine& ln);
    FancyLine& operator= (const FancyLine&);
    LineType GetType () const;
    string GetType_name() const;
    void SetType (LineType lt);
    void Draw () const;
    Shape* MakeCopy () const;
    ~FancyLine ();
private:
    LineType  line_type;
};

// Class "Circle" is derived from "Shape".
class Circle: public Shape
{
public:
    Circle (const Point& c = Point(0,0),double r = 0.0);
    Circle (const Circle& c);
    Circle& operator= (const Circle&);
    void SetCenter (const Point& c);
    void SetRadius (double r);
    Point GetCenter () const;
    double GetRadius () const;
    void Draw () const;
    Shape* MakeCopy () const;
    ~Circle ();
private:
    Point  center;
    double radius;
};

// Class "Rectangle" is derived from "Shape".
class Rectangle: public Shape
{
public:
    Rectangle ();
    Rectangle (const Point& p1, const Point& p2);
};
```

```
Rectangle (const Rectangle& r);
Rectangle& operator= (const Rectangle&);
void SetP1 (const Point& p1);
void SetP2 (const Point& p2);
Point GetP1 () const;
Point GetP2 () const;
void Draw () const;
Shape* MakeCopy () const;
~Rectangle ();
private:
    Point    pt1;
    Point    pt2;
};

#endif /* SHAPE_H */
```

```
// File: shape.cpp
// Copyright 1997 by Mark J. Sebern, Milwaukee School of Engineering
// Implementation of "shape" classes.
//
// Revision history:
//
// Version 1.1
// 30 April 1999
// Mark Sebern
// Updated insertion operators to current practice, avoiding "friend"
// declarations.

#include <iostream>
#include <cassert>
using namespace std;

#include "shape.h"

// Point

Point::Point ()
: x(0.0), y(0.0)
{ }

Point::Point (double xval, double yval)
: x(xval), y(yval)
{ }

Point::Point (const Point& pt)
: x(pt.x), y(pt.y)
{ }

double Point::GetX () const
{
    return x;
}

double Point::GetY () const
{
    return y;
}

void Point::SetX (double xval)
{
    x = xval;
}

void Point::SetY (double yval)
{
    y = yval;
}

Point& Point::operator= (const Point& pt)
{
    x = pt.x;
    y = pt.y;
    return *this;
}

void Point::Insert (ostream& os) const
{
    os << "(" << x << ", " << y << ")";
}

ostream& operator<< (ostream& os, const Point& pt)
{
    pt.Insert(os);
    return os;
}

// Shape
int Shape::shape_count = 0;
ostream* Shape::outp = &cout;

Shape::~Shape ()
```

```

{
    Out() << "Destroying Shape " << shape_num << endl;
}

Shape::Shape ()
{
    shape_num = ++shape_count;
    Out() << "Constructing Shape " << shape_num << " ... ";
}

Shape::Shape (const Shape& s)
{
    // Private: should not be called
    shape_num = ++shape_count;
    Out() << "Copy Shape " << shape_num << " from " << s.shape_num << " ... ";
}

Shape& Shape::operator= (const Shape& s)
{
    Out() << "Assign Shape " << Num() << " from " << s.Num();
    if (this == &s) Out() << " (self)";
    Out() << endl;
    return *this;
}

int Shape::Num () const
{
    return shape_num;
}

ostream& Shape::Out()
{
    return *outp;
}

void Shape::SetOut(ostream& os)
{
    outp = &os;
}

// Line
Line::Line ()
{
    Out() << "Construct default Line from " << pt1 << " to " << pt2 << endl;
}

Line::Line (const Point& p1, const Point& p2, double wid)
    : pt1 (p1), pt2(p2), width (wid)
{
    Out() << "Construct Line from " << pt1 << " to " << pt2 << endl;
}

Line::Line (const Line& ln)
    : pt1 (ln.pt1), pt2(ln.pt2)
{
    Out() << "Copy Line {" << Num() << "} from {" << ln.Num() << "}" << endl;
}

Line& Line::operator= (const Line& ln)
{
    Out() << "Assign Line " << Num() << " from " << ln.Num();
    if (this == &ln)
    {
        Out() << " (self)" << endl;
    }
    else
    {
        Out() << " ... ";
        Shape::operator= (ln);
        pt1 = ln.pt1;
        pt2 = ln.pt2;
    }
    return *this;
}

```

```
void Line::SetP1 (const Point& p1)
{
    pt1 = p1;
}

void Line::SetP2 (const Point& p2)
{
    pt2 = p2;
}

Point Line::GetP1 () const
{
    return pt1;
}

Point Line::GetP2 () const
{
    return pt2;
}

void Line::SetWidth (double w)
{
    width = w;
}

double Line::GetWidth () const
{
    return width;
}

void Line::Draw () const
{
    Out() <<"Draw Line {"<<Num()<<} from "<< pt1;
    Out() << " to " << pt2 <<endl;
}

Shape* Line::MakeCopy () const
{
    Shape* pshape = new Line (*this);
    Out() << "... MakeCopy Line" << endl;
    return pshape;
}

Line::~Line ()
{
    Out() << "Destroy Line ... ";
}

// FancyLine

FancyLine::FancyLine () : line_type (SOLID)
{
    Out() << "Construct default FancyLine from " << GetP1();
    Out() << " to " << GetP2() << endl;
}

FancyLine::FancyLine (const Point& p1,
                      const Point& p2,
                      LineType lt,
                      double wid)
    : Line(p1,p2,wid), line_type (lt)
{
    Out() << " ... Construct FancyLine from " << GetP1() << " to " << GetP2();
    Out() << ", type = " << GetType() << endl;
}

FancyLine::FancyLine (const FancyLine& ln)
    : Line(ln), line_type (ln.line_type)
{
    Out()<<"... Copy FancyLine {"<<Num()<<} from {"<<ln.Num()<<}"<<endl;
}

FancyLine& FancyLine::operator= (const FancyLine& ln)
{
    Out() << "Assign FancyLine " << Num() << " from " << ln.Num();
}
```

```

    if (this == &ln)
    {
        Out() << " (self)" << endl;
    }
    else
    {
        Out() << " ... ";
        Line::operator= (ln);
        line_type = ln.line_type;
    }
    return *this;
}

FancyLine::LineType FancyLine::GetType () const
{
    return line_type;
}

string FancyLine::GetType_name () const
{
    string line_type_name;
    switch (line_type)
    {
    case SOLID:
        line_type_name = "Solid";
        break;
    case DASHED:
        line_type_name = "Dashed";
        break;
    case DOTTED:
        line_type_name = "Dotted";
        break;
    }
    return line_type_name;
}

void FancyLine::SetType (LineType lt)
{
    line_type = lt;
}

void FancyLine::Draw () const
{
    Out() << "Draw FancyLine {" << Num() << "} from " << GetP1();
    Out() << " to " << GetP2();
    Out() << ", type = " << GetType_name() << endl;
}

Shape* FancyLine::MakeCopy () const
{
    Shape* pshape = new FancyLine (*this);
    Out() << "... MakeCopy FancyLine" << endl;
    return pshape;
}

FancyLine::~FancyLine ()
{
    Out() << "Destroy FancyLine ... ";
}

// Circle
Circle::Circle (const Point& c, double r)
: center (c), radius (r)
{
    Out() << "Construct Circle: center = " << center;
    Out() << ", radius = " << radius << endl;
}

Circle::Circle (const Circle& c)
: center (c.center), radius (c.radius)
{
    Out() << "Copy Circle {" << Num() << "} from {" << c.Num() << "}" << endl;
}

```

```
void Circle::SetCenter (const Point& c)
{
    center = c;
}

void Circle::SetRadius (double r)
{
    radius = r;
}

Point Circle::GetCenter () const
{
    return center;
}

double Circle::GetRadius () const
{
    return radius;
}

void Circle::Draw () const
{
    Out() << "Draw Circle {" << Num() << "} at " << center;
    Out() << ", radius " << radius << endl;
}

Shape* Circle::MakeCopy () const
{
    Shape* pshape = new Circle (*this);
    Out() << "... MakeCopy Circle" << endl;
    return pshape;
}

Circle::~Circle ()
{
    Out() << "Destroy Circle ... ";
}

// Rectangle

Rectangle::Rectangle ()
{
    Out() << "Construct default Rectangle from " << pt1 << " to " << pt2 << endl;
}

Rectangle::Rectangle (const Point& p1, const Point& p2) : pt1 (p1), pt2(p2)
{
    Out() << "Construct Rectangle from " << pt1 << " to " << pt2 << endl;
}

Rectangle::Rectangle (const Rectangle& r) : pt1 (r.pt1), pt2(r.pt2)
{
    Out() << "Copy Rectangle {" << Num() << "} from {" << r.Num() << "}"<< endl;
}

void Rectangle::SetP1 (const Point& p1)
{
    pt1 = p1;
}

void Rectangle::SetP2 (const Point& p2)
{
    pt2 = p2;
}

Point Rectangle::GetP1 () const
{
    return pt1;
}

Point Rectangle::GetP2 () const
{
    return pt2;
}
```

```
void Rectangle::Draw () const
{
    Out() << "Draw Rectangle {" << Num() << "} from " << pt1 << " to " << pt2 << endl;
}

Shape* Rectangle::MakeCopy () const
{
    Shape* pshape = new Rectangle (*this);
    Out() << "... MakeCopy Rectangle" << endl;
    return pshape;
}

Rectangle::~Rectangle ()
{
    Out() << "Destroy Rectangle ... ";
}
```

```
// File: shptest.cpp
// Copyright 1997 by Mark J. Sebern, Milwaukee School of Engineering
// Test operation of shape class hierarchy.
// Demonstrates run-time polymorphism and virtual functions.

#include <cassert>
#include <iostream>
#include <fstream>
#include <list>
using namespace std;
#include "shape.h"

void main ()
{
    ofstream outfile ("shptest.txt");
    assert (outfile);
    Shape::SetOut(outfile); // Set Shape log output file

    list<Shape*>    ptr_list;

    outfile << "Creating shapes." << endl;

    ptr_list.push_back (new Circle (Point(1,3),2));
    ptr_list.push_back (new Line (Point(0,0), Point (3,4.5)));
    ptr_list.push_back (new Rectangle (Point(1,2),Point(5,10)));
    ptr_list.push_back (new FancyLine (Point(2,5),Point(3,6),FancyLine::DASHED));
    outfile << endl;

    outfile << "Test object creation/assignment";
    outfile << endl;
    FancyLine line1 (Point (99,88),Point (-10,-2),FancyLine::DOTTED);
    FancyLine line2;
    line2 = line1;
    outfile << "Test self-assignment:" << endl;
    line2 = line2;

    // MakeCopy is a "virtual copy constructor".
    // When we reference "line1" through a "Shape*" pointer, we don't
    // directly know what type of shape it actually is. The virtual
    // function "MakeCopy" is overridden in all derived classes. When
    // called for an object, it makes another object just like it, and
    // returns a base class pointer (Shape*).
    Shape* sp1 = &line1;
    ptr_list.push_back (sp1->MakeCopy());
    outfile << endl;

    // Now "draw" the shapes in the list. We can't actually draw, but the
    // "Draw" virtual member function writes out information that would be
    // used by an actual drawing program.
    outfile << "Drawing list of shapes." << endl;

    list<Shape*>::const_iterator iter1;
    for (iter1 = ptr_list.begin(); iter1 != ptr_list.end(); iter1++)
    {
        Shape* ptr = *iter1;
        ptr->Draw();
    }

    outfile << endl;
    outfile << "Deleting shapes from list." << endl;
    // Now empty list, deleting the shapes referenced by
    // each pointer removed from the list.
    // Note that the list destructor would free the pointers,
    // but will not release the shapes that the pointers
    // point to. So we have to do it explicitly.
    while (ptr_list.size() > 0)
    {
        Shape* p = ptr_list.front();
        ptr_list.pop_front();
        delete p;
    }
}
```

Creating shapes.

```
Constructing Shape 1 ... Construct Circle: center = (1,3), radius = 2
Constructing Shape 2 ... Construct Line from (0,0) to (3,4.5)
Constructing Shape 3 ... Construct Rectangle from (1,2) to (5,10)
Constructing Shape 4 ... Construct Line from (2,5) to (3,6)
... Construct FancyLine from (2,5) to (3,6), type = Dashed
```

Test object creation/assignment

```
Constructing Shape 5 ... Construct Line from (99,88) to (-10,-2)
... Construct FancyLine from (99,88) to (-10,-2), type = Dotted
Constructing Shape 6 ... Construct default Line from (0,0) to (0,0)
Construct default FancyLine from (0,0) to (0,0)
Assign FancyLine 6 from 5 ... Assign Line 6 from 5 ... Assign Shape 6 from 5
Test self-assignment:
Assign FancyLine 6 from 6 (self)
Constructing Shape 7 ... Copy Line {7} from {5}
... Copy FancyLine {7} from {5}
... MakeCopy FancyLine
```

Drawing list of shapes.

```
Draw Circle {1} at (1,3), radius 2
Draw Line {2} from (0,0) to (3,4.5)
Draw Rectangle {3} from (1,2) to (5,10)
Draw FancyLine {4} from (2,5) to (3,6), type = Dashed
Draw FancyLine {7} from (99,88) to (-10,-2), type = Dotted
```

Deleting shapes from list.

```
Destroy Circle ... Destroying Shape 1
Destroy Line ... Destroying Shape 2
Destroy Rectangle ... Destroying Shape 3
Destroy FancyLine ... Destroy Line ... Destroying Shape 4
Destroy FancyLine ... Destroy Line ... Destroying Shape 7
Destroy FancyLine ... Destroy Line ... Destroying Shape 6
Destroy FancyLine ... Destroy Line ... Destroying Shape 5
```