Polymorphism

Much of this material taken from Horstmann, Ch. 8.

I. Let's say we want to make a FilledRectangle class to use in making graphs such as

We derive from the existing Rectangle class, giving us:

```cpp
class FilledRect : public Rectangle {
public:
    FilledRect( Point, Point, GraphicsContext::Brushstyle);
    void plot( Graphicscontext& gc) const;
private:
    GraphicsContext::Brushstyle _pattern;
};
```

where BrushStyle is an enumerated type defined in the GraphicsContext module. The graph above might be given by the code:

```cpp
FilledRect chart1 (Point(1.0, 7.0), Point(2.0, 2.0), GraphicsContext::FHORIZ_BRUSH);
FilledRect chart2 (Point(2.0, 7.0), Point(3.0, 1.0), GraphicsContext::HATCHED_BRUSH);
FilledRect chart3 (Point(3.0, 7.0), Point(4.0, 3.0), GraphicsContext::LDIAG_BRUSH);
Rectangle chart4 (Point(0.5, 0.5), Point(4.5, 7.5));
chart1.plot( gc); chart2.plot( gc);
chart3.plot( gc); chart4.plot( gc);
```

The problem is that this isn’t very flexible, since we don’t know at compile time necessarily how many filled rectangles we’ll need, and how many "regular" rectangles.

II. Conversion between Base and Derived-Class Objects
   A. Below we can see the data layouts for a plain Rectangle and a FilledRect object:

   ![Data Layouts](image)

   B. Note that derived-class objects have at least as many data items as the base class, since fields can only be added, not taken away in the derivation process.

   C. A derived class object can be assigned to a base class object, however some information is lost during the assignment:

   ```cpp
   FilledRect d( ulPt, lrPt, GraphicsContext::HORIZ_BRUSH);
   Rectangle b = d; //OK, but information lost
   b.plot( gc);    //plots as Rectangle only
   ```

   Similarly info. is lost when a derived class object is sent as a parameter where a base-class object is expected.
D. A base class object cannot be assigned to a derived-class object

```cpp
Rectangle b(ulPt, lrPt);
FilledRect d = b;       // Error
```

E. Let's say we wanted to keep track of various FilledRect and Rectangle objects in an array so that we can later go through the array and print each of the objects. This might look like:

```cpp
array<Rectangle> chart;     // Array template
chart.grow( 1,4);           // Set size and subscripts
chart[1] = FilledRectPoint(1.0, 7.0), Point( 2.0, 2.0),
          GraphicsContext::PHORIZ_BRUSH);
chart[2] = FilledRect(Point(2.0, 7.0), Point( 3.0, 1.0),
          GraphicsContext::HATCHED_BRUSH);
chart[3] = FilledRectPoint(3.0, 7.0), Point( 4.0, 3.0),
          GraphicsContext::LDIAG_BRUSH);
chart[4] = Rectangle(Point(0.5, 0.5), Point( 4.5, 7.5));
```

where plotting these objects is attempted by:

```cpp
for (int i = chart.low(); i <= chart.high(); i++)
  chart[i].plot(gc);
```

The problem is that all objects are truncated to the base class. Before seeing how we can keep track of instances of different related classes, we must understand pointers to objects.

III. Pointers to Objects

A. Pointers are a mechanism to share values. Objects in C++ are normally copied by value, where each copy would have its own distinct identity.

B. We can declare pointers to objects by following the type (class name) with an asterisk as in:

```cpp
Employee* pe;
```

Note that just as in C, multiple pointer declarations are not written as:

```cpp
Employee* pe, pf;
```

But rather as

```cpp
Employee *pe, *pf;
```

C. There are three methods for obtaining pointer values

1. The & operator returns a pointer to an existing object.
2. The new operator creates an object on the heap and returns a pointer to it.
3. The value 0 denotes a pointer value that currently points to no object.
4. Examples:

```cpp
Employee dale("Dale Reed");
Employee* pd = &dale;
Employee* pg = new Employee("Guillermo Sanchez");
Employee* pn = 0;
```

D. Dereferencing

Applying the * unary operator transforms a pointer into the value to which it points

```cpp
Employee e = *pg;
```

and is known as dereferencing the pointer.
E. Copying a pointer gives two access paths to the same object:

```cpp
Employee* p = new Employee("Guillermo Sanchez");
Employee* q = p;
(*p).set_salary(40000); // Sets Guillermo’s salary
(*q).raise_salary(.10); // raises Guillermo’s salary
```

Note the parenthesis in `(*p).set_salary` are necessary because otherwise "." has precedence over ".". Since dereference and access field can be combined into ":->" , this could be written as

```cpp
p->set_salary(40000);
```

Remember that set_salary here has two arguments: the explicit argument 40000 and the implicit argument *p.

IV. Conversion between base and derived-class pointers

A. Given the structure of Rectangle and FilledRect we saw before:

A pointer to a derived-class object can be converted to a base class pointer:

```cpp
FilledRect* pd = new FilledRect(ul, ur, Graphicscontext::HORIZ_BRUSH);
Rectangle* pb = pd;
```

Note that pb, however, can only access those fields present in the base class

B. If we’ve already converted a derived class pointer into a base class pointer, we can again convert it back using a cast, such as in:

```cpp
FilledRect* pd = ...;
Rectangle* pb = pd; // convert to base class pointer
//...
FilledRect* pe = (FilledRect*) pb; //OK
```

C. By using base-class pointers to keep track of various related objects, we can do use a single array to keep track of these objects as we had attempted before. E.g.

```cpp
array<Rectangle*> chart; // Array template for array of pointers
chart.grow( 1,4); // Set size and subscripts
chart[1] = new FilledRectPoint(1.0, 7.0), Point( 2.0, 2.0),
      GraphicsContext::FHORIZ_BRUSH);
chart[2] = new FilledRect(Point(2.0, 7.0), Point( 3.0, 1.0),
      GraphicsContext::HATCHED_BRUSH);
chart[3] = new FilledRectPoint(3.0, 7.0), Point( 4.0, 3.0),
      GraphicsContext::LDIAG_BRUSH);
chart[4] = new Rectangle(Point(0.5, 0.5), Point( 4.5, 7.5));
for (int i = chart.low(); i <= chart.high(); i++)
  chart[i]->plot(gc); //used ":->" rather than "."
```

1. Unfortunately all the rectangles when plotted appear the same (blank). The compiler translates the call chart[i]->plot(gc) into a call to Rectangle::plot, because the type of each chart[i] is statically determined to be Rectangle.
2. We would like to select the appropriate plot function dynamically at run-time. This is made possible in C++ using what are called virtual functions.
V. Dynamic Binding
A. Static & Dynamic binding
1. To select the correct version of \textit{plot} at run-time we make \textit{plot} into a \textit{virtual function}. The compiler translates a virtual function call into code that selects the correct operation at run time.
   
   E.g. if \textit{plot} is a virtual function, then the code
   
   \begin{verbatim}
   chart[i]->plot(gc);
   \end{verbatim}
   
   will call the correct version of \textit{plot} either Rectangle::Plot or FilledRect::plot, depending on the actual contents of chart[i]

2. This is referred to as \textit{dynamic binding}, which is different from \textit{static} binding

3. How does the compiler know whether to use dynamic or static binding? It depends on the type of the object. In the call \texttt{chart[i]->plot(gc)} the type of \texttt{chart[i]} is \texttt{Rectangle*}.
   
   a. If \textit{plot} is a nonvirtual operation, it is bound statically to Rectangle::plot
   
   b. If \textit{plot} is a virtual operation, then either Rectangle::plot or FilledRect::plot is invoked, depending on the type of \texttt{chart[i]}.

B. Declaring virtual functions
1. To enable dynamic binding, the operation must be declared as \textit{virtual} in the base class.
   
   \begin{verbatim}
   class Rectangle {
       public:
       Rectangle( Point, Point);
       virtual void plot(GraphicsContext&) const;
       //...
   }
   \end{verbatim}

   The keyword \textit{virtual} is not replicated in the function definition.

2. In the derived-class declaration, the keyword \textit{virtual} is optional, however it is considered good style to provide it as documentation.
   
   \begin{verbatim}
   class FilledRect : public Rectangle {
       public:
       FilledRect( Point, Point, FillPattern);
       virtual void plot(GraphicsContext&) const;
       //...
   }
   \end{verbatim}

   Once a function is \textit{virtual}, it remains \textit{virtual} in all derived classes.

C. Recognizing Dynamic Binding
1. If an \textit{object} invokes an operation, the call is always statically bound
   
   \begin{verbatim}
   Rectangle r;
   r.plot(gc);       // static binding, calls Rectangle::plot
   \end{verbatim}

2. If a \textit{pointer} invokes an operation, the binding depends on the nature of the operation. If the operation has been declared \textit{virtual}, then the binding is dynamic, otherwise it is static.
   
   \begin{verbatim}
   Rectangle *r;
   r->plot(gc);      // dynamic binding, calls Rectangle::plot
   // or FilledRect::plot
   r->scale( p, 0.9);  // static binding to Rectangle::scale
   \end{verbatim}

   In the above example we assume \textit{scale} is not virtual.
3. If a virtual function is invoked on the *implicit argument* of a class operation, the binding is also dynamic. E.g.

```cpp
class Window {
public:
    virtual void paint( Rectangle r); //Repaint window inside r
    void scroll( double dx, double dy);
    Rectangle size() const;
    //...
};
```

```cpp
void Window::scroll( double dx, double dy)
{
    if (dy > 0) {    // scroll window contents
        Rectangle s = size(); //static binding
        Point p = s.left_top();
        Rectangle r(p, Point( s.xright(), p.y() + dy));
        paint(r);    // dynamic binding
    }
    else
        //...
}
```

Note the call `paint(r)`.

a. If the object invoking scroll is a plain Window, then `Window::paint` is called.
b. If `GraphWindow` is derived from Window, and thus inherits `Window::scroll`, then the code:

```cpp
GraphWindow gw;
gw.scroll(0,dy);    //dynamic binding, calls `Window::scroll`
```

invokes `GraphWindow::paint`

4. Only if no virtual ancestor is found anywhere can we be assured that `paint` is not a virtual function.

VI. Polymorphism

A. Imagine that we not only have Rectangles and Filled Rectangles, but also lines, circles, text, etc. as part of a complete chart. We might store the various elements of the chart in an array, and then plot the array using

```cpp
for ( int i = chart.low(); i <= chart.high(); i++)
    chart[i]->plot(gc);
```

and scale the chart using

```cpp
for ( int i = chart.low(); i <= chart.high(); i++)
    chart[i]->scale(c, 0.5);
```

The problem is, what sort of type is `chart[i]`? Its type needs to be a pointer to a class that supports virtual functions `plot` and `scale`. This type must be the *lowest common denominator* of the objects we wish to display and scale.

B. We call this new class `Shape`

```cpp
class Point;    //forward declaration (prototype)

class Shape {
public:
    virtual void plot(GraphicsContext& gc) const;
    virtual void scale(Point center, double s);
    //...
};
```
where all other shape classes derive from it:

```cpp
class Point : public Shape { ... };
class Rectangle : public Shape { ... };
class FilledRect : public Rectangle { ... };
```

Each of these classes would then have its own versions of `plot` and `scale` functions appropriate to its type. The complete inheritance hierarchy is shown below:

![Inheritance Hierarchy Diagram]

and the entire chart can be plotted using

```cpp
Array<Shape*> chart;
//... construct chart here
for (int i = chart.low(); i <= chart.high(); i++)
    chart[i]->plot(gc);  
```
VII. Additional Example
(This example is taken from Winston, P.H. *On To C++*. Addison Wesley, 1995, Section 29.)

A. Problem Description: You want to write a program that classifies railroad cars as they come into the railroad switching yard. There is a sensor by the track that can read a number associated with each car as it comes, so sensor readings for 5 cars of the integers:

```
0 1 1 2 3
```

might correspond to:

- engine
- boxcar
- boxcar
- tank
- caboose

B. The program to do this simply could be given as:

```c++
#include <iostream.h>
main ( ) {
    int type_code;
    while (cin >> type_code)
        if (type_code == 0)        cout << "engine" << endl;
        else if (type_code == 1)   cout << "boxcar" << endl;
        else if (type_code == 2)   cout << "tank" << endl;
        else if (type_code == 3)   cout << "caboose" << endl;
}
```

C. Since it is likely that much more data will be gathered and much more analysis will be done, you decide to represent railroad cars as objects in a hierarchy as:

```
railroad_car
   ↓   ↓   ↓   ↓
boxcar  tank  engine  caboose
```

For the sake of simplicity for now, we will assume that each of these derived classes only have a single constructor that does nothing.

D. Our first inclination to accumulate information for the cars on a train is to use some sort of array to hold engines, tanks, boxcars, and cabooses. This doesn’t work, however, since an array must hold objects of the same type so that the compiler knows how to statically allocate space for the array.

E. We can, however, use an array of pointers to boxcars to store the information, such as

```c++
railroad_car *train[ 100];
```

where each pointer points to a railroad_car that is also either a boxcar, tank, engine, or caboose. (For now assume trains are of length 100 cars or less.) For instance as the program runs, the first car of the train might be a caboose, identified by the sensor with the integer 3. This could be stored in the array as

```
train[0] = new caboose;
```

Note that train[0] is a railroad_car*, although the memory it points to is actually a caboose*. 

Polymorphism / page 7
F. In this way we can write a program that contains pointers to the various types of cars as they come in:

```cpp
#include <iostream.h>

class railroad_car {  
  public: railroad_car ( ) { } 
};
class box_car : public railroad_car {  
  public: box_car ( ) { } 
};
class tank_car : public railroad_car {  
  public: tank_car ( ) { } 
};
class engine : public railroad_car {  
  public: engine ( ) { } 
};
class caboose : public railroad_car {  
  public: caboose ( ) { } 
};

// Define railroad car array:
railroad_car *train[100];

main ( ) {
  int car_count, type_code;  // Declare various integer variables

  // Read type number and create corresponding objects:
  for (car_count = 0; cin >> type_code; ++car_count)
    if (type_code == 0)       train[car_count] = new engine;
    else if (type_code == 1)   train[car_count] = new box_car;
    else if (type_code == 2)   train[car_count] = new tank_car;
    else if (type_code == 3)   train[car_count] = new caboose;
  // Display car count:
  cout << "There are " << car_count << " cars in the array." << endl;
}
```

Now we need to store the name of the car somewhere so that we can iterate through the array of cars, printing out the name of each as we go. We can think of two approaches:

1. We could add a private field called `name` to the railroad_car base class, along with corresponding accessor and mutator functions.
2. We could choose to not store the information, but instead add a `display_short_name` member function to the derived classes. E.g.
   ```cpp
   class box_car : public railroad_car {  
     public:  
       box_car ( ) { } // Constructor  
       virtual void display_short_name ( ) { cout << "boxcar"; }  
   };
   ```

   We choose this second approach, adding similar `display_short_name` member functions to tank_car, engine, and caboose classes.
G. With this done, we now attempt to display the names using a for loop

```cpp
for (n = 0; n < car_count; n++) {
    train[n]->display_short_name( );
    cout << endl;
}
```

The C++ compiler rejects this, however. The `train` array is supposed to contain pointers to `railroad_car` objects, but there is no `display_short_name` function defined for the `railroad_car` class. At compile time the compiler has no way of knowing that the `train` array may actually point to subclass objects.

H. We could attempt to add a `display_short_name` function to the `railroad_car` class, such as

```cpp
class railroad_car {
public:
    railroad_car() { }
    void display_short_name() { cout << "Railroad Car "; }
};
```

but now the output would be

```
Railroad Car
Railroad Car
Railroad Car
Railroad Car
Railroad Car
```

since at compile time the compiler decides that all array elements belong to the `railroad_car` class.

I. The method used to tell the compiler that the member function selection should be done at runtime (dynamically) rather than at compile time (statically) is by declaring the function to be `virtual` (see more detailed comments in previous pages of notes). Another way to think of a virtual function is that it is a "multiversion function for which the version to be called is determined by object class at run time".
J. This then gives us the following:

```cpp
#include <iostream.h>

class railroad_car {
public:
    railroad_car ( ) { }
    virtual void display_short_name ( ) {cout << "Railroad Car";}
};
class box_car : public railroad_car {
public:
    box_car ( ) { }
    virtual void display_short_name ( ) {cout << "boxcar";}
};
class tank_car : public railroad_car {
public:
    tank_car ( ) { }
    virtual void display_short_name ( ) {cout << "tank";}
};
class engine : public railroad_car {
public:
    engine ( ) { }
    virtual void display_short_name ( ) {cout << "engine";}
};
class caboose : public railroad_car {
public:
    caboose ( ) { }
    virtual void display_short_name ( ) {cout << "caboose";}
};

// Define railroad car pointer array:
railroad_car *train[100];

main ( ) {
    int n, car_count, type_code; // Declare various integer variables

    // Read car-type number and create car class objects:
    for (car_count = 0; cin >> type_code; ++car_count)
        if (type_code == 0) train[car_count] = new engine;
        else if (type_code == 1) train[car_count] = new box_car;
        else if (type_code == 2) train[car_count] = new tank_car;
        else if (type_code == 3) train[car_count] = new caboose;

    // Display classes using display_short_name virtual function:
    for (n = 0; n < car_count; ++n)
        (train[n] -> display_short_name ( )); cout << endl;
}
```

where the input of

0 1 1 2 3

gives the output:

engine
boxcar
boxcar
tank
caboose

K. Note that if you accidentally leave off the display_short_name member function from a derived class, such as engine, then that function call will match on that of the base class, displaying "Railroad Car" rather than "engine" in the output. To force an error message in this scenario, we can make the base-class version into a pure virtual function.
L. A \textit{pure virtual function} is one that is needed so that derived classes can "shadow" this base class virtual function.

1. A pure virtual function should never be called, however, and in fact should signal a compiler error when it is called.
2. Pure virtual functions are just like "regular" virtual functions, except the body is replaced by "= 0".
3. For example:

\begin{verbatim}
class railroad_car {
public:
    railroad_car ( ) { } // virtual void display_short_name ( ) = 0;
};
\end{verbatim}
VIII. Additional Example [See separate overheads]
(This example is taken from Deitel & Deitel, Ch. 20.)

A. Type fields & the *switch* statement
   1. Rather than use inheritance & polymorphism to call the correct version of a function, we could store the type of the object in a base-class private field, then use a *switch* statement to explicitly call the appropriate function.
   2. This approach is error-prone, as switch conditions may be inadvertently left out.
   3. This approach also requires recompiling every time a new condition (class) is added.
   4. Polymorphism promotes extensibility (Deitel, p. 773): Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be added into such a system without modifying the base system. Except for client code that instantiates new objects, programs need not be recompiled (but must be relinked).

B. Abstract Base classes and Concrete Classes
   1. In an inheritance hierarchy there are situations where a class is used only by inheritance, and is never directly instantiated itself. Such a class is called an *abstract base class*.
      Attempting to instantiate this class results in an error
      a. A class with virtual functions is made *abstract* by declaring one or more of its functions to be *pure* (by giving an initializer in the body of "= 0").
      b. If a class is derived from a class with a pure virtual function, and if no definition is supplied for that pure virtual function in the derived class, then that virtual function remains pure in the derived class. Consequently the derived class is also an *abstract* class.
      c. Good object-oriented systems tend to have class hierarchies headed by an abstract base class, such as a hierarchy of geometric objects, where we might have the abstract head base class *Shape*, which might be inherited by the abstract classes *TwoDimensionalShape* and *ThreeDimensionalShape*, which in turn might be inherited by concrete classes such as *circle* and *sphere*.
   2. Classes from which objects can be instantiated are called *concrete classes*.

C. Example of a Payroll System
   1. Base class *Employee* has first & last names, & pure virtual functions, making it an *abstract* class. Note that the virtual functions have no definition, only a declaration.
   2. Derived class *HourlyWorker* inherits *Employee* and has wage/hr. and hours/week. It also implements virtual functions *earnings()* and *print()*.
   3. Derived class *Boss* inherits *Employee* and has weeklySalary. It also implements virtual functions *earnings()* and *print()*.
   4. The driver shows how instances referring to the *earnings()* and *print()* functions can be bound statically or dynamically.

D. Mechanism for implementing routing of virtual function calls
   1. A *virtual function table* called the *vtable* is an array with function pointers
   2. The *vtable* points to each accessible member function, be it in the referent’s class or in some ancestor class

E. Virtual Destructors
   1. Given some array of base-class pointers, when the destructor is called for this array, the base-class destructor will be called regardless of the type of object.
   2. To avoid this problem, destructors may also be made *virtual*, so that the appropriate destructor is called for each object.