# Chapter 10- Virtual Functions and Polymorphism

**Outline**

**10.1 Introduction**
**10.2 Type Fields and `switch` Statements**
**10.3 Virtual Functions**
**10.4 Abstract Base Classes and Concrete Classes**
**10.5 Polymorphism**
**10.6 Case Study: A Payroll System Using Polymorphism**
**10.7 New Classes and Dynamic Binding**
**10.8 Virtual Destructors**
**10.9 Case Study: Inheriting Interface and Implementation**
**10.10 Polymorphism, `virtual` Functions and Dynamic Binding “Under the Hood”**

# 10.1 Introduction

- **`virtual`** functions and polymorphism
  - Design and implement systems that are more easily extensible
  - Programs written to generically process objects of all existing classes in a hierarchy

# 10.2 Type Fields and `switch` Statements

- **`switch`** statement
  - Take an action on a object based on its type
  - A switch structure could determine which **`print`** function to call based on which type in a hierarchy of shapes
- Problems with **`switch`**
  - Programmer may forget to test all possible cases in a **`switch`**
    - Tracking this down can be time consuming and prone to error
  - **`virtual`** functions and polymorphic programming can eliminate the need for **`switch`** logic

# 10.3 `virtual` Functions

- **`virtual`** functions
  - Suppose a set of shape classes such as **`Circle`**, **`Triangle`**, etc.
  - Every shape has own unique draw function but possible to call them by calling the **`draw`** function of base class **`Shape`**
    - Compiler determines dynamically (i.e., at run time) which to call
  - In base-class declare **`draw`** to be **`virtual`**
  - Override **`draw`** in each of the derived classes
  - **`virtual`** declaration:
    - Keyword **`virtual`** before function prototype in base-class

```
virtual void draw() const;
```

  - A base-class pointer to a derived class object will call the correct **`draw`** function

```
Shape->draw();
```

  - If a derived class does not define a **`virtual`** function, the function is inherited from the base class

# 10.3 Virtual Functions

```
ShapePtr->Draw();
```

- Compiler implements dynamic binding
- Function determined during execution time

```
ShapeObject.Draw();
```

- Compiler implements static binding
- Function determined during compile-time

◀ ▶

# 10.4 Abstract and Concrete Classes

- Abstract classes
  - Sole purpose is to provide a base class for other classes
  - No objects of an abstract base class can be instantiated
    - Too generic to define real objects (i.e., **`TwoDimensionalShape`**)
    - Can have pointers and references
- Concrete classes
  - Classes that can instantiate objects
  - Provide specifics to make real objects (i.e., **`Square`**, **`Circle`**)
- Making abstract classes
  - Declare one or more **`virtual`** functions as "pure" by initializing the function to zero
  - Example of a pure **`virtual`** function:

```
virtual double earnings() const = 0;
```

# 10.5 Polymorphism

- Polymorphism
  - Ability for objects of different classes to respond differently to the same function call
  - Implemented through **`virtual`** functions
    - Base-class pointer (or reference) calls a **`virtual`** function
    - C++ chooses the correct overridden function in object
  - If non-**`virtual`** member function defined in multiple classes and called from base-class pointer then the base-class version is used
    - If called from derived-class pointer then derived-class version is used
  - Suppose **`print`** is not a **`virtual`** function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();    // call base-class print function
hPtr->print();    // call derived-class print
function
ePtr=&h;          // allowable implicit conversion
ePtr->print();    // still calls base-class print
```

# 10.6 Case Study: A Payroll System Using Polymorphism

- The following example is a payroll system
  - Uses **`virtual`** functions and polymorphism to perform payroll calculations based on the type of an employee

```
// Fig. 10.1: employ2.h
// Abstract base class Employee
#ifndef EMPLOY2_H
#define EMPLOY2_H

class Employee {
public:
   Employee( const char *, const char * );
   ~Employee();   // destructor reclaims memory
   const char *getFirstName() const;
   const char *getLastName() const;

   // Pure virtual function makes Employee abstract base class
   virtual double earnings() const = 0;   // pure virtual
   virtual void print() const;            // virtual
private:
   char *firstName;
   char *lastName;
};

#endif
```

**1. Employee Definition (base class)**

**`earnings`** is declared pure **`virtual`** because the implementation will depend on which derived class it will be used in.

**`Employee`** is an abstract base class.

```
// Fig. 10.1: employ2.cpp
// Member function definitions for
// abstract base class Employee.
// Note: No definitions given for pure virtual functions.
#include <iostream>

using std::cout;

#include <cstring>
#include <cassert>
#include "employ2.h"

// Constructor dynamically allocates space for the
// first and last name and uses strcpy to copy
// the first and last names into the object.
Employee::Employee( const char *first, const char *last )
{
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );    // test that new worked
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );     // test that new worked
   strcpy( lastName, last );
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   delete [] firstName;
   delete [] lastName;
}
```

**1.1 Function Definitions**

## 1.1 Function Definitions

```

// Return a pointer to the first name
// Const return type prevents caller from modifying private
// data. Caller should copy returned string before destructor
// deletes dynamic storage to prevent undefined pointer.
const char *Employee::getFirstName() const
{
   return firstName;   // caller must delete memory
}

// Return a pointer to the last name
// Const return type prevents caller from modifying private
// data. Caller should copy returned string before destructor
// deletes dynamic storage to prevent undefined pointer.
const char *Employee::getLastName() const
{
   return lastName;   // caller must delete memory
}

// Print the name of the Employee
void Employee::print() const
   { cout << firstName << ' ' << lastName; }
```

```
// Fig. 10.1: boss1.h
// Boss class derived from Employee
#ifndef BOSS1_H
#define BOSS1_H
#include "employ2.h"

class Boss : public Employee {
public:
   Boss( const char *, const char *, double = 0.0 );
   void setWeeklySalary( double );
   virtual double earnings() const;
   virtual void print() const;
private:
   double weeklySalary;
};

#endif
```

**1. Boss Definition (derived class)**

**1.1 Function Definitions**

```
// Fig. 10.1: boss1.cpp
// Member function definitions for class Boss
#include <iostream>

using std::cout;

#include "boss1.h"

// Constructor function for class Boss
Boss::Boss( const char *first, const char *last, double s )
   : Employee( first, last )  // call base-class constructor
{ setWeeklySalary( s ); }

// Set the Boss's salary
void Boss::setWeeklySalary( double s )
   { weeklySalary = s > 0 ? s : 0; }

// Get the Boss's pay
double Boss::earnings() const { return weeklySalary; }

// Print the Boss's name
void Boss::print() const
{
   cout << "\n             Boss: ";
   Employee::print();
}
```

Notice the overriden **`earnings`** and **`print`** functions.

They were declared **`virtual`** in the base class.

```
// Fig. 10.1: commis1.h
// CommissionWorker class derived from Employee
#ifndef COMMIS1_H
#define COMMIS1_H
#include "employ2.h"

class CommissionWorker : public Employee {
public:
   CommissionWorker( const char *, const char *,
                     double = 0.0, double = 0.0,
                     int = 0 );
   void setSalary( double );
   void setCommission( double );
   void setQuantity( int );
   virtual double earnings() const;
   virtual void print() const;
private:
   double salary;          // base salary per week
   double commission;      // amount per item sold
   int quantity;           // total items sold for week
};

#endif
```

**1. CommissionWorker Definition (derived class)**

```cpp
// Fig. 10.1: commis1.cpp
// Member function definitions for class CommissionWorker
#include <iostream>

using std::cout;

#include "commis1.h"

// Constructor for class CommissionWorker
CommissionWorker::CommissionWorker( const char *first,
        const char *last, double s, double c, int q )
   : Employee( first, last )  // call base-class constructor
{
   setSalary( s );
   setCommission( c );
   setQuantity( q );
}

// Set CommissionWorker's weekly base salary
void CommissionWorker::setSalary( double s )
   { salary = s > 0 ? s : 0; }
```

**1.1 Function Definitions**

### 1.1 Function Definitions

```

// Set CommissionWorker's commission
void CommissionWorker::setCommission( double c )
   { commission = c > 0 ? c : 0; }

// Set CommissionWorker's quantity sold
void CommissionWorker::setQuantity( int q )
   { quantity = q > 0 ? q : 0; }

// Determine CommissionWorker's earnings
double CommissionWorker::earnings() const
   { return salary + commission * quantity; }

// Print the CommissionWorker's name
void CommissionWorker::print() const
{
   cout << "\nCommission worker: ";
   Employee::print();
}
```

Notice the overriden **`earnings`** and **`print`** functions.

They were declared **`virtual`** in the base class.

```
// Fig. 10.1: piece1.h
// PieceWorker class derived from Employee
#ifndef PIECE1_H
#define PIECE1_H
#include "employ2.h"

class PieceWorker : public Employee {
public:
   PieceWorker( const char *, const char *,
                double = 0.0, int = 0);
   void setWage( double );
   void setQuantity( int );
   virtual double earnings() const;
   virtual void print() const;
private:
   double wagePerPiece; // wage for each piece output
   int quantity;        // output for week
};

#endif
```

## Outline

**1. `PieceWorker` Definition (derived class)**

## 1.1 Function Definitions

```cpp
// Fig. 10.1: piece1.cpp
// Member function definitions for class PieceWorker
#include <iostream>

using std::cout;

#include "piece1.h"

// Constructor for class PieceWorker
PieceWorker::PieceWorker( const char *first, const char *last,
                          double w, int q )
   : Employee( first, last )  // call base-class constructor
{
   setWage( w );
   setQuantity( q );
}

// Set the wage
void PieceWorker::setWage( double w )
   { wagePerPiece = w > 0 ? w : 0; }

// Set the number of items output
void PieceWorker::setQuantity( int q )
   { quantity = q > 0 ? q : 0; }

// Determine the PieceWorker's earnings
double PieceWorker::earnings() const
   { return quantity * wagePerPiece; }

// Print the PieceWorker's name
void PieceWorker::print() const
{
   cout << "\n     Piece worker: ";
   Employee::print();
}
```

Again, notice the overridden **`earnings`** and **`print`** functions.

They were declared **`virtual`** in the base class.

```
// Fig. 10.1: hourly1.h
// Definition of class HourlyWorker
#ifndef HOURLY1_H
#define HOURLY1_H
#include "employ2.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char *, const char *,
                 double = 0.0, double = 0.0);
   void setWage( double );
   void setHours( double );
   virtual double earnings() const;
   virtual void print() const;
private:
   double wage;   // wage per hour
   double hours;  // hours worked for week
};

#endif
```

**1. `HourlyWorker` Definition (derived class)**

1.1 Function Definitions

```cpp
// Fig. 10.1: hourly1.cpp
// Member function definitions for class HourlyWorker
#include <iostream>

using std::cout;

#include "hourly1.h"

// Constructor for class HourlyWorker
HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double w, double h )
   : Employee( first, last )   // call base-class constructor
{
   setWage( w );
   setHours( h );
}

// Set the wage
void HourlyWorker::setWage( double w )
   { wage = w > 0 ? w : 0; }
```

## 1.1 Function Definitions

```

// Set the hours worked
void HourlyWorker::setHours( double h )
   { hours = h >= 0 && h < 168 ? h : 0; }

// Get the HourlyWorker's pay
double HourlyWorker::earnings() const
{
   if ( hours <= 40 ) // no overtime
      return wage * hours;
   else               // overtime is paid at wage * 1.5
      return 40 * wage + ( hours - 40 ) * wage * 1.5;
}

// Print the HourlyWorker's name
void HourlyWorker::print() const
{
   cout << "\n    Hourly worker: ";
   Employee::print();
}
```

Overridden functions.

```
// Fig. 10.1: fig10_01.cpp
// Driver for Employee hierarchy
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include "employ2.h"
#include "boss1.h"
#include "commis1.h"
#include "piece1.h"
#include "hourly1.h"

void virtualViaPointer( const Employee * );
void virtualViaReference( const Employee & );

int main()
{
   // set output formatting
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

```

## Outline

**1. Load headers**

1.1 Initialize objects

2. Print

```
   Boss b( "John", "Smith", 800.00 );
   b.print();
   cout << " earned $" << b.earnings();    // static binding
   virtualViaPointer( &b );             //
   virtualViaReference( b );            // uses dynamic binding

   CommissionWorker c( "Sue", "Jones", 200
   c.print();                              // static binding
   cout << " earned $" << c.earnings();
   virtualViaPointer( &c );             // u
   virtualViaReference( c );            // u

   PieceWorker p( "Bob", "Lewis", 2.5, 200
   p.print();
   cout << " earned $" << p.earnings();
   virtualViaPointer( &p );             // u
   virtualViaReference( p );            // u

   HourlyWorker h( "Karen", "Price", 13.75
   h.print();
   cout << " earned $" << h.earnings();
   virtualViaPointer( &h );             // uses dynamic binding
   virtualViaReference( h );
   cout << endl;
   return 0;
}

// Make virtual function calls off a base-class pointer
// using dynamic binding.
void virtualViaPointer( const Employee *baseClassPtr )
{
   baseClassPtr->print();
   cout << " earned $" << baseClassPtr->earnings();
}
```

```
Boss: John Smith earned $800.00
```

```
Boss: John Smith earned $800.00
```

```
Boss: John Smith earned $800.00
```

Call function **print** using the object itself.

```
Commission worker: Sue Jones earned $650.00

Commission worker: Sue Jones earned $650.00

Commission worker: Sue Jones earned $650.00
```

dynamic binding

```
Piece worker: Bob Lewis earned $500.00

     Piece worker: Bob Lewis earned $500.00

     Piece worker: Bob Lewis earned $500.00
```

This uses **virtual** functions and dynamic binding.

```
Hourly worker: Karen Price earned $550.00

    Hourly worker: Karen Price earned $550.00

    Hourly worker: Karen Price earned $550.00
```

Take in a baseclass pointer, call the **virtual** function **print**.

**3.  Function Definitions**

```

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Employee &baseClassRef )
{
   baseClassRef.print();
   cout << " earned $" << baseClassRef.earnings();
}
```

Take in base class reference, call the **`virtual`** function **`print`**.

**Program Output**

```
Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
             Boss: John Smith earned $800.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
Commission worker: Sue Jones earned $650.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
     Piece worker: Bob Lewis earned $500.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
    Hourly worker: Karen Price earned $550.00
```

# 10.7 New Classes and Dynamic Binding

- Polymorphism and virtual functions
  - Work well when all classes are not known in advance
  - Use dynamic binding to accommodate new classes being added to a system
- Dynamic binding (late binding)
  - Object's type need not be know at compile time for a **`virtual`** function
  - **`virtual`** function call is matched at run time

# 10.8 Virtual Destructors

- Problem:
  - If a base-class pointer to a derived object is deleted, the base-class destructor will act on the object
- Solution:
  - declare a **`virtual`** base-class destructor to ensure that the appropriate destructor will be called

# 10.9 Case Study: Inheriting Interface and Implementation

- Extension of point, circle, cylinder hierarchy
  - Use the abstract base class **`Shape`** to head the hierarchy
    - Two pure virtual functions **`printShapeName`** and **`print`**
    - Two other virtual functions **`volume`** and **area**
  - **`Point`** is derived from **`Shape`** and inherits these implementations

```cpp
// Fig. 10.2: shape.h
// Definition of abstract base class Shape
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
public:
   virtual double area() const { return 0.0; }
   virtual double volume() const { return 0.0; }

   // pure virtual functions overridden in derived classes
   virtual void printShapeName() const = 0;
   virtual void print() const = 0;
};

#endif
// Fig. 10.2: point1.h
// Definition of class Point
#ifndef POINT1_H
#define POINT1_H

#include <iostream>

using std::cout;

#include "shape.h"

class Point : public Shape {
public:
   Point( int = 0, int = 0 );  // default constructor
   void setPoint( int, int );
   int getX() const { return x; }
   int getY() const { return y; }
```

Notice the **virtual** functions which will be overridden by each class.

**Point** inherits from the abstract base class.

**1. Shape Definition (abstract base class)**

**---------------------**

**1. Point Definition (derived class)**

```
   virtual void printShapeName() const { cout << "Point: "; }
   virtual void print() const;
private:
   int x, y;   // x and y coordinates of Point
};

#endif
// Fig. 10.2: point1.cpp
// Member function definitions for class Point
#include "point1.h"

Point::Point( int a, int b ) { setPoint( a, b ); }

void Point::setPoint( int a, int b )
{
   x = a;
   y = b;
}

void Point::print() const
   { cout << '[' << x << ", " << y << ']'; }
```

Outline

**1. Point Definition (derived class)**

**1.1 Function Definitions**

```
// Fig. 10.2: circle1.h
// Definition of class Circle
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
   // default constructor
   Circle( double r = 0.0, int x = 0, int y = 0 );

   void setRadius( double );
   double getRadius() const;
   virtual double area() const;
   virtual void printShapeName() const { cout << "Circle: "; }
   virtual void print() const;
private:
   double radius;   // radius of Circle
};

#endif
```

**Circle** inherits from **Point**.

**1. Circle Definition (derived class)**

Outline

```
// Fig. 10.2: circle1.cpp
// Member function definitions for class Circle
#include <iostream>

using std::cout;

#include "circle1.h"

Circle::Circle( double r, int a, int b )
   : Point( a, b )  // call base-class constructor
{ setRadius( r ); }

void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }

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

double Circle::area() const
   { return 3.14159 * radius * radius; }

void Circle::print() const
{
   Point::print();
   cout << "; Radius = " << radius;
}
```

**1.1 Function Definitions**

## Outline

1. Cylinder Definition (derived class)

```
// Fig. 10.2: cylindr1.h
// Definition of class Cylinder
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
   // default constructor
   Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

   void setHeight( double );
   double getHeight();
   virtual double area() const;
   virtual double volume() const;
   virtual void printShapeName() const { cout << "Cylinder: "; }
   virtual void print() const;
private:
   double height;   // height of Cylinder
};

#endif
```

**Cylinder** inherits from **Circle**.

```
// Fig. 10.2: cylindr1.cpp
// Member and friend function definitions for class Cylinder
#include <iostream>

using std::cout;

#include "cylindr1.h"

Cylinder::Cylinder( double h, double r, int x, int y )
   : Circle( r, x, y )  // call base-class constructor
{ setHeight( h ); }

void Cylinder::setHeight( double h )
   { height = h > 0 ? h : 0; }

double Cylinder::getHeight() { return height; }

double Cylinder::area() const
{
   // surface area of Cylinder
   return 2 * Circle::area() +
          2 * 3.14159 * getRadius() * height;
}

double Cylinder::volume() const
   { return Circle::area() * height; }

void Cylinder::print() const
{
   Circle::print();
   cout << "; Height = " << height;
}
```

**1.1 Function Definitions**

Outline

```cpp
// Fig. 10.2: fig10_02.cpp
// Driver for shape, point, circle, cylinder hierarchy
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::ios;
using std::setiosflags;
using std::setprecision;

#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );

int main()
{
   cout << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 );

   Point point( 7, 11 );                  // create a Point
   Circle circle( 3.5, 22, 8 );           // create a Circle
   Cylinder cylinder( 10, 3.3, 10, 10 );  // create a Cylinder

   point.printShapeName();    // static binding
```

1. Load headers

1.1 Function prototypes

1.2 Initialize objects

2. Function calls

```
   point.print();               // static binding
   cout << '\n';

   circle.printShapeName();   // static binding
   circle.print();
   cout << '\n';

   cylinder.printShapeName(); // static binding
   cylinder.print();            // static binding
   cout << "\n\n";

   Shape *arrayOfShapes[ 3 ];  // array of base-class pointers

   // aim arrayOfShapes[0] at derived-class Point object
   arrayOfShapes[ 0 ] = &point;

   // aim arrayOfShapes[1] at derived-class Ci
   arrayOfShapes[ 1 ] = &circle;

   // aim arrayOfShapes[2] at derived-class Cy
   arrayOfShapes[ 2 ] = &cylinder;

   // Loop through arrayOfShapes and c
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
   cout << "Virtual function calls made off "
        << "base-class pointers\n";

   for ( int i = 0; i < 3; i++ )
      virtualViaPointer( arrayOfShapes[ i ] );

   // Loop through arrayOfShapes and call virtu
   // to print the shape name, attributes, area, and volume
   // of each object using dynamic binding.
```

```
Point: [7, 11]
```

Print

```
Circle: [22, 8]; Radius = 3.50
```

```
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
```

Create an array of base class pointers. Assign these to the objects, then call the **print** functions again, using the base class pointers. The appropriate **virtual** functions will be called.

```
Virtual function calls made off base-class pointers
```

```
Point: [7, 11]
Area
Volu
```

```
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
```

```
ght =
10.00
Area = 275.77
Volume = 342.12
```

2. Function calls

Definitions

```
   cout << "Virtual function calls made off "
        << "base-class references\n";

   for ( int j = 0; j < 3; j++ )
      virtualViaReference( *arrayOfShapes[ j ] );

   return 0;
}

// Make virtual function calls
// using dynamic binding.
void virtualViaPointer( const S
{
   baseClassPtr->printShapeName
   baseClassPtr->print();
   cout << "\nArea = " << baseC
        << "\nVolume = " << bas
}

// Make virtual function calls off a base-class reference
// using dynamic binding.
void virtualViaReference( const Shape &baseClassRef )
{
   baseClassRef.printShapeName();
   baseClassRef.print();
   cout << "\nArea = " << baseClassRef.area()
        << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```

Repeat process using base-class

```
Virtual function calls made off base-class
references

Point: [7, 11]

Area = 0.00

Volume = 0.00

Circle: [22, 8]; Radius = 3.50

Area = 38.48

Volume = 0.00

Cylinder: [10, 10]; Radius = 3.30; Height =
10.00

Area = 275.77

Volume = 342.12
```

**Program Output**

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.50
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Virtual function calls made off base-class pointers
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
Virtual function calls made off base-class references
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: [22, 8]; Radius = 3.50
Area = 38.48
Volume = 0.00
Cylinder: [10, 10]; Radius = 3.30; Height = 10.00
Area = 275.77
Volume = 342.12
```

# 10.10 Polymorphism, `virtual` Functions and Dynamic Binding “Under the Hood”

- When to use polymorphism
  - Polymorphism requires a lot of overhead
  - Polymorphism is not used in STL (Standard Template Library) to optimize performance
- **`virtual`** function table (vtable)
  - Every class with a **`virtual`** function has a vtable
  - For every **`virtual`** function, its vtable has a pointer to the proper function
    - If a derived class has the same function as a base class, then the function pointer points to the base-class function
  - Detailed explanation in Fig. 10.3