Chapter 13 | Polymorphism

CS185

Virtual Methods and Polymorphism

The specification (**Employee.h**) for an Employee class:

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class Employee
{
private:
	std::string firstName;
	std::string lastName;
	float salary;
	int years;

public:
	Employee(const std::string& firstName_, const std::string& lastName_,
	         float salary_, int years_);
	void setName(const std::string& firstName_, const std::string& lastName_);
	void setSalary(float salary_);
	void setYears(int years_);
	void Display(void) const;
};
#endif
```

The *specification* (**Manager.h**) for the `Manager` class:

```
#ifndef MANAGER_H
#define MANAGER_H
#include "Employee.h"

class Manager : public Employee
{
public:
	Manager(const std::string& firstName_, const std::string& lastName_,
	        float salary_, int years_, int deptNumber_, int numEmployees_);
	void setDeptNumber(int deptNumber_);
	void setNumEmployees(int numEmployees_);
	void Display(void) const;

private:
	int deptNumber;    // department managed
	int numEmployees;  // employees in department
};
#endif
```

Does the following code compile as is? If not, make the necessary changes so it will compile then trace the execution of the program. What is the output? Why?

Code

```
#include "Employee.h"
#include "Manager.h"
#include <iostream>

void func1(const Employee& emp_)
{
	emp_.Display();
	std::cout << std::endl;
}

void func2(const Manager& mgr_)
{
	mgr_.Display();
	std::cout << std::endl;
}

int main()
{
	Employee emp1("John", "Doe", 30000, 2);
	Manager mgr1("Mary", "Smith", 50000, 10, 5, 8);

	func1(emp1);  // pass an Employee object
	func2(mgr1);  // pass a Manager object

	func1(mgr1);  // pass a Manager object
	func2(emp1);  // pass an Employee object
	return 0;
}
```

Output (Once the code is fixed!)

```
  Name: Doe, John
Salary: $30000.00
 Years: 2

  Name: Smith, Mary
Salary: $50000.00
 Years: 10
  Dept: 5
  Emps: 8

  Name: Smith, Mary
Salary: $50000.00
 Years: 10
```

The following code won't compile. Remove the offending line(s) and then trace the execution of the program. What is the output? Why?

Code	
	```
	#include "Employee.h"
	#include "Manager.h"
	#include <iostream>

	int main()
	{
	        Employee emp1("John", "Doe", 30000, 2);
	        Manager mgr1("Mary", "Smith", 50000, 10, 5, 8);

	        Employee* empPtr1 = &emp1;
	        Manager* mgrPtr1 = &mgr1;

	        empPtr1->Display();
	        std::cout << std::endl;

	        mgrPtr1->Display();
	        std::cout << std::endl;

	        empPtr1 = &mgr1;
	        empPtr1->setYears(11);
	        empPtr1->setNumEmployees(12);
	        empPtr1->Display();
	        std::cout << std::endl;
	        return 0;
	}
	```

Output (Once the code is fixed!)	
	```
	  Name: Doe, John
	Salary: $30000.00
	 Years: 2

	  Name: Smith, Mary
	Salary: $50000.00
	 Years: 10
	  Dept: 5
	  Emps: 8

	  Name: Smith, Mary
	Salary: $50000.00
	 Years: 11
	```

This program creates an array of pointers to Employee objects. It displays each object using a for loop. Make sure you understand what the program is trying to do.

Code

```cpp
#include "Employee.h"
#include "Manager.h"
#include <iostream>

int main()
{
	// Create the personnel
	Employee emp1("John", "Doe", 30000, 2);
	Employee emp2("Nigel", "Tufnel", 35000, 4);
	Manager mgr1("Mary", "Smith", 50000, 10, 5, 8);
	Manager mgr2("Derek", "Smalls", 60000, 13, 6, 5);

	// Create an array to hold pointers to the 4 objects
	Employee* personnel[4];

	// Assign a pointer for each object
	personnel[0] = &emp1;
	personnel[1] = &emp2;
	personnel[2] = &mgr1;  // a Manager is an Employee
	personnel[3] = &mgr2;  // a Manager is an Employee

	// Loop through and display each object
	for (int i = 0; i < 4; i++)
	{
		personnel[i]->Display();
		std::cout << std::endl;
	}

	return 0;
}
```

Output

```
  Name: Doe, John
Salary: $30000.00
 Years: 2

  Name: Tufnel, Nigel
Salary: $35000.00
 Years: 4

  Name: Smith, Mary
Salary: $50000.00
 Years: 10

  Name: Smalls, Derek
Salary: $60000.00
 Years: 13
```

What we really wanted was to have each object display all of its data. The Employee objects displayed all of their data, but the Manager objects only displayed the data that they have in common with an Employee. We really wanted this to display:

```
  Name: Doe, John       Name: Smith, Mary
Salary: $30000.00     Salary: $50000.00
 Years: 2              Years: 10
                        Dept: 5
                        Emps: 8

  Name: Tufnel, Nigel   Name: Smalls, Derek
Salary: $35000.00     Salary: $60000.00
 Years: 4              Years: 13
                        Dept: 6
                        Emps: 5
```

Because the *personnel[]* array is an array of pointers to Employee objects, when the compiler sees the statement:

personnel[i]->Display();

- The compiler generates code to call the ***Display()*** method of the Employee class, regardless of what type of object is being pointed at in position ***"i"*** of the ***personnel[]*** array. (Static type vs. dynamic type)
- This is the "normal" or default behavior of the C++ compiler.
- This type of code generation is called *static binding* or *early binding* because it is done at compile time.
- We need a way to tell the compiler *not* to generate the function call at compile time, but wait until run-time to do so.
- Delaying the decision until runtime is called *dynamic binding* or *late binding* because the decision is made at the last possible moment.

This should bring up these points:

- Heterogeneous arrays in C++?
- How do we tell the compiler to use dynamic binding?
- Why isn't dynamic binding the default?
- What makes it work?

The How is pretty simple:

- The mechanism in C++ is *virtual functions*.
- In order for our example to work, we need to make the ***Display()*** method in the Employee class a virtual function.
- To do so, we merely add the ***virtual*** keyword to the function declaration in the specification (header) file:
- ***virtual void Display() const;***
- That's all there is to it! Now, if you run the previous example, you will get the correct output.

Why isn't dynamic binding the default? There's a couple of reasons:

- Efficiency. Virtual methods require more memory and processor time. (The overhead is very slight and almost never poses a problem in actual use.)
- You might not want to redefine it in a derived class.

So, in a nutshell, what is a virtual method? *A virtual method allows a derived class to replace the implementation that was provided by the base class.*

A Closer Look at Static Binding vs. Dynamic Binding (Polymorphism)

On the surface, the concept of virtual methods seems strange, complex, or even magical. In fact, it's all three. To understand dynamic binding of methods (virtual methods) you must first understand static binding.

- In C++ code, all method names (e.g. Employee::Display) are actually pointers that contain the address of the area of memory that contains the executable code for the method.
- Statically bound methods are methods which point to the correct area of memory at compile time (link time, actually), meaning that they are initialized with the address of the method at compile (link) time and *always* contain the same address.
- Until now, all methods that we have been dealing with have been statically bound.
- Dynamically bound methods (virtual methods) are not initialized with the address of the method at compile time.
- At runtime, when a virtual method is called, the code "looks up" the address of the virtual method and calls that address.
- A method is statically bound by default.
- You must explicitly mark a method as ***virtual*** for it to be dynamically bound.

Notes about virtual methods:

- In order for the virtual mechanism to work properly, the base class must specify which methods are virtual.
- You can't simply add the ***virtual*** keyword to a method in the derived class and expect to be able to call the corresponding method in the base class.
- The virtual mechanism only works with pointers because calls to class methods via non-pointers are always statically bound. (Objects can't change, pointers can):

```
Manager mgr("Mary", "Smith", 50000, 10, 5, 8);
Employee *pm = &mgr; // OK, a Manager is an Employee

mgr.Display(); // Always calls Manager::Display() regardless of virtual keyword
pm->Display(); // Depends whether or not Display() is virtual in the base class
```

- Converting a derived class reference or pointer to a base class reference or pointer is called ***upcasting*** and is allowed by the compiler automatically. (implicit conversion) (Think of casting *up* the hierarchy.)

  This is simply because a derived class ***is-a-kind-of*** base class.

- Converting a base class reference or pointer to a derived class reference or pointer is called ***downcasting*** and can only be done with an explicit cast. (Think of casting *down* the hierarchy.) It may not be safe to do so.

  This is because a base class ***is-NOT-a-kind-of*** derived class.

- ***Upcasting*** is always safe. ***Downcasting*** is unsafe and can easily lead to program crashes at runtime.
- Simple rule of thumb: If you intend to redefine a method in a derived class, make the method ***virtual*** in the base class. If you don't want to redefine it, don't make it ***virtual***.

Additional notes:

- If it's not obvious, only member functions can be ***virtual***.
- The ***virtual*** keyword must only appear in the class declaration.

  Correct:

```
virtual void Display() const;
```

  Incorrect:

```
virtual void Employee::Display() const
{
  // Code to print the object goes here
}
```

- You only have to specify the ***virtual*** keyword in the base class. In our example, the ***Manager::Display*** does not have to be tagged as ***virtual***, although it is.

  Note: ***Although the `virtual` keyword is not required in the derived classes, it's a good idea to include it as a way to document the code.***

- Once a function is marked as ***virtual***, it will always be ***virtual*** in all derived classes. There is no way to "undo" this in a descendant class. (Changed in C++11 with the **final** identifier)
- The virtual mechanism is apparent when a pointer to a base class is pointing at an object of a derived class:

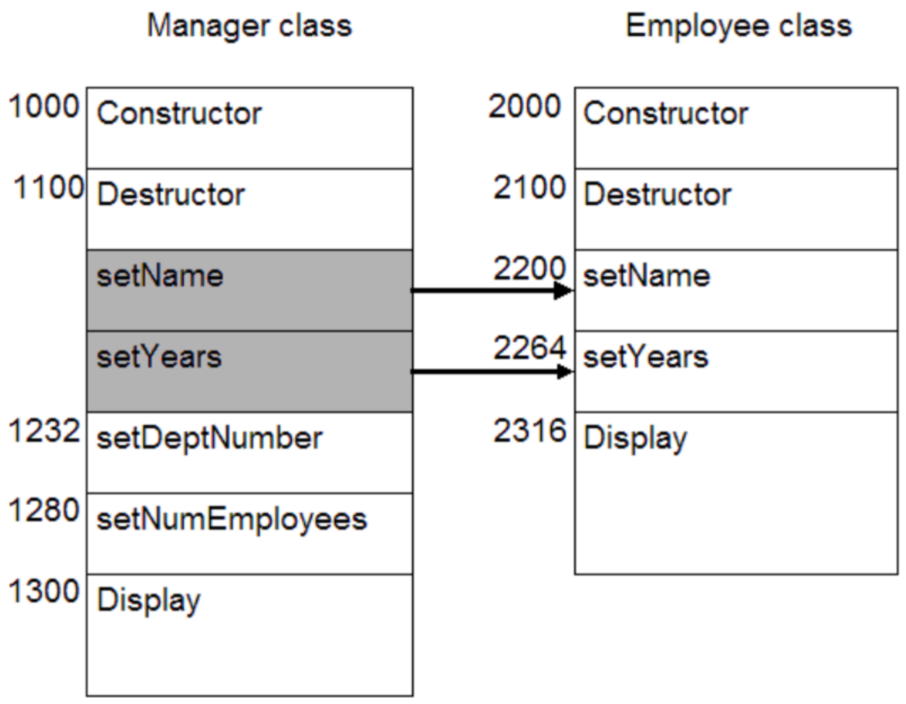
```
Employee emp("John", "Doe", 30000, 2);
Manager mgr("Mary", "Smith", 50000, 10, 5, 8);

Employee* empPtr = &emp; // Nothing fancy here
Manager* mgrPtr = &mgr;  // Nothing fancy here

emp.Display();      // Employee::Display()
empPtr->Display(); // Employee::Display()
mgr.Display();      // Manager::Display()
mgrPtr->Display(); // Manager::Display()

// Polymorphism is realized now
empPtr = &mgr;      // Safe and legal (Manager is an Employee)
empPtr->Display(); // Depends on "virtualness" of Display()
                   //   if Display() is virtual, Manager::Display()
                   //   if Display() is non-virtual, Employee::Display()
```

- This concept of having base class pointers point to objects of the derived class is what ***polymorphism*** is all about.
- A *very oversimplified* example:



Without virtual methods: (i.e. Employee::Display() is **NOT** marked as **virtual**)

```
int main()
{
        // Create an Employee and a Manager
        Employee emp("John", "Doe", 30000, 2);
        Manager mgr("Mary", "Smith", 50000, 10, 5, 8);

        // Display them
        emp.Display(); // Compiler/linker generated JUMP to address 2316
        mgr.Display(); // Compiler/linker generated JUMP to address 1300

        Employee *pe = &emp; // OK
        Employee *pm = &mgr; // OK, a Manager is an Employee

                                                    // Display them
        pe->Display(); // Compiler/linker generated JUMP to address 2316
        pm->Display(); // Compiler/linker generated JUMP to address 2316
                       //(pm is an Employee pointer)

        return 0;
}
```

With virtual methods: (i.e. Employee::Display() is marked as **virtual**)

```
int main()
{
        // Create an Employee and a Manager
        Employee emp("John", "Doe", 30000, 2);
        Manager mgr("Mary", "Smith", 50000, 10, 5, 8);

        // Display them
        emp.Display(); // Compiler/linker generated JUMP to address 2316
        mgr.Display(); // Compiler/linker generated JUMP to address 1300

        Employee *pe = &emp; // OK
        Employee *pm = &mgr; // OK, a Manager is an Employee

        // Display them
        pe->Display(); // Compiler generated code to perform lookup at runtime.
                       //   Finds Display() at address 2316
        pm->Display(); // Compiler generated code to perform lookup at runtime.
                       // Finds Display() at address 1300
        return 0;
}
```

Virtual Function Tips

Virtual Method Tables

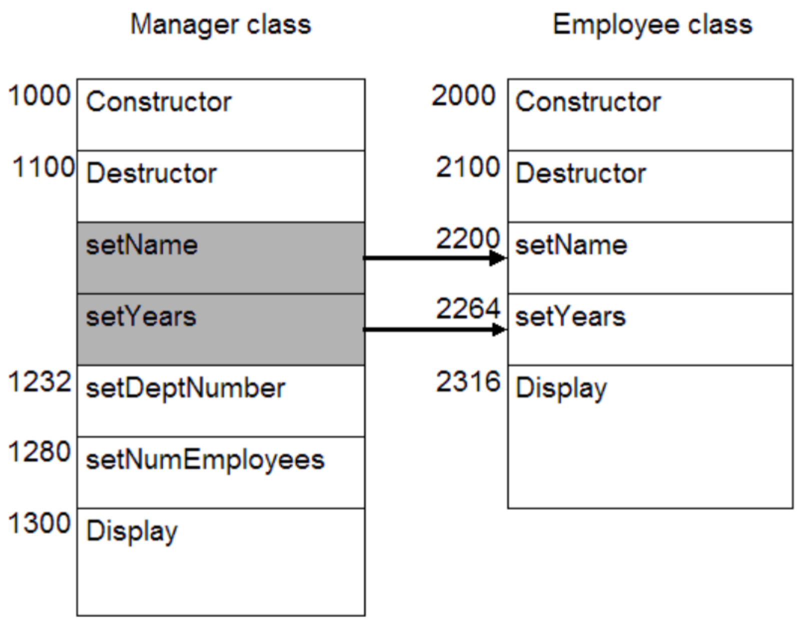
- Each class that contains virtual methods has a *Virtual Method Table* or VMT.
- A VMT is basically a mechanism that allows a method's address to be located (looked up) at runtime.
- The VMT is generated at compile and doesn't change.
- It takes more time to call a virtual method than a non-virtual method because of the lookup time. (The added time is slight and almost never poses a problem.)

Special Functions

- **Constructors** Can't be virtual since derived classes don't inherit them from the base class. (Now in C++11 with **inheriting constructors**)
- **Destructors** Any class intended for use as a base class should have a virtual destructor. I would read ***should*** as ***must***.

```
Manager *mgr = new Manager("Mary", "Smith", 50000, 10, 5, 8);
Employee *pe = mgr;
...
delete pe; // what gets called? ~Employee() or ~Manager()
           // pe->~Employee() or pe->~Manager()
```

Using the following diagram, without virtual:



```
delete pe; // Compiler generated JUMP to address 2100
```

However, with virtual destructor:

```
delete pe; // Compiler generated code to lookup function at 1100
```

- This becomes an issue if the derived class dynamically allocates memory (**new**) or has some other resource that needs to be released (e.g. open file handle)
- Failing to make the base class destructor virtual will prevent the derived class' resources to be released. The base class sub-object will get destroyed, but not the derived portion. Resource leak.
- Even if the base class doesn't need a destructor (default is adequate) you need to define one to make it virtual (remember: static binding is default):

```
virtual ~Base() {};
```

- The compiler automatically calls the destructors for the base objects.

Base Class

We know that all squares are rectangles, but all rectangles are not squares. Sounds like a perfect example of an "is-a" relationship (read: inheritance).

So, we sketch out the interface to our base class *Rectangle*:

```
class Rectangle
{
private:
   double center_x; // x-coordinate of center point
   double center_y; // y-coordinate of center point
   double length;   // "long" sides
   double width;    // "short" sides

public:
   // Constructor (default)
   Rectangle(double x_ = 0, double y_ = 0, double length_ = 0, double width_ = 0);

   // Rectangle-specific get/set methods
   double getLength() const;
   double getWidth() const;
   void setLength(double length_);
   void setWidth(double width_);
   double getCenterX() const;
   double getCenterY() const;
   void SetCenter(double x_, double y_);

   // Need to be redefined in derived classes
   virtual double Area() const;
   virtual void Draw() const;
   virtual void Scale(double scale_x_, double scale_y_);
};
```

Once we've got that scoped out, we can work on the *Square* class:

```
class Square : public Rectangle
{
private:
   double side;  // length of a side

public:
   // Constructor
   Square(double x_, double y_, double side_);

   // Square-specific Get methods
   double GetSide() const;
   void SetSide(double side_);

   // Methods from Rectangle that we need to specialize
   virtual double Area() const;
   virtual void Draw() const;
   virtual void Scale(double scale_);
};
```

But, something's not right. This is what the *Square* class "*sort of looks*" like now:

```
class Square : public Rectangle
{
private:
   // Sort of Inherited from Rectangle (We have gettors and settors for them).
   double center_x; // x-coordinate of center point
   double center_y; // y-coordinate of center point
   double length;   // "long" sides
   double width;    // "short" sides

   // Derived member
   double side;  // length of a side

public:
   // Inherited from Rectangle
   double getLength() const;
   double getWidth() const;
   void setLength();
   void setWidth();
   double getCenterX() const;
   double getCenterY() const;
   void SetCenter(double x_, double y_);

   // Derived members
   Square(double x_, double y_, double side_);
   double GetSide() const;
   void SetSide(double side_);
   virtual void Scale(double scale_); // Hides the base class function

   // Redefined
   virtual double Area() const;
   virtual void Draw() const;
};
```

There are several problems:

- Acess to length *and* width
- Redundant members
- Square::Scale hides Rectangle::Scale (no polymorphism)

Even though in the "real world" this relationship seems straight-forward, we need to re-think the design. It would be simpler to just create the *Square* class "from scratch":

```
class Square
{
private:
	double center_x; // x-coordinate of center point
	double center_y; // y-coordinate of center point
	double side;     // length of a side

public:
	// Constructor
	Square(double x_, double y_, double side_);

	double getCenterX() const;
	double getCenterY() const;
	void SetCenter(double x_, double y_);
	double GetSide() const;
	void SetSide(double side_);

	double Area() const;
	void Draw() const;
	void Scale(double scale_);
};
```

We ’ve traded one set of "problems" for another.

- This time, we've got a lot of duplicated functionality.
- We should factor out the commonality into its own class.
- We can actually simplify the design by adding a third class.

```
class Figure
{
private:
	double center_x; // x-coordinate of center point
	double center_y; // y-coordinate of center point

public:
	// Constructor
	Figure(double x_ = 0, double y_ = 0);

	// get/set
	double getCenterX() const;
	double getCenterY() const;
	void SetCenter(double x_, double y_);

	// Virtual methods common to both
	virtual double Area() const;
	virtual void Draw() const;
};
```

Rectangle and *Square* are now derived from *Figure*:

```
#include "Figure.h"

class Rectangle : public Figure
{
private:
	double length_; // "long" sides
	double width_;  // "short" sides

public:
	// Constructor (default)
	Rectangle(double x = 0,
		double y = 0,
		double length = 0,
		double width = 0);

	// Rectangle-specific get/set methods
	double getLength() const;
	double getWidth() const;
	void setLength();
	void setWidth();
	void Scale(double scale_x, double scale_y);

	// Methods from Figure that
	// we need to specialize
	virtual double Area() const;
	virtual void Draw() const;
};
```

```
#include "Figure.h"

class Square : public Figure
{
private:
	double side; // length of a side

public:
	// Constructor (default)
	Square(double x_ = 0, double y_ = 0, double side_ = 0);


	// Square-specific get/set methods
	double GetSide() const;
	void SetSide(double side_);
	void Scale(double scale_);

	// Methods from Figure that
	// we need to specialize
	virtual double Area() const;
	virtual void Draw() const;
};
```

Sample client code:

Code	`Rectangle r(4, 5, 10, 3);`
	`Square s(2, 3, 6);`
	`Figure *figs[2] = { &r, &s };`
	
	`for (int i = 0; i < 2; i++)`
	`{`
	` figs[i]->Draw();`
	` cout << "Area: " << figs[i]->Area() << endl;`
	`}`

Output	**`Drawing the rectangle: 10x3`**
	`Area: 30`
	`Drawing the square: 6`
	`Area: 36`

Here are the simplistic `Draw` methods:

```
void Rectangle::Draw() const
{
	cout << "Drawing the rectangle: " << length_ << "x" << width_ << endl;
}

void Square::Draw() const
{
	cout << "Drawing the square: " << side_ << endl;
}
```

Notes:

- A class hierarchy goes from the more general to the more specific.
- Derived classes are more specific than their base classes.
- The more "derived" a class is, the more concrete it is (represents a specific type)
- Some base classes are so general that they do not (and cannot) represent anything specific enough.
- Base classes that cannot represent any object should not be instantiated.
- We call an ***"overly general"*** base class an ***abstract base class***.

Abstract Base Class

If I said: "Everyone take out a pencil and draw a figure centered at (0, 0) on an X-Y grid.", what would we see?

An (incomplete) example:

- **Figure** - a generic figure with properties shared by all figures: a **center** and **Draw()** and **Area()** methods
- **Circle** - A specific kind of Figure, has an additional unique attribute, **radius**
- **Square** - A specific kind of Figure, has an additional unique attribute, **length** of a side

```
class Figure
{
private:
       double center_x; // x-coord center
       double center_y; // y-coord center

public:
       Figure(double x_ = 0, double y_ = 0);
       virtual void Draw() const;
       virtual double Area() const;
};
```

Derived classes *Circle* and *Square*:

```
class Circle : public Figure
{
private:
       double radius;

public:
       Circle(double x_ = 0, double y_ = 0, double radius = 0);
       virtual double Area() const;
       virtual void Draw() const;
};
```

```
class Square : public Figure
{
private:
	double side; // length of a side

public:
	Square(double x_ = 0, double y_ = 0, double side_ = 0);
	virtual double Area() const;
	virtual void Draw() const;
};
```

Sample client code:

```
int main()
{
	Circle circle(0, 0, 5); // Circle at (0,0) with radius=5
	Square square(0, 0, 8); // Square at (0,0) with side=8
	Figure figure(3, 9);    // Figure at (3, 9)

	circle.Draw(); // draws the Circle
	square.Draw(); // draws the Square
	figure.Draw(); // ???
	return 0;
}
```

The implementation of the *Figure* class:

```
#include "Figure.h"

Figure::Figure(double x, double y)
{
	center_x_ = x;
	center_y_ = y;
}

double Figure::Area() const
{
	// What's the area of a Figure?
}

void Figure::Draw() const
{
	// How do you draw a Figure?
}
```

There's obviously a problem with implementing some methods of the *Figure*:

- It makes no sense to ever instantiate a Figure object.
- We want to prevent the user from creating one.
- We need to make the base class *abstract*; you can't instantiate an object of an abstract class

- C++ uses a quirky syntax to mark a class as *abstract*

```
virtual void Draw() const = 0;
virtual double Area() const = 0;
```

- Yes, setting the *method* declaration to 0 makes the *class* abstract. Funky? Yes, but that's the way it is so we must deal with it.
- This syntax makes the *Draw* and *Area* methods *pure* virtual methods.

Notes on abstract classes:

- At least one virtual method in a class must be *pure virtual* in order for the class to be abstract.
- Once a class is abstract, you cannot instantiate any objects of that class.
- If a base class has a pure virtual method and you derive a class from it:
    - You must override the pure virtual method or else the derived class itself becomes abstract (this may be desirable)
    - If the base class has multiple pure virtual methods, the derived class must override all of them, else the derived class becomes abstract
    - Pure virtual methods can be implemented and invoked by the other methods of the class or derived classes (if the methods are visible).
- In our example, we must redefine both the *Draw* method and the *Area* method in the derived classes in order to instantiate them.

Figure is now an abstract base class

```
class Figure
{
private:
	double center_x; // x-coord center
	double center_y; // y-coord center

public:
	Figure(double x_ = 0, double y_ = 0);
	virtual void Draw() const = 0;
	virtual double Area() const = 0;
};
```

Code	`int main() { Circle circle(0, 0, 5); // Circle at (0,0) with radius=5 ...`

```
int main()
{
	Circle circle(0, 0, 5); // Circle at (0,0) with radius=5
	Square square(0, 0, 8); // Square at (0,0) with side=8
	Figure figure(3, 9);    // Compile error
}
```