Chapter 15: Object-Oriented Programming

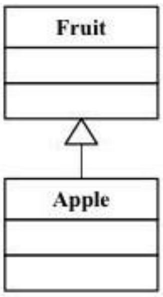
15.2 Inheritance 繼 承

繼承（inheritance）是物件導向語言重要的機制，繼承讓我們在建立類別時，可以不用宣告全新的成員，反之，你可以指定新類別繼承現有類別的成員。我們稱現有類別為父類別（superclass/parent class/**base class**），新 延伸 類別 為子類別（subclass/child class/extended class/**derived class**）。

繼承（inheritance）主要目的有三

1. 子類別可再利用父類別，吸收其成員，然後增添新的功能或覆寫（override）父類別的功能。
2. 繼承詮釋 is-a（是一種）的關係，例如父類別是 Fruit，子類別是 Apple 或 Orange，透過繼承詮釋 Apple is a Fruit（蘋果是一種水果）、Banana is a Fruit（香蕉是一種水果）的關係。而兩者的共通性則出現在父類別成員。
3. 物件導向語言最重要的機制：多型（polymorphism），需依賴繼承的覆寫（override）來完成。

In UML (Unified Modeling Language), we use an empty triangle to represent the **is-a** relationship between the derived class and base class.



15.2.1 C++繼承的基本語法

Classes related by **inheritance** form a hierarchy. Typically there is a **base class** at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as **derived classes**.

The base class defines those members that are **common** to the types in the hierarchy (e.g., `Student`). Each derived class defines those members that are **specific** to the derived class itself (e.g., `Student_Under`, `Student_Grad`)

每一語言在處理繼承（inheritance）的原理雖然類似，但語法不盡相同，以下針對 C++ 的繼承語法做一說明。

Base Class (父類別)

Suppose we would like to model different kinds of pricing strategies for our bookstore. We'll define a class named `Quote`, which will be the base class of our hierarchy. A `Quote` object will represent books in general. We will inherit a class `Bulk_quote` from `Quote` to represent specialized books that can be sold with a discount.

The `Quote` and `Bulk_quote` classes will have the following two member functions:

- `isbn()`, which will return the ISBN. This operation does not depend on the specifics of the inherited class(es); it will be defined only in class `Quote`.
- `net_price(size_t)`, which will return the price for purchasing a specified number of copies of a book. This operation is type specific; both `Quote` and `Bulk_quote` will define their own version of this function.

Quote.h

```
#ifndef QUOTE_H
#define QUOTE_H

#include <string>

class Quote
{ public:
    Quote() = default;
    Quote(const std::string &book, double sales_price) :
        bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    // returns the total sales price for the specified number of 
items
    // derived classes will override and apply different discount 
algorithms
    virtual double net_price(std::size_t n) const
    {
        return n * price;
    }
    virtual ~Quote() = default; // dynamic binding for the 
destructor
private:
    std::string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
```

```
};

#endif
```

Q: What's new?

A:

public, protected, private members

1 When designing a class to serve as a base class, the criteria for designating a member as `public` do not change: interface functions should be `public` and data generally should **not be** `public`.

2 A class designed to be inherited from must decide which parts of the implementation to declare as `protected` and which should be `private`.

3 A member should be made `private` if we wish to prevent derived classes from having access to that member. On the other hand, a member should be made `protected` if a derived class will need to use it in its implementation.

4 In other words, the interface to the derived type is the combination of both the `protected` and `public` members.

virtual member functions

In C++, the base class defines **`virtual`** member functions if it expects its derived classes to define the specific versions for themselves (we will call the derived class **override**s the virtual member function in the base class). In our case, `net_price` member function is type specific; both `Quote` and `Bulk_quote` will define their own version of this function. Thus, the `net_price` is a **`virtual`** member function and a keyword **`virtual`** is inserted at the beginning of function declaration.

```
class Quote
{ public:
  std::string isbn() const;
  virtual double net_price(std::size_t n) const;
  ...
};
```

Example: we can use the base class as usual.

```
#include "Quote.h"
#include <iostream>

using namespace std;

int main(){
    Quote q("032-171-4113", 42.38); // C++ Primer
    cout << "Net price for three copies of " << q.isbn()
      << " is: " << q.net_price(3) << endl;
    return 0;
}
```

```
Net price for three copies of 032-171-4113 is: 127.14
Press any key to continue . . .
```

Derived Class (子類別)

Bulk_Quote.h

```
#ifndef BULKQUOTE_H
#define BULKQUOTE_H
#include "Quote.h"
#include <string>

class Bulk_quote : public Quote { // Bulk_quote inherits from
Quote
public:
  Bulk_quote() = default;
  Bulk_quote(const std::string& book, double p, std::size_t qty,
double disc): Quote(book, p), min_qty(qty), discount(disc) { }
  // overrides the base version
  double net_price(std::size_t) const override;
private:
  std::size_t min_qty = 0; // minimum purchase for discount to
apply
  double discount = 0.0; // fractional discount to apply
};
#endif
```

A derived class must specify the class(es) from which it intends to inherit. It does so in a class derivation list, which is a colon followed by a comma-separated list of base classes, each of which may have an optional access specifier:

```
class Bulk_quote : public Quote { // Bulk_quote inherits from
Quote
```

```
public:
    ...
};
```

Because `Bulk_quote` uses **`public`** in its derivation list, we can use objects of type `Bulk_quote` as if they were `Quote` objects (**public inheritance means is-a relationship**, more on this later).

A derived class must include in its own class body a declaration of all the `virtual` functions it intends to define for itself. A derived class may include the `virtual` keyword on these functions but is not required to do so.

```
class Bulk_quote : public Quote { // Bulk_quote inherits from
Quote
public:
double net_price(std::size_t) const;
...
};
```

Good Practice: The new standard lets a derived class explicitly note (compiling check) that it intends a member function to override a `virtual` member function that it inherits. It does so by specifying `override` after the parameter list, or after the `const` qualifier if the member is a `const` function.

```
class Bulk_quote : public Quote { // Bulk_quote inherits from
Quote
public:
double net_price(std::size_t) const override;
...
};
```

Remarks:

1 Ordinarily, **derived classes redefine the `virtual` functions that they inherit**, although they are not required to do so. If a derived class does not redefine a `virtual`, then the version it uses is the one defined in its base class.

2 When a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in all the derived classes.

3 A derived-class function that overrides an inherited virtual function must have **exactly** the same parameter type(s) as the base-class function that it overrides. If somehow the derived class defines a function that has different parameters than the virtual function in the base class, it overloads rather than overrides. We will discuss more on overriding vs. overloading later.

We are now ready to **redefine** `net_price` in `Bulk_quote`:

Bulk_Quote.cpp

```
#include "Bulk_Quote.h"

double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

Remarks:

1 A derived class can access the `public` and `protected` members (e.g., `price`) of its base class.

2 There is no distinction between how a member of the derived class uses members defined in its own class (e.g., `min_qty` and `discount`) and how it uses members defined in its base (e.g., `price`).

15.2.2 子類別的建構子 (derived-class constructor) and Constructor Chaining

The constructors of a base class are **NOT** inherited by a derived class, but each derived-class constructor explicitly or implicitly calls its base-class constructor.

Explicitly: using the derived class constructor initializer list to pass values to a base-class constructor.

```
Bulk_quote(const std::string& book, double p, std::size_t qty,
double disc) :
Quote(book, p), min_qty(qty), discount(disc) { }
// as before
};
```

Implicitly: C++ automatically calls the base-class **default** constructor if we do not specify the base-class constructor.

The derived-class constructor explicitly or implicitly calls its base-class constructor is known as constructor chaining.

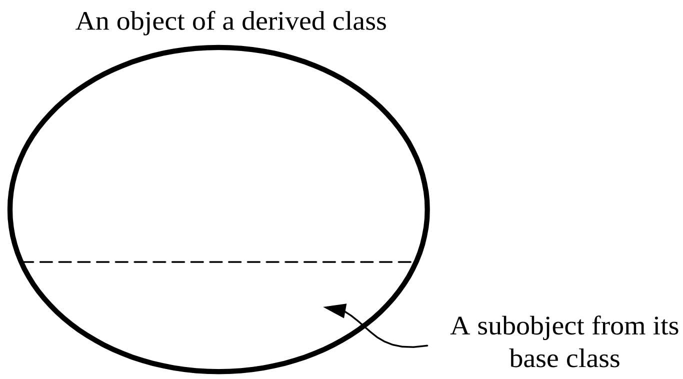
Example: we are now ready to use the base class and the derived class.

```
#include "Quote.h"
#include "Bulk_Quote.h"
#include <iostream>

using namespace std;

int main(){
   Quote q("032-171-4113", 42.38);
cout << "Net price for three copies of " << q.isbn() << " is: "
<< q.net_price(3) << endl;
   Bulk_quote bq1("032-171-4113", 42.38, 10, 0.2);
cout << "Net price for three copies of " << bq1.isbn() << " is: "
<< bq1.net_price(3) << endl;
   Bulk_quote bq2("032-171-4113", 42.38, 10, 0.2);
   cout << "Net price for 30 copies of " << bq2.isbn() << " is: "
<< bq2.net_price(30) << endl;
   return 0;
}
```

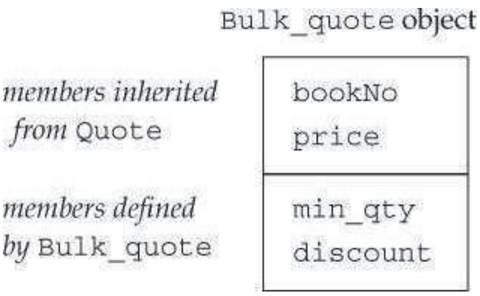
```
Net price for three copies of 032-171-4113 is: 127.14
Net price for three copies of 032-171-4113 is: 127.14
Net price for 30 copies of 032-171-4113 is: 1017.12
Press any key to continue . . .
```

15.2.3 子類別的物件

When you create an object of the derived class, it contains within it **a sub-object of the base class**.



Conceptually, we can think of a `Bulk_quote` object consisting of two parts as shown below:



This sub-object from its base class is created by <u>implicitly or explicitly</u> calling the base-class constructor. If you think of a base class as the <u>parent</u> to the derived class the <u>child</u>, you know that **the base class parts of an object have to be fully-formed before the derived class parts can be constructed**.

15.2.4 Inheritance: an `is-a` Relationship and Dynamic Binding

Because a derived object contains a **subpart** corresponding to its base class, we can use an object of a derived type **as if it were an object of its base type**. This is a very famous ***is-a*** relationship in OOP:

A derived object **is a** base object.

Ordinarily, C++ requires that references and pointer types match the assigned types, but this rule is relaxed for inheritance. In particular, we can bind a base-class reference or pointer to the base-class part of a derived object.

```
Quote item; // object of base type
Bulk_quote bulk; // object of derived type
Quote *p = &item; // p points to a Quote object
Quote *p2 = &bulk; // p2 points to the Quote part of bulk
Quote &r = bulk; // r bound to the Quote part of bulk
```

This derived-to-base conversion is implicit. It means that we can use **an object of derived type** or **a reference to a derived type** when **a reference to the base type is required**. Similarly, we can use a pointer to a derived type where a pointer to the base type is required.

Example
(IsAExample.cpp)

```
class Person{};
class Student : public Person{};

void dance(const Person& p){} // anyone can dance
void study(const Student& s){} // only students study

int main(){
    Person p;
    Student s;
    dance(p);
    dance(s);
    // study(p);  // COMPILING ERROR
    study(s);
    return 0;
}
```

Overriding vs. Overloading

Overriding means to provide a new implementation for a method in the derived class and is a very **IMPORANT** mechanism in inheritance and polymorphism. Overloading simply means to define multiple methods with the same name but different parameter lists. Overloading has nothing to do with inheritance.

Overriding Example (OverrideEx.cpp)

```
#include "iostream"
using namespace std;

class Base
{ public:
```

```
    virtual void p(int i) {
        cout << "Base::p(int)" << endl;
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(int i) {
        cout << "Derived::p(int)" << endl;
    }
};

int main(){
    Base b;
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    return 0;
}
```

Q: What is the output?

A:

<u>Possible unintentional mistake</u>: you are meant for overriding but unintentionally perform overloading:

OverrideExWrong1.cpp

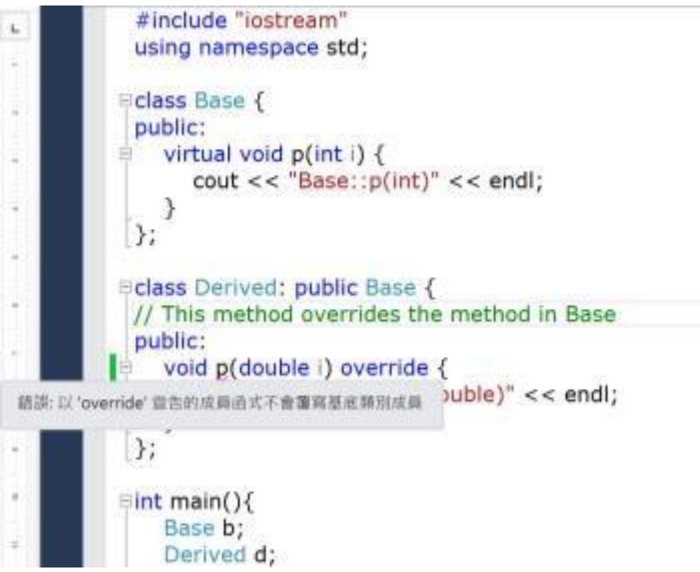
```
#include "iostream"
using namespace std;

class Base
{ public:
    virtual void p(int i) {
        cout << "Base::p(int)" << endl;
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(double i) {
        cout << "Derived::p(double)" << endl;
    }
};

int main(){
    Base b;
```

```
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    return 0;
}
```

Q: What are the outputs?

A:

To avoid this unintentional mistakes, you can specify `override` in the derived class after the parameter list in C++11 for compiling check:



<u>Possible unintentional mistake</u>: you forget to define `virtual` member function in the base class. Again this will unintentionally perform overloading:

OverrideExWrong2.cpp

```
#include "iostream"
using namespace std;

class Base
{ public:
    virtual void p(int i) {
        cout << "Base::p(int)" << endl;
```

```
    }
};

class Derived: public Base {
// This method overrides the method in Base
public:
    void p(int i) {
        cout << "Derived::p(int)" << endl;
    }
};

int main(){
    Base b;
    Derived d;
    Base* dp = new Derived();
    b.p(10);
    d.p(10);
    dp->p(10);
    return 0;
}
```

Q: What are the outputs?

A:

(see ppt)

15.3 Dynamic Binding 動態繫結

The aforementioned implicit derived-to-base conversion is the key behind **dynamic binding**. Through **dynamic binding**, we can use the same code to process objects of either type `Quote` or `Bulk_quote` **interchangeably**. For example, the following function prints the total price for purchasing the given number of copies of a given book:

```
//
double print_total(ostream& os, const Quote& item, size_t n)
{
// depending on the type of the object bound to the item
parameter
// calls either Quote::net_price or Bulk_quote::net_price
  double ret = item.net_price(n);
  os << "ISBN: " << item.isbn() // calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
  return ret;
}
```

This function is pretty simple—it prints the results of calling `isbn` and `net_price` on its parameter and returns the value calculated by the call to `net_price`. Because the `item` parameter is a **reference** to `Quote`, we can call this function on either a `Quote` object or a `Bulk_quote` object.

And because `net_price` is a `virtual` member function, the version of `net_price` that is run will depend <u>on the type of the object</u> that we pass to `print_total`:

```
// basic has type Quote; bulk has type Bulk_quote
Quote basic;
Bulk_quote bulk;
print_total(cout, basic, 20); // calls Quote version of net_price
print_total(cout, bulk, 20); // calls Bulk_quote version of
net_price
```

<u>Example:</u> we are now ready to put these codes together
UseDynamicBinding.cpp

```
#include "Quote.h"
#include "Bulk_Quote.h"
#include <iostream>

using namespace std;

// calculate and print the price for the given number of copies,
applying any discounts
double print_total(ostream& os, const Quote& item, size_t n)
{
    // depending on the type of the object bound to the item
parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() // calls Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}

int main(){
    // basic has type Quote; bulk has type Bulk_quote
    Quote basic("032-171-4113", 42.38); // C++ Primer ISBN-10
    Bulk_quote bulk("032-171-4113", 42.38, 10, 0.2); // C++ Primer
ISBN-10
    Quote *pBulk = new Bulk_quote("978-0321714114", 42.38, 10,
0.2);
// C++ Primer ISBN-13
    print_total(cout, basic, 20); // calls Quote version of
net_price
```

```
    print_total(cout, bulk, 20); // calls Bulk_quote version of
net_price
    print_total(cout, *pBulk, 20); // calls Bulk_quote version of
net_price
    delete pBulk;
    return 0;
}
```

```
ISBN: 032-171-4113 # sold: 20 total due: 847.6
ISBN: 032-171-4113 # sold: 20 total due: 678.08
ISBN: 978-0321714114 # sold: 20 total due: 678.08
Press any key to continue . . .
```

Remarks:

1 In C++, dynamic binding happens when <u>a virtual member function is called through a reference (or a pointer) to a base class</u>. The run-time selection of `virtual` functions to run is relevant only the function is called through a reference or a pointer.

2 If we call the `virtual` function on behalf of an object (as opposed through a reference or a pointer), then we know the exact type of the object at the compile time. <u>The type is fixed and it does not vary at run time</u>.

3 But in contrast, a reference or pointer to a base-class object may refer or point to a base-class object or to an object of a type derived from the base class.

4 This means that the type of object to which a reference or a pointer is bound may differ at run time. We call this **dynamic binding** or **late binding**.

Interesting (Ironic) Observation: In C++, we cannot achieve **object**-oriented programming through **object**. We **MUST** to use reference or pointer!

<u>Ex15-1.cpp</u>

<u>In-Class Exercise 15.1</u>: Write two classes, `Electricity` and `Heat`, which are both derived from a base class `Energy`. Make your code support the following `main` function and produce the following sample run.

```
int main ()
{
    Energy* e = new Electricity ;
    cout << e << endl ;
    Energy* h = new Heat ;
    cout << h << endl ;
    delete e;
    delete h;
    return 0 ;
}
```

```
Electricy works
Heat works
請按任意鍵繼續 . . .
```

A:

15.4 Polymorphism 多型

Polymorphism (多型) 是物件導向語言最重要的機制，多型可以讓我們以相同的方法處理父類別（base class）與子類別（derived class）的物件。我們可以運用多型的機制寫出非常有彈性的程式。

15.4.1 The Basics

Let us consider a base class `Shape` with derived classes `Circle` and `Triangle` to demonstrate the key mechanism of polymorphism:

Shape.h

```
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>

class Shape
{ public:
    Shape() { std::cout << "Shape::Shape()" << std::endl; }
    virtual void draw() const { std::cout << "Shape::draw()" <<
std::endl; }
};

class Circle : public Shape
{ public:
    Circle() { std::cout << "Circle::Circle()" << std::endl; }
    void draw() const { std::cout << "Circle::draw()" <<
std::endl; }
};

class Triangle : public Shape
{ public:
    Triangle() { std::cout << "Triangle::Triangle()" <<
std::endl; }
    void draw() const { std::cout << "Triangle::draw()" <<
std::endl; }
};

#endif
```

UseShape.cpp

```
#include "Shape.h"

void display(const Shape&
    s){ s.draw();
}

int main(){
    Shape* s = new Shape();
    display(*s);
    Shape* c = new Circle();
```

```
    display(*c);
    Shape* t = new Triangle();
    display(*t);
    return 0;
}
```

Q: What are the outputs?

A:

The method `display` takes a parameter of the `Shape` type. We can call `display` by passing any object of `Shape` and any object from its derived class (`Circle`, `Triangle`).

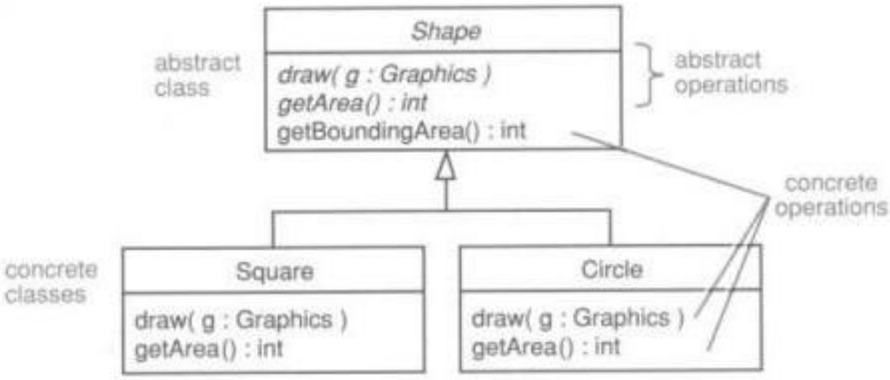
This is commonly known as polymorphism (from a Greek word meaning “many forms”). In simple terms, **polymorphism means an object of a derived class can be used wherever its base class object is used**.

Q: 那 … 彈性在哪裡?

A:

15.4.2 Serious Polymorphism: Abstract Base Class

(see ppt)



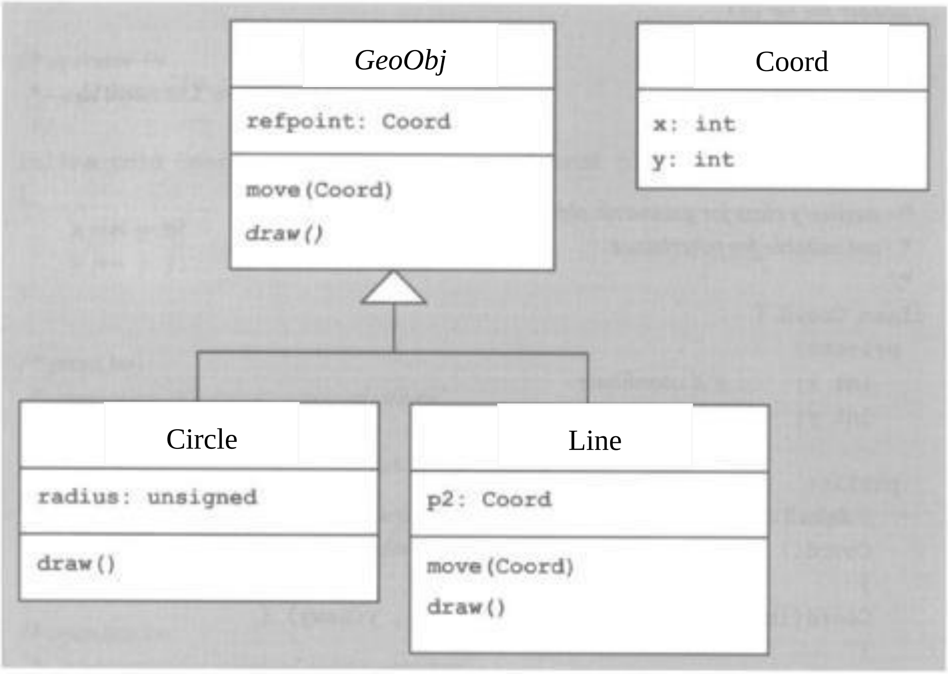
The *Shape* class is an abstract base class (ABC) in C++. Its definition looks like:

```
class Shape
{ public:
    virtual void draw(Graphics g) = 0;
    virtual int getArea() = 0;
    int getBoundingArea(); // to be implemented in Shape.cpp
    virtual ~Shape() = default;
}
```

Remarks:

1. In C++, an **abstract method** is implemented through pure virtual member function. It means the member function must be **overridden**.
2. In C++, a class with **abstract methods** is an abstract class.
3. With an abstract base class, we guarantee that anyone doing the work at runtime is **an object of a derived class** of the **abstract base class**. The creation of derived class is allowed only if all the pure virtual functions have been implemented.

Example: Let us now work through a complete example to demonstrate polymorphism through the abstract base class design. Again let us consider a `GeoObj` class hierarchy shown below.



Before we dive into abstract base class, let's take a quick look of the auxiliary class `Coord`

Coord.h

```
#ifndef COORD_H
#define COORD_H
#include <iostream>

class Coord
  { private:
    int x;     // X coordinate
    int y;     // Y coordinate
  public:
    // default constructor, and two-parameter constructor
    Coord() : x(0), y(0) {     // default values: 0
    }
    Coord(int newx, int newy) : x(newx), y(newy) {
    }

    Coord operator + (const Coord&) const;    // addition
    Coord operator - () const;               // negation
    void  operator += (const Coord&);        // +=
    void  printOn(std::ostream& strm) const;  // output
};

inline Coord Coord::operator + (const Coord& p) const
{
    return Coord(x+p.x,y+p.y);
}

inline Coord Coord::operator - () const
{
    return Coord(-x,-y);
}

inline void Coord::operator += (const Coord& p)
```

```
{
    x += p.x;
    y += p.y;
}

inline void Coord::printOn(std::ostream& strm) const
{
    strm << '(' << x << ',' << y << ')';
}

inline std::ostream& operator<< (std::ostream& strm, const Coord&
p)
{
    p.printOn(strm);
    return strm;
}

#endif // COORD_H
```

In this example, two types of geometric objects `Line` and `Circle` are regarded as being geometric objects under the **common term** `GeoObj` (abstract base class). All the geometric objects have a reference point. All the geometric objects can be moved with `move()` and drawn with `draw()`. For `move()`, there is a default implementation that simply moves the reference point accordingly.

A `Circle` additionally has a radius and implements the `draw()` function (is it necessary?). The function for moving is inherited (what does it mean?).

A `Line` has a second point (the first point is the reference point) and implements the `draw()` function. `Line` also reimplements the function for moving (what does it mean?).

The abstract base class `GeoObj` defines the **common** attributes and operations:

GeoObj.h

```
#ifndef GEOOBJ_H
#define GEOOBJ_H

#include "Coord.h"

class GeoObj {
  protected:
    // every GeoObj has a reference point
    Coord refpoint;
    GeoObj(const Coord& p) : refpoint(p) {
    }

  public:
    virtual void move(const Coord& offset)
        { refpoint += offset;
    }
```

```
    virtual void draw() const = 0;
    virtual ~GeoObj() = default;
};

#endif  // GEOOBJ_H
```

Circle.h

The derived class `Circle`

```
#ifndef CIRCLE_H
#define CIRCLE_H

// header file for I/O
#include <iostream>

#include "GeoObj.h"

class Circle : public GeoObj
  { protected:
    unsigned radius;    // radius

  public:
    // constructor for center point and radius
    Circle(const Coord& m, unsigned r)
         : GeoObj(m), radius(r) {}

    // draw geometric object (now implemented)
    virtual void draw() const;

    // virtual destructor
    virtual ~Circle() {}
};

inline void Circle::draw() const
{
    std::cout << "Circle around center point " << refpoint
              << " with radius " << radius << std::endl;
}

#endif // CIRCLE_H
```

Line.h

The derived class `Line`

```
#ifndef LINE_H
#define LINE_H

#include <iostream>
#include "GeoObj.h"

class Line : public GeoObj
  { protected:
    Coord p2;    // second point, end point
```

```
  public:
    Line(const Coord& a, const Coord& b)
         : GeoObj(a), p2(b) {}

    virtual void draw() const;
    virtual void move(const Coord&);
    virtual ~Line() {}
};

inline void Line::draw() const
{
    std::cout << "Line from " << refpoint
            << " to " << p2 << std::endl;
}

inline void Line::move(const Coord& offset)
{
    refpoint += offset;    // represents GeoObj::move(offset);
    p2 += offset;
}

#endif // LINE_H
```

<u>UseGeoObj.cpp</u>

Application example

```
#include "Line.h"
#include "Circle.h"
#include "GeoObj.h"

// forward declaration
void printGeoObj(const GeoObj&);

int main()
{
    Line l1(Coord(1,2), Coord(3,4));
    Line l2(Coord(7,7), Coord(0,0));
    Circle c(Coord(3,3), 11);

    // array as an inhomogenous collection of geometric objects:
    GeoObj* coll[10];

    coll[0] = &l1;    // collection contains: - line l1
    coll[1] = &c;     //     - circle c
    coll[2] = &l2;    //     - line l2

    /* move and draw elements in the collection
     * - the correct function is called automatically
     */
    for (int i=0; i<3; i++)
        { coll[i]->draw();
        coll[i]->move(Coord(3,-3));
    }

    // output individual objects via auxiliary function
```

```
    printGeoObj(l1);
    printGeoObj(c);
    printGeoObj(l2);
}

void printGeoObj(const GeoObj& obj)
{
    /* the correct function is called automatically
     */
    obj.draw();
}
```

Q: What are the outputs?

A:

```
Line from (1,2) to (3,4)
Circle around center point (3,3) with radius 11
Line from (7,7) to (0,0)
Line from (4,-1) to (6,1)
Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)
Press any key to continue . . .
```

Another case to illustrate the advantages of polymorphism: a derived class capable of combining multiple geometric objects together to form a group of geometric objects.

<u>GeoGroup.h</u>

```
#ifndef GEOGROUP_H
#define GEOGROUP_H

// include header file of the base class
#include "GeoObj.h"

// header file for the internal management of the elements
#include <vector>


/* class GeoGroup
 * - derived from GeoObj
 * - a GeoGroup consists of:
 *     - a reference point (inherited)
 *     - a collection of geometric elements (new)
 */
class GeoGroup : public GeoObj
  { protected:
    std::vector<GeoObj*> elems;   // collection of pointers to
GeoObjs

  public:
    // constructor with optional reference point
    GeoGroup(const Coord& p = Coord(0,0)) : GeoObj(p) {
    }

    // output (now also implemented)
```

```
    virtual void draw() const;

    // insert element
    virtual void add(GeoObj&);

    // remove element
    virtual bool remove(GeoObj&);

    // virtual destructor
    virtual ~GeoGroup() = default;
};

#endif  // GEOGROUP_H
```

GeoGroup.cpp

```
#include "GeoGroup.h"
#include <algorithm>

/* add
 * - insert element
 */
void GeoGroup::add(GeoObj& obj)
{
    // keep address of the passed geometric object
    elems.push_back(&obj);
}

/* draw
 * - draw all elements, taking the reference points into account
 */
void GeoGroup::draw() const
{
   for (const auto& e : elems) {
       e->move(refpoint);   // add offset for the reference point
       e->draw();          // draw element
       e->move(-refpoint);  // subtract offset
   }
}

/* remove
 * - delete element
 */
bool GeoGroup::remove(GeoObj& obj)
{
    // find first element with this address and remove it
    // return whether an object was found and removed
    std::vector<GeoObj*>::iterator pos;
    pos = std::find(elems.begin(),elems.end(),&obj);
    if (pos != elems.end()) {
        elems.erase(pos);
        return true;
    }
    else {
        return false;
```

```
    }
}
```

UseGeoGroup.cpp

Application example with `GeoGroup` derived class

```
#include <iostream>

// header files for used classes
#include "Line.h"
#include "Circle.h"
#include "GeoGroup.h"

int main()
{
   Line l1(Coord(1, 2), Coord(3, 4));
   Line l2(Coord(7, 7), Coord(0, 0));
   Circle c(Coord(3, 3), 11);

   GeoGroup g;

   g.add(l1);           // GeoGroup contains: - line l1
   g.add(c);            //     - circle c
   g.add(l2);           //     - line l2

   g.draw();            // draw GeoGroup
   std::cout << std::endl;

   g.move(Coord(3, -3));  // move offset of GeoGroup
   g.draw();                // draw GeoGroup again
   std::cout << std::endl;

   g.remove(l1);         // GeoGroup now only contains c and l2
   g.draw();            // draw GeoGroup again
}
```

Q: What are the outputs?

A:

```
Line from (1,2) to (3,4)
Circle around center point (3,3) with radius 11
Line from (7,7) to (0,0)

Line from (4,-1) to (6,1)
Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)

Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)
Press any key to continue . . .
```

Remarks: What's the beauty of introducing `GeoGroup`?

1 Note that interface of the `GeoGroup` hides the internal use of pointers. Thus the application programmers need only pass the objects that need to get inserted or removed.

2 The `GeoGroup` contains no code that refers to **any concrete type** of the geometric objects it contains. Thus if a new geometric object, such as `Triangle` is introduced, we only need to make sure that `Triangle` is derived from `GeoObj` and that's it.

(see ppt)

15.4.3 Container and Inheritance

Let us consider our bookstore example with the base class `Quote` and the derived class `Bulk_quote` that offers different pricing policies:

```
class Quote
{ public:
  Quote() = default;
  Quote(const std::string &book, double sales_price):
    bookNo(book), price(sales_price) { }
  std::string isbn() const { return bookNo; }
// returns the total sales price for the specified number of
items
// derived classes will override/apply different discount
algorithms
virtual double net_price(std::size_t n) const
 { return n * price; }
virtual ~Quote() = default; // dynamic binding for the destructor
private:
  std::string bookNo; // ISBN number of this item
protected:
  double price = 0.0; // normal, undiscounted price
};
```

```
class Bulk_quote : public Quote { // Bulk_quote inherits from
Quote
public:
  Bulk_quote() = default;
  Bulk_quote(const std::string& book, double p, std::size_t qty,
double disc) : Quote(book, p), min_qty(qty), discount(disc) { }
// overrides the base version in order to implement the bulk purchase discount
policy
  double net_price(std::size_t) const override;
private:
  std::size_t min_qty = 0; // minimum purchase for the discount to
apply
  double discount = 0.0; // fractional discount to apply
};
```

```
double Bulk_quote::net_price(size_t cnt) const
{
if (cnt >= min_qty)
return cnt * (1 - discount) * price;
else
return cnt * price;
}
```

Before C++11: Use Base-class **Raw** Pointer for Polymorphism

```
int main()
{
    vector<Quote*> basket;
    Quote* q0 = new Quote("0-201-82470-1", 50.0);
    basket.push_back(q0);
    Quote* q1 = new Bulk_quote("0-201-54848-8", 50.0, 10, .2);
    basket.push_back(q1);
    double totalPrice = 0.0;
    for (auto e: basket) {
        totalPrice += e->net_price(10);
    }
    cout << "Total price: " << totalPrice << endl;
    delete q0;
    delete q1;
}
```

Q: What is the output?

A:

```
Total price: 900
```

After C++11: Use Base-class **Smart** Pointer for Polymorphism (self study)