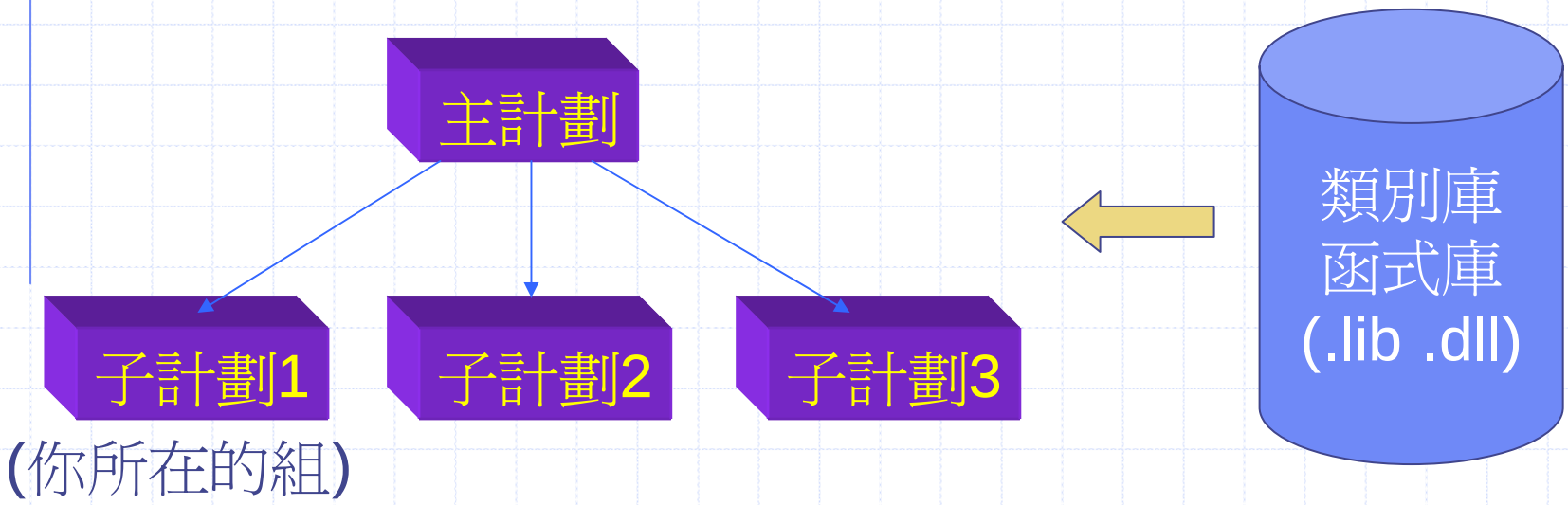
繼承的優點

- 程式碼再使用 (code reuse)
- 抽象概念再使用
- 類別階層化
  - 澄清物件間的關係

繼承與Code Reuse(被動)



繼承與Code Reuse

```
class List {
    ……
    void insert() {…}
    void delete() {…}
} ;
```

如果你對class list 感到
(1) 功能不足 (2) 原有功能不佳

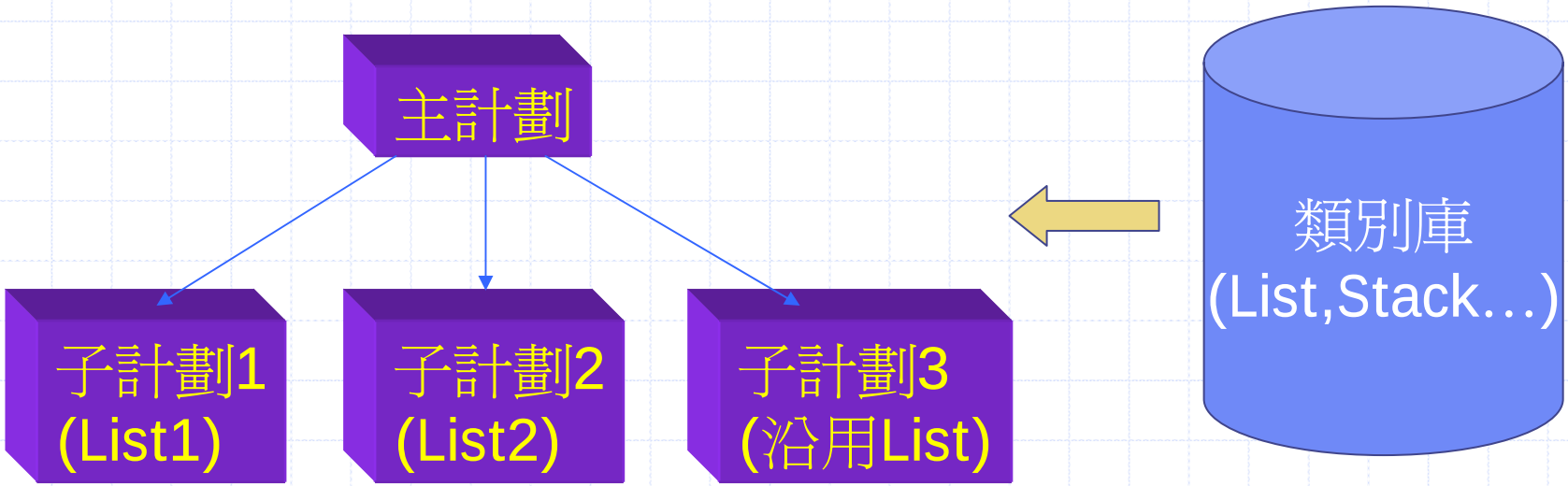
重新改寫???
(1) 原始碼在哪?
(2) 還有其他使用者，是否都同意改寫?

使用繼承來改善

繼承與Code Reuse

```
class List {….. insert(int n) ;…. delete(int pos);};

class List1: public List {
    node& operator[](int index) ; //新增
    void insert(int n) ; // 修改: 更高效率的實作方式
};
void main() {
    List1 L;      …..      ; L.insert(15) ;
    cout << L[3] ;
}
```

可能結果



繼承與Code Reuse(主動)

Q: 設計一物件導向資料庫: 儲存學校學生資料

[先修生]	[僑生]	[交換學生]	[一般生]
學號	學號	學號	學號
系級	系級	系級	系級
高中校名	e-mail	e-mail	e-mail
	國籍	國籍	
選課()	選課()	選課()	選課()
註冊()	註冊()	註冊()	註冊()
	工讀()	定期約談()	工讀()

作業: list ← stack

```
struct node {int info; node* next; };
class list {
    node *head, *tail ; int node_no ;
public:
    list() ;          list(const node& n) ;      list(const list& L) ;
    ~list() ;
    int getSize() ;
    void insert(int pos, int value) ; // 0: first, node_no:last
    void delete(int pos) ; // 刪除第pos個元素
    void show(string msg) ; //印出串列內容
    list& operator=(const list& L) ;
    friend list operator+(const list& L1, const list& L2) ; //聯集
    friend list operator*(const list& L1, const list& L2) ; //交集
    friend list operator-(const list&L1, const list& L2) ;  //差集
} ;
```

使用list產生stack

```
class stack: public list {
    stack() ;
    ~stack() ;
    operator=(const stack& s) ;
    void push(int x) ; //加在list的最前端
    int pop() ;        //刪除list的第一個元素
    // list中的operator+, -, * 是否也被繼承?
} ;
```

測試class list

```
void main() {
    list L1, L2, L3 ;
    for (int i = 101; i<=108; i++) L1.insert(L1.getSize(),i) ;
    for (int j = 110; j>=105; j--) L2.insert(L2.getSize(), j) ;
    L1.show("L1="); L2.show("L2=") ;
    L3 = L1 + L2 ;          L3.show("L3=L1+L2=") ;
    L3 = L1 * L2;  L3.show("L3=L1*L2=") ;
    L3 = L1 – L2;  L3.show("L3=L1-L2=") ;
    L3.delete(1) ; L3.delete(2) ;  L3.show("after 2 delete, L3=") ;
    stack s1, s2 ;
    for (int k=1; k<=10; k++) {
        if (k%3==0) s1.pop();
        s1.push(k) ; s1.show("s1=") ;
    }
    s2 = s1;        s2.show("s2=") ;
}
```

第十章 多型與虛擬函數 (Polymorphism & Virtual Functions)

10-1 衍生類別的指標

10-2 簡介虛擬函數

10-3 虛擬函數的細節

10-4 應用多型

多型

- 編譯時期多型(靜態多型)
  - function overloading
    - 如何正確呼叫同名的函數? 利用參數個數與型態
  - operator overloading
    - 其實同function overloading
- 執行時期多型 (或動態多型)
  - 如何正確呼叫不同物件的相同名稱的成員函數 → 利用繼承與多型

衍生類別與基底類別物件間的指定(assignment)

```
class base {
    int x ;
public:
    setx(int n) { x=n;}
} ;
class derived: public base {
    int y ;
public:
    setx(int n) { base::setx(3*n);}
    sety(int n) { y = n;}
} ;
```

```
void main() {
    base b ;
    derived d ;
    b = d ; // 可乎?
    b.setx(5) ; // 哪個setx()
    b.sety(10); //?

    d = b ; // ?
    d.setx(5) ; d.sety(8); // ?
}
```

結論

- base Obj = derived Obj (可)
- derived Obj = base Obj (否)

```
void main() {
        base b ;
        derived d ;
        b = d ; // 可
        b.setx(5) ;
      // 哪個setx()
        b.sety(10); //? 否

        d = b ; // ?否
        d.setx(5) ; d.sety(8);
// ?可
```

10-1 衍生類別的指標

- **Case 1**

```
void main() {
    base *pb ;
    base b;  derived d ;
    pb = &b ; // Sure!
    pb->setx(5) ;

    pb = &d ; // 可乎?
    pb->setx(5) ; // 哪個?
    pb->sety(10); // ? 否
}
```

- case 2

```
void main() {
    derived *pd ;
    base b;  derived d ;
    pd = &b ; // ??? 否
    pd->setx(5) ; // ?否
}
```

衍生類別的參考(reference)

- **Case 1**

```
void main() {
    base b;  derived d ;
    base &refb1 = b ; // sure
    refb1.setx(5) ;
   // 哪個? base

    base& refb2 = d ; // ?? 可
    refb2.setx(5) ;
    refb2.sety(10); // ?? 否
}
```

- case 2

```
void main() {
    base b;  derived d ;
    derived &refd1 = b ;
    //?否
    refb1.setx(5) ;
}
```

結論: 自己寫

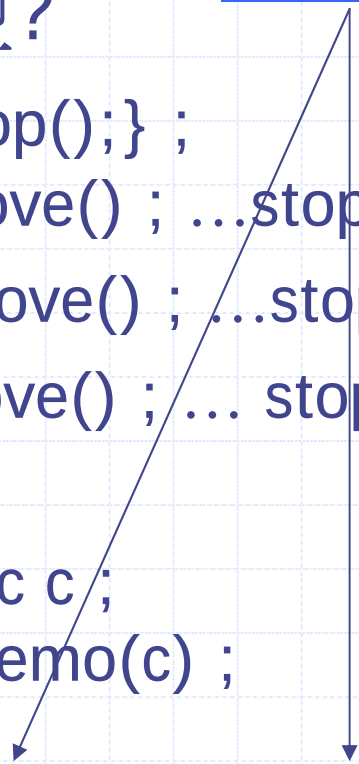
- base-pointer = &derived-Obj (可)
- base-reference = derived-Obj (可)
- derived-pointer = &base-Obj (否)
- derived-reference = base-Obj (否)

10-2 多型與虛擬函數

是否能呼叫到正確的move()與stop();

- 甚麼是執行時期的多型?

```
class car { … move() ; …stop();} ;
class Benz: public car {…move() ; …stop();} ;
class Volvo: public car {…move() ; …stop(); } ;
class Civic: public car {…move() ; … stop(); } ;
void main() {
        Benz b;  Volvo v; Civic c ;
        demo(b); demo(v); demo(c) ;
}
void demo(car& c) {  c.move() ;  c.stop() ;}
```



不使用虛擬函數

實際try!
輸出結果爲何?

```
class car {
    public:void move() { cout << “car move”;}
} ;
class Benz: public car {
    public:void move() { cout << “Benz move”;}
} ;
class Volvo: public car {
    public:void move() { cout << “Volvo move”;}
} ;
void demo(car& c) { c.move() ; }
void main() { Benz b; Volvo v; demo(b); demo(v) ; }
```

甚麼是虛擬函數?

- 是一種宣告在基底類別中的成員函數
- 提供執行時期多型的機制
- 使用virtual保留字
- 通常衍生類別會override它

使用虛擬函數 (配合reference)

```
class car {
    virtual void move() { cout << “car m
} ;
class Benz: public car {
    void move() { cout << “Benz move”;}
} ;
class Volvo: public car {
    void move() { cout << “Volvo move”;}
} ;
void demo(car& c) { c.move() ; }
void main() { Benz b ;Volvo v; demo(b); demo(v) ; }
```

實際try!
輸出結果爲何?

使用虛擬函數(配合pointer)

實際try!
輸出結果爲何?

```
class car {
    virtual void move() { cout << “car move”;}
} ;
class Benz: public car {
    void move() { cout << “Benz move”;}
} ;
class Volvo: public car {
    void move() { cout << “Benz move”;}
} ;
void demo(car *pc) { pc->move() ; }
void main() { Benz b ;Volvo v; demo(&b); demo(&v) ; }
```

使用虛擬函數 (配合物件傳遞)

實際try!
輸出結果爲何?

```
class car {
    virtual void move() { cout << “car move”;}
} ;
class Benz: public car {
    void move() { cout << “Benz move”;}
} ;
class Volvo: public car {
    void move() { cout << “Benz move”;}
} ;
void demo(car c) { c.move() ; }
void main() { Benz b ;Volvo v; demo(b); demo(v) ; }
```

不使用多型可以嗎?

多型: 一個介面多種用法

```
void move(car& c) { c.move() ; ….}
```

不多型:

```
void move(void *p, int type) {
        switch(type){
                case 1: ((Benz *)p)->move(); break ;
                case 2: ((Volvo *)p)->move(); break ;
                ……
        }
}
```

練習

```
class plane {
    virtual void fly() { takeoff(); onAir(); landing();}
    void onAir() {……}
    void takeoff() {……}
    void landing() {……}
} ;
// 你不滿意takeoff的行爲該如何?
```

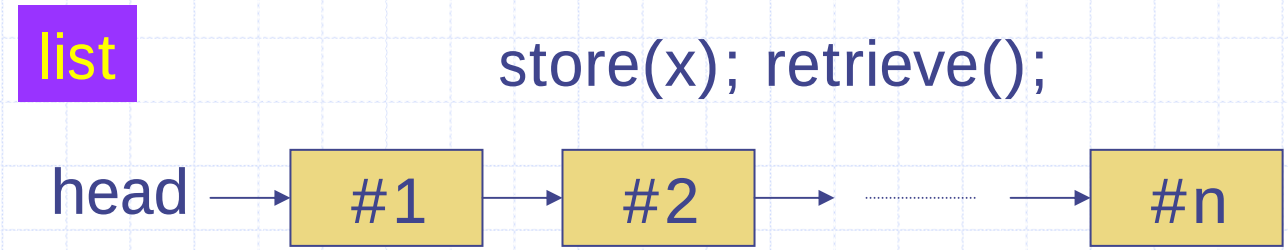
Plane

```
class plane{
public:
    virtual void fly()
    { takeoff(); onAir();
    landing();}
    void onAir()
    {cout<<"onAir"<<endl;}
virtual   void takeoff()
    {cout<<"takeoff"<<end
    l;}
    void landing()
    {cout<<"landing"<<end
    l;}
};
```

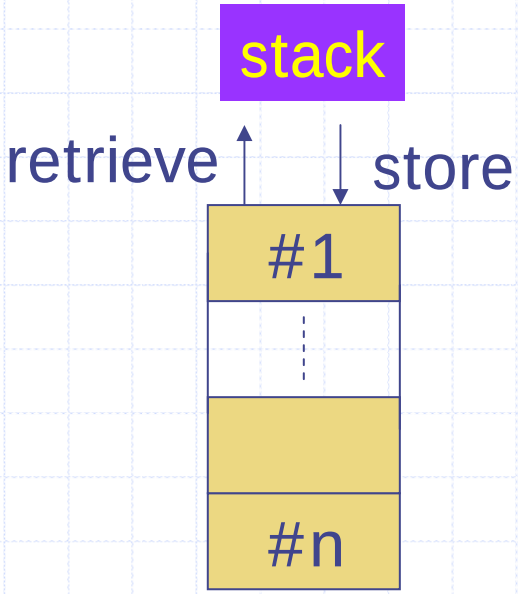
```
class F16:public plane{
public:
//          void fly(){cout<<"F16 fly"<<endl;}
            void takeoff(){cout<<"F16
takeoff"<<endl;}
};

class B747:public plane{
public:
//          void fly(){cout<<"B747 fly"<<endl;}
            void takeoff(){cout<<"B747
takeoff"<<endl;}
};
```

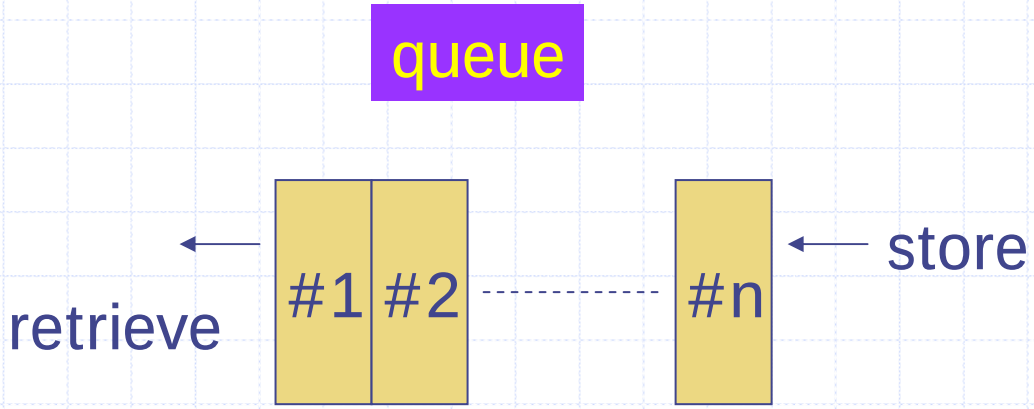
```
void demo(plane &p){p.fly();}
void main()
{
    B747 b;
            F16 f;
            demo(b);
            demo(f);
}
```

Case Study: p. 10-19



如何利用list來模擬(實作) stack與queue





繼承示意圖

```
class list {
        …
        virtual void store(int i) ;
        virtual int retrieve() ;
} ;
```

```
class stack: public list {
        …
        void store(int i) ;
        int retrieve() ;
} ;
```

```
class queue: public list {
        …
        void store(int i) ;
        int retrieve() ;
} ;
```

10-3 更多虛擬函數的細節

- 純粹虛擬函數(pure virtual function)

```
class printer {
    string filename ;
public:
    void reset() {… }
    virtual void print(int m)=0 ;
}
```

- 抽象類別(abstract class)
  - 當類別至少含有一個純粹虛擬函數時
  - 不能用來產生物件

    e.g. printer p ; //XX

純粹虛擬函數的內容

- 純粹虛擬函數(pure virtual function)

```
class printer {
    string filename ;
public:
    virtual void reset()=0;
    virtual void print(int mode)=0 ;
}
```

純粹虛擬函數的特性

- 衍生類別一定要override 基底類別中所有的純粹虛擬函數

```
class printer {……};
class HPLaserJet6L: public printer {
    ……
    void reset() { …自己的版本…}
    void print(int mode) {…自己的版本…}
};
```

抽象類別的用途(一)

- 設計共同的使用介面的類別(衍生類別負責實作)

```
class shape {
    string name ;
public:
    virtual void draw(char b[][80])=0 ; // 不必有實作
    virtual void clear()=0 ; //不必有實作
};
class triangle: public shape {……} ;
class circle: public shape {……};
```

抽象類別的用途(二)

- 防止使用者產生不允許存在的物件

```
class shape {
    string name ;
public:
    virtual void draw()=0 ;
};
class triangle: public shape {……} ;
class circle: public shape {……};
void main() { shape s ; /* what ??? */ ….. }
```

10-4 應用多型

- 早期繫結(early binding)或編譯時期繫結(compiling time binding)
  - 一般函數
  - 超載函數
  - 夥伴函數
  - 非虛擬之成員函數
- 晚期繫結(late binding)或執行時期繫結(run-time binding)
  - 虛擬函數 (效率較差)

例子

- early binding

```
void fun(int x) {
    cout << x;
}
void main() {
    fun(5) ; // early binding
}
```

- late binding

```
// 承前例
void move(car& c) {
    c.move() ; // late binding
}
void main() {
    Benz b; Volvo v ; int x ;
    cin >> x ;
    if (x%2) b.move(); …..
```

繼承與Code Reuse(主動)

Q: 設計一物件導向資料庫: 儲存學校學生資料

[先修生]	[僑生]	[交換學生]	[一般生]
學號	學號	學號	學號
系級	系級	系級	系級
高中校名	e-mail	e-mail	e-mail
	國籍	國籍	
選課()	選課()	選課()	選課()
註冊()	註冊()	註冊()	註冊()
	工讀()	定期約談()	工讀()

練習

```
class student {
protected:
    string studID, name, eMail ;
public:
    void fillData()=0 ;
    void getID() { return ID; }
    void show()=0 ;
} ;
class LocalStudent: public student {
    string ID ; ……
}
class AbroadStudent: public student {
    string passportID; string country ;……
}
```

續

```
class IMStudents {
    const int MAX_STUD ;
    student *stud[720] ;
public:
    students(string filename=“”):MAX_STUD(720) { }
    void addData(string ID) ;
    int search(string ID) ;
} ;
void main() {
    IMStudents ims(“Imdata.txt”) ;
    while (true) {

    }
}
```