第十三章　再論繼承與類別樣版

私有繼承

C++ 的繼承方式除了公用繼承外，還有**私有繼承**（private inheritance）。因為私有繼承可使基礎類別的公用及保護成員變成衍生類別的私有成員，意思是基礎類別的成員函數不是衍生類別的公用介面，它們只能在衍生類別的成員函數中使用。

若為公用繼承，基礎類別的公用成員函數會成為衍生類別的公用成員函數。簡而言之，衍生類別繼承基礎類別的介面。而私有繼承，基礎類別的公用成員函數就成為衍生類別的私有成員函數。

私有繼承是在定義類別時用關鍵字 private 而非 public。（實際上，private 為預設值，若無此關鍵字，系統自動設為私有繼承。）以 C 類別為例，要繼承 A、B 兩個類別，其片段程式如下：

```
class C : private A, private B
{
   public:
   …
};
```

表示 C 類別以私有繼承的方式繼承了 A 和 B 兩個類別。而

```
class C : public A, B
{
   public:
   …
};
```

則表示 C 以公用繼承的方式繼承 A，而以私有繼承的方式繼承 B。

有一個以上的基礎類別，此種繼承稱為**多重繼承**（Multiple inheritance）或 MI。多重繼承，特別是公用的多重繼承會導致一些問題，需要額外的規則來解決。稍後會討論這些問題。

保護繼承

C++ 第三種繼承的方式是**保護繼承**（Protected inheritance），它是私有繼承的變形，所用的關鍵字 protected，如下所示：

```
class C : protected A, protected B
{
public:
    …
};
;
```

保護繼承使基礎類別的公用和保護成員變成衍生類別的保護成員。私有和保護繼承的不同處是若從衍生類別中再衍生另一個類別。私有繼承的第三代類別就不能使用第一代基礎類別的公用介面，因為第一代基礎類別的公用成員函數在衍生類別中變成私有，第三代衍生類別就不能直接存取上一代的私有成員。保護繼承使第一代基礎類別的公用成員函數在第二代變成保護模式，在第三代還是可以直接使用了。

表 13-1 總結了公用，私有和保護繼承的一些性質。

表 13-1 各種的繼承

性質	公用繼承	保護繼承	私有繼承
基礎類別的公用成員變成	衍生類別的公用成員	衍生類別的保護成員	衍生類別的私有成員
基礎類別的保護成員變成	衍生類別的保護成員	衍生類別的保護成員	衍生類別的私有成員
基礎類別的私有成員變成	只能透過基礎類別介面存取	只能透過基礎類別介面存取	只能透過基礎類別介面存取

多重繼承

多重繼承是繼承一個以上基礎類別的衍生類別。例如，若有 Waiter 和 Singer 類別，衍生出 SingingWater 類別，表示如下：

```
class SingingWaiter : public Waiter, public Singer{…};
```

每個基礎類別前加關鍵字 public，是因為編譯程式的預設模式為私有繼承：

```
class SingingWaiter : public Waiter, Singer{…}
                                        //Singer is a private base
```

則表示 SingingWaiter 繼承 Waiter 和 Singer 類別，但繼承的方式 Waiter 是公有繼承，而 Singer 是私有繼承。

多重繼承會帶給程式設計師新的問題，如從不同的基礎類別繼承同名的成員函數…等等，因此使用多重繼承會比單一繼承來得困難，且易發生問題。

接下來，我們以一個範例 13-1 來說明多重繼承有那些問題和解決之道。假設有一 Worker 類別，直接衍生 Waiter 和 Singer 類別，再用多重繼承衍生出 SingingWaiter 類別，如圖 13-1。

範例程式 13-1　workerfd.h

```
// workerfd.h  -- working classes, first draft
#include "strng2.h"
class Worker
{
private:
    String fullname;
    long id;
protected:
    virtual void data() const;
public:
    Worker() : fullname("no one"), id(0L) {}
    Worker(const String & s, long n)
```

```
                : fullname(s), id(n) {}
    virtual void set();
    virtual void show() const;
};

class Waiter : public Worker
{
private:
    int panache;
protected:
    void data() const;
public:
    Waiter() : Worker(), panache(0) {}
    Waiter(const String & s, long n, int p = 0)
                : Worker(s, n), panache(p) {}
    Waiter(const Worker & wk, int p = 0)
                : Worker(wk), panache(p) {}
    void set();
    void show() const;
};

class Singer : public Worker
{
public:
    enum {Vtypes = 7};
protected:
enum {other, alto, contralto, soprano,
                  bass, baritone, tenor};
private:
    static char *pv[Vtypes];   // string equivs of voice types
    int voice;
protected:
    void data() const;
public:
    Singer() : Worker(), voice(other) {}
    Singer(const String & s, long n, int v = other)
                : Worker(s, n), voice(v) {}
    Singer(const Worker & wk, int v = other)
                : Worker(wk), voice(v) {}
    void set();
    void show() const;
};
```

圖 13-1　共用祖先的多重繼承

多少個工作者？

先從衍生自 Singer 和 Waiter 的 SingingWaiter 開始：

```
class SingingWaiter:public Singer, public Waiter{…};
```

因為 Singer 和 Waiter 繼承了 Worker 的類別，故 SingingWaiter 有兩份 Worker 元件，如圖 13-2。

正如所料，這是一個問題。例如，要將衍生類別物件的位址設給基礎類別的指標時會產生模稜兩可：

```
SlngingWaiter ed;
Worker *pw = &ed;          //產生模稜兩可
```

此設定的一般作法是將基礎類別指標指向衍生類別內基礎類別物件的位址。但 ed 含有兩個 Worker 物件，該選擇那一個物件的位址呢？此時必須使用型態轉換來指明是那一個物件，如下所示：

```
Worker * pw1 = (Waiter *) &ed;     //the Worker in Waiter
Worker * pw2 = (Singer *) &ed;     //the Worker in Singer
```

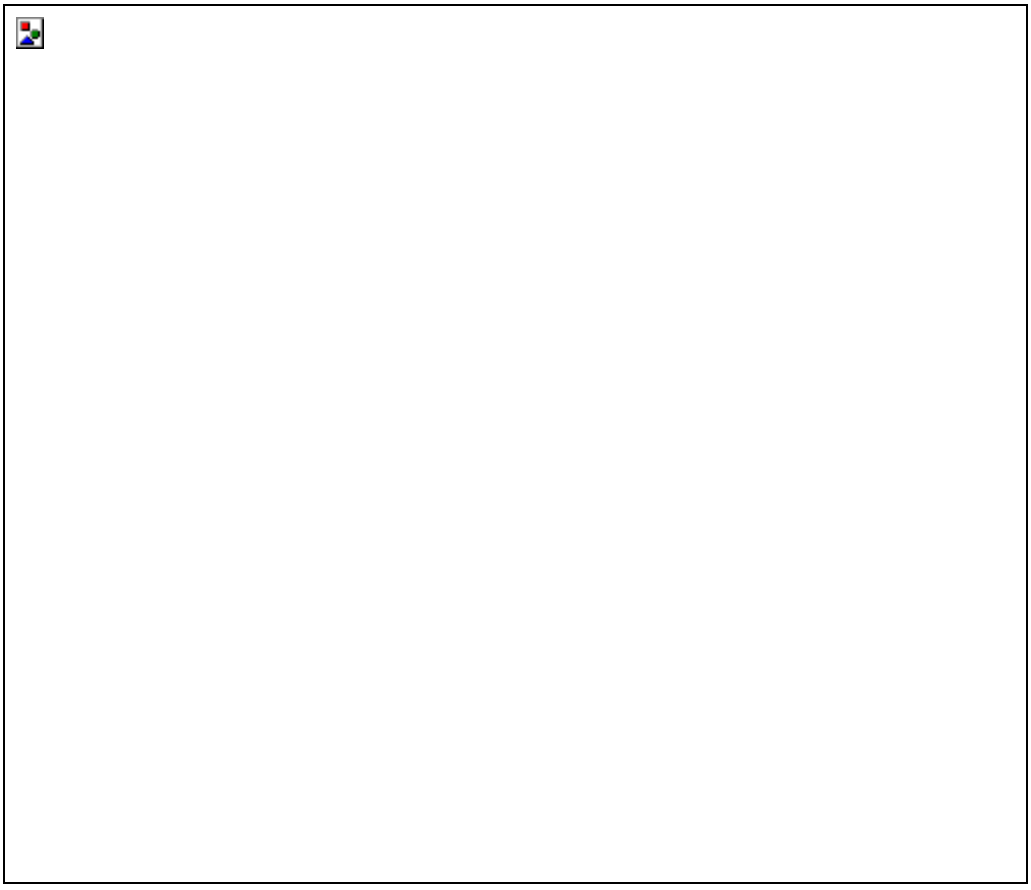

圖 13-2　繼承兩個基礎物件

有兩份 Worker 物件亦會引起其它問題。真正的問題是為何要有兩份 Worker 物件？為解決此一問題，C++ 提供了一個新技巧，虛擬基礎類別（Virtual base class）。

虛擬基礎類別

虛擬基礎類別的功能為：衍生自多個基礎類別的物件，且這些基礎類別有共同的祖先。此例中，對 Singer 和 Waiter 使用關鍵字 virtual 將 Worker 宣告成虛擬基礎類別：

```
class Singer : virtual public Worker{…};
class Welter : virtual public Worker{…};
```

SingingWaiter 可和以前一樣宣告：

```
class SingerWaiter : public Singer, public Waiter{…};
```

如此，衍生的 Singer 和 Waiter 物件共用一個 Worker 物件，而非各自擁有一個，所以 SingingWaiter 物件只含一個 Worker 物件。如圖 13-3 所示。

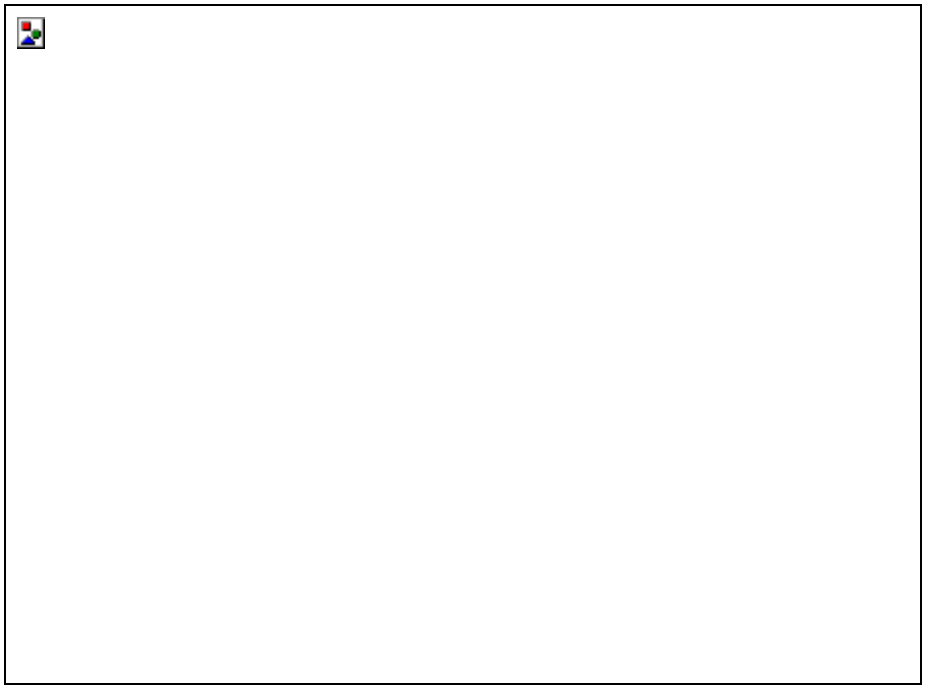

圖 13-3　虛擬基礎類別的繼承

範例程式 13-2 是每一類別的完整宣告。

範例程式 13-2　workermi.h

```
// workermi.h  -- working classes
#include "strng2.h"
class Worker
{
private:
    String fullname;
    long id;
protected:
    virtual void data() const;
    virtual void get();
public:
    Worker() : fullname("no one"), id(0L) {}
    Worker(const String & s, long n)
              : fullname(s), id(n) {}
    virtual void set();
    virtual void show() const;
};

class Waiter : virtual public Worker
{
private:
    int panache;
protected:
    void data() const;
    void get();
public:
    Waiter() : Worker(), panache(0) {}
    Waiter(const String & s, long n, int p = 0)
              : Worker(s, n), panache(p) {}
    Waiter(const Worker & wk, int p = 0)
              : Worker(wk), panache(p) {}
    void set();
    void show() const;
};

class Singer : virtual public Worker
{
protected:
enum {other, alto, contralto, soprano,
                  bass, baritone, tenor};
```

```
    enum {Vtypes = 7};
    void data() const;
    void get();
private:
    static char *pv[Vtypes];   // string equivs of voice types
    int voice;
public:
    Singer() : Worker(), voice(other) {}
    Singer(const String & s, long n, int v = other)
              : Worker(s, n), voice(v) {}
    Singer(const Worker & wk, int v = other)
              : Worker(wk), voice(v) {}
    void set();
    void show() const;
};

class SingingWaiter : public Singer, public Waiter
{
protected:
    void data() const;
    void get();
public:
    SingingWaiter()  {}
    SingingWaiter(const String & s, long n, int p = 0,
                      int v = Singer::other)
              : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) {}
    SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
              : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
    SingingWaiter(const Waiter & wt, int v = other)
              : Worker(wt),Waiter(wt), Singer(wt,v) {}
    SingingWaiter(const Singer & wt, int p = 0)
              : Worker(wt),Waiter(wt,p), Singer(wt) {}
    void set();
    void show() const;
};
```

呼叫那一個成員函數

在 SingingWaiter 有成員函數 show()。因 SingingWaiter 無新的資料成員，故可省略這些成員函數，而用繼承過來的成員函數就可。若 SingingWaiter 類別省略這 show() 成員函數，則 SingingWaiter 物件應如何地呼叫 show()成員函數呢？

```
SingingWaiter newhire("Elise Hawk", 2005, 6, soprano);
newhire.show();        //產生模稜兩可
```

若為單一繼承，則會使用最近祖先的定義，但在多重繼承的情況下，此呼叫會造成混淆。

因此，需要以範疇運算子來標明要使用哪一個成員函數：

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Singer::show();      //use Singer version
```

較好的方法是重新定義 SingingWaiter 的 show()函數，在其中標明要使用那一個 show()。若 SingingWaiter 物件要用 Singer 的版本，則程式碼為：

```
void SingingWaiter::show()
{
    Singer::show();
}
```

若要 SingingWaiter::show()顯示其所有祖先的資料，則程式要改成：

```
void SingingWaiter::show()
{
     Singer::show();
     Waiter::show();
}
```

Waiter::show()可設計成如下：

```
void Waiter::show() const
{
     cout<<" Category: waiter\n";
     Worker::data();
     data();  //Waiter::data()
}
```

注意，最後的 data() 是呼叫 Waiter 的 data()。

同樣的，SingingWaiter 的 data()和 show()可設計成如下：

```
void SingingWaiter::data() const
{
    Singer::data();
    Waiter::data();
};

void SingingWaiter::show() const
{
    cout<<"Category: singingWaiter\n";
    Worker::data();
    data(); // SingingWaiter::data()
}
```

範例程式 13-3 為各種類別的成員函數之實作。

範例程式 13-3　workermi.cpp

```
// workermi.cpp -- working class methods
#include "workermi.h"
#include <iostream>
using namespace std;
// Worker methods
void Worker::set()
{
    cout << "Enter worker's name: ";
    get();

}

void Worker::show() const
{
    cout << "Category: worker\n";
    data();
}

// protected methods
void Worker::data() const
```

```
{
    cout << "Name: " << fullname << "\n";
    cout << "Employee ID: " << id << "\n";
}

void Worker::get()
{
    cin >> fullname;
    cout << "Enter worker's ID: ";
    cin >> id;
    while (cin.get() != '\n')
         continue;
}

// Waiter methods
void Waiter::set()
{
    cout << "Enter waiter's name: ";
    Worker::get();
    get();
}


void Waiter::show() const
{
    cout << "Category: waiter\n";
    Worker::data();
    data();
}

// protected methods
void Waiter::data() const
{
    cout << "Panache rating: " << panache << "\n";
}

void Waiter::get()
{
    cout << "Enter waiter's panache rating: ";
    cin >> panache;
    while (cin.get() != '\n')
         continue;
}

// Singer methods
```

```
char * Singer::pv[Singer::Vtypes] = {"other", "alto", "contralto",
             "soprano", "bass", "baritone", "tenor"};

void Singer::set()
{
    cout << "Enter singer's name: ";
    Worker::get();
    get();
}

void Singer::show() const
{
    cout << "Category: singer\n";
    Worker::data();
    data();
}

// protected methods
void Singer::data() const
{
    cout << "Vocal range: " << pv[voice] << "\n";
}

void Singer::get()
{
   int i;
    cout << "Enter number for singer's vocal range:\n";
    for (i = 0; i < Vtypes; i++)
    {
         cout << i << ": " << pv[i] << "   ";
         if ( i % 4 == 3)
             cout << '\n';
    }
    if (i % 4 != 0)
         cout << '\n';
    cin >> voice;
    while (cin.get() != '\n')
         continue;
}

// SingingWaiter methods
void SingingWaiter::data() const
{
    Singer::data();
```

```
    Waiter::data();
}

void SingingWaiter::get()
{
    Waiter::get();
    Singer::get();
}

void SingingWaiter::set()
{
    cout << "Enter singing waiter's name: ";
    Worker::get();
    get();
}

void SingingWaiter::show() const
{
    cout << "Category: singing waiter\n";
    Worker::data();
    data();
}
```

範例程式 13-3 為測試程式。我們必須將此程式和 workermi.cpp 和 strng2.cpp 一起編譯。

範例程式 13-4　workmi.cpp

```
// workmi.cpp -- multiple inheritance
// compile with workermi.cpp, strng2.cpp
#include <iostream>
using namespace std;
#include <cstring>
#include "workermi.h"
const int SIZE = 5;
int main(void)
{

    Worker * lolas[SIZE];
    int ct;
    for (ct = 0; ct < SIZE; ct++)
    {
         char choice;
```

```
        cout << "Enter the employee category:\n"
             << "e: worker  w: waiter  s: singer  "
             << "t: singing waiter  q: quit\n";
        cin >> choice;
        while (strchr("ewstq", choice) == NULL)
        {
             cout << "Please enter an e, w, s, t, or q: ";
             cin >> choice;
        }
        if (choice == 'q')
             break;
        switch(choice)
        {
             case 'e':lolas[ct] = new Worker;
                    break;
             case 'w':lolas[ct] = new Waiter;
                    break;
             case 's':lolas[ct] = new Singer;
                    break;
             case 't':lolas[ct] = new SingingWaiter;
                    break;
        }
        cin.get();
        lolas[ct]->set();
    }

    cout << "\nHere is your staff:\n";
    int i;
    for (i = 0; i < ct; i++)
    {
         cout << '\n';
         lolas[i]->show();
    }
    for (i = 0; i < ct; i++)
         delete lolas[i];
    system("PAUSE");
    return 0;
}
```

程式的輸出結果如下：

```
Enter the employee category:
e: worker  w: waiter  s: singer  t: singing waiter  q: quit
e
Enter worker's name: Wilmot Snipside
Enter worker's ID: 2004
```

```
Enter the employee category:
e: worker  w: waiter  s: singer  t: singing waiter  q: quit
t
Enter singing waiter's name: Natasha Gargalova
Enter worker's ID: 2005
Enter waiter's panache rating: 6
Enter number for singer's vocal range:
0: other   1: alto   2: contralto   3: soprano
4: bass   5: baritone   6: tenor
3
Enter the employee category:
e: worker  w: waiter  s: singer  t: singing waiter  q: quit
q

Here is your staff:

Category: worker
Name: Wilmot Snipside
Employee ID: 2004

Category: singing waiter
Name: Natasha Gargalova
Employee ID: 2005
Vocal range: soprano
Panache rating: 6
```

類別樣版

在 Stack 類別中可存 unsigned long 值，亦可容易地定義存 double 值或 String 物件的堆疊類別，除了儲存的物件型態外，程式碼均相同。因此，若定義一個一般的（即型態獨立）堆疊，將型態以參數傳入類別中，就毋須撰寫新的類別宣告。在第 9 章，曾用 typedef 達到這種需求，但有一些缺點:首先，每次改變型態都要編輯標頭檔。其次，這技術對每支程式只能產生一種堆疊，即 typedef 不能同時表示兩種型態，所以不能在同一程式中定義 int 堆疊及 String 堆疊。

C++ 的類別樣版（class template）提供較好的方法，即產生一般性的類別宣告。

定義類別樣版

我們以第 9 章的 Stack 類別為例來建立樣版。

原來的類別宣告如下：

```
#includ "booly.h"           // define Bool, False, True
typedef unsigned long Item;
class Stack
{
private:
    enum {MAX = 10};        // constant specific to class
    Item items[MAX];        // holds stack items
    int top;                // index for top stack item
public:
    Stack();
    Bool  isempty()  const;
    Bool  isfull()  const;
    // push() returns False if stack already is full,  True otherwise
    Bool push(const Item & item);  //add item to stack
    // pop() returns False if stack already is empty, True otherwise
    Bool pop(Item & item);  // pop top into item
};
```

樣版的方式會用樣版定義和樣版成員函數取代 Stack 定義及成員函數。在樣版類別前以下列程式碼為前言：

```
template <class Type>
```

關鍵字 template 會通知編譯程式即將定義樣版，角括號內有如函數的引數列。Class 是一關鍵字，而 Type 表示變數的型態名稱。（此處的 class 不是指 Type 為一類別，意思是 Type 為一般資料型態的名稱，當使用樣版時，會有真正的型態取代之）。在引用樣版時，Type 會置換成特殊的型態值，如 int 或 String。此例中，將用 Type 取代所有的以 typedef 所定義的 Item。例如：

```
Item items[MAX];         //holds stack items
```

變成

```
Type items[MAX];         //holds stack items
```

同樣的，要用樣版的成員函數取代原類別的成員函數。每個函數標題都用相同的樣版宣告：

```
template <class Type>
```

之後，還需修改類別修飾子，將 Stack::改成 Stack<Type>::。例如，

```
Bool Stack::push(const Item & item)
{
…
}
```

變成：

```
Template <class Type>
Bool Stack<Type>::push(const Type & item)
{
…
}
```

若在類別宣告中有定義成員函數，則可省略樣版前言和類別修飾子。範列程式 13-5 結合類別和成員函數樣版。

範例程式 13-5　stacktp.h

```
// stacktp.h
#include "booly.h"

template <class Type>
class Stack
{
private:
```

```
    enum {MAX = 10};   // constant specific to class
    Type items[MAX];   // holds stack items
    int top;        // index for top stack item
public:
    Stack();
    Bool isempty();
    Bool isfull();
    Bool push(const Type & item);// add item to stack
    Bool pop(Type & item);          // pop top into item
};

template <class Type>
Stack<Type>::Stack()
{
    top = 0;
}

template <class Type>
Bool Stack<Type>::isempty()
{
    return top == 0? True: False;
}

template <class Type>
Bool Stack<Type>::isfull()
{
    return top == MAX? True :False;
}

template <class Type>
Bool Stack<Type>::push(const Type & item)
{
    if (top < MAX)
    {
         items[top++] = item;
         return True;
    }
    else
         return False;
}

template <class Type>
Bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
```

```
    {
        item = items[--top];
        return True;
    }
    else
        return False;
}
```

使用樣版類別

下例將產生兩個堆疊，一存 int，一存 String 物件：

```
Stack<int> kernels;        //create a stack of  ints
Stack<String> colonels;    //create a stack of  String of objects
```

從這兩宣告，編譯程式會按照 Stack<Type>樣版，而產生兩個不同的類別宣告及兩組不同的類別成員函數。Stack<int>類別宣告會將所有的 Type 都換成 int，而 Stack<String>宣告則將 Type 換成 String。

範例程式 13-6 乃是修改第九章的堆疊測試程式（範例程式 9-13），用字串記錄購買訂單的 IDs。

因為使用到 String 類別，所以要和 strng2.cpp 一起編譯。

範例程式 13-6　stacktem.cpp

```
// stacktem.cpp -- test template stack class
// compiler with strng2.cpp
#include <iostream>
using namespace std;
#include <ctype.h>
#include "stacktp.h"
#include "strng2.h"
int main(void)
{
    Stack<String> st;  // create an empty stack
    char c;
    String po;
```

```
    cout << "Please enter A to add a purchase order,\n"
         << "P to process a PO, or Q to quit.\n";
    while (cin >> c && toupper(c) != 'Q')
    {
         while (cin.get() != '\n')
             continue;
         if (!isalpha(c))
         {
             cout << '\a';
             continue;
         }
         switch(c)
         {
             case 'A':
             case 'a':cout << "Enter a PO number to add: ";
                      cin >> po;
                      if (st.isfull())
                         cout << "stack already full\n";
                      else
                         st.push(po);
                      break;
             case 'P':
             case 'p':if (st.isempty())
                         cout << "stack already empty\n";
                      else {
                         st.pop(po);
                         cout << "PO #" << po << " popped\n";
                         break;
                      }
         }
         cout << "Please enter A to add a purchase order,\n"
              << "P to process a PO, or Q to quit.\n";
    }
    cout << "Bye\n";
    system("PAUSE");
    return 0;
}
```

程式的執行結果為：

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: red911porsche
```

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: green325bmw
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: silver747boing
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #silver747boing popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #green325bmw popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #red911porsche popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

正確的使用堆疊存指標

要使用堆疊存指標的方法之一是呼叫程式提供指標陣列，每個指標指向不同的字串，將這些指標放入堆疊中才有意義。

例如，要模擬下述的情形：有個人送了一車的公文給 Plodson。若 Plodson 的籃內是空的，它就從車上取出最上面的公文置於籃內。若籃內已滿則移走籃內的最上面公文，處理此檔案，然後置於籃外。若籃內即不空也不滿，Plodson 可能會處理籃內的頂端檔案，或從車上取下檔案置於籃內；他會擲硬幣決定該如何做。問題是他的方法對原始的檔案順序有何影響。

我們可用字串的指標陣列來表示這一車的檔案；每個字串表示檔案所描述的人名。以堆疊表示籃子，並用第二指標陣列表示籃外的檔案。把檔案放到籃中就是將輸入陣列的指標推入堆疊中，處理檔案就是從堆疊中取出元素，然後加入籃外的陣列中。

範例程式 13.7 重新定義 Stack<Type>類別，使 Stack 的建構子接受堆疊大小的參數，因此內部就需改成動態陣列，故亦須解構子。

範例程式 13.7　stcktp1.h

```
// stcktp1.h
#include "booly.h"

template <class Type>
class Stack
{
private:
    enum {MAX = 10};   // constant specific to class
    int stacksize;
    Type * items;   // holds stack items
    int top;      // index for top stack item
public:
    Stack(int ss = MAX);
    ~Stack() { delete [] items; }
    Bool isempty() { return top == 0? True: False; }
    Bool isfull() { return top == stacksize? True :False; }
    Bool push(const Type & item);// add item to stack
    Bool pop(Type & item);       // pop top into item
};

template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
    items = new Type [stacksize];
}

template <class Type>
Bool Stack<Type>::push(const Type & item)
{
    if (top < stacksize)
    {
```

```
        items[top++] = item;
        return True;
    }
    else
        return False;
}

template <class Type>
Bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return True;
    }
    else
        return False;
}
```

範例程式 13.8 的程式用新的堆疊樣版完成模擬 Plodson。它用 rand()、srand()和 time()來產生隨機數字，這裡產生 0 或 1 模擬丟硬幣的結果。

範例程式 13.8　stkoptr1.cpp

```
// stkoptr1.cpp -- test stack of pointers
#include <iostream>
using namespace std;
#include <stdlib.h>    // for rand(), srand()
#include <time.h>   // for time()
#include "stcktp1.h"
const int Stacksize = 4;
const int Num = 10;
int main(void)
{
    srand(time(0)); // randomize rand()
    cout << "Please enter stack size: ";
    int stacksize;
    cin >> stacksize;
    Stack<char *> st(stacksize); // create an empty stack with 4 slots


    char *in[Num] = {
             " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
```

```
            " 3: Betty Rocker", " 4: Ian Flagranti",
            " 5: Wolfgang Kibble", " 6: Portia Koop",
            " 7: Joy Almondo", " 8: Xaverie Paprika",
            " 9: Juan Moore", "10: Misha Mache"
            };
    char *out[Num];

    int processed = 0;
    int nextin = 0;
    while (processed < Num)
    {
         if (st.isempty())
             st.push(in[nextin++]);
         else if (st.isfull())
             st.pop(out[processed++]);
         else if (rand() % 2  && nextin < Num) // 50-50 chance
             st.push(in[nextin++]);
         else
             st.pop(out[processed++]);
    }
    for (int i = 0; i < Num; i++)
         cout << out[i] << "\n";

    cout << "Bye\n";
    system("PAUSE");
    return 0;
}
```

下面為程式執行兩次的結果，檔案最後的順序都不同。

```
Please enter stack size: 5
 1: Hank Gilgamesh
 4: Ian Flagranti
 5: Wolfgang Kibble
 6: Portia Koop
 7: Joy Almondo
 3: Betty Rocker
 9: Juan Moore
 8: Xaverie Paprika
 2: Kiki Ishtar
10: Misha Mache
Bye

Please enter stack size: 5
```

```
 2: Kiki Ishtar
 1: Hank Gilgamesh
 7: Joy Almondo
 8: Xaverie Paprika
 6: Portia Koop
 5: Wolfgang Kibble
 4: Ian Flagranti
 3: Betty Rocker
 9: Juan Moore
10: Misha Mache
Bye
```

程式摘要

字串本身不會移動。將字串放到堆疊中會產生一個新指標指向存在的字串，即其值為字串的位址。從堆疊中取出字串，將位址值複複至 out 陣列。

因類別建構子用 new 產生指標陣列，類別的解構子刪除此陣列，而非陣列元素所指的字串。

樣板特定化

以某一型態實體化樣板時，可修改樣板使其行為不同。例如，樣板定義如下:

```
Template <class T>
class SortedArray
{
    …//details omitted
};
```

若元素加入陣列時，樣板會為其排序，若樣板使用 > 運算子比較元素值，數字就不會有問題，但若為類別物件就需定義此類別的運算子。假設 T 是以 char *來表示字串，此時若要作字串陣列的排序，就需要修改樣板的程式碼，以 strcmp( )取代 >。因為以>運算子來處理字串排序，只會比較字串在記憶體的位址大小而已。此時程式設計師需提供樣板特定化(template specializations)，以定義某特殊型態而非一般型態的樣板。若特定化的樣板和一般化的樣板皆符合實體化的要求時，編譯程式會先採用特殊化的版本。

特殊化的類別樣板定義格式如下:

```
template <> class classname <speclallzed-type-name>{…};
```

例如，對 SortedArray 樣板的特殊型態 char *，其程式碼如下:

```
template <> class SortedArray <char *>
{
   …//details omitted
};
```

省略的程式碼中利用 strcmp( )來代替>比較字串值。若以 char*將 SortedArray 實體化，會用此特殊化的定義:

```
SortedArray<int> scores;        //使用一般定義的類別樣版
SortedArray<char *> dates;      //使用樣版特殊化的版本
```

問題回顧

1. 下列每組類別中，在第二欄位的類別較適合以公用或私有來繼承：

class Bear	class PolarBear
class Kitchen	class Home
class Person	class Programmer
class Person	class HorseAndJocky
class Person, class Automobile	class Driver

2. 假設有下述定義：

```
class Frabjous {
private:
   char fab[20];
public:
   Frabjous(const char *s = "C++") : fab(s){}
   virtual void tell(){cout << fab;}
};
```

```
class Gloam {
private:
    int glip;
    Frabjous fb;
public:
    Gloam(int g = 0, const char *s = "C++");
    Gloam(int g, const Frabjous & f);
    void tell();
};
```

已知 Gloam 中的 tell()會顯示 glip 和 fb 值，試寫出 Gloam 三個成員函數的定義。

3. 假設有下述定義：

```
class Frabjous {
private:
    char fab[20];
public:
    Frabjous(const char *s = "C++") : fab(s){}
    virtual void tell(){cout << fab;}
};

class Gloam : private Frabjous {
private:
    int glip;
public:
    Gloam(int g = 0, const char *s = "C++");
    Gloam(int g, const Frabjous & f);
    void tell();
};
請撰寫 Gloam 的成員函數，以 tell()來顯示 glip 和 fab 的值
```

4. 假設下述定義是以範例程式 13.5 的 Stack 和範例程式 13.3 的 Worker 類別為基礎：

```
Stack<worker *>sw;
```

寫出即將產生的類別宣告內容，不用包含非內嵌的類別成員函數。

5. 用本章的樣版來定義以下面敘述：

   String 物件的陣列
   double 陣列的堆疊
   Worker 物件指標之堆疊的陣列

6. 描述虛擬和非虛擬基礎類別的不同？

程式設計練習

1. Person 類別儲存人的姓和名，除了建構函數外，還有成員函數 show()可以顯示姓名。Gunslinger 以虛擬方式衍生自 Person 類別，有 Draw()成員回傳 double 值表示一個槍手的拔槍次數，還有一個 int 成員表示此槍手命中的次數，以及 Show()函數顯示所有的資訊。

   PokePlayer 類別以虛擬方式衍生自 Person 類別，有一 Draw()成員，回傳一個介於 1 到 52 之間的隨機值，表一張牌的數字（你可以隨意地定義 Card 類別具有花色和面值的成員，並使用 Card 作為 Draw()的回傳值）。PokePlayer 類別沿用 Person 的 show()函數。BadDude 以公用的方式衍生自 Gunslinger 和 PokerPlayer 類別，有 Gdraw()成員回傳一個人拔槍的次數，而成員 Cdraw()回傳下一張抽出的牌，且有適當的 Show()函數。定義所有的類別和成員函數，包括其它必須的成員函數（例如設定物件值），並以類似範例程式 13.3 的程式測試之。

2. 下列為一些類別宣告：

```
// emp.h-- header file for employee class and children
#include<ctring>
#include<iostream>
using namespace std;

const int SLEN = 20;
class employee
{
protected:
    char fname[SLEN];
```

```
    char lname[SLEN];
    char job[SLEN];
public:
    employee();
    emlployee(char * fn, char * ln, char * j);
    employee(const employee & e);
    virtual void ShowAll() const;
    virtual void SetAll();  // prompts user for values
    friend ostream & operator << (ostream & os, const employee & e);
};

class manager: virtual public employee
{
protected:
    int inchargeof;
pulic:
    manager();
    manager(char * fn, char * ln, char * j, int ico = 0);
    manager (const employee & e, int ico);
    manager (const manager & m);
    void ShowAll() const;
    void SetAll();
};

class fink: virtual public employee
{
protected:
    char reportsto[SLEN];
public:
    fink();
    fink(char * fn, char * ln, char * j, char * rpo);
    fink(const employee & e, char * rpo);
    fink(const fink & e);
    void ShowAll() const;
    void SetAll();
};

class highfink: public manager, public fink
{
public:
    highfink();
    highfink(char * fn, char * ln, char * j, char * rpo, int ico);
    highfink(const employee & e, char * rpo, int ico);
    highfink(const fink & f, int ico);
    highfink(const manager & m, char * rpo);
    highfink(const highfink & h);
```

```
    void ShowAll()const;
    void SetAll();
};
```

注意此類別階層架構使用具有虛擬基礎類別的多重繼承，所以要記住類別建構函數初始化串列的特殊規則。還要注意資料成員是宣告為 protected 而不是 private。這可以簡化 highfink 成員函數的程式碼（例如，若 highfink::ShowAll()只要呼叫 fink::ShowAll()和 manager::ShowAll()，它不會呼叫 employee::ShowAll()兩次）。可如 worker 類別一般在多重繼承中使用私有的資料並提供其它保護的成員函數。請提供 類別成員函數的實作，並在程式中測試類別。這裡是極短的測試程式，你至少要加入一個 SetAll()成員函數。

```
// useemp1.cpp use employee classes
#include<iostream>
using namespace std;
#include "emp.h"

int main ()
{
    employee th ("Trip", "Harris", "Thumper");
    cout << th << '\n';
    th.ShowAll();

    manager db("Debbie","Bolt", "Twigger", 5);
    cout << db << '\n';
    db.ShowAll();

    cout << "Press a key for next batch of output:\n"
    cin.get();

    fink mo("Matt", "Oggs", "Oiler", "Debbie Bolt");
    cout << mo << '\n';
    mo.ShowAll();
    highfink hf(db, "Curly Kew");
    hf.ShowAll();

    cout<< "Using an employee * pointer:\n";
    employee * tri[4] = { &th, &db, &mo, &hf };
    for (int i = 0; I < 4; i++)
    tri[i]->ShowAll();
```

```
    system("PAUSE");
    return 0;
}
```

3. 為何沒有定任何設定運算子？

4. 為何 ShowAll()和 SetAll()為虛擬函數？

5. 為何 employee 是虛擬基礎類別？

6. 為何 highfink 類別無資料區段？

7. 為何只需要一種版本的 operator<<()？

8. 若程式結束處換成下列程式碼，會有什麼差異？

```
employee tri[4] = {th, db, mo, hf};
for (int i = 0; i < 4; i++)
    tri[i].ShowAll();
```