13. More on Inheritance in C++

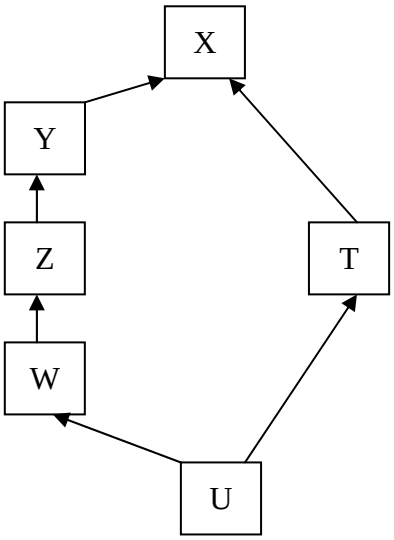
13.1 Virtual Bases and Diamond Hierarchies

13.1.1 Virtual Bases and Constructor

Diamond hierarchies 為 C++多重繼承之一大問題。

程式範例: oop_ex116.cpp

對於此，C++提供了 virtual base 之方式來解決，例如以下之繼承架構，Y 與 T 繼承 X 時，最好將 X 宣告為 virtual base，避免重複繼承造成 ambiguous，其作法為於宣告繼承時使用 virtual 關鍵字：




```
class Y : virtual public X
{
      ……
}
```

如 Y 與 T 均將 X 宣告為 virtual base，則同時繼承了 Y 與 T 的類別其中只有一份 X 物件，由 Y 與 T 共享，例如 U 之物件中具有 Y 與 T 物件，Y 與 T 共有一份 X 物件。

但如將 X 宣告為 virtual base，則 X 下的所有衍生類別，無論是 direct 或 indirect 繼承它，其建構函式均需要明確地呼叫 X 的建構函式。

如非 virtual base class，衍生類別是不能呼叫其 indirect base class 的建構函式，只能透過呼叫 immediate base class 的建構函式來間接引用(invoke)以初始化 indirect base class 的資料成員，故如 X 非 virtual base，則 Z、W、與 U 的建構函式不能呼叫 X 的建構函式，但如 X 為 virtual base，則 Z、W、與 U 的建構函式均必須明確地呼叫 X 的建構函式。

程式範例: oop_ex117.cpp，oop_ex118.cpp

注意要點:

1. 架構中每個類別的建構函式均必須明確地呼叫 X 的建構函式。
2. 當 Y 類別宣告其基本類別 X 為 virtual，則繼承 Y 之衍生類別 Z 的建構函式對 Y 建構函式之引用，將忽略其中對 X 建構函式之引用。
3. U 只包含了一個 X 物件，由 Y 與 T 共享，故 X::print( )不會造成 ambiguous call。
4. TA 只包含了一個 Person 物件，故原 oop_ex116.cpp 中 ambiguous calls 均不會發生。

13.1.2 Virtual Bases and Copy Constructor

由於 virtual base 的產生將使的其下各層類別的建構函式均需要明確地呼叫 virtual base 的建構函式，故此形成的規則，是 virtual base 下各層類別的拷貝建構函式均需要明確地呼叫 virtual base 的拷貝建構函式。

程式範例: oop_ex119.cpp

13.1.3 Virtual Bases and Copy Assignment Operators

我們可看到將基本類別宣告為 virtual base 將影響到其下 direct 或 indirect 衍生類別的建構函式的定義方式，包括拷貝建構函。那拷貝指定運算子應為何?

程式範例: oop_ex120.cpp

注意要點:

1. 拷貝指定運算子非建構函式。
2. 拷貝指定運算子不受基本類別是否為 virtual base 所影響。
3. 執行 U 之拷貝指定運算子，U 物件中之 x 資料成員將分別被 W 與 T 之拷貝指定運算子設定其值兩次。
4. 如 U 之拷貝指定運算子中沒有呼叫 W 之拷貝指定運算子，觀察執行之結果。
5. Y 與 T 之拷貝指定運算子寫法不同，但功能完全相同。

13.1.4 Summary for Diamond Hierarchies

先前的 Diamond Hierarchies 繼承架構雖於設計上較直觀，但卻有需將基本類別宣告為 virtual base 的必要，此使得程式難以延伸發展(extend)。

Main Disadvantages of OO Design That Involves Diamond Hierarchies：

1. Difficult to extend existing code.
2. Excessively rigid categorization.
3. Type conversion inefficiencies.

13.2 Mixin Classes

Mixin class 為只有純虛擬函式(pure virtual function)所構成的類別，其有如 Java 的 Interface，其作用是規定/賦與繼承其之衍生類別需有的能力(之函式原型)，繼承其之衍生類別必須對其所定義之純虛擬函式提出實作，才能實體化。

程式範例: oop_ex121.cpp

Project 2 問題二的 Mixin class 參考: oop_ex122.cpp

使用 Mixin class 之設計雖比先前的 Diamond Hierarchies 繼承架構要有彈性，但仍有以下缺點。

Main Disadvantages of OO Design Based on Mixin Classes：

1. Excessively rigid categorization.

2. Type conversion inefficiencies.

13.3 Type Conversion Inefficiencies

假設某大學部學生完成了其大學學業，並準備進入研究所，成為研究生，究程式的觀點，我們想要達成的是，將原來為 Student 之物件，轉換成為 GraduateStudent 之物件，並為其新加入屬於 GraduateStudent 之資料。

首先就將 Student 之物件轉換成為 GraduateStudent 之物件而言，雖 GraduateStudent 為 Student 之衍生類別，但就物件的轉換上，並沒有記憶體操作上有效的方法，以下為可能的方式之一。

```
GraduateStudent*
CreateGraduateStudentFromStruent( Student* doneStudent,
                                  Department newDepartment,
                                  Teacher& theAdvisor )
{
    GraduateStudent * gp =
        new GraduateStudent(   doneStudent->getName( ),
                               doneStudent->getIdentification( ),
                               doneStudent->getBirthDate( ),
                               doneStudent->getAddress( ),
                               doneStudent->getStudentStatus( ),
                               newDepartment,
                               theAdvisor );
    delete doneStudent;

    return gp;
}
```

以下為另一個方法，但與上面的方法一樣沒效率。

```
GraduateStudent*
CreateGraduateStudentFromStruent( Student* doneStudent,
                                  Department newDepartment,
                                  Teacher& theAdvisor )
```

```
{
    //first create an empty GraduateStudent Object
    GraduateStudent * gp =
        new GraduateStudent(   0,                  // name
                               0,                  // ssn
                               0,                  // birthdate
                               0,                  // address
                               eUnknown,           // studentStatus
                               newDepartment,
                               theAdvisor );

    Student* tsp = gp;

    *tsp = *doneStudent;

    delete doneStudent;

    return gp;
}
```

13.4 OO Design Using Role Playing Classes

使用 Role Playing Classes，即是以”has a”取代”is a”關係來設計，某個 Person 可以有一個以上的角色物件，角色物件則定義了此類角色應(可)有的能力，擁有某角色物件即具有扮演此類角色應有的能力，例如某人為 TA，則其為同時擁有 Student 與 Teacher 此兩個 Role Playing Classes 的 Person 物件，具有 Student 物件則有 enrollForClass 等能力，具有 Teacher 物件則有 teachForClass 等能力。

假設某大學部學生完成了其大學學業，並準備進入研究所，成為研究生，我們只要將此 Person 物件其角色(UniversityMember*成員所指物件)由原為 Student 之物件，改成 GraduateStudent 之物件即可，不需改變原為 Person 型態物件之型別，亦不需拷貝 Person 物件其他非角色之資料成員。

使用 Role Playing Classes 之設計，所有 Role Playing Classes 的物件在擁有者物件中均是以基本類別的指標來參照，例如 Student、Teacher 等物件，於 Person 中均是以 UniversityMember 的指標來定位。問題在於，對於一個物件，我們如何

得知其是否具有某種角色，而有此角色之能力。例如對於某 Person 物件，我們須能知道其是否具有 Student 之角色，有的話才能修課。

13.4.1 dynamic_cast Operation of RTTI

C++之"Run Time Type Identification (RTTI)"可用於此一判斷，假設我們想知道某個 UniversityMember 指標所指的物件是否為 Student 之物件，我們可使用 C++之 RTTI 所提供之 dynamic_cast 運算如下。

```
Student* st = dynamic_cast<Student*>(um);
```

其中 um 為 UniversityMember*之型態，如 um 所指的物件非 Student，則 dynamic_cast 傳回 0。

例：

```
void enrollStudentInCourse(const Courses& newCourse, Person* aStudent)
{
    UniversityMember* role = aStudent.getActiveRole( );

    Student* studentPtr = dynamic_cast<Student*>(role);

    if(studentPtr != 0)
        studentPtr->enrollForCourse( newCourse);
    else
        cout<<"The person is not a student. Cannot enroll in a course."<<endl;
}
```

程式範例: oop_ex123.cpp

注意要點:

1. 欲使用 RTTI，有些編譯器需先啟動 RTTI 功能，例如 Visual C++，其設定為：Project->Settings->C/C++->Category 選擇 C++ Language->勾選 Enable RTTI。
2. Dynamic casts 只可作用於 polymorphic types(具有至少一個 virtual function 的類別)。此並非真正的問題，因為如沒有任何虛擬函式，至少可將其解構函

式宣告為 virtual。

3. Dynamic casts 只能用於某繼承架構中之類別間的轉換。
4. 除指標型態外，Dynamic casts 也可作用於物件之參考(object references)，但此轉換只能於 try-catch 區塊中使用，例如：

```
void f(UniversityMember&    roleRef)
{
    try{
        Student& studentRef = dynamic_cast<Student&>(roleRef);
        // if flow of control gets here, that means yes, the true identity of
        // the object refered by roleRef is indeed a Student.
        // So it can be enrolled for a course.
    }
    catch(bad_cast& b){
        // means the object refered by roleRef is not a Student.
        // so do something here.
    }
```

5. C++之 RTTI 機制同時提供了 static_cast 運算子以供型別之轉換，其為強制的靜態轉換，此常會有問題，例如我們自然可用其將 Derived*轉成 Base*，但其亦可做強制的 downcasting 將 Base*轉成 Derived*，不像 dynamic_cast 會根據物件的實際型態判斷此轉換是否可行。
6. C++之 RTTI 機制的另一個 casting operator 為 reinterpret_cast，其主要作用是將某 int 轉換成記憶體位址。

13.4.2 typeid of RTTI

RTTI 之 typeid 函式，可得到某物件所屬類別之 type_info，以用於判斷某物件實際為何種類別，或兩物件是否實際為同類別等比較。

程式範例: oop_ex124.cpp

注意要點:

1. 欲使用 typeid，需引用<typeinfo.h>。
2. 其輸入參數可以是物件(例如*(ab))，或者類別名稱(例如 printable)，傳回值是一個指向 type_info 物件的參考，type_info 物件是一個系統支援的物件，

代表一種類別。

3. 由 type_info 物件之 name 函式成員所傳回的字串是由系統所保有，無法由 delete 所釋放。
4. 其對物件所屬類別之判斷不具多型之行為，例如 printandscaleable 之物件非 printable 之物件，typeid 對輸入物件之傳回值只有一個，即其實際所屬類別之 type_info 物件。
5. typeid 對輸入類別名稱之傳回值即此類別之 type_info 物件

Project 2 問題三的參考範例: oop_ex125.cpp