

# 你所不知道的 C 語言：物件導向程式設計篇

「慣 C」當然可以進行物件導向

Copyright (慣C) 2016, 2018 宅色夫

直播錄影 (上)

直播錄影 (下)





## 先來看一個範例

- [bash-oo-framework](#): 引入進階的語言特徵到 GNU bash
  - 物件導向著重於「思維」，語法只是「輔助」

```
import util/namedParameters util/class

class:Human() {
    public string name
    public integer height
    public array eaten

    Human.__getter__() {
        return name;
    }
}
```

```

        echo "I'm a human called $(this.name), $(this.height) cm tall."
    }

Human.Example() {
    [array]      someArray
    [integer]    someNumber
    [...rest]   arrayOfOtherParams

    echo "Testing $(var: someArray toString) and $someNumber"
    echo "Stuff: ${arrayOfOtherParams[*]}"

    # returning the first passed in array
    @return someArray
}

Human.Eat() {
    [string] food

    this.eaten.push "$food"

    # will return a string with the value:
    @return:value "$this just ate $food, which is the same as $1"
}

Human.WhatDidHeEat() {
    this.eaten.toString
}

# this is a static method, hence the :: in definition
Human::PlaySomeJazz() {
    echo "$(UI.Powerline.Saxophone)"
}
}

# required to initialize the class

```

```
Type::Initialize Human

class:SingletonExample() {
    private integer YoMamaNumber = 150

    SingletonExample.PrintYoMama() {
        echo "Number is: $(this YoMamaNumber)!"
    }
}

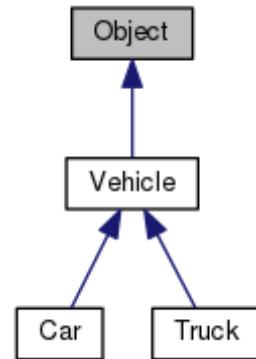
# required to initialize the static class
Type::InitializeStatic SingletonExample
```

- Doxygen 可掃描給定原始程式碼內容、註解，之後將自動產生程式文件，其中一種輸出就是 HTML，可透過瀏覽器來閱讀文件和程式碼
  - 延伸閱讀: [Learning Doxygen](#)
- 修改自 Doxygen 的範例: [doxygen-oop-in-c](#)

```
$ git clone https://github.com/jserv/doxygen-oop-in-c.git
$ cd doxygen-oop-in-c
$ make
$ make doc
```

用網頁瀏覽器打開 `doc/html/index.html`

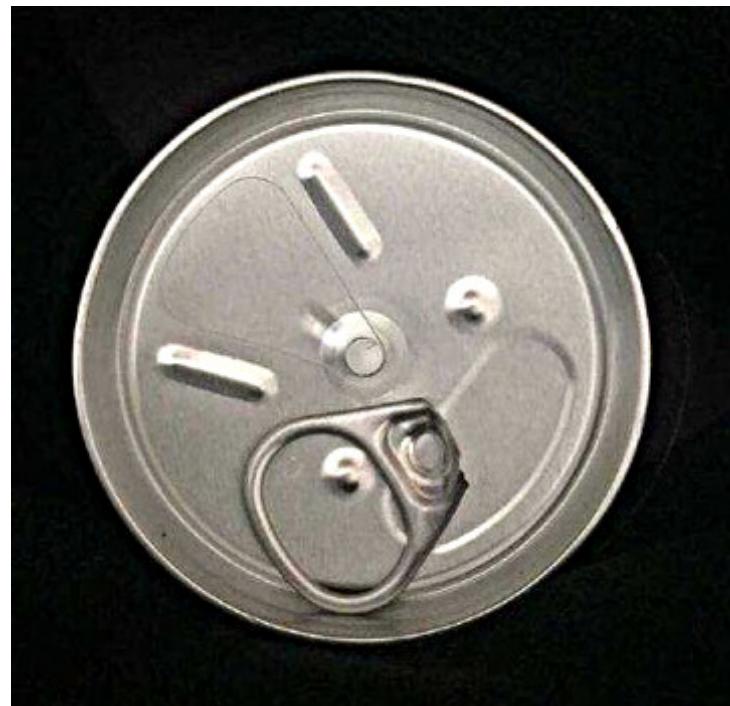
點擊 “Classes” -> “Object”



檔案 [manual.c](#) 定義了物件的繼承

這跟一般的 C 語言程式有什麼差別？

(你乾脆問「你媽媽跟林志玲絕大多數的生理機能都一樣，有什麼差別？」)



(圖片來源)

「資料 + 動作」包裝成「物件」

## 物件導向是一種態度

只要有心，Brainf\*ck 語言也能作 Object-Oriented Programming (OOP) !

摘自「無拘的物件導向」：

如同數學上的複數，是由實數與虛數組合而成，而就物件而言，單純就是整體地看待一組資料，也就是個資料複合體。

因此，像 `[5, 10]` 就算是個物件了，也許這代表了一個複數，或者是一個平面直角座標，端看開發者怎麼去接受、運算這類資料。當然，使用可以令這個物件的意圖更明確，或者進一步地，使用個鍵值結構來儲存，例如 `{'x' : 5, 'y' : 10}` 會更加方便。這也是許多語言會用來呈現物件的基本語法形式，然而，不代表一定必須是這種形式，才是物件。

像 `[5, 10]`、或 `{'x' : 5, 'y' : 10}`，就是所謂**被封裝的資料**。基本上，資料的封裝並非不可見，而是整體地看待這組資料，私有性（private）這類強封裝則是為了工程、團隊合作才出現的要求，一些比較彈性的語言，如Python，就不強調私有性。那麼，程式上的封裝呢？只要能夠處理這組資料的任何函式（Function）都可以！

真正封裝程式的是函式，如果有一組函式，可以接受物件（資料複合體）進行運算，就可以說這組函式是物件的方法（Method），也就是說，函式與物件是可以分別看待的，只不過概念上會稱這類函式為「物件可用的方法」。

## Data Encapsulation

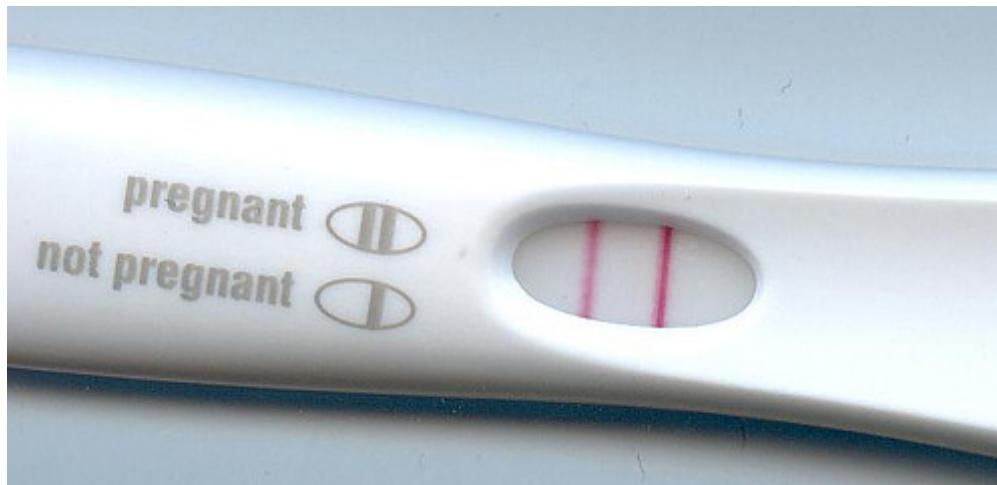


善用 forward declaration 與 function pointer，用 C 語言實現 Encapsulation，示範程式碼如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* forward declaration */
5 typedef struct object Object;
6 typedef int (*func_t) (Object *);
7
8 struct object {
9     int a, b;
10    func_t add, sub;
11 };
12
13 static int add_impl(Object *self) { // method
14     return self->a + self->b;
15 }
16 static int sub_impl(Object *self) { // method
17     return self->a - self->b;
18 }
19
```

```
20 // & : address or
21 // * : value of // indirect access
22 int init_object(Object **self) { // call-by-value
23     if (NULL == (*self = malloc(sizeof(Object)))) return -1;
24     (*self)->a = 0; (*self)->b = 0;
25     (*self)->add = add_impl; (*self)->sub = sub_impl;
26     return 0;
27 }
28
29 int main(int argc, char *argv[])
30 {
31     Object *o = NULL;
32     init_object(&o);
33     o->a = 9922; o->b = 5566;
34     printf("add = %d, sub = %d\n", o->add(o), o->sub(o));
35     return 0;
36 }
```

注意，為何要 `Object **self` 呢？



透過 pointer to pointer 才能確認是否有效

Inheritance and Polymorphism in C

Object-oriented techniques in C

延伸閱讀：

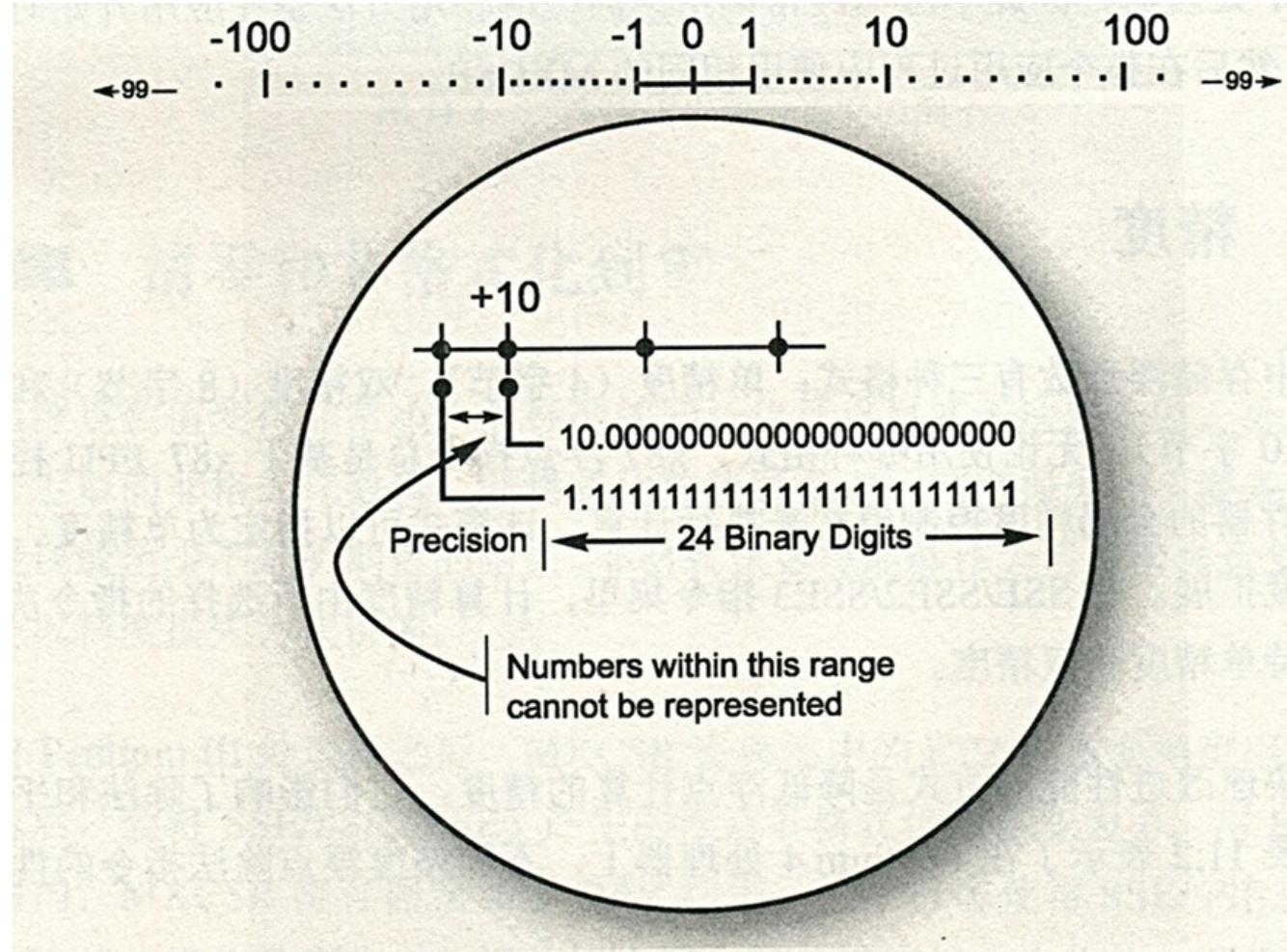
- 探尋 C 語言名稱空間
- Application NoteObject-Oriented Programming in C

## 複習 C 語言程式設計

---

- designated initializers [C99]
  - 在 C90 時，初始化的順序必須要照著宣告時的順序，在 C99 後可以任意指定 member 來初始化
  - 在 struct 的初始化中，Omitted field members are implicitly initialized the same as objects that have static storage duration. (也就是會被初始化成跟 static variable 相同的值)
    - [index] , .fieldname 這樣的形式被稱作 designator
    - 這兩個 designator 也可同時使用，如 struct array 的初始化 `struct point ptarray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };`。union 的初始化方式跟 struct 一樣
- 你所不知道的 C 語言：前置處理器篇
- 你所不知道的 C 語言：指標篇
- C99 [3.14] **object**
  - region of data storage in the execution environment, the contents of which can represent values
  - 在 C 語言的物件就指在執行時期，**資料**儲存的區域，可以明確表示數值的內容

- 很多人誤認在 C 語言程式中，(int) 7 和 (float) 7.0 是等價的，其實以資料表示的角度來看，這兩者截然不同，前者對應到二進位的“111”，而後者以 IEEE 754 表示則大異於“111”



- C99 [6.2.4] ***Storage durations of objects***

- An object has a storage duration that determines its lifetime. There are three storage durations: static, automatic, and allocated.

注意生命週期 (lifetime) 的概念，中文講「初始化」時，感覺像是「盤古開天」，很容易令人誤解。其實 initialize 的英文意義很狹隘：“to set (variables, counters, switches, etc.) to their starting values at the beginning of a program or subprogram.”

- The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined.

在 object 的生命週期以內，其存在就意味著有對應的常數記憶體位址。注意，C 語言永遠只有 call-by-value

- The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

作為 object 操作的「代名詞」(alias) 的 pointer，倘若要在 object 生命週期以外的時機，去取出 pointer 所指向的 object 內含值，是未知的。考慮以下操作 `ptr = malloc(size); free(ptr);` 倘若之後做 `*ptr`，這個 allocated storage 已經超出原本的生命週期

- An object whose identifier is declared with no linkage and without the storage-class specifier static has automatic storage duration.

- C99 [6.2.5] **Types**

- A pointer type may be derived from a function type, an object type, or an incomplete type, called the referenced type. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type T is sometimes called “pointer to T”. The construction of a pointer type from a referenced type is called “pointer type derivation”.

注意到術語！這是 C 語言只有 call-by-value 的實證，函式的傳遞都涉及到數值

這裡的 “incomplete type” 要注意看，稍後會解釋。要區分 `char []` 和 `char *`

- Arithmetic types and pointer types are collectively called scalar types. Array and structure types are collectively called aggregate types.

注意“scalar type”這個術語，日後我們看到 `++`, `--`, `+=`, `-=` 等操作，都是對 scalar (純量)

[來源] 純量只有大小，它們可用數目及單位來表示(例如溫度=30°C)。純量遵守算數和普通的代數法則。注意：純量有「單位」(可用 `sizeof` 操作子得知單位的「大小」)，假設 `ptr` 是個 pointer type，對 `ptr++` 來說，並不是單純 `ptr = ptr + 1`，而是遞增或遞移 1 個「單位」

- An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

這是 C/C++ 常見的 forward declaration 技巧的原理，比方說我們可以在標頭檔宣告

```
struct GraphicsObject; (不用給細部定義) 然後 struct GraphicsObject  
*initGraphics(int width, int height); 是合法的，但 struct GraphicsObject  
obj; 不合法
```

- Array, function, and pointer types are collectively called derived declarator types. A declarator type derivation from a type T is the construction of a derived declarator type from T by the application of an array-type, a function-type, or a pointer-type derivation to T.

這句話很重要，看起來三個不相關的術語「陣列」、「函式」，以及「指標」都稱為 derived declarator types，讀到此處會覺得驚訝的人，表示不夠理解 C 語言

## 案例探討: oltk

再以「[你所不知道的 C 語言：指標篇](#)」提到的 oltk 為例，Polymorphism 的應用在於顯示裝置的支援。試想，如果我們能為 oltk 設計一組高階的 API (“tk” 就是 toolkit 的意思)，然後實做本身可支援 fbdev (Linux framebuffer device) 和 X11 (Linux 桌面系統)，那就可以確保在開發階段和實際在裝置執行的一致性。

### [ [gr\\_impl.h](#) ]

```
1  struct gr_backend {
2      const char *name;
3      struct gr *(*open)(const char *, int width, int height, int nonblock);
4  };
5  extern struct gr_backend gr_x11_backend;
6  extern struct gr_backend gr_fb_backend;
```

雖然我們看到 x11 與 fb 字樣，顯然表示兩種不同的環境，但有趣的是，根本沒看到具體實做，僅有 struct gr\_backend 以及裡頭的 open 與 name，我們稱 open 為 method / virtual function (OOP 用語)。而在 oltk 銜接的地方，是這樣作：

### [ [gr.c](#) ]

```
1 static const struct gr_backend *backends[] = { &gr_x11_backend, &gr_fb_backend };
2 static const int n_backends = sizeof(backends) / sizeof(backends[0]);
3 struct gr *gr_open(const char *dev, int nonblock)
4 {
5     struct gr *gr = NULL;
6     for (int i = 0; i < n_backends; i++) {
7         gr = backends[i]->open(dev, 480, 640, nonblock);
8         if (gr) break;
9         printf("gr_open: backend %d bad\n", i);
10    }
11    return gr;
12 }
```

在此可見到，oltk 初始化時，會逐一去呼叫 graphics backend (也可以說“provider”) 的 open 這個 method，界面都是一樣的，但實際上的平台相依程式碼卻大相逕庭。oltk 可專注在視覺表現和事件處理，而 framebuffer 到底怎麼提供呢？自行看 [gr\\_fb.c](#) 的實做細節即可。

值得注意的是，無論是 [gr\\_fb.c](#) 或 [gr\\_x11.c](#)，這兩個平台相依的實做，在定義實做本體時，都有 gr 的實例 (instance):

```
1 struct gr_fb {
2     struct gr gr;
3     ...
4 };
5 struct gr_x11 {
6     struct gr gr;
7     ...
8 };
```

以 OOP 術語來看，我們可說 gr\_fb 和 gr\_x11 「繼承」 (inherit) 了 gr，熟悉 C++ 程式設計的朋友可能會不以為然，這是“has-a”，並非“is-a”的關聯。的確，我們需要更精確的說法，這裡只是展示概念。

無論實做怎麼變動，oltk 大部份的程式碼只需要在意將按鈕、文字方塊、底圖等視覺元件描繪在 `gr_open` 所指向的真正實做中，定義在 gr.h 的界面：

#### [ gr.h ]

```
1 struct gr {
2     int width; int height;
3     int bytes_per_pixel; int depth;
4
5     void (*close)(struct gr *);
6     void (*update)(struct gr*, struct gr_rectangle *, int);
7
8     void (*set_color)(struct gr *, unsigned int, struct gr_rgb *, unsigned int);
9     int (*get_color)(struct gr *, unsigned int);
10
11    int (*sample)(struct gr* gr, struct gr_sample *);
12};
```

然後在 oltk.c 中，如果想變更按鈕視覺元件的顏色，可以這麼實做：

#### [ oltk.c ]

```
void oltk_button_set_color(struct oltk_button *b,
                           enum oltk_button_state_type state,
                           enum oltk_button_color_type color,
                           int rgb)
{
```

```
    struct gr_rgb grrgb;
    struct gr *gr = b->oltk->gr;
    grrgb.red = ((rgb >> 16) & 0xff) << 8; grrgb.green = ((rgb >> 8) & 0xff) << 8;
    gr->set_color(gr, b->oltk->color_index, &grrgb, 1);
```

顯然，上述程式呼叫的“set\_color”，也是個 method，完全看 gr 這個 instance 指向哪一種平台相依的裝置。

更多的案例可見以下：

- [Inheritance and Polymorphism in C](#)
  - [中文翻譯](#)
- [Object-Oriented Structures in C](#)
- [Monk-C : a toolkit for OOP in pure C](#)
- [Design patterns of 1972](#)
- [我所偏愛的 C 語言物件導向設計模式](#)
- [OOP in C \(非常詳盡\)](#)

## Design Patterns in C

抽象化各種常見的問題，並提供抽象化的解決方案

20年前GoF提出的設計模式，對這個時代是否還有指導意義？

- 二十年前，軟體設計領域的四位大師 (GoF，「四人幫」，又稱 Gang of Four，即 Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides) 通過著作《**Design Patterns: Elements of Reusable Object-Oriented Software**》闡述了設計模式領域的開創性成果
- 最後一章（遺憾的是，讀者們大多直接將其忽略），他們指出：

「這本書的實際價值也許還值得商榷。畢竟它並沒有提出任何前所未有的演算法或者程式設計技術。它也沒能給出任何嚴格的系統設計方法或者新的設計開發理論——它只是**對現有設計成果的一種審視**。大家當然可以將其視為一套不錯的教材，但它顯然無法為經驗豐富的物件導向設計人員帶來多少幫助。」

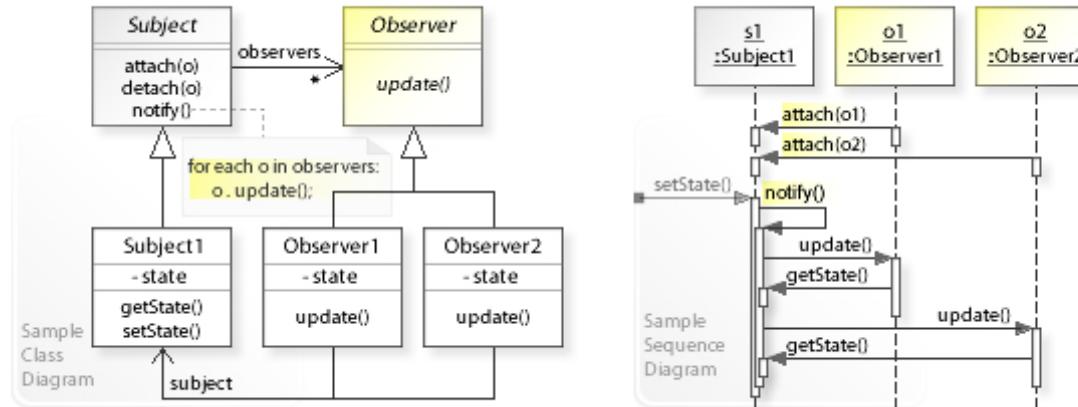
「人們很容易將模式視為一種解決方案，或者說一種能夠進行實際採納及重複使用的技術。相較之下，人們很難弄清其更為確切的核心——即定義其能夠解決的問題以及在哪些背景下才屬於最佳解決方案。總體來講，大多數人都能弄清他人正在做什麼，卻不太了解這樣做的理由——而所謂『理由』對於模式而言正是其需要解決的主要問題。理解模式的目標同樣重要，因為這能幫助我們選定適合自己的模式，也能幫助我們理解現有系統的設計方式。模式作者必須弄清並定義該模式所能解決的問題，或者至少在將其作為解決方案之後做出進一步思考。」（第 393 頁）

模式是一套立足於特定背景，且擁有一整套可預測結果的解決方案

實際案例學習 Design Patterns。事先準備工作：

```
$ git clone https://github.com/QMonkey/OOC-Design-Pattern.git  
$ cd OOC-Design-Pattern  
$ make
```

## Observer Pattern [ [source](#) ]



### Name : Observer

**Context :** 你在購物網站上看到一個很熱門且價格超便宜的產品，正當你想要下單購買的時候，你發現這個產品目前「售完補貨中」。

**Problem :** 如何得知商品已到貨？

### Force :

- 你可以定期連回這個網站查詢產品是否到貨，但是這樣做真的很浪費時間。
- 為了知道商品是否到貨而搬到購物網站的倉庫去住（緊密耦合），是一件很愚蠢的行為。
- 除了你以外，還有很多人都對這個商品有興趣
- 如果商品到貨而你沒有適時地收到通知，你會很火大

**Solution :** 請購物網站 (Subject) 提供「貨到通知」功能，讓你 (Observer) 可以針對有興趣的產品主動加入 (attach) 到「貨到請通知我」的名單之中。你必須將你的電子郵件地址 (update) 提供給購物網站，以便貨品到貨的時候購物網站可以立即通知 (notify) 你。當你對產品不再有興趣的時候，購物網站也要讓你可以從「貨到請通知我」的名單中移除 (detach)。

[ include/iobserver.h ]: 檔名開頭的 `i` 字母表示 interface

```
typedef struct _IObserved IOObserved;
typedef struct _IObserver IOObserver;
struct _IObserved { /* 商品 */
    void (*registerObserver)(IOObserved *, IOObserver *);
    void (*notifyObservers)(IOObserved *);
    void (*removeObserver)(IOObserved *, IOObserver *);
};

struct _IObserver { /* 顧客 */
    void (*handle)(IOObserver *);
};
```

[ include/observed.h ]

```
typedef struct _Observed Observed;
struct _Observed {
    IOObserver **observers;
    size_t count;
    size_t size;
    union {
        IOObserved;
        IOObserved iobserved;
    };
};

extern Observed *Observed_construct(void *);
extern void Observed_destruct(Observed *);
```

[ include/observer.h ]

```
#include "iobserver.h"
typedef struct _Observer Observer;
struct _Observer {
    union {
        IObserver;
        IObserver iobserver;
    };
};

extern Observer *Observer_construct(void *);
extern void Observer_destruct(Observer *);
```

Observer 是種被動呼叫的典範，當有份資料會不定期的更新，而許多人都想要最新版的資料時，有兩種做法，最簡單的是寫個執行緒不斷去確認，但這致使 CPU 的 busy wait；另一種做法就是當資料更新時，由資料擁有者去通知大家「嘿！新貨到！」，這時在意的人就會跑過來看，如此模式即是“observer”。

要留意的是，現實生活中要讓跑過來的眾人不要撞在同一個時間點，如果是網路模組也是，可能會造成阻塞。

[ src/main.c ]

```
const int OBSERVER_SIZE = 10;
int main() {
    Observed *observed = new (Observed);
    IObserved *iobservered = &observed->iobservered;
    Observer *observers[OBSERVER_SIZE];
    for (int i = 0; i < OBSERVER_SIZE; ++i) {
        observers[i] = new (Observer);
        observed->registerObserver(iobservered, &observers[i]->iobserver);
        printf("handle: %p\n", &observers[i]->iobserver);
    }
    printf("\n");
```

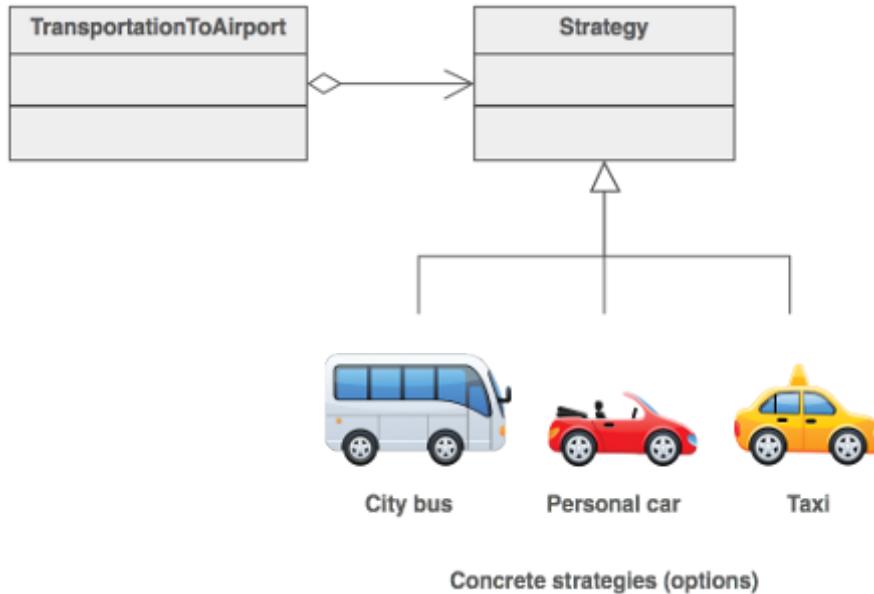
```
iobserved->notifyObservers(iobserved);
for (int i = 0; i < OBSERVER_SIZE; ++i)
    delete(Observer, observers[i]);
delete(Observed, observed);
return 0;
}
```

在 GNOME/Gtk+ 中，Observer pattern 很常見。依據 [The GLib/GTK+ Development Platform](#) 第 51 頁：

In fact, creating a GObject signal or property is a nice way to implement the Observer design pattern; that is, one or several objects observing state changes of another object, by connecting function callbacks. The object emitting the signal is not aware of which objects receive the signal. GObject just keeps track of the list of callbacks to call. So adding a signal permits to decouple classes

- 搭配閱讀: [Discovering GObject signals](#)

## Strategy Pattern [ [source](#) ]



**Name :** Strategy

**Context :** 為了達到相同的目的，物件可以**因地制宜**，讓行為擁有多種不同的實作方法。例如，一個壓縮檔案物件，可以採用 zip、arj、rar、tar、7z 等不同的演算法來執行壓縮工作。

**Problem :** 如何讓物件自由切換演算法或行為實作？

**Force :**

- 你可以把所有的演算法全部寫進同一個物件，然後用條件式判斷來選用所要執行的版本，但是：
  - 物件的程式碼很容易變得過於複雜與肥大，不好理解與修改。
  - 物件占用過多的記憶體空間，因為可能不會使用到全部的演算法。
  - 擴充新的演算法必須要修改既有的程式碼。
  - 不容易分別開發、修改與測試每一個演算法。

- 你可以透過繼承，讓子類被重新定義自己的演算法。但是這樣會產生許多類似的類別，但僅僅只有行為上些微的差別。

**Solution :** 將演算法從物件 (Context) 身上抽離出來。為演算法定義一個 Strategy 介面，針對每一種演算法，新增一個實作 Strategy 介面的 ConcreteStrategy 物件。把物件中原本的演算法實作程式碼移到相對應的 ConcreteStrategy 物件身上。讓 Context 物件擁有一個指向 Strategy 介面的成員變數，在執行期間藉由改變 Strategy 成員變數所指向的 ConcreteStrategy 物件，來切換不同的演算法演算法。

[ include/itravel\_strategy.h ]

```

1  typedef struct _ITravelStrategy ITravelStrategy;
2  struct _ITravelStrategy {
3      void (*travel)(ITravelStrategy *);
4 }
```

[ include/person.h ]

```

1  typedef struct _Person Person;
2  struct _Person {
3      ITravelStrategy* travelStrategy;
4      void (*setTravelStrategy)(Person*, ITravelStrategy*);
5      void (*travel)(Person*);
6  };
7  extern Person *Person_construct(void *, ITravelStrategy *);
8  extern void Person_destruct(Person *);
```

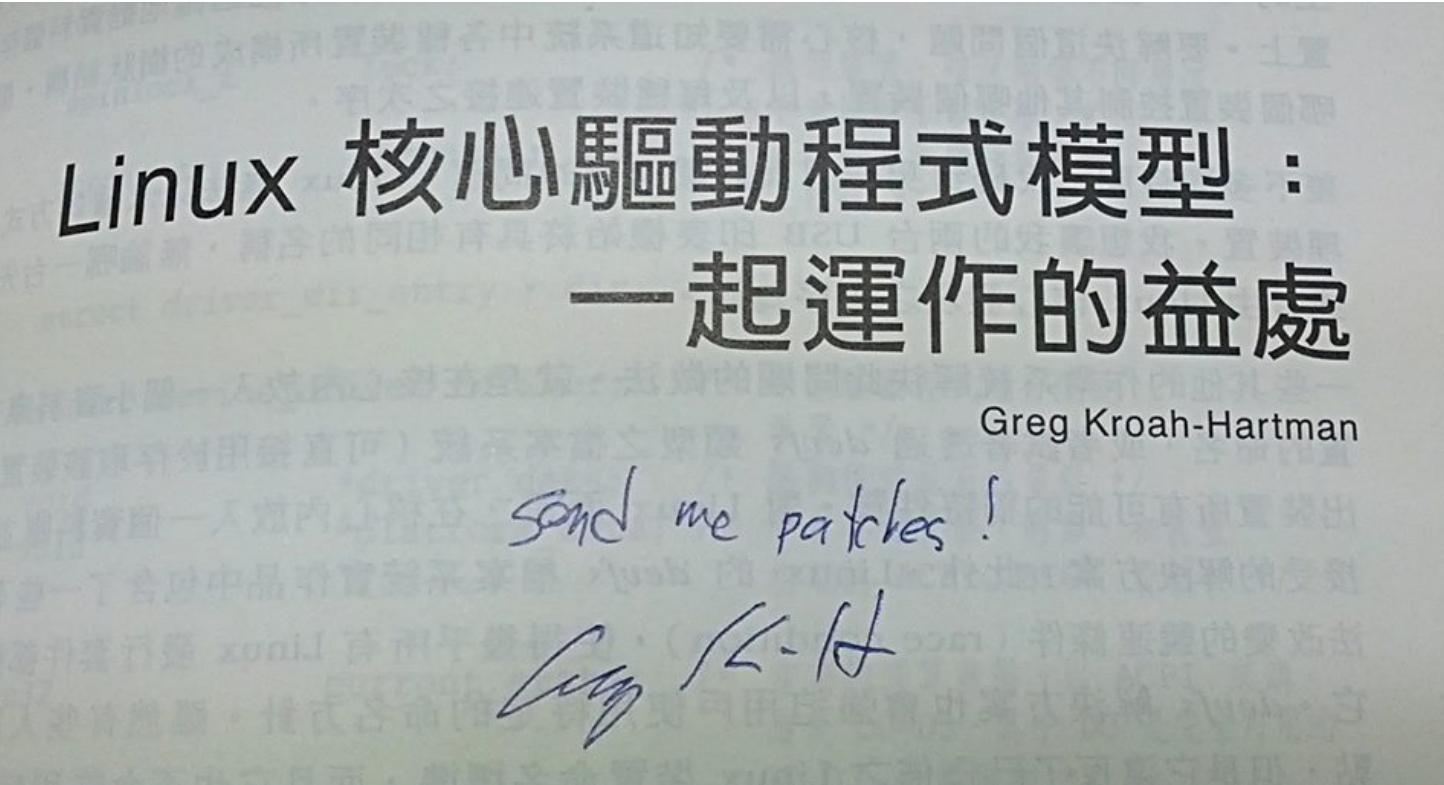
旅行時留下深刻的像 (?!)

[ src/main.c ]

```
1 #include "base.h"
2 #include "itravel_strategy.h"
3 #include "airplane_strategy.h"
4 #include "train_strategy.h"
5 #include "person.h"
6 int main() {
7     TrainStrategy *trainStrategy = new (TrainStrategy);
8     ITravelStrategy *itravelStrategy = &trainStrategy->itravelStrategy;
9     Person* person = new (Person, itravelStrategy);
10    person->travel(person);
11
12    AirplaneStrategy *airplaneStrategy = new (AirplaneStrategy);
13    itravelStrategy = &airplaneStrategy->itravelStrategy;
14    person->setTravelStrategy(person, itravelStrategy);
15    person->travel(person);
16
17    delete (Person, person);
18    delete (AirplaneStrategy, airplaneStrategy);
19    delete (TrainStrategy, trainStrategy);
20    return 0;
21 }
```

Strategy pattern 的重要範本，很多 pattern 其實是從 Strategy pattern 延伸，基本概念是：當一件事情有很多種做法的時候，可在執行時期選擇不同的作法，而非透過 if-else 去列舉，這樣能讓程式碼更容易維護、增修功能。

## Object-Oriented Programming in Linux Kernel



取自《Beautiful Code》

- Object-oriented design patterns in the kernel (1)
- Object-oriented design patterns in the kernel (2)

## 案例分析: 通訊系統的可擴充界面



[ [source](#) ]

定義一個通用的通訊界面:

```
typedef struct {
    int (*open) (void *self, char *fspec);
    int (*close) (void *self);
    int (*read) (void *self, void *buff, size_t max_sz, size_t *p_act_sz);
    int (*write) (void *self, void *buff, size_t max_sz, size_t *p_act_sz);
} tCommClass;

tCommClass commRs232; /* RS-232 communication class */
commRs232.open = &rs232Open;
commRs232.write = &rs232Write;
```

```
tCommClass commTcp; /* TCP communication class */
commTcp.open = &tcpOpen;
commTcp.write = &tcpWrite;
```

對應的通訊實做: [TCP subclass]

```
static int tcpOpen (tCommClass *tcp, char *fspec) {
    printf ("Opening TCP: %s\n", fspec);
    return 0;
}
static int tcpInit (tCommClass *tcp) {
    tcp->open = &tcpOpen;
    return 0;
}
```

當然你也可以定義 HTTP subclass

```
static int httpOpen (tCommClass *http, char *fspec) {
    printf ("Opening HTTP: %s\n", fspec);
    return 0;
}
static int httpInit (tCommClass *http) {
    http->open = &httpOpen;
    return 0;
}
```

測試程式:

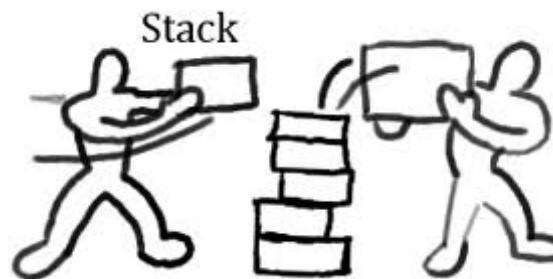
```
int main (void) {
    int status;
    tCommClass commTcp, commHttp;

    // Same 'base' class but initialized to
    // different sub-classes.
    tcpInit (&commTcp);
    httpInit (&commHttp);

    // Called in exactly the same manner.
    status = (commTcp.open)(&commTcp, "bigiron.box.com:5000");
    status = (commHttp.open)(&commHttp, "[ ](http://www.microsoft.com)http://www.mi

    return 0;
}
```

## 案例分析: Stack ADT



[ [source](#) ]

namespace:

- `stack_push(thing *); /* C */`
- `stack::push(thing *); // C++`

用 C++ 表示:

```
class stack {
public:
    stack();
    void push(thing *);
    thing *pop();
    static int an_example_entry;
private:
    ...
};
```

「慣 C」表示:

```
struct stack {
    struct stack_type *self;
    /* Put the stuff that you put after private: here */
};

struct stack_type {
    void (*construct)(struct stack *self); /*< initialize memory */
    struct stack *(*operator_new)(); /*< allocate a new struct to construct */
    void (*push)(struct stack *self, thing *t);
    thing * (*pop)(struct stack *self);
    int as_an_example_entry;
} Stack = {
    .construct = stack_construct,
    .operator_new = stack_operator_new,
    .push = stack_push,
```

```
.pop = stack_pop  
};
```

使用方式:

```
struct stack *st = Stack.operator_new(); /* a new stack */  
if (!st) {  
    /* Do something about error handling */  
} else {  
    stack_push(st, thing0); /* non-virtual call */  
    Stack.push(st, thing1); // cast *st to a Stack and push  
    st->my_type.push(st, thing2); /* a virtual call */  
}
```

## Prototype-based OO

- 以 C 語言實做 Javascript 的 prototype 特性

## 案例分析: Server Framework in modern C

- **server-framework**: 用 C99 撰寫、具備物件導向程式設計風格的伺服器開發框架
- 由三部份組成:
  - **async**
  - **reactor**
  - **protocol-server**
- 取得原始碼並編譯:

```
$ git clone https://github.com/embedded2016/server-framework.git  
$ cd server-framework  
$ make
```

- 測試 [protocol-server](#)

```
$ ./test-protocol-server
```

- 預期將看到類似以下訊息:

```
(pid 4130 : 8 threads) Listening on port 8080 (max sockets: 65536, ~5 used)
```

- 這時開啟新的終端機，輸入 `telnet localhost 8080`，可以發現原本執行 `test-protocol-server` 的終端機印出以下訊息:

```
A connection was accepted on socket #7.
```

- 回到執行 `telnet` 的終端機，可發現以下輸出:

```
Trying 127.0.0.1...  
Connected to localhost.  
Escape character is '^]'.  
Simple echo server. Type 'bye' to exit.
```

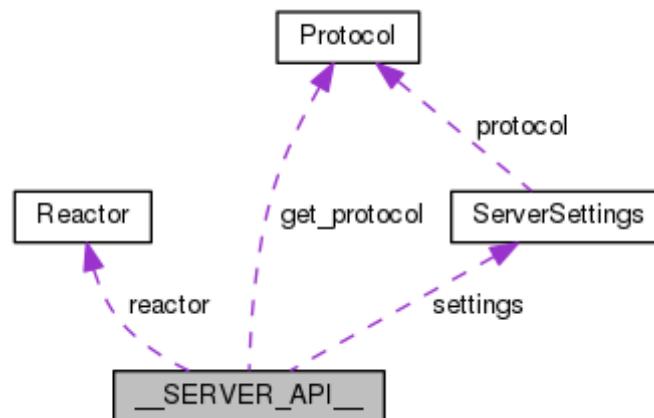
- 試著打字 (記得按下 Enter)，會發現會重複輸出，直到輸入 `bye` 時，才會結束連線

- 當對伺服器按下 Ctrl-C 時，預期將看到類似以下輸出:

```
server done  
  
(4223) Stopped listening for port 8080
```

## 使用 Doxygen 自動產生文件

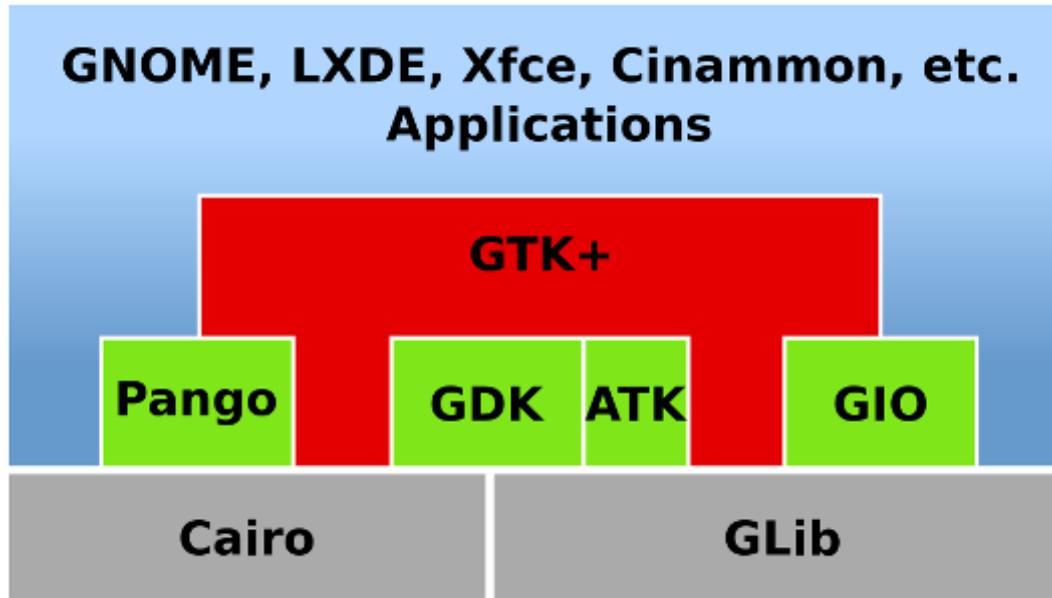
- Doxygen 可掃描給定原始程式碼內容、註解，之後將自動產生程式文件，其中一種輸出就是 HTML，可透過瀏覽器來閱讀文件和程式碼
- 執行 make doc，之後會產生新的目錄 html，可用 Firefox 或 Chrome browser 開啟該目錄底下的 index.html 檔案
- 類似以下網頁畫面，注意右邊有搜尋功能：
- 按下“Data Structures”頁面可探究物件相依性示意圖：



- 延伸閱讀: [你所不知道的 C 語言: 從打造類似 Facebook 網路服務探討整合開發](#)

## GObject 設計與實做

- GObject 是 **GLib** 的核心技術，而 GLib 又是 Gtk+ 的重要根基，GNOME 桌面系統建構於 Gtk+



- Glib-C: C as an alternative Object Oriented Environment

## CHAPTER 6. UNIT- AND REGRESSION TESTING

qemu object model: <https://github.com/Gyumeijie/qemu-object-model>

## Q & A

- 所以Observer Pattern就是，原本多人去觀察一個『事件』，變成由擁有者去觀察『事件』，再通知有追蹤的人？所以，擁有者還是有需要一個Loop去觀察這個『事件』？

我想Observer Pattern想解決的問題是「減少追蹤者的麻煩」，事件擁有者如何觀察事件並不是Observer Pattern想要關心的。若事件擁有者是事件的起源（或即事件本身），更新事件與通知同時一起做。

— 楊日新

- ZeroMQ 的 C 語言設計原則：<http://rfc.zeromq.org/spec:21> 也相當值得一讀，其他極簡化的 C library 有：
  - klib: <http://attractivechaos.github.io/klib/> (主要是提供 function)
  - libmowgli (OO interface for C, 可惜沒有 active development)
  - Cello for C, 介面非常漂亮，不過 GC 會停住整個世界。
- 如果希望程式能在 32bit 跟 64bit 機器上都能最佳化，有什麼要注意的？
- ~~Visitor pattern 是個相當實用的模式 (例如實現一個解釋執行 AST 的 Interpreter，我們能讓樹中不同類型的節點都有個名為 eval 的 method，該 method 的作用是對節點自身解釋求值，只要遞迴地呼叫各個節點的 eval，最終就能得到執行結果)，請問如何用 C 漂亮地實現這個模式？~~
  - 補充：我已經去 RTFM、STFW 了，找到了示範 <https://github.com/QMonkey/OOC-Design-Pattern/tree/master/Visitor/src>
- 以 libmodbus 為例探討物件導向設計
- <https://nullprogram.com/blog/2014/10/21/>