實機演練題目

- 撰寫你的第一支 JNI 本地程序
- 使用你的 Android NDK 和 SDK 環境，實際執行下述的應用程序。
- 並核對你的程序的執行結果。

1. 介紹 LW_OOPC

認識 OOPC

OOPC 是指 OOP(Object-Oriented Programming)與 C 語言的結合，藉由 C 語言\在一般 Embedded 系統開發上，經常只用到其中的小部分功能而已，而不需要用到全部的機制，例如多重繼承(Multiple Inheritance)、運算子的重複定義(Overloading)等等。此時，許多 Embedded 系統開發者就捨棄 C++的龐大身軀而回歸到精簡的 C 環境裏。只是捨棄 C++的同時也捨棄了珍貴的 OOP 能力，實在太可惜了。

OOPC 就是要彌補這個缺憾。它是以 C 的巨集寫成的 Header 檔案，可以任由 C 程式師去對它瘦身美容，刪去不需要的部分，挑出自己所需要的 OOP 特性，以小而美的身材滿足 Embedded 系統開發的需要。

認識 LW_OOPC

LW_OOPC 是一種輕便又快速的面向對象 C 語言。在嵌入式程式師還是蠻青睞 C 語言的，只是 C 語言沒有對象、類等概念，程式很容易變成義大利面型的結構，維護上比較費力。在 1986 年 C++上市時，希望大家改用 C++，但是 C++的效率不如 C，並不受嵌入式程式師的喜愛。於是，MISOO 團隊設計一個輕便又高效率的 OOPC 語言。輕便的意思是：它只用了約 20 個 C 宏敍述而已，簡單易學。其宏如下：

```
/*    lw_oopc.h */    /*  這就MISOO團隊所設計的C宏  */
#include "malloc.h"
#ifndef LOOPC_H
#define LOOPC_H

#define CLASS(type)\
typedef struct type type; \
struct type

#define CTOR(type) \
void* type##New() \
{ \
```

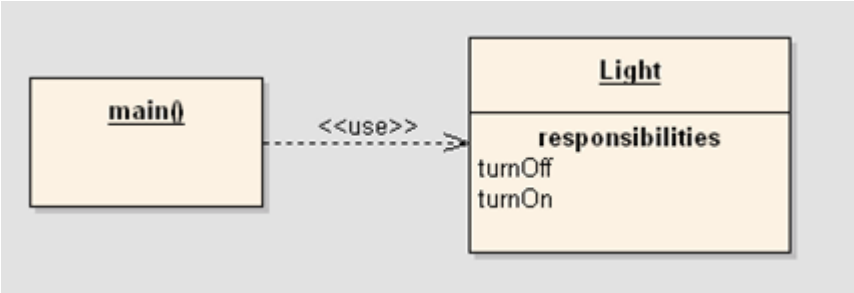
```
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define CTOR2(type, type2) \
void* type2##New() \
{ \
  struct type *t; \
  t = (struct type *)malloc(sizeof(struct type));

#define END_CTOR return (void*)t;    };
#define FUNCTION_SETTING(f1, f2)    t->f1 = f2;
#define IMPLEMENTS(type) struct type type
#define INTERFACE(type) struct type
#endif
/*          end          */
```

其高效率的意思是，它沒提供類繼承，內部沒有虛擬函數表(Virtual Function Table)，所以仍保持原來 C 語言的高效率。除了沒有繼承機制之外，它提供有類、對象、資訊傳遞、介面和介面多型等常用的機制。目前受到不少 C 程式師的喜愛。

簡單示例

由於一般 C 語言並沒有使用 OO 觀念，從軟體設計圖實現爲 C 程式過程中，讓人感到不順暢。所以 MISOO 團隊將 OO 觀念添加到 C 語言，讓人們既以 OO 觀念去分析及設計，並以 OO 觀念去編寫 C 程式，則從分析、設計到程式編寫的過程就非常直截了當。如下述的步驟：

Step-1: 分析出一個類叫 Light，它提供兩項服務，以 UML 表達如下：



Step-2：實現爲 LW_OOPC 程式：

基於上述的lw_oopc.h就可以定義出類了，例如定義一個Light類，其light.h內容爲：

```
/*    Ex_light.h      */
#include "lw_oopc.h"

CLASS(Light) {
        void (*turnOn)();
        void (*turnOff)();
```

```
};
```

類裏的函數定義格式爲：

回傳值的型態 (*函數名稱)();

類定義好了，就開始編寫函數的實現內容：

```
/* Ex_light.c      */
#include "stdio.h"
#include "Ex_light.h"

static void turnOn()
    {    printf("Light is ON\n");    }
static void turnOff()
    {    printf("Light is OFF\n");    }

CTOR(Light)
    FUNCTION_SETTING(turnOn, turnOn)
    FUNCTION_SETTING(turnOff, turnOff)
END_CTOR
```

這個 FUNCTION_SETTING(turnOn, turnOn)宏的用意是：讓類定義(.h檔)的函數名稱能夠與實現的函數名稱不同。例如在light.c裏可寫爲：

```
static void TurnLightOn()
{ ….. }

CTOR(Light)
{
    FUNCTION_SETTING(turnOn, TurnLightOn);
    …..
}
```

這是創造.c檔案自由抽換的空間，這是實踐介面的重要基礎。最後看看如何編寫主程序：

```
#include "lw_oopc.h"
#include "Ex_light.h"

extern void* LightNew();
void main()
{
      Light* light = (Light*)LightNew();
      light->turnOn();
      light->turnOff();
```

```
    getchar();
    return;
}
```

LightNew()是由 CTOR 宏所生成的類構造器(Constructor)。由於它是定義於別的檔案，所以必需加上 extern void* LightNew();指令。生成對象的基本格式爲：

類名稱*　對象指標 = (類名稱*)類名稱 New();

示例：Light light = (Light*)LightNew()*

然後就能透過對象指標去調用成員函數了。

簡介 LW_OOPC 的宏

LW_OOPC 的主要宏是：CLASS(類別名稱)和 CTOR(類別名稱)。

- **CLASS 宏**

它用來定義類別的函數和資料。定義好了，程式裏就可以拿它來誕生對象，並透過對象而呼叫到類別的函數。其使用格式是：

```
CLASS(類別名稱) {
    回傳值型態  (*函數名稱 1)(參數名稱 1,  參數名稱 2, ……);
    回傳值型態  (*函數名稱 2)(參數名稱 1,  參數名稱 2, ……);
    ……
    變數型態  變數名稱 1;
    變數型態  變數名稱 2;
     ……
};
```

例如，

```
CLASS(Rectangle) {
    int (*get_area)();
    int (*get_perimeter)();
    int length;
    int width;
};
```

這定義了 Rectangle 類別，它含有 2 個函數和 2 個變數。這兩個變數就是這些函數所共用的變數，這些變數之值就是此類別的資料值，爲此類別各函數所共用。

- **CTOR 宏**

它用來定義類別的建構式(Constructor)函數。建構式的任務是誕生對象，在對象誕生時它會安排所需要的記憶體空間。CTOR 宏的使用格式是：

```
CTOR(類別名稱)
    FUNCTION_SETTING(類別的函數名稱 1, 實作函數名稱 1)
    FUNCTION_SETTING(類別的函數名稱 2, 實作函數名稱 2)
    …….
END_CTOR
```

例如，

```
CTOR(Rectangle)
    FUNCTION_SETTING(get_area, area)
    FUNCTION_SETTING(get_perimeter, perimeter)
END_CTOR
```

這個 FUNCTION_SETTING()就是將類別函數對應到實作的函數。前面已經說明過，LW_OOPC 能對既有的 C 程式黃袍加身，此時 area()通常是既有的函數，而像 get_area()只是類別內的函數名稱，但 get_area()本身並沒有實做指令，所以依賴 FUNCTION_SETTING()將 get_area()對應 area()，這樣 get_area()就有實作部份，類別完整了，也將 area()函數歸類了。

CTOR 宏會產生出另一個建構式，此建構式的格式爲：

類別名稱 New()

你的程式可以呼叫它來誕生一個對象，它會將新對象的指標傳遞回來。例如，

```
Rectangle *pr = (Rectangle*)RectangleNew();
```

此時，pr 就指向這個新誕生的對象了。此外，CTOR 宏也會產生退化型的建構式，其格式爲：類別名稱 Setting(&對象名稱)

例如，

```
Rectangle rect;
RectangleSetting( &rect );
```

其中，RectangleNew()擔任兩件工作：

---- 呼叫 malloc()函數分派記憶體給新對象。

---- 執行 FUNCTION_SETTING()的函數對應設定。

至於，RectangleSetting(&rect)則只擔任後者：FUNCTION_SETTING()的函數對應設定而已。因爲指令----Rectangle rect; 已經誕生了對象，RectangleSetting()只是替此既有的對象設定其函數對應而已。

[注] 關於 LW_OOPC 詳細用法請參閱:

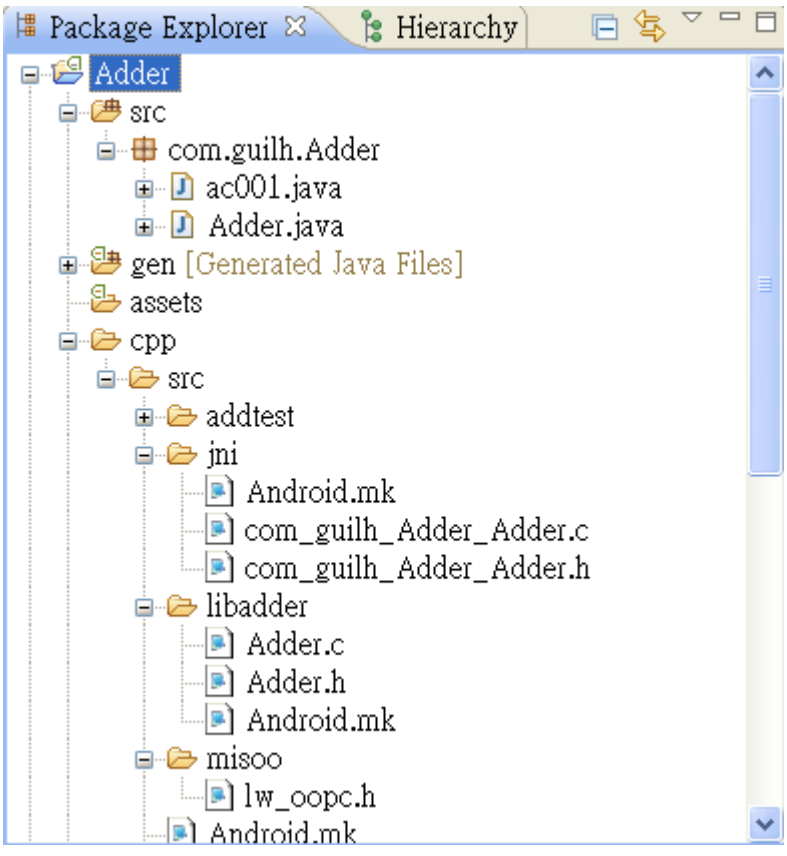
<<UML+OOPC 嵌入式 C 語言開發精講>> 一書(博文視點出版)

2. Android 的本地代碼範例

Android 的 NDK 本地程式開發是以 JNI 爲橋樑，將 Java 與 C/C++結合起來。雖然 C++是個面向對象的語言，但是有很多 Android 底層的服務(Service)或驅動(Driver)仍只提供 C 的介面，有時不得不用 C 來編寫代碼時，可使用 OOPC 來寫出易讀易懂的面向對象代碼。本節就來舉個非常簡單的範例，說明如何以 OOPC 來撰寫 Android 的本地代碼。其步驟如下：

Step-1. 建立一個 Adder 開發專案

在 workspace 裏，建立一個專案，如下：



Step-2. 撰寫 Java 程式

撰寫 Adder.java 介面定義類別，其內容如下：

/* Adder.java */

```
package com.guilh.Adder;
public class Adder {

	static {
		System.loadLibrary("adder_jni");
	}
	int sum(int a, int b)
	{
		setA(a);
		setB(b);
		return getSum();
	}
	private native void setA(int a);
	private native void setB(int b);
	private native int getSum();
}
```

這個 JNI 介面定義類別含有 3 個本地函數：setA()、setB()和 getSum()。

Step-3. 編譯上述的 Java 程式碼，產出 Adder.class 檔案

以下步驟，將由 Adder.class 來產生.h 介面定義標頭檔。

Step-4. 以 javah 轉譯 Adder.class 檔

使用 javah 轉譯，而產出 com_guilh_Adder_Adder.h 標頭檔，其內容如下：

/* com_guilh_Adder_Adder.h */

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_guilh_Adder_Adder */

#ifndef _Included_com_guilh_Adder_Adder
#define _Included_com_guilh_Adder_Adder
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_guilh_Adder_Adder
 * Method:    setA
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_com_guilh_Adder_Adder_setA
  (JNIEnv *, jobject, jint);

/*
 * Class:     com_guilh_Adder_Adder
 * Method:    setB
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_com_guilh_Adder_Adder_setB
  (JNIEnv *, jobject, jint);
```

```
/*
  * Class:          com_guilh_Adder_Adder
  * Method:        getSum
  * Signature: ()I
  */
JNIEXPORT jint JNICALL Java_com_guilh_Adder_Adder_getSum
   (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Step-5. 撰寫 com_guilh_Adder_Adder.c 程式碼

現在看看如何撰寫 com_guilh_Adder_Adder.c 程式碼，如下：

/* com_guilh_Adder_Adder.c */

```
#include "com_guilh_Adder_Adder.h"
#include <libadder/Adder.h>

static Adder* adder = 0;
static Adder* getAdder()
       {
          if (adder == 0)
                 adder = Adder_new();
          return adder;
       }
JNIEXPORT void JNICALL Java_com_guilh_Adder_Adder_setA
          (JNIEnv* env, jobject job, jint a)
         {
             getAdder()->setA(adder, a);
         }
JNIEXPORT void JNICALL Java_com_guilh_Adder_Adder_setB
          (JNIEnv* env, jobject job, jint b)
         {
             getAdder()->setB(adder, b);
         }
JNIEXPORT jint JNICALL Java_com_guilh_Adder_Adder_getSum
         (JNIEnv* e, jobject job)
         {
            return getAdder()->getSum(adder);
       }
```

Step-6. 撰寫 Adder.h 及 Adder.c 程式碼

現在看看如何撰寫 Adder.h 和 Adder.c 程式碼，如下：

/* Adder.h */

```
#ifndef ANDROID_GUILH_OOPC_ADDER_H
#define ANDROID_GUILH_OOPC_ADDER_H
```

```
#include <misoo/lw_oopc.h>

CLASS(Adder)
{
	int a;
	int b;

	void (*setA)(Adder* thiz, int a);
	void (*setB)(Adder* thiz, int b);
	int (*getSum)(Adder* thiz);
};

#endif // ANDROID_GUILH_OOPC_ADDER_H
```

/* Adder.c */

```
#include "Adder.h"
void setA(Adder* thiz, int a)
{
	thiz->a = a;
}

void setB(Adder* thiz, int b)
{
	thiz->b = b;
}

int getSum(Adder* thiz)
{
	return thiz->a + thiz->b;
}

CTOR(Adder)
	FUNCTION_SETTING(setA, setA);
	FUNCTION_SETTING(setB, setB);
	FUNCTION_SETTING(getSum, getSum);
END_CTOR

DTOR(Adder)
END_DTOR
```

由 JNI 的 getAdder()來誕生 Adder 對象。

Step-7. 進行編譯(產出.o 檔)

對 Adder.c 和 com_guilh_Adder_Adder.c 進行編譯，可產出 Adder.o 和 com_guilh_Adder_Adder.o 目的程式檔。

Step-8. 進行連結(產出.so 檔)

對 Adder.o 進行連結，產出 libAdder.so 共用程式檔。而且，對 com_guilh_Adder_Adder.o 進行連結，產出 libAdder_jni.so 共用程式檔。

Step-9. 將.so 檔與*.apk 打包起來

詳細做法請參閱後續的 NDK 介紹。

Step-10. 執行 Adder.apk

執行的結果是：輸出兩個整數的總和。

~ END ~