嵌入式系統

單元三：即時作業系統基本觀念

授課教師：雲林科技大學 張慶龍 老師

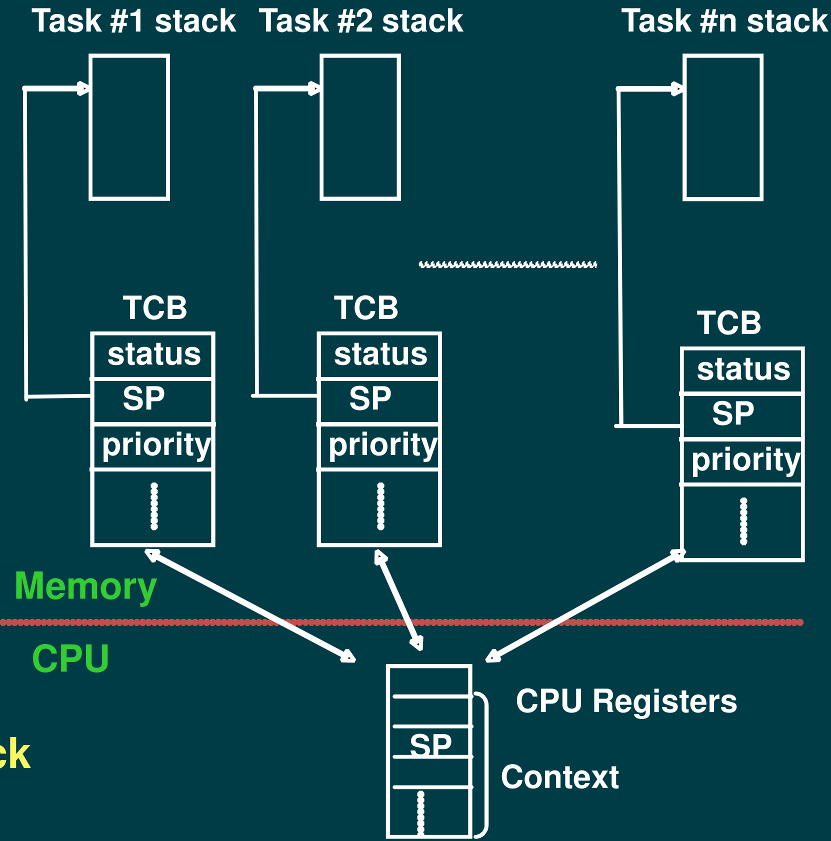
單元學習目標與大綱

- 什麼是即時作業系統
- Blocking/non-blocking函數
- Reentrancy/non-reentrancy函數
- C語言之區域/靜態/全域變數

Part 1
什麼是即時作業系統

什麼是即時作業系統

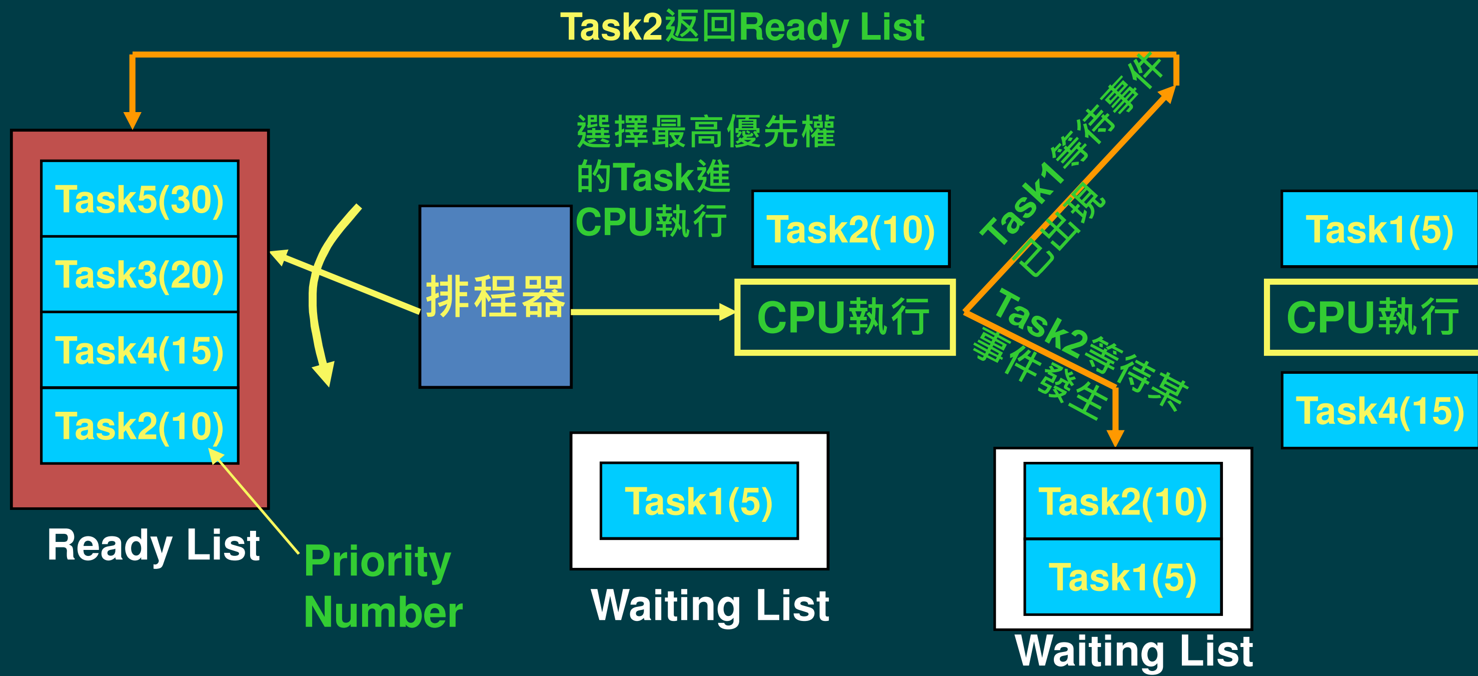
- 一個系統功能是由幾個獨立執行的工作(Task)互相合作來完成
  - 此處Task、Process、Thread皆看成相同的意義
  - 多個Task平行執行
- 每個Task會設定其對應的堆疊(Stack)與執行的優先權
- Task的執行順序是依實際事件(Event)發生的順序與優先權決定
  - Event Driven
  - Priority-based排程
- 每個Task基本上會有三個狀態
  - Ready state: 等待進入CPU執行，由Ready-list管理
  - Running state: 正在CPU上執行
  - Waiting state: 等待事件發生，由Waiting-list管理



TCB: Task Control Block

即時作業系統運作的正確性

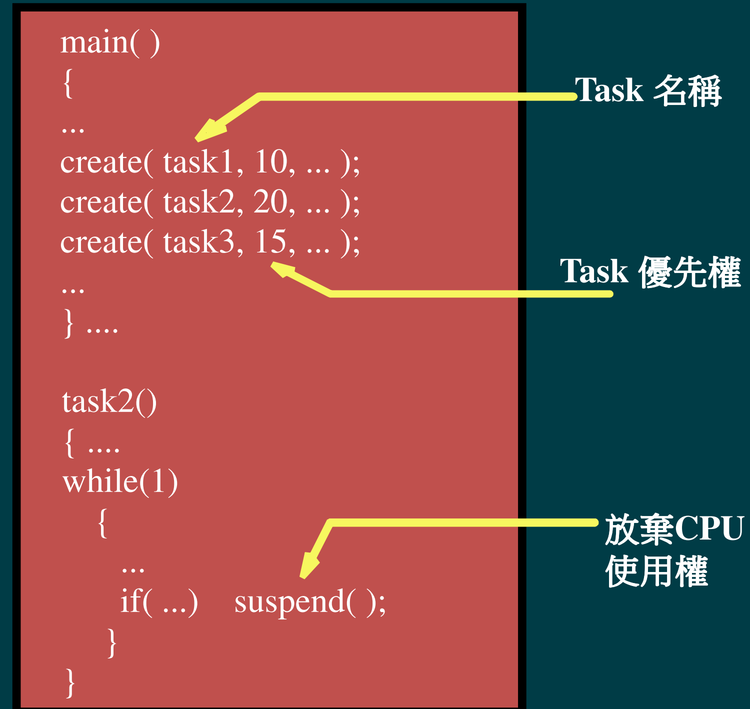
- 邏輯運作的正確性
- 反應的即時性
  - 只要能在事件的Deadline前反應，皆滿足反應的即時性
  - 根據每一事件即時性要求(Deadline的時間長短)，決定對應的Task的優先權
  - 優先權愈高的，即擁有愈高的CPU使用權➔對事件的反應時間亦較短



什麼是即時作業系統
Task2返回Ready List
選擇最高優先權
的Task進
CPU執行
Task5(30)
Task3(20)
Task4(15)
Task2(10)
排程器
Task2(10)
CPU執行
Task1等待事件
已出現
Task2等待某
事件發生
Task1(5)
CPU執行
Task4(15)
Ready List
Priority
Number
Task1(5)
Waiting List
Task2(10)
Task1(5)
Waiting List

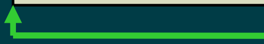
Task程式架構

- 一個系統功能是由多個Tasks共同合作完成
- 每個Task有
  - Task 名稱
  - Task 優先權
  - Task 堆疊
- 每個Task為各別獨立運作之程式碼
  - 為一無窮迴圈
  - 需呼叫blocking function
    - 做context switch
    - 讓其它Task亦有執行機會
  - Task與Task溝通方式稱為Inter-process Communication



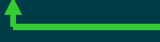
Tasks in RTOS

High Priority Task



importance

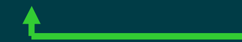
Low Priority Task



Each task is event driven

event event

Task



Infinite loop

Example of Tasks

Tasks

```
Void App_TaskADC (void *p_arg)
{
    while (1) {
        ADC_Read();
        sleep for 1 ms;
    }
}
```

```
Void App_TaskUSB (void *p_arg)
{
    while (1) {
        Wait for signal from ISR;
        USB_packet();
    }
}
```

ISRs

```
Void App_ISRUSB (void)
{
    clear interrupt;
    signal USB Task
}
```

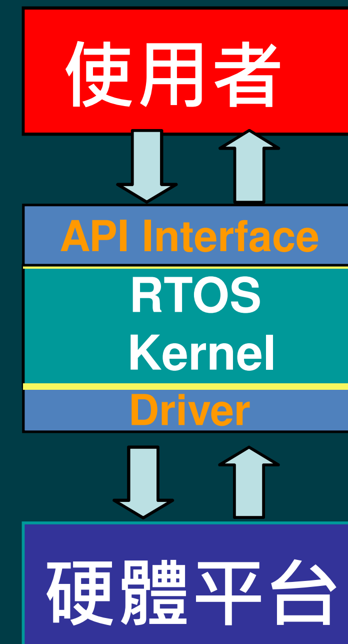
Part 2

Blocking/Non-blocking Function

Synchronous I/O與asynchronous I/O

RTOS作業環境

- 即時性作業系統最主要的功能作就是管理Tasks間之CPU使用時間
- RTOS Kernel提供相關的排程工作、資源管理與相關服務給使用者程式
- 藉由application program interface (API)提供相關之功能服務給使用者呼叫/使用
  – 所謂API即是系統所提供之function calls
- API functions可分為
  – Blocking function call
  – Non-blocking function call



Blocking function

- RTOS環境, 資源是各個Task所共享
  - 當有一Task A佔用某一資源(resource)時(如:印表機)，若Task B也要使用相同資源時
    - Task B需等待Task A釋放出資源時，才可繼續執行
  - 或Task A要讀取網路封包，此時網路卡尚未收到網路封包，則Task A需等待，直至網路卡收到封包
- Task主要是靠API function call取得資源
  - 若function call無法立即取得資源，則呼叫的Task必需停在那(waiting state)，會停多久不確定➔會做context switch
  - 此種無法立即完成工作的function call稱為blocking function

Non-blocking function

- Function call會立即返回，即不會被blocking住
- 如:某一Task有很多事情要做
  - 接收網路封包
  - 讀取鍵盤按鍵值
  - ...
- Task不直接呼叫讀取封包之function call(可能會被blocking)
  - 先讀取網路卡接收狀態暫存器
  - 先確認網路卡是否有收到封包(如同polling)
    - 若有封包，才呼叫讀取封包的function call
    - 若沒有封包，就去做其它事情
- 一直到此Task都沒有事情做時，才會去呼叫blocking call

Synchronous I/O與asynchronous I/O

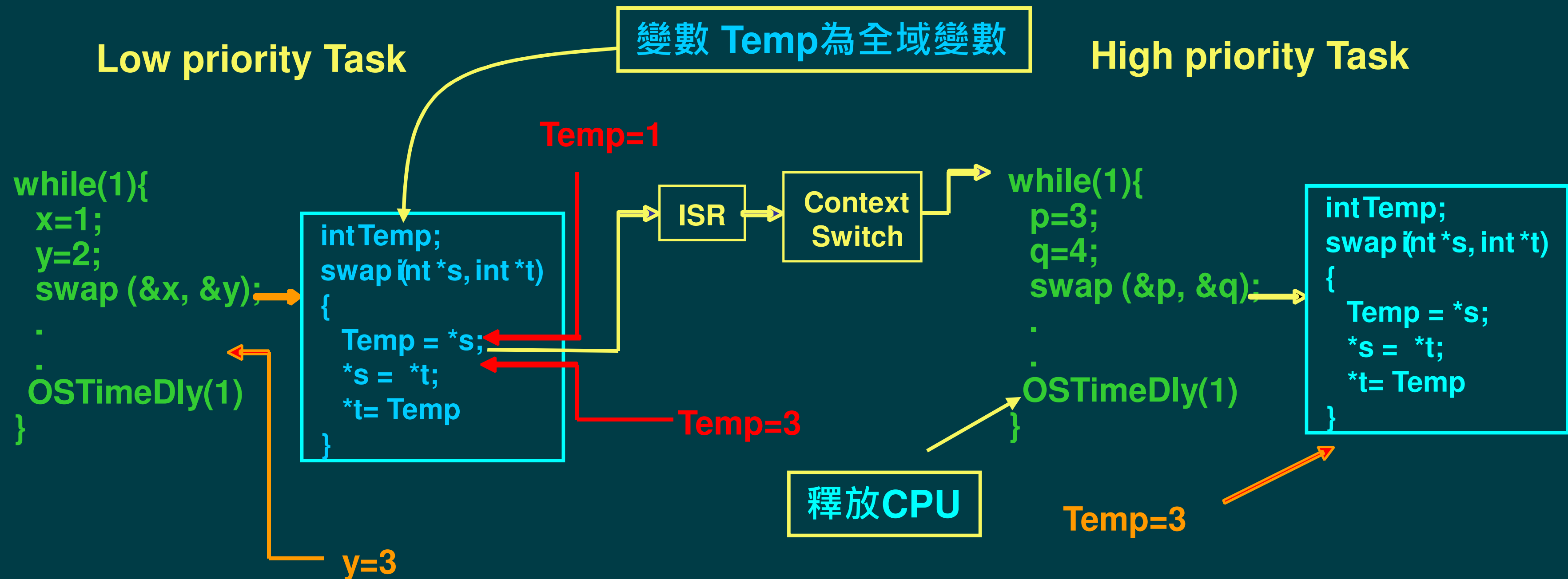
- Synchronous I/O
  - 以blocking function call去讀取I/O資料
  - 若沒有資料，會停在那裡，一直到I/O資料產生時，才會返回繼續執行
  - 資料一產生，Task會立即取得處理
- Asynchronous I/O
  - 呼叫non-blocking function判斷是否有I/O資料 (如同polling)
  - 若無資料，即去做其它事
  - 資料產生即，Task可能在做其它事，無法立即處理I/O資料

Part 3

Non-reentrant function & reentrant function

Non-reentrant function

- RTOS之多工環境，Task的執行是依事件發生的時機而定(event driven)
  - Task執行過程，隨時可能被其它較高優先權的Task所插斷(preemptive)
- Non-reentrant function
  - Function執行結果，受執行過程中是否有被插斷所影響
    - 同時有多個Task在執行此function
  - Example:
    - Task A執行一個function到一半時，被Task B所插斷
    - Task B也執行相同function
    - 當Task B釋放CPU使用權時(context switch)，Task A完成未執行完的function
    - Task A執行function結果受此function執行過程是否被插斷所影響


Example
變數 Temp為全域變數
Low priority Task
High priority Task
Temp=1
while(1){
x=1;
y=2;
swap (&x, &y);
.
.
OSTimeDly(1)
}
int Temp;
swap (int *s, int *t)
{
Temp = *s;
*s = *t;
*t= Temp
}
ISR
Context Switch
while(1){
p=3;
q=4;
swap (&p, &q);
.
.
OSTimeDly(1)
}
int Temp;
swap (int *s, int *t)
{
Temp = *s;
*s = *t;
*t= Temp
}
Temp=3
釋放CPU
Temp=3
y=3

Reentrant function

- 同時有多個Task執行相同function
- 每一Task所執行之function結果不受影響
  - 不管執行過程是否被插斷
- Reentrant function內部不會使用到全域變數(global variable)

Non-reentrant function

全域變數

```
int  Temp;
Swap (int *s, int *t)
{
    Temp = *s;
    *s = *t;
    *t = Temp;
}
```

Reentrant function

區域變數

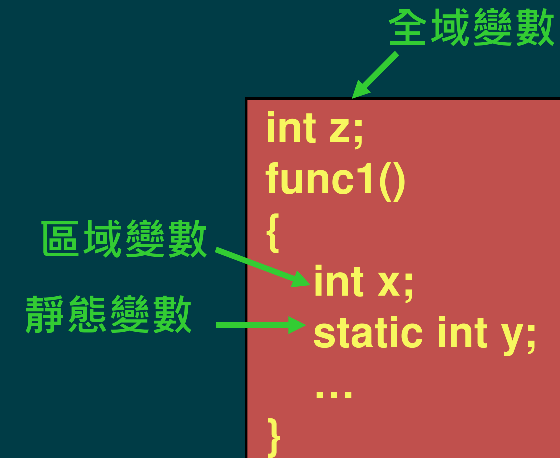
```
Swap (int *s, int *t)
{
    int  Temp;
    Temp = *s;
    *s = *t;
    *t = Temp;
}
```

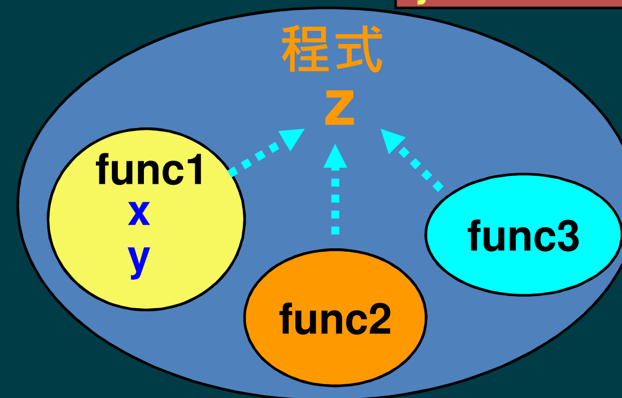
Part 4

C語言之區域/靜態/全域變數

C語言之變數

- C語言程式主要是由函數(function)組成
- 變數依可被使用範圍，可分為
  - 區域變數(local variable)
    - 在函數內宣告
    - 僅在宣告的函數內可使用
    - 函數返回(return)後，變數即消失
  - 靜態變數(static variable)
    - 在函數內宣告
    - 僅在宣告的函數內可使用
    - 函數返回(return)後，變數的值會維持住
  - 全域變數(global variable)
    - 在函數外宣告
    - 任何一個函數皆可使用



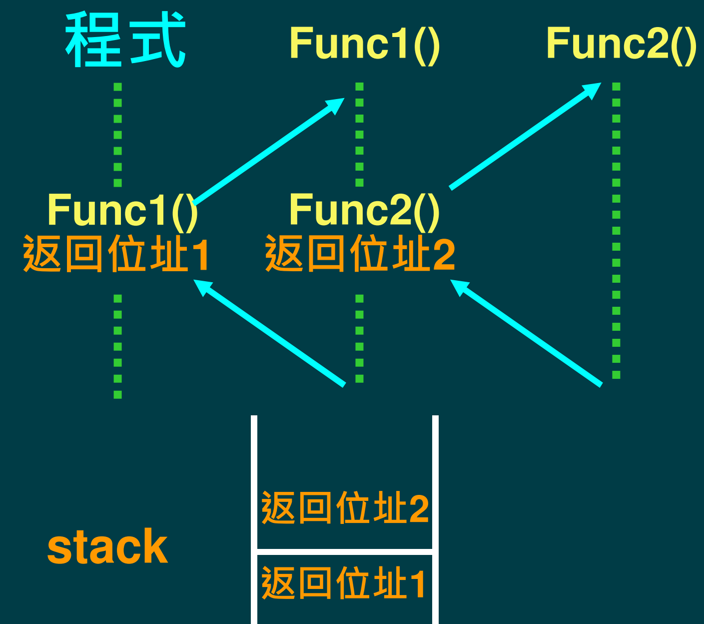


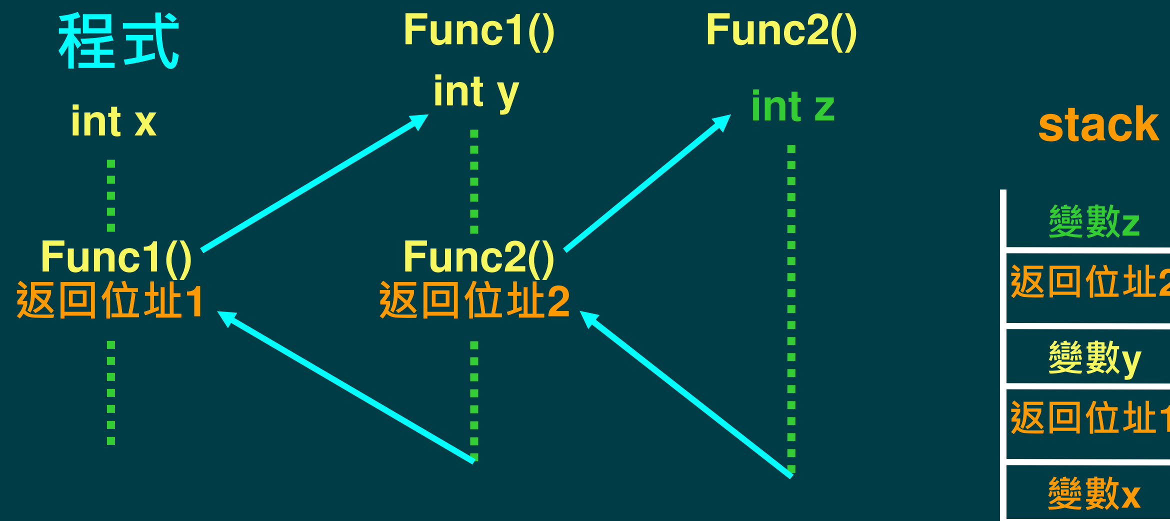
C語言之記憶體分類

- C語言環境下，記憶體可分為三部分
  - Global記憶體
    - 存放全域變數與靜態變數
    - 程式編譯時，即確認變數位址
  - Stack記憶體
    - 存放函數返回位址
    - 存放呼叫函數時所傳遞之參數
    - 存放區域變數
      - 函數被呼叫時才確認變數位址，變數才存在
  - Heap記憶體
    - 動態記憶體管理區域
    - 動態記憶體要求時，所提供之記憶體空間

C語言之函數呼叫

- 程式主要是由多個函數所組成
- 每個程式有其專屬的stack
- 當呼叫函數時，會將返回位址記錄在stack




函數呼叫與區域變數
程式
int x
Func1()
返回位址1
Func1()
int y
Func2()
返回位址2
Func2()
int z
stack
變數z
返回位址2
變數y
返回位址1
變數x

多工環境之Task

- 每個task就是一獨立程式
- 皆有自己的stack
  - 存放返回位址
  - 存放區域變數
  - 存放函數呼叫之參數
- Stack overflow
  - 宣告的區域變數或函數呼叫層數多於stack容量