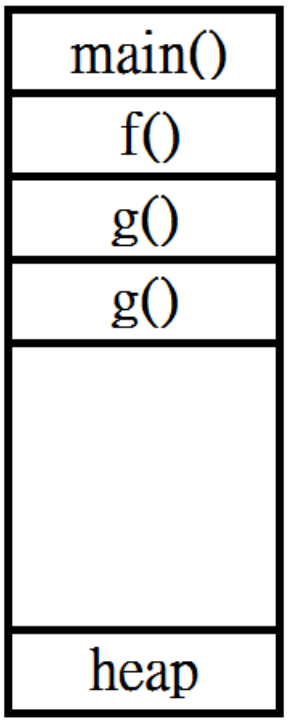
1 記憶體佈局 - 續集

在「記憶體佈局」那份作業中，我們已經簡單看過 stack memory 的運作大方向，也就是每當呼叫函數的時候，我們就會把這個函數的區域變數和一些不是使用者宣告的系統變數加到 stack 頂端。這份作業裡，我們就要來仔細看看實際上被放在 stack memory 的到底有什麼東西。

除非特別說明，不然我們介紹的內容都是以沒有編譯器優化的情況為主。如果你開啟了編譯器的優化選項，那麼有些記憶體佈局可能就會因為執行效能的關係而被改動，不會跟這份作業介紹的完全相同。

1.1 Stack Frame & Calling Convention

複習一下，在「記憶體佈局」的作業中有一張圖：



當時我們說：「每次呼叫函數時，該函數內的變數和一些系統變數就會被存到 stack 的頂端。」在這邊我們更仔細地來說，上半張圖的每個長方形叫作一個 frame（frame 的中文直翻叫作框框），代表那個函數的記憶體使用區，所有的區域變數和一些系統變數都會被存放在這個函數的 frame 裡面。可是，stack 上面有很多函數的 frame，要怎麼知道知道現在執行的函數是哪個 frame 呢？所以，在你的程式執行時，會有兩個系統變數來負責紀錄現在的 frame 在哪裡。請參考下面的 Figure 1。

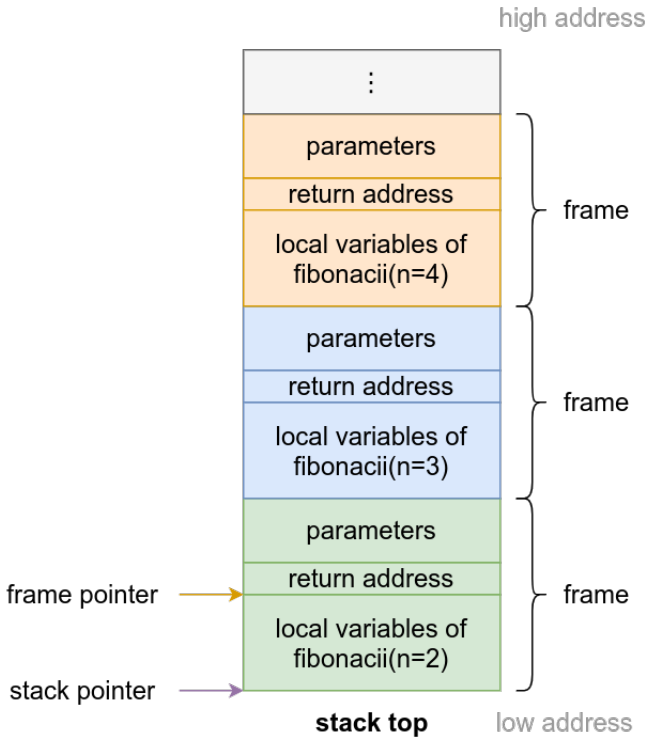


Figure 1: Stack 上不同函數的 frame 與兩個系統變數

其中一個指著 stack top 的系統變數叫作 stack pointer，另一個指著區域變數開始存放的位址的系統變數則叫作 frame pointer。被這兩個指標夾起來的記憶體區間，存放著當前執行的函數的區域變數。要注意的是，一個函數的 frame 還包含著其他的資訊，例如圖中的 return address 和參數（parameters）。

除了 stack pointer 和 frame pointer 這兩個系統變數，還有一個系統變數記錄著當前程式執行到哪一行，這個系統變數叫做 program counter。還記得在「記憶體佈局」的作業中，我們說一隻程式的 text segment 存放著程式指令碼，實際上，program counter 就是一個指標，指著一個 text segment 裡面的一條指令（其實就是接下來要執行的那條指令，如果你好奇的話）。

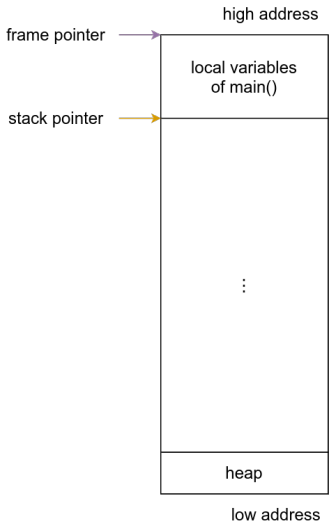
現在，我們有了 stack pointer、frame pointer 和 program counter 這三個系統變數，可以來看點例子了：

```c
#include <stdio.h>

int func (int x, int y) {
    int a = 3;
    int b = 5;
    int c = 7;
    printf("%p %p %p %p %p\n", &x, &y, &a, &b, &c);
    return 0;
}

int main () {
    func(1, 2);
    return 0;
}
```

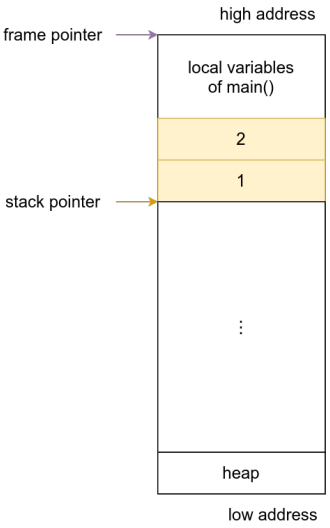
整個程式的執行流程可以簡化成下面的步驟（請搭配下面的附圖服用）：

1. 在 `main` 裡面準備呼叫 `func`
   (a) 把參數 push 到 stack memory 上
   (b) 把 `return 0` 這條指令的位址 push 到 stack memory 上
   (c) 把程式的執行權交給 `func`。換句話說，就是把 program counter 設為 `func` 函數的第一條指令
2. 執行 `func`
   (a) 把舊的 frame pointer 給 push 到 stack memory 上
   (b) 把 frame pointer 設成 stack top
   (c) 移動 stack pointer，在 stack memory 上劃分出一塊記憶體空間可以儲存 `func` 的所有區域變數。在這個例子的話是 12 bytes
   (d) 初始化 `a`, `b`, `c` 的值
   (e) 執行 `printf` （細節略）
   (f) 把回傳值 `0` 放到一個用來存放回傳值的系統變數裡
   (g) 把 stack pointer 設成 frame pointer 的值
   (h) 把 stack memory 上儲存的舊 frame pointer 給 pop 出來，把 pop 出來的這個值存給 frame pointer。換句話說，就是把 frame pointer 還原成舊的 frame pointer
   (i) 把程式執行權交還給 `main`。換句話說，就是把 stack memory 上儲存的 `return 0` 位址給 pop 出來，把 pop 出來的這個值存給 program counter
3. 回去執行 `main` 剩下的東西
   (a) 把 stack memory 上殘留的 `func` 的參數 pop 掉
   (b) 執行 `return 0`（細節略）

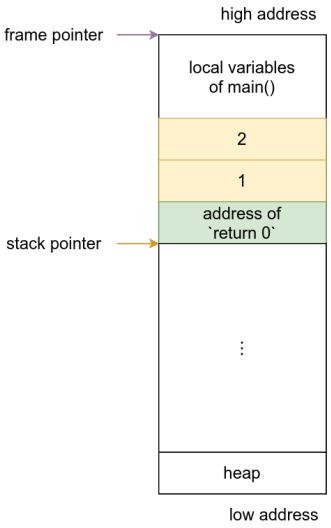
下面的圖是各步驟執行完之後的 stack memory 長相：



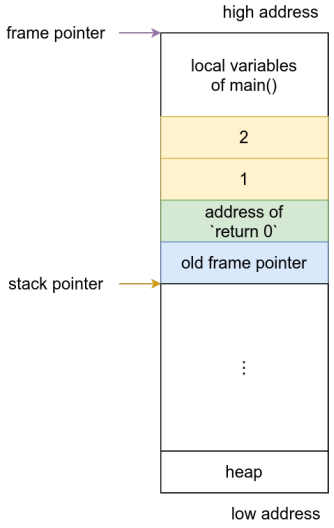
步驟 1



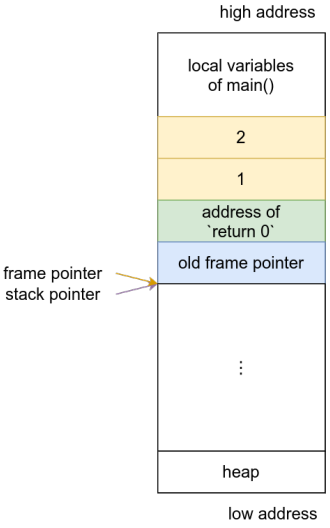
步驟 1.a



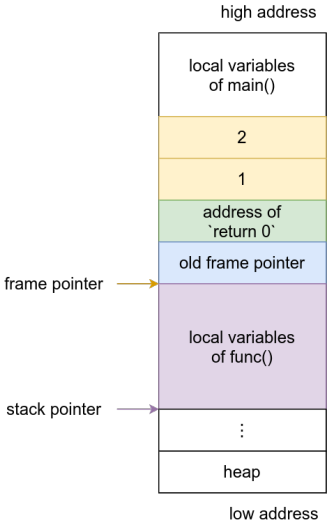
步驟 1.b



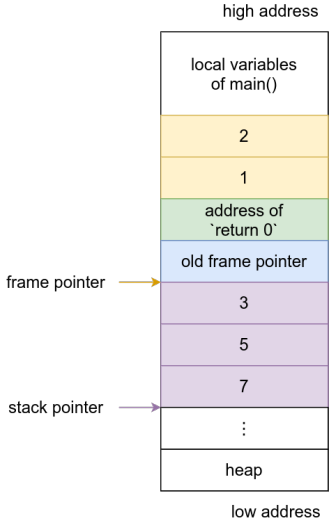
步驟 2.a



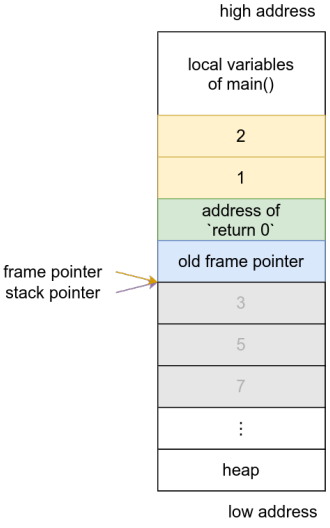
步驟 2.b



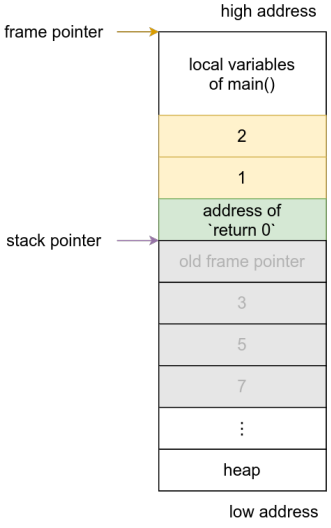
步驟 2.c



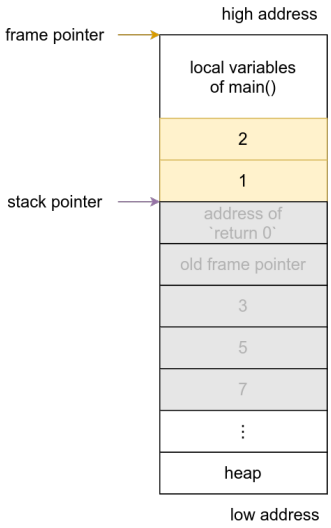
步驟 2.d



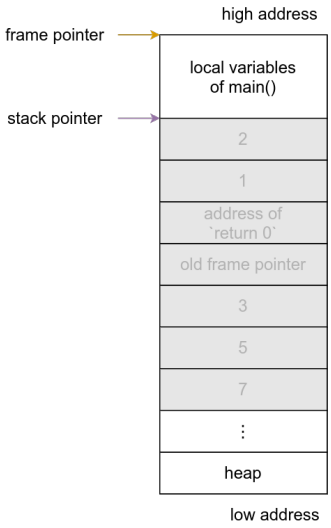
步驟 2.g



步驟 2.h



步驟 2.i



步驟 3.a

看完了以上圖例，我們來觀察底下幾件事：

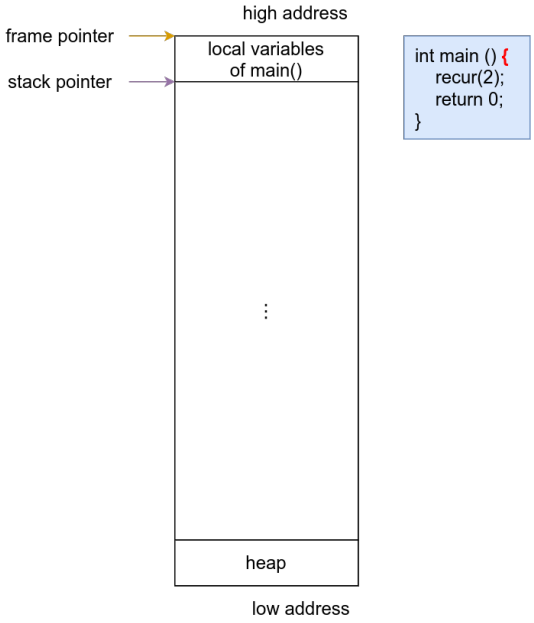
- 比較步驟 1 和步驟 3.a。對 `main` 來說，在呼叫 `func` 前後的記憶體配置是不是長得一樣？
- 觀察上列示意圖，在步驟 2.c 和 2.d 前後，記憶體的配置是不是對稱的？例如 2.b 和 2.g，或是 1.b 和 2.h
- 觀察文字敘述的步驟，是不是只有步驟 2.d、2.e，和 2.c 的區域變數大小會跟我們寫的程式原始碼有關？其他步驟是不是可以直接被套用在任何函數的呼叫？（其實這些可以被重複套用的步驟被統稱為 calling convention，不同函數的 calling convention 都是相同的）

1.2 遞迴函數

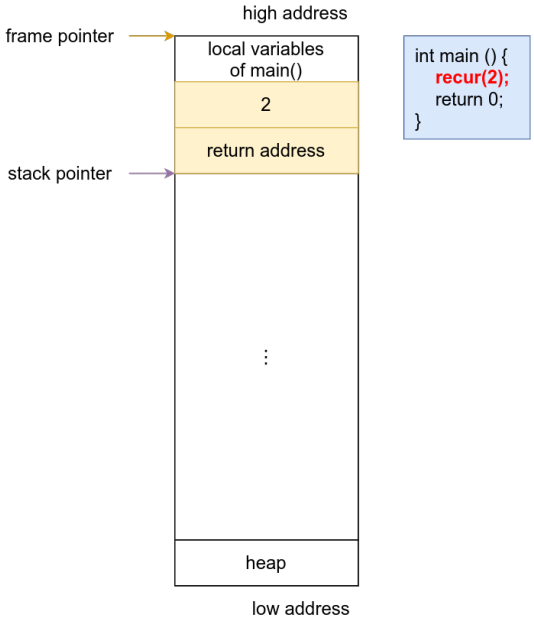
在前一節裡面，我們用了一個簡單的小函數 `func` 來介紹了電腦是怎麼呼叫函數的，現在我們可以來看有趣一點的例子：遞迴函數。考慮底下的程式：

```
1 void recur (int n) {
2     if (n == 0) return;
3     recur(n-1);
4 }
5
6 int main () {
7     recur(2);
8     return 0;
9 }
```

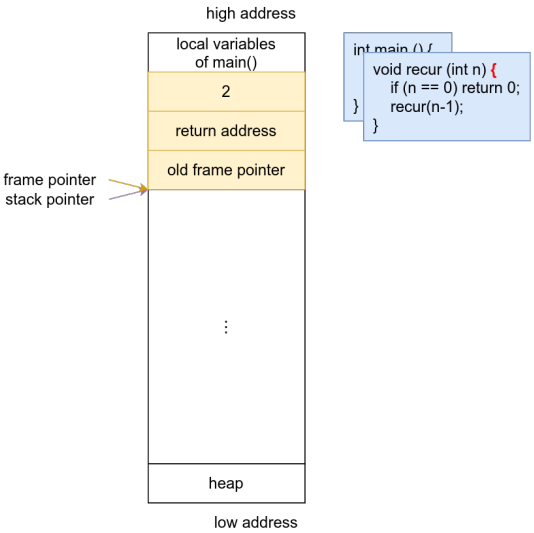
仿造前一節的步驟，我們可以得知 stack memory 在程式執行的不同時間點的樣子。在下面的圖中，紅色粗體的字是程式當前執行到的地方。特別注意到，步驟 3, 5, 7 的 stack pointer 和 frame pointer 都指向 stack top，是因為 `recur` 函數沒有區域變數，所以他們之間不需要有任何的記憶體空間。換句話說，前一節的步驟 2.c 需要新增的記憶體空間是 0 byte。



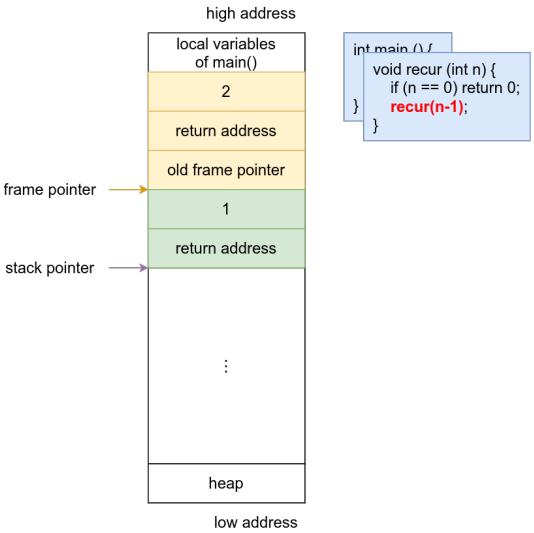
步驟 1



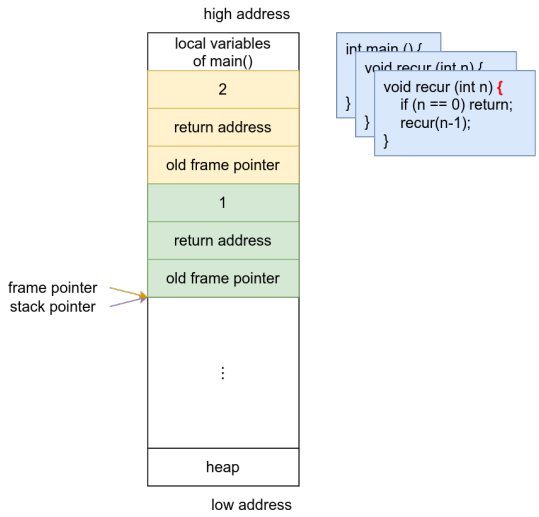
步驟 2



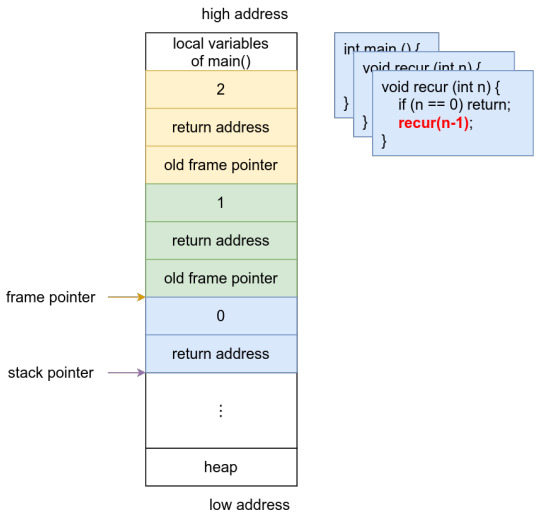
步驟 3



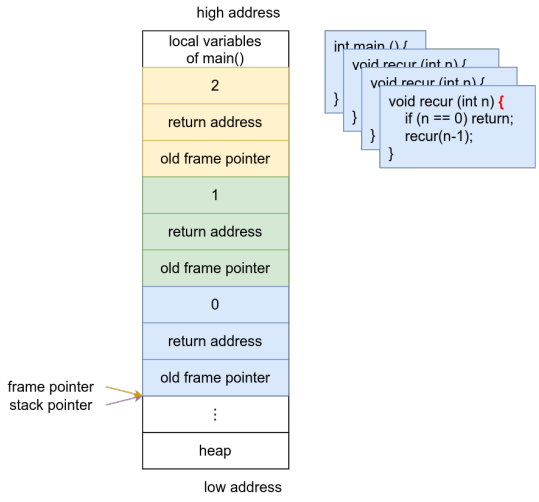
步驟 4



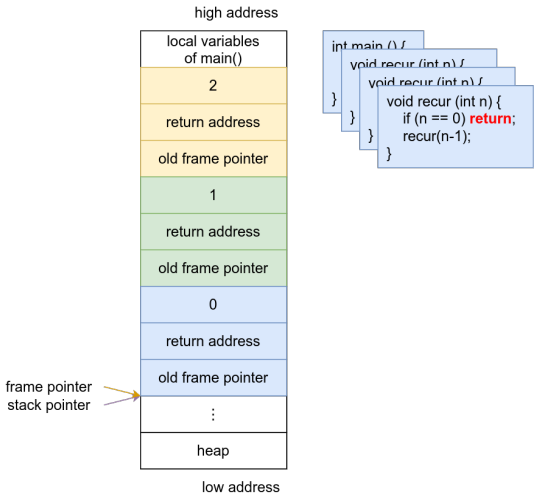
步驟 5



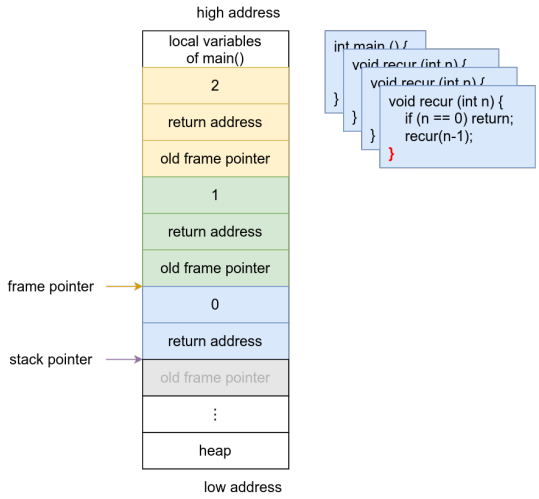
步驟 6



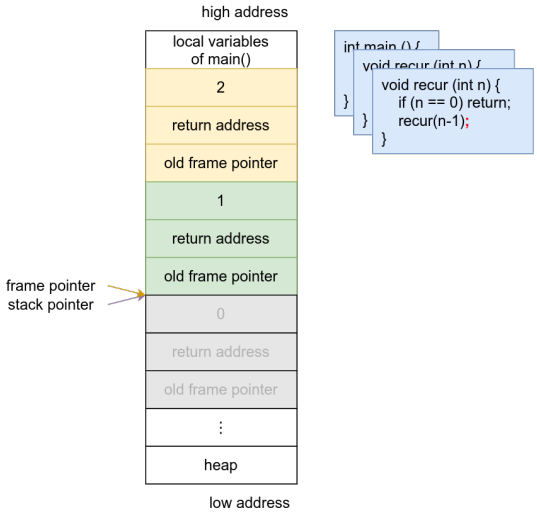
步驟 7



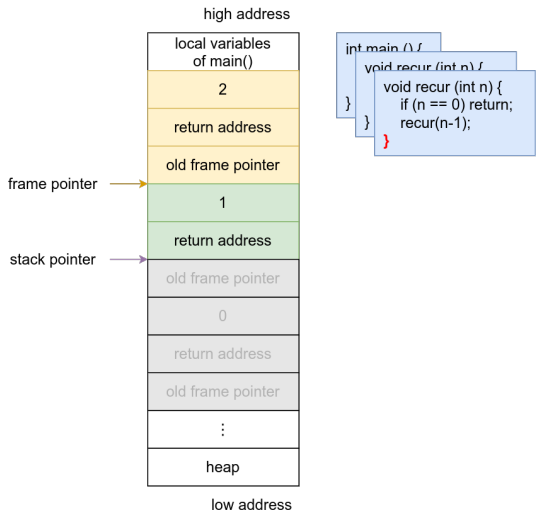
步驟 8



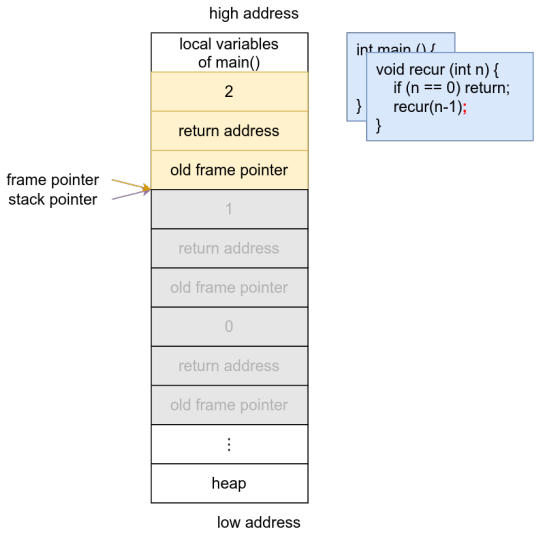
步驟 9



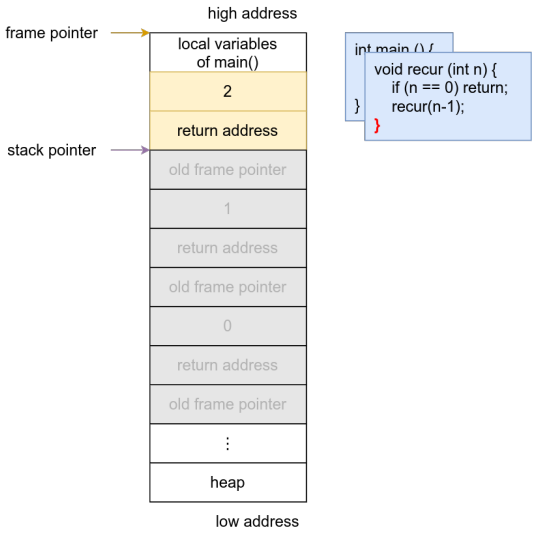
步驟 10



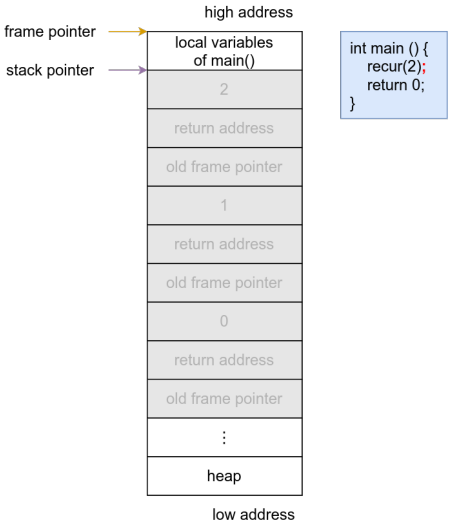
步驟 11



步驟 12



步驟 13



步驟 14

所以，如果你寫了一個會無窮遞迴的函數，stack memory 就會一直被拿來新增新的 frame，直到記憶體用量超過一個門檻（作業系統設的門檻），然後作業系統就會把你的程式砍掉。現在，你終於知道 stack overflow 是什麼意思了。

1.3 後記

這份作業只是提供一個大概念，讓你大致上知道函數是怎麼在電腦裡面被呼叫的，還有遞迴函數是怎麼在電腦裡跑起來的。現實生活中，不同的編譯器的實作細節不同，就算是同一個編譯器，不同的優化參數也會造成編譯出來的執行檔不相同。所以，真正在你電腦上面跑的程式不一定會跟這份作業介紹的機制完全相同，但大方向和概念上都是類似的。

習題

1. (25 pts) 請選出所有錯誤的敘述（複選題）：

   (A) 開發程式時，因為別的函數的變數作用域（scope）跟當前函數的變數作用域不相同，所以如果當前函數把敏感資料存放在區域變數的話，其他函數不會有機會可以存取到。

   (B) 如果在一個函數的區域變數宣告一個很大的陣列，那麼這個函數的 frame 就會很大。

   (C) 對一個宣告在區域變數的陣列 `int a[10]`，如果存取 `a[20]`，有可能會拿到當前函數上一個呼叫的函數的區域變數。

2. 簡答題：

   (a) (13 pts) 請用最多兩句話簡述何謂 function inlining，並用最多兩句話簡述為何該技術可以減少程式執行時間。

   (b) (14 pts) 俗話說：「遞迴只應天上有，凡人應當用迴圈。」然而，同樣的目的，使用迴圈的效能通常會比使用遞迴來得好一點點，請用最多五句話說明為什麼。

   (c) (20 pts) 以下為計算費氏數列的一個遞迴函數。考慮在 `main` 函數呼叫 `fib(4)`，請依照「1.2 遞迴函數」的圖例，畫出第二次呼叫到的 `fib(0)` 在 return 之前的 stack memory layout。不需要標出 stack pointer 和 frame pointer 指向哪裡，但需要把「整個」stack 的內容都寫出來（包含上面圖例中的灰底格子）。

```c
int fib (int n) {
    int sum = 0;
    if (n <= 0) return 0;
    if (n == 1) return 1;
    sum += fib(n-1);
    sum += fib(n-2);
    return sum;
}
```

   (d) (13 pts) 編譯器的優化選項有一個叫作 `omit-frame-pointer`，可以讓編譯出來的程式不會使用到 frame pointer。換句話說，就是只使用 stack pointer。請把「1.1 stack rame & calling convention」章節裡的步驟改成沒有使用到 frame pointer。