

我很棒，快
來學我

你看不到我？
你看的到我？！

107年度教育部扎根高中職資訊科學教育計畫 程式設計基礎課程Topic 6

Kuan-Teng Liao (廖冠登)

2018/10/20

大綱

- 作用範圍(scope)與生命週期(lifetime)
- extern與static
- 練習與作業

作用範圍(scope)與生命週期(lifetime)

- 一般來說，程式裡的變數、函式及所有結構等都有其作用範圍及生命週期
 - ✓ 作用範圍基本上分成
 - 區域型(local)
 - 一般來說是用來探討變數
 - 顧名思義代表其宣告之物只限程式局部區域
 - 通常是針對大括弧、結構，或是函式有關，其有效範圍將各自侷限在大括弧、結構，或是函式中
 - 全域型(global)，在前面沒有亂七八糟的修飾詞下(像是extern const, static)
 - 代表該變數、函式，或是結構等進行宣告，只能被該檔案(不是專案)看到
 - 一般來說，全域型在函式與結構等必定宣告成該檔案類型的全域型，且只能被需要的標頭檔和cpp檔案引入(即#include“某檔案.h”)
 - 全域型的變數通常是在探討全域型特性最容易考的
 - ✓ 生命週期
 - 即為該變數、函式，或是結構的作用時間，意指變數、函式，或是結構在該段期間存活狀態，進而可以進行呼叫或存取

作用範圍(scope)(1)

- 區域性

- 一般來說，探討區域性是用來討論變數之關聯
- 若變數符合滿足於下面條件之一，則稱該變數為區域變數
 - 宣告或宣告並定義於區塊之間(即大括弧，{...})，則稱該變數為區域變數
 - 宣告或宣告並定義於函式內
 - 不會出現在最外層區塊外圍
 - 如下範例中，int_Var即為區域變數

```
1  ...
2  int main(){
3      int int_Var=10;
4      {
5          std::cout<< int_Var <<"\n";
6          {
7              std::cout<< int_Var <<"\n";
8          }
9      }
10     return 0;
11 }
```

作用範圍(scope)(2)

- 區域變數特性：主要與區塊(block)相關
 - 可影響內層區塊，即在內層區塊可以讀到該變數值

```
1 ...  
2 int main(){  
3     int int_Var=10;  
4     {  
5         std::cout<< int_Var <<"\n";  
6         {  
7             std::cout<< int_Var <<"\n";  
8         }  
9     }  
10    return 0;  
11 }
```

- 每個區塊中同名稱變數只能被宣告定義一次

```
1 ...  
2 int main(){  
3     int int_Var=10;  
4     {  
5         std::cout<< int_Var <<"\n";  
6         {  
7             int int_Var=-30;  
8             std::cout<< int_Var <<"\n";  
9         }  
10    }  
11    return 0;  
12 }
```

可以看出Line 3 int_Var被定義宣告一次，Line 7 int_Var又被宣告定義一次;然而這隻程式在編譯時不會出錯，原因在於該同名宣告定義於不同區塊中，使得在編譯時不會出錯。

作用範圍(scope)(3)

- 當區塊內的程式執行結束，同屬該區塊的區域變數將會隨著區塊內的程式進行釋放

```
1  ...
2  int main(){
3      int int_Var=10;
4      {
5          std::cout<< int_Var <<"\n";
6          {
7              int int_Var=-30;
8              std::cout<< int_Var <<"\n";
9          }
10     }
11     return 0;
12 }
```

Line 3的 `int_Var`作用範圍為黃色部分所示(即Lines 3-12);至於Line 7的`int_Var`會因於Line 9讀到區塊終結後會將Lines 7-9，進行指令與變數`int_Var`進行記憶體釋放，因此可得知**Line 7**的`int_Var`作用範圍為綠色部分區塊所示。

- 區域變數亦可作用在不同地方，如函式中的參數亦屬區域變數，其作用範圍為整個函式本身，如下圖

```
1  void fn_PrintVariable(int int_Var=10);
2
3  int main(){
4      ...
5  }
6
7  void fn_PrintVariable(int int_Var){
8      {
9          std::cout<< int_Var <<"\n";
10         {
11             int int_Var=-30;
12             std::cout<< int_Var <<"\n";
13         }
14     }
15 }
```


作用範圍(scope)(4)

- 全域性

- 一般來說，探討全域性及討論變數、函式，或是結構等的宣告，可想而知，該些全域變數、函式，或是結構等只能被該檔案(不是專案)看到
 - 函式與結構等基本上是被引用才能使用/看到(#include 方法)，如之前所提，業界多把該函式與結構等宣告放在標頭檔
 - 容易造成標頭檔或函式庫被循環式的引用(如main.cpp引用A.h與B.h，B.h內又引用了A.h)，而在編譯時造成重複宣告，因此需要用#ifdef xxx #define xxx 函式庫或標頭檔 #endif 或是#pragma once置於各標頭檔中保護宣告以避免該標頭檔被多次引入，產生重複宣告的錯誤
 - 變數的全域性稱為全域變數，其可宣告或宣告並定義放置的檔案有
 - 可以放在(1)main.cpp中或(2)各種.cpp檔案中(最好不要使用後者方案，當專案越做越大變數散在各處，若要修改會變得很xx的亂)
 - 若放全域變數放在標頭檔內要有計畫，不能隨便亂塞在各個標頭檔
 - 若亂塞，會發生重複定義問題

[p8](#)
 - 可以請指定一個獨立的標頭檔，將其放入，所有cpp中只有由main.cpp引入，其它標頭不能引入該獨立標頭檔

作用範圍(scope)(4)

- 重複定義問題

- 回顧Topic 2變數部分，當時我們為了讓大家更好了解
 - `int int_Var; //稱為宣告`
 - `int_Var=1; //稱為定義`
- 該概念也被廣泛應用在之後的函式與結構等，如將含式分開寫需要**先宣告**之後**再進行定義**。
- 但嚴謹來說，C/C++只有在變數議題上宣告和定義上與函式與結構等有些微的不同
 - 一般來說，宣告只是告訴編譯器未來會有這東西，而定義是真正將這東西配置記憶體
 - C/C++的變數則是在變數上進行宣告時，就順手配置了記憶體。**因此嚴謹的來說，C/C++宣告時就同時默默進行了定義。然而為了加深同學之後在其他部分印象，因此Topic 2就先利用該概念說明宣告與定義之不同

作用範圍(scope)(4)

- 全域變數必出現一般來說出現於main()外面或是函式外面(建議不要把全域變數放的到處都是的，最後撰寫會變得很混亂)
 - 全域變數如下圖所示， gblint_Var為一全域變數，即黃色部分

```
1  int gblint_Var =10;
2  int main(){
3      ...
4  }
```

- 若其他函式、結構等同時宣告並定義於與全域變數相同的檔案，皆可利用在區塊內使用該全域變數(練習6(1)，於[p10](#))
- 全域變數可以被其他檔案中有使用extern修飾字之同名宣告，使得該全域變數可以與其他檔案分享其全域變數，使得全域變數可以在不同檔案中進行存取，該部分會於後面介紹extern修飾詞進行討論

練習6(1)

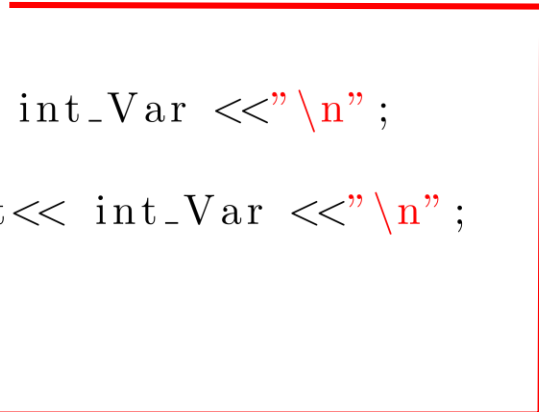
- 請將下圖程式於main.cpp中完成，並加上 `fn_ShowGobalVarInFunction` 函式，該函式無回傳及傳入參數，並於函式區塊內敘述執行 `std::cout<< gblint_Var << std::endl;` 並於主程式中呼叫，看會該全域變數是否能被函式內部所使用

```
1 int gblint_Var =10;
2 int main(){
3     ...
4 }
```

生命週期(lifetime)(1)

- 即為該物件有效的作用時間，意指該物件在該段期間將為存活狀態，進而可以進行呼叫及存取
- 整體來說**生命週期多數(真的只有多數)**皆與**作用範圍相關**，如以單純區域變數為例，其作用範圍為該區塊(block)並向下支援，且其該變數生命週期為從區塊開始到結束，如下圖所示

```
1  ...
2  int main(){
3      int int_Var=10;
4      {
5          std::cout<< int_Var <<"\n";
6          {
7              std::cout<< int_Var <<"\n";
8          }
9      }
10     return 0;
11 }
```



int_Var 存活的時間

生命週期(lifetime)(2)

- 以函數中的變數為例，其生命週期也將與函式被呼叫時執行到該變數才開始，以函式結束後死亡

```
1 void fn_PrintVariable(int int_Var=10);  
2  
3 int main(){  
4     ...  
5 }  
6  
7 void fn_PrintVariable(int int_Var){  
8     {  
9         std::cout<< int_Var <<"\n";  
10        {  
11            int int_Var=-30;  
12            std::cout<< int_Var <<"\n";  
13        }  
14    }  
15 }
```

參數的int_Var 存活的時間

外部宣告鏈結(extern)(1)

- extern這關鍵字基本上是衝著作用範圍之**全域性**的分享而來
 - ✓由於全域性可視範圍為單一檔案，若要在其它檔案使用單一檔案的全域變數/函式，可以在其他檔案使用extern關鍵字將全域變數進行共享
 - ✓以變數為例

```
main.cpp
1 #include <iostream>
2 #include "fn_GetExternVariable.h"
3
4 int gblint_Var= -1;
5
6 int main(){
7     fn_GetExternVariable();
8     return 0;
9 }
```

```
fn_GetExternVariable.h
1 #pragma once
2 #include <iostream>
3
4 extern int gblint_Var;
5 void fn_GetExternVariable();
```

```
fn_GetExternVariable.cpp
1 #include "fn_GetExternVariable.h"
2
3 void fn_GetExternVariable(){
4     std::cout << gblint_Var<< std::endl;
5 }
```



```
C:\Users\Nick\Documents\Workshop\Test\bin\Debug\Test.exe
-1
Process returned 0 (0x0)   execution time : 0.222 s
Press any key to continue.
```

外部宣告鏈結(extern)(2)

✓以函式為例，當然也一定可以啊！不過表演當然要讓學生印象深刻，所以

- 請參照作業6(1)的[p18](#)與[p19](#)

靜態(static)(1)

- static這關鍵字用在類別(class)與不用在類別意義不同

- ✓用在類別內(C++專有)

- 進行宣告並定義的靜態變數和函式是不論該類別生成多少個物件(object)，被宣告成靜態的變數和函數都只會共享有一份變數、函式(透過靜態方式呼叫，和物件本身無關與類別有關，之後會再教)

- ✓不用在類別內(C/C++共有)，如用在函式中的變數

- 基本上是衝著**生命週期**而來

- 由於函式中的區域變數生命週期，主要隨著呼叫時該變數宣告並定義與執行結束兩者區段，執行使用與釋放

- 但在函式中宣告靜態成靜態的變數，它會將該變數放在全域資料區域(global data area)中，且**只有再第一次進入函式內會被宣告及定義**，但該靜態變數，會一直保留到整個程式執行結束(基本上就是main結束)

- **原理是它是靜態的，但它會被宣告的作用範圍給局限住(很重要，請念100遍)**
 - 現象是意味著每次呼叫該函式，它都會保持著上一次的結果

靜態(static)(2)

- 靜態變數範例

main.cpp

```
1 #include <iostream>
2
3 void fn_CallStaticVar ();
4
5 int main(){
6     fn_CallStaticVar ();
7     fn_CallStaticVar ();
8     fn_CallStaticVar ();
9     fn_CallStaticVar ();
10    return 0;
11 }
12
13 void fn_CallStaticVar(){
14     static int int_Var=10;
15     std::cout<< int_Var << "\n";
16     int_Var=int_Var+2;
17 }
```



```
C:\Users\Nick\Documents\Workshop\Test\bin\Debug\Test.exe
10
12
14
16
Process returned 0 (0x0)   execution time : 0.061 s
Press any key to continue.
```

靜態(static)(3)

✓不用在類別內(C/C++共有)，如static用在全域變數上

- 思考方向：

- 全域變數代表於該檔案作用範圍為整個檔案，並且可以利用extern修飾詞分配給其他.cpp檔案使用。

- static特性原理是它原理是它是靜態的，但它會被宣告的作用範圍給侷限住，並且其現象是意味著每次呼叫該函式，它都會保持著上一次的結果

- 因此考慮結合上面兩者特性，如宣告全域變數為靜態的，意指它會被侷限在宣告並定義它的檔案內，且不能被其他檔案extern，若有其他檔案對其使用extern，則會出現錯誤

作業6(1)(2)

- 紙本作業，請使用 **外部鏈結方式** 進行串接Topic 5的p46、p47及p48頁的程式，並將結果印出

✓提示：

- 該程式原始方式使用#include“標頭檔”如下圖程式碼，請用extern改寫使其main.cpp與GetSum.h中個別使用extern 函式進行外部鏈結，個別原始程式碼如下所示，而圖解關聯於[p19](#)呈現

```
main.cpp
1 #include <iostream>
2 #include "fn_GetSum.h"
3
4 int main(){
5     int int_Var= 10, int_Var2= -5;
6     fn_GetSum(10, -5);
7     return 0;
8 }
```

```
fn_GetSum.h
1 #ifndef FN.GETSUM.HINCLUDED
2 #define FN.GETSUM.HINCLUDED
3
4 #include <iostream>
5 #include "fn_GetSumR.h"
6 void fn_GetSum(int int_X, int int_Y);
7
8
9 #endif // FN.GETSUM.HINCLUDED
```

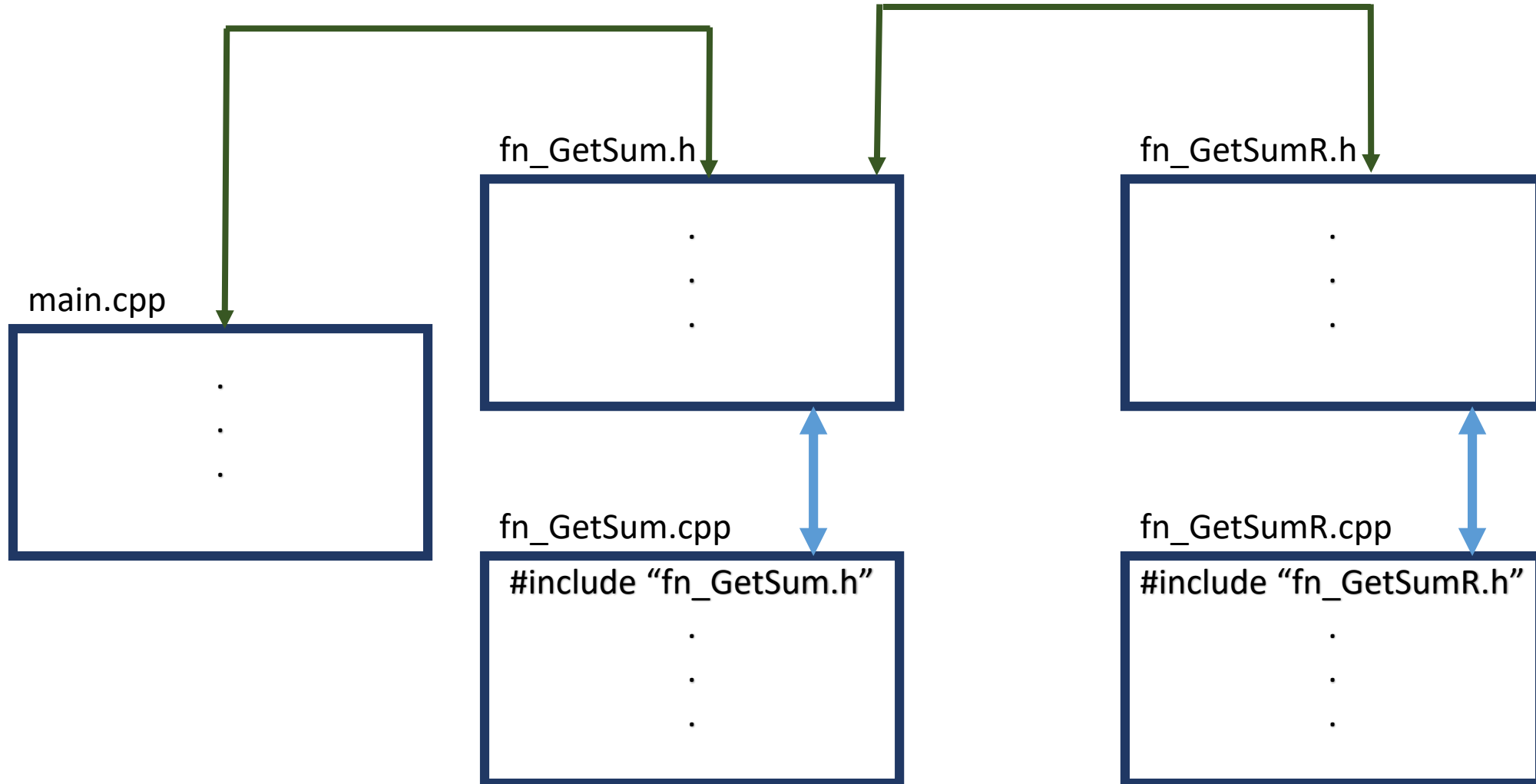
```
fn_GetSum.cpp
1 #include "fn_GetSum.h"
2
3 void fn_GetSum(int int_X, int int_Y){
4     std::cout<< (int_X+int_Y)<< std::endl;
5     int int_Result= fn_GetSumR(int_X, int_Y);
6     std::cout<< "The sum from fn_GetSum calling fn_GetSumR "
7     << int_Result<< std::endl;
8 }
```

```
fn_GetSumR.h
1 #ifndef FN.GETSUMR.HINCLUDED
2 #define FN.GETSUMR.HINCLUDED
3
4 #include <iostream>
5 int fn_GetSumR(int int_Var, int int_Var2);
6
7 #endif // FN.GETSUMR.HINCLUDED
```

```
fn_GetSumR.cpp
1 #include "fn_GetSumR.h"
2
3 int fn_GetSumR(int int_Var, int int_Var2){
4     // declare a variable and then initialize the variable
5     int int_Temp = -1;
6     int_Temp=(int_Var+int_Var2);
7     return int_Temp;
8 }
```

作業6(1)(2)

- 於編譯時期圖解



藍色箭頭作用在於編譯專案時進行的繫結。綠色箭頭為使用者利用extern宣告進行外部鏈結之意