```
回顧 - 逐式

• 函式 - 用來將程式獨立成為小單元、模組

• 標準函式庫函式 - 提供 C/C++認為基本、必要的功能 - 輸入輸出、數學運算、...

• 自訂函式 (user-defined functions)

• 函式的目的主要主於簡化程式發展的難度、並使多人可容易可以互相合作以發展大型程式

• 各別擊破法 (divide and concur)
```

```
回顧 - 函式
   • 由於函式之間互相為獨立的單元,因此:
     。 函式定義時可以互相呼叫,但是任何一個函式定義不會把別的函式
      定義包起來
     。 函式定義時所使用的變數大部份是 local 的,亦即函式內所使用的
變數互相獨立而不互相干擾 。
     。一個函式要呼叫任何其它函式之前,至少要先有它的原型宣告被定
      義出來。
                        void funA(int a, double b) {
void funA(int, double);
void funB(void);
                          int c;
double d;
                                                  這些區
塊出現
int main() {
                                                  在同一
個檔案
   funA(3, 5.0);
                       void funB() {
   funB();
                           double c:
                                                  裡哦!
                           int d;
```

補充

• 目前為止,所有的函式原型與函式定義都請大家寫在一個原始程式碼裡 (source code)。而真正大的程式開發:
。 原型宣告被放在標頭檔 (header file) 裡,通常為副檔名為.h,或沒有副檔名 - e.g. iostream, cstdlib, cmath, ...。用到的程式把它含括進來 (#include ...)

• 而函式定義被放在

• 分開的原始程式檔裡

• 函式庫檔案裡 (library file)

• 例如大家作業使用的堆土機專案 ...

```
回顧 - 指標
• 為一特別的資料型別,代表一個變數存放的資料是記憶體位址,用來
 記錄一個資料存在電腦裡的位置(印出時為16進位數字)。
• 宣告:
  • int A=3; double B=3.14; float D=1.732;
  • int *ptrA; double *ptrB, *ptrC; float *ptrD, ***pptrD, ***ppptrD;
   給指標變數值 – 使用取址運算子 &∶ptrA = &A; ptrB = &B;
   ptrC=ptrB; ptrD=&D; pptrD = &ptrD; ppptrD = &pptrD;
   // 印出 0x7000fe24: 3
                                 // 一樣印出 3.14
   cout << *ptrD;
                                 // 印出 1.732
   cout << *pptrD << ": " << **pptrD;
cout << ***ppptrD;
                                 // 印出 0x7000a000: 1.732
                                // 印出 1.732
```

```
| D| 順 - 傳址呼叫 (以傳遞陣列為例)

#include <iostream>
using namespace std;
void printArray(int n, int *a);
int main() {
  int A[5] = {5,4,3,2,1};

  printArray(5, A);
  printArray(3, A);
  printArray(2, &A[2]);

  return 0;
}

| D| 随 - 傳址呼叫 (以傳遞陣列為例)

| 5 4 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 4 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3 3 2 1
| 5 3 3
```

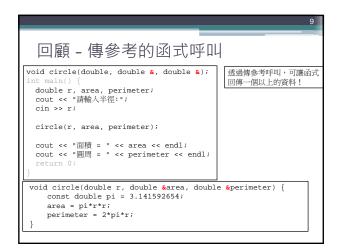
```
● 也是一特別的資料型別,代表一個變數為另一個變數的分身、別名。
● 宣告:

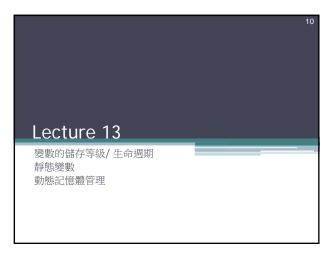
● int A=3; double B=3.14; float D=1.732;

● int &secretA=A; double &agentB = B; float &aliasD=D;

● 使用上和一般變數沒有兩樣!

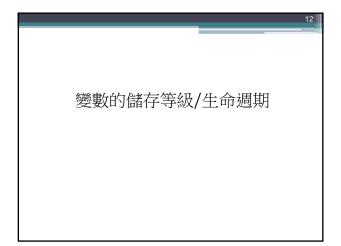
int a = 3;
int &b = a;
cout << a << endl;
b = 4;
cout << a << endl;
b = 4;
cout << a << endl;
```





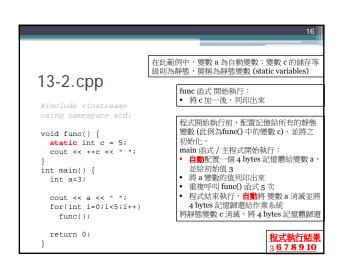
今日內容

· 變數的儲存等級/生命週期
。自動變數、靜態變數、動態記憶體管理
· 靜態變數 (static) 的使用
· 動態記憶體管理
。 記憶體配置 (memory allocation)
。 判斷記憶體配置的成功與否
。 記憶體釋放 (memory de-allocation/release)
。 多維陣列的記憶體管理





靜態變數 (static) 的使用



一种態變數在程式載入開始執行**前**,即已被創造出來並佔據記憶體

- 雖然靜態變數宣告在函式內,並不會隨著函式開始執行而被重新創造出來,也不會因為函式結束執行而被消滅掉。

- 由於變數的初始化 (int c=5;) 是在變數被創造出來佔據記憶體時所執行的動作,因此在函式被重複呼叫時,變數的初始化是不會再被執行的。

- 靜態變數若沒有初始化時,會被初始化為 o。

動態記憶體管理

Dynamic Memory Management

```
13-3.cpp

#include <iostream>
using namespace std;

int main() {
    int a=3;
    int *b = new int(4);

    cout << a << ", " << *b;

    delete b;
    return 0;
}
```

```
13-3a.cpp
#include <iostream>
using namespace std;

int main() {
    int a=3;
    int *b = new int;
    *b = 4;

    cout << a << ", " << *b;

    delete b;
    return 0;
}
```

13-3.cpp

- 在之前使用指標變數時,都是用別的變數的儲存空間來儲存資料。int a;
int *b = &a; *b = 3; // 將 3 存入 a 變數的儲存空間內

- 使用動態記憶體配置 (new),指標變數可以取得一個獨立的資料儲存空間,不會將資料存到其到變數的儲存空間。
。 int *b = new int(3); // 配置一整數空間,並初始其資料為 3。

- 動態配置出來的儲存空間必需歸還給作業系統,一個 new 一定會有一個相對應的 delete 才正確。
。 delete b;



動態 陣列的配置

• 陣列 (Array)

• 用來儲存大量相同型別的資料

· int a[100];

· double b[40];

• 但陣列在宣告時即必需知道大小,且大小必需是常數

• 且陣列大小一經宣告即無法改變

• 有時,所需要的陣列大小無法預知。

• 透過動態記憶體管理,我們可依程式與處理資料量的需求,適當地配置記憶體空間。

```
13-4.cpp

for(i=0;i<size;i++) {
    cout << "請輸入第" << i+1 << "個數字:";
    cin >> myData[i];
}

for(i=size-1;i>=0;i--) {
    cout << myData[i] << ", ";
}

delete []myData;

return 0;
}
```

```
new 運算子的語法二:配置陣列

- 用來配置(allocate) —塊記憶體
- 配置到的記憶體應該要用 delete 運算子釋放掉(release, de-allocate)。

- 語法二:指標變數 = new 變數型別[陣列大小];
- e.g. myData = new int[100];
- 動態地向作業系統要一可容納 100 個整數的記憶體空間。
- 配置完成後即可將指標變數當成一般的一維陣列進行操作。
- 無法指定初始值!
```





記憶體配置的成功與失敗

· 記憶體配置的成功與失敗

· 說憶體配置的成功與失敗

· 成功時,代表程式已成功地向作業系統從 heap memory 中配置一塊 記憶體資源供程式應用

。 heap 記憶體獨立於程式運行之記憶體空間,可取得的大小遠多於程式內宣告的陣列 (使用 stack memory)

· 在 C++ 裡,有兩種方式處理記憶體配置失敗
。 (nothrow) 參數 → 13-5.cpp
。 例外處理 (Exception handling) → 13-6.cpp (僅供參考...)

```
13-5.cpp

#include <iostream>
using namespace std;

int main() {
    char *mem;
    unsigned long size, i;

    cout << "請問你要配置幾 MB 的記憶體?";
    cin >> size;

size *= 1024 * 1024;
    cout << "\n配置" << size << " bytes 的記憶體";
```

```
13-5.cpp

mem = new (nothrow) char[size];
if(mem==0) {
    cout << "記憶體配置失敗,程式結束!";
    return 255;
}

for(i=0;i<size;i++) {
    mem[i] = (char) (i*6);
    }
    cout << "輸入任意資料後按 Enter 结束";
    cin >> mem;
    delete []mem;
    return 0;
}
```

```
13-5.cpp 的說明

• mem = new (nothrow) char[size];

• 透過讓程式執行時,若記憶體配置失敗時,不要抛出例外狀況 (exception)。

• 若記憶體配置失敗且程式不允許抛出例外狀況時,會回傳一個 o pointer (null pointer),null pointer 其值為 o。
```

```
13-6.cpp

#include <iostream>
using namespace std;

int main() {
   char *mem;
   unsigned long size, i;

try {
   cout << "請問你要配置幾 MB 的記憶體?";
   cin >> size;

   size *= 1024 * 1024;
   cout << "\n配置" << size << " bytes 的記憶體";
```

```
13-6.cpp

mem = new char[size];
for(i=0;i<size;i++) {
    mem[i] = i;
}
cout << "按Enter 结束";
cin >> mem;
delete []mem;
return 0;
}
catch(bad_alloc&) {
    cout << "記憶體配置失敗,程式結束!";
    return 255;
}
}
```

```
13-6.cpp 的說明

• 此範例利用 C++ 的例外處理的機制來處理記憶體配置的錯誤。

• 例外處理是 C++ 新加入的功能,用來將所有程式錯誤狀況 (e.g. 除以0) 當成一個一個的例外狀況 (exception),在正常程式流程的最後再一起加以處理。

• 例外處理機制可使正常程式與錯誤處理程式分開撰寫而不會交織在一起,使程式更容易理解與閱讀。

• 目前還沒有正式學到例外處理,以後請自修 ... : p

• 當記憶體配置失敗時,程式會跳躍到 catch (bad_alloc&) 的區塊執行,印出記憶體配置失敗的訊息。
```

```
適要
動態記憶體管理

new ← → delete
a = new int; ← → delete a;
new ...[] ← → delete []...
a = new int[100]; ← → delete []a;

記憶體配置失敗

a = new (nothrow) int; ← → if(a==0) 表失敗
例外處理
```





```
請輸入你要輸入的整數資料個數: 10
請輸入第 1 個資料: 1
請輸入第 2 個資料: 2
請輸入第 3 個資料: 4
請輸入第 6 個資料: 5
請輸入第 6 個資料: 7
請輸入第 6 個資料: 10
請輸入第 7 個資料: 10
請輸入第 9 個資料: 12
請輸入第 9 個資料: 14
請輸入第 10 個資料: 15
你輸入的資料為: 12 4 5 7 9 10 12 14 15
其中共有 5 個偶數。他們為: 2 4 10 12 14
```