Appendix J

問題回顧解答

第 2 章

1. 函數
2. 在編譯之前，此指令被換成 iostream 檔案的內容。
3. 它使程式可以使用 std 名稱空間中的定義。
4. `cout << "Hello, world\n";`

   或是

   `cout << "Hello, world" << endl;`
5. `int cheese;`
6. `cheese = 32;`
7. `cin >> cheese;`
8. `cout << "We have " << cheese << " varieties of cheese\n";`
9. 這說明函數 froop() 被呼叫時，將有一個 double 型態的引數，而其回傳型態為 int。舉例來說，它可以下列的方式使用之：

   `int gval = froop(3.14159);`

   函數 rattle() 不具有回傳值，但需要一個 int 引數。舉例來說，它可以這樣來使用：

   `rattle(37);`

函數 prune() 會回傳一個 int，並且不需要使用引數。舉例來說，它可以這樣來使用：

```
int residue = prune();
```

10. 當函數使用 void 回傳型態，表示函數內不必使用 return 敘述。但是若你不提供回傳值，你可以使用它：

```
return;
```

11. 將會出現 cout 未定義。有下列方法解決，先將下一行敘述匯入

```
#include <iostream>
```

然後在將

```
using namespace std;
```

置於 main()函數外或函數內，或是省略上一敘述，改以

```
using std::cout;
```

也是可以的。

第 3 章

1. 因為有多種整數型態可以選擇，所以你可以選擇最適合特定需求的型態。例如，short 可以節省空間，long 可以保證容量大小，或是找到特定型態可以加速某種計算。

2.
```
short rbis = 80;            //or short int rbis = 80;
unsigned int q = 42110;     //or unsigned q = 42110;
unsigned long ants = 3000000000;
```

註釋：int 不夠大不能儲存 3,000,000,000。若您的系統有支援通用串列初值設定，則也可以使用如下的敘述：

```
short rbis = 80;            // = is optional;
unsigned int q {42110};     // could use = {42110}
unsigned long ants {3000000000};
```

3. C++ 沒有自動的防護方法，可以避免超出整數的限制，你可以用 climits 標頭檔決定此限制為何。

4. 常數 33L 為 long 型態，常數 33 為 int 型態。

5. 雖然這兩個敘述在某些系統上結果相同，但它們並不完全相等。重點是，第一個敘述只有在系統使用 ASCII 碼時，才會指定字母 A 給 grade；但第二個敘述無論系統採用何種內碼均適用。其次，65 為 int 常數，而 'A' 為 char 常數。

6. 以下是 4 種方式：

```
char c = 88;
cout << c << endl;            // char type prints as character

cout.put(char(88));           // put() prints char as character

cout << char(88) << endl;     // new-style type cast value to char

cout << (char)88 << endl;     // old-style type cast value to char
```

7. 這答案得視兩種型態有多大，如果 long 佔 4 個位元組，則資料不會遺失。原因是 long 最大值約達 20 億，為 10 位數，而 double 至少有 13 位數，所以沒有四捨五入的問題。另一方面，long long 可以達到 19 位數，超過 double 的 13 位有效位數。

8. a. 8 * 9 + 2 為 72 + 2 等於 74

   b. 6 * 3 / 4 為 18 / 4 等於 4

   c. 3 / 4 * 6 為 0 * 6 等於 0

   d. 6.0 * 3 / 4 為 18.0 / 4 等於 4.5

   e. 15 % 4 等於 3

9. 下列二個方法均可行：

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

   To add them as type double and then convert, you could do either of the following:

```
int pos = (int) (x1 + x2);
int pos = int(x1 + x2);
```

10. a. int

    b. float

    c. char

    d. char32_t

    e. double

第 4 章

1. a. char actors[30];

   b. short betsie[100];

c. `float chuck[13];`

d. `long double dipsea[64];`

2. a. `array<char, 30> actors;`

b. `array<short, 100> betsie;`

c. `array<float, 13> chuck;`

d. `array<long double, 64> dipsea;`

3. `int oddly[5] = {1, 3, 5, 7, 9};`

4. `int even = oddly[0] + oddly[4];`

5. `cout << idea[1] << "\n";  //or << endl;`

6. `char lunch[13] = "cheeseburger"; // number of character + 1`

或

```
char lunch[] = "cheeseburger";  // let the complier count elements
```

7. `string lunch = ”Waldorf Salad”;`

若沒有使用 using 指令，

```
std::string lunch = ”Waldorf Salad”;
```

8.
```
struct fish {
   char kind[20];
   int weight;
   float length;
};
```

9.
```
fish petes =
{
    "trout",
    12,
    26.25
};
```

10. `enum Response{No, Yes, Maybe};`

11.
```
double * pd = &ted;
cout << *pd << "\n";
```

12.
```
float * pf = treacle;   // or = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
                          // or use *pf and *(pf + 9)
```

13. 假設 iostream 與 vector 標頭檔已匯入，並且使用 using 指令

```
unsigned int size;
cout << "Enter a positive integer: ";
cin >> size;
```

```
int * dyn = new int [size];
vector<int> dv(size);
```

14. 正確。運算式 "Home of the jolly bytes" 為字串常數，以字串的起始位來運作。cout 物件原先會將 char 位址，解釋為印出字串，但因為型態轉換 (int *)，將位址轉換成 int 指標。因此，此敘述印出字串的位址。

15.
```
struct fish
{
    char kind[20];
    int weight;
    float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

16. 用 cin >> address 使程式跳過正常空白，直到找到非正常空白值。接著讀取字元，直到再度碰到正常空白。此法會跳過數值輸入後面的換行字元。另一方面，這種作法只讀取一個單字，並非一整行。

17.
```
#include <string>
#include <vector>
#include <array>
cons tint Str_num[10]; // or = 10
…
std::vector<std::string> vstr(Str_num);
std::array<std::string>, Str_num> astr;
```

第 5 章

1. 條件進入迴圈（entry-condition loop）是於程式進入迴圈主體之前，先測試運算式。如果條件為 false，則程式不會進入迴圈主體。條件離開迴圈（exit-condition loop）則於執行迴圈主體後，測試運算式。如果條件最初為 false，則迴圈主體仍會執行一次。for 和 while 迴圈屬於條件進入的迴圈，而 do while 屬條件離開的迴圈。

2. 程式會印出：

```
01234
```

注意，cout << endl; 未在迴圈主體之內（因為沒有大括號）。

3. 程式會印出：

```
0369
12
```

4. 程式會印出：

```
6
8
```

5. 程式會印出：

```
k = 8
```

6. 最簡單的方式是用 *= 運算子：

```
for (int num=1; num <=64; num *=2)
   cout << num << " ";
```

7. 你可以利用大括弧，將一些敘述組成複合敘述，或區段。

8. 是的，第一個敘述是正確的。運算式 1, 024 是由兩個運算式組成，1 和 024，由逗號運算子將二者結合起來。其值為右邊運算式的值。所以值為 024，這是 20 的八進位表示式，所以此宣告將 20 指定給 x。第二個敘述同樣也正確，但因運算子優先權使敘述變成：

```
(y = 1), 024;
```

   也就是說，左邊運算式將 y 設成 1，而整個運算式是右邊的運算式 024，或是 20。

9. cin >> ch 形式會跳過空白鍵、換行符號和 tab 鍵，而另外二者則讀入這些字元。

第 6 章

1. 二個版本結果相同，但 if else 版本較具效率。想想若 ch 為空白會發生什麼情形。版本 1，增加空白數目後，仍會測試 ch 是否為換行字元，這實在浪費時間，因為程式早知 ch 為空白，而且不會是換行字元。版本 2 在這種情形就跳過換行測試了。

2. ++ch 與 ch + 1 會得到相同的數值，但 ++ch 為 char 型態，所以印出字元，但 ch + 1 是 char 與 int 相加，結果為 int 型態，所以印出一個數值。

3. 因為程式使用 ch = '$' 而非 ch == '$'，結合輸入和輸出的樣子如下：

```
H!!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

每個字元在印出第二次之前會轉成 $ 字元。而且運算式 ch = $ 之值是 $ 字元的代碼，是非 0 值，所以為真，因此 ct2 會每次遞增。

4. a. `weight >= 115 && weight < 125`

b. `ch == 'q' || ch == 'Q'`

c. `x % 2 == 0 && x != 26`

d. `x % 2 == 0 && !(x % 26 == 0)`

e. `donation >= 1000 && donation <= 2000 || guest == 1`

f. `(ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')`

5. 未必。例如，若 x 等於 10，則 !x 為 0，而 !!x 為 1。但是若 x 是 bool 變數，則 !!x 仍為 x。

6. `( x < 0 ) ? -x : x`

或是

```
( x >= 0 ) ? x : -x;
```

7.
```
switch (ch)
{
    case 'A': a_grade++;
              break;
    case 'B': b_grade++;
              break;
    case 'C': c_grade++;
              break;
    case 'D': d_grade++;
              break;
    default:  f_grade++;
              break;
}
```

8. 如果使用整數標籤，而且使用者鍵入如 **q** 的非整數值，因為整數輸入無法處理字元，所以程式停在那兒。如果使用字元標籤，當使用者鍵入整數如 **5**，則字元輸入會將 5 當作字元處理。然後 switch 的 default 敘述會建議再輸入另一字元。

9. 以下為一種版本：

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
    if (ch == '\n')
        line++;
}
```

第 7 章

1. 這三步驟是定義函數，提供函數原型，呼叫函數。

2. a. `void igor(void);  // or void igor()`

   b. `float tofu(int n);  // or  float tofu(int);`

   c. `double mpg(double miles, double gallons);`

   d. `long summation(long harray[], int size);`

   e. `double doctor(const char * str);`

   f. `void ofcourse(boss dude);`

   g. `char * plot(map *pmap);`

3.
```
void set_array(int arr[], int size, int value)
{
    for (int i = 0; i < size; i++)
        arr[i] = value;
}
```

4.
```
void set_array(int * begin, int * end, int value)
{
    for (int * pt = begin; pt != end; pt++)
        pt* = value;
}
```

5.
```
double biggest (const double foot[], int size)
{
    double max;
    if (size < 1)
    {
        cout << "Invalid array size of " << size << endl;
        cout << "Returning a value of 0\n";
        return 0;
    }
    else    // not necessary because return terminates program
    {
```

```
        max = foot[0];
        for (int i = 1; i < size; i++)
            if (foot[i] > max)
                max = foot[i];
        return max;
    }
}
```

6. 利用 const 修飾指標，以避免指標所指的原始資料被更改。當程式傳入如 int 或 double 基本型態的引數時，會以值傳遞，所以函數是處理副本，原始的資料是受到保護的。

7. 字串可以存在 char 陣列，可以用雙引號包住的字串常數表示之，也可以使用指向字串第一個字元的指標表示之。

8.
```
int replace(char * str, char c1, char c2)
{
    int count = 0;
    while (*str)     // while not at end of string
    {
        if (*str == c1)
        {
            *str = c2;
            count++;
        }
        str++;       // advance to next character
    }
    return count;
}
```

9. 因為 C++ 將 "pizza" 解釋成第一個元素的位址，所以運用 * 運算子得到第一個元素值，亦即字元 p。因為 C++ 將 "taco" 解釋為第一個元素的位址，所以 "taco"[2] 代表第 3 個元素的值，即字元 c。換句話說，字串常數如同一個陣列的名稱。

10. 以值傳遞，只需傳遞結構名稱 glitz；要傳遞位址，則使用取址運算子 &glitz。以值傳遞會自動保護原始資料，但比較消耗時間和記憶體；以位址傳遞較節省時間和記憶體，但除非使用 const 修飾函數參數，否則無法保護原始資料。此外，以值傳遞表示可以使用一般的結構成員表示法，但傳遞指標時要記得用間接成員運算子。

11. `int judge (int (*pf)(const char *));`

12. a. 若 ap 是一 applicant 結構變數，則 ap.credit_ratings 表示陣列名稱，ap.credit_ratings[i]表示陣列元素。

```
void display(applicant ap)
{
    cout << ap.name << endl;
    for (int i = 0; i < 3; i++)
        cout << ap.credit_ratings[i] << endl;
}
```

b. 若 ap 是一指向 applicant 結構的指標變數，則 ap->credit_ratings 表示陣列名稱，ap->credit_ratings[i]表示陣列元素。

```
void show(const applicant * pa)
{
    cout << pa->name << endl;
    for (int i = 0; i < 3; i++)
        cout << pa->credit_ratings[i] << endl;
}
```

13.
```
typedef void (*p_f1)(applicant *)
p_f1 p1 = f1;
typedef const char * (*p_f2)(const applicant *, const applicant *);
p_f2 p2 = f2;
p_f1 ap[5];
p_f2 (*pa)[10];
```

第 8 章

1. 短而非遞迴的函數，而且可寫成一行的程式碼，都適合 inline 表示。

2. a. `void song(char * name, int times = 1);`

b. 沒有，只是函數原型包含了預設值的資訊。

c. 可以，前提是仍保留 times 的預設值。.

```
void song(char * name = "O, My Papa", int times = 1);
```

3. 程式可以利用字串 "\"" 或字元 '"' 印出雙引號。以下程式片段為兩種表示法。

```
#include <iostream.h>
void iquote(int n)
```

```
{
   cout<< "\"" << n << "\"";
}
void iquote(double x)
{
    cout<< '"' << x << '"';
}
void iquote(const char * str)
{
    cout << "\"" << str << "\"";
}
```

4. a. 這函數不應改變結構成員，所以加入 const 修飾元：

```
void show_box(const box & container)
{
    cout << "Made by " << container. maker << endl;
    cout << "Height = " << container.height << endl;
    cout << "Width = " << container.width << endl;
    cout << "Length = " << container.length << endl;
    cout << "Volume = " << container.volume << endl;
}
```

b.
```
void set_volume(box & crate)
{
    crate.volume = crate.height * crate.width * crate.length;
}
```

5. 首先，將函數原型改為

```
// function to modify array object
void fill(std::array<double, Seasons> & pa);
// function that uses array object without modifying it
void show(const std::array<double, Seasons> & da);
```

注意，show()應加入 const，以保護物件，防止被更改。

接下來，在 main() 函數中，將 fill()函數改為

```
fill(expenses);
```

而 show()函數的呼叫沒有改變。

接下來，fill() 函數應如下所示：

```
void fill(std::array<double, Seasons> & pa) // changed
{
    using namespace std;
    for (int i = 0; i < Seasons; i++)
    {
        cout << "Enter " << Snames[i] << " expenses: ";
        cin >> pa[i];     // changed
    }
}
```

注意，(*pa)[i] 改為 pa[a]。

最後，改變 show() 函數的原型為

```
void show(std::array<double, Seasons> & da)
```

6. a. 這可藉由第二個引數的預設值達成：

```
double mass(double d, double v = 1.0);
```

或者以函數的多載達成：

```
double mass(double d, double v);
double mass(double d);
```

b. 不能對列印重複次數的變數指定預設值，原因是指定預設值必須從右至左。但可使用多載：

```
void repeat(int times, char* str);
void repeat(char* str);
```

c. 可利用函數的多載：

```
int average(int a, int b);
double average(double x, double y);
```

d. 不能這樣使用，因為兩個版本有相同的簽名。

7.
```
template<class T>
T max(T t1, T t2) // or T max(const T & t1, const T & t2)
{
    return t1 > t2? t1 : t2;
}
```

8.
```
template<> box max(box b1, box b2)
{
    return b1.volume > b2.volume? b1 : b2;
}
```

9. v1 是 float 型態，v2 是 float &型態，v3 是 float &型態，v4 是 int 型態，v5 是 double 型態。2.0 是 double 型態，所以 2.0 * m 是 double 型態。

第 9 章

1. a. homer 自動地成為為自動變數。

   b. secret 應在一個檔案中定義為外部變數，而於其它檔案宣告為 extern 變數。

   c. topsecret 要在其外部定義前放置關鍵字 static，將它定義為具有內部連結性的 static 變數。或是它可以定義在不具名的名稱空間中。

   d. beencalled 必須定義成區域的 static 變數，也就是在函數中其宣告前面放置關鍵字 static。

2. using 宣告是使名稱空間的單一名稱變成可用，而且其範疇對應於 using 宣告發生處所屬的宣告區域。using 指令會使名稱空間的所有名稱都變成可用。使用 using 指令時就好像是在，包含 using 宣告和名稱空間本身的最小宣告區域中宣告名稱。

3.
```
#include <iostream>
int main()
{
    double x;
    std::cout << "Enter value: ";
    while (! (std::cin >> x) )
    {
        std::cout << "Bad input. Please enter a number: ";
        std::cin.clear();
        while (std::cin.get() != '\n')
           continue;
    }
    std::cout << "Value = " << x << std::endl;
    return 0;
}
```

4. 以下是修改後的程式碼：

```
#include <iostream>
int main()
{
    using std::cin;
    using std::cout;
    using std::endl;
```

```
    double x;
    cout << "Enter value: ";
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";
        cin.clear();
       while (cin.get() != '\n')
           continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

5. 在每個檔案中，你可以有獨立的靜態函數定義。或是每個檔案可以定義，位於不具名之名稱空間的適當 average() 函數。

6.
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7.
```
1
4, 1, 2
2
2
4, 1, 2
2
```

第 10 章

1. 類別為使用者自定型態的定義，類別宣告說明了類別儲存的資料，和可以用來處理這些資料的成員函數（類別成員函數）。

2. 類別用類別成員函數的公用介面，表示可以執行在類別物件上的操作，稱為抽象化。類別的資料成員預設為私有可見性（private visibility），所以資料是隱藏的，只能透過成員函數存取之。至於實作細節，例如資料表示法或成員函數則被隱藏著，這是封裝。

3. 類別定義一種型態，包括如何使用。物件可視為一個變數，或是另一種資料物件，如由 new 產生的物件，物件建立和使用方式均視類別定義而定。這關係就如同標準型態與其變數的關係。

4. 如果建立許多屬同一種類別的物件，則每個物件擁有自己一組類別資料，但所有物件使用一組函數成員（同常，成員函數是公用，資料成員是私有，但這些政策事務並非一定，不是類別的需求）。

5. 這個範例使用 char 陣列來儲存字元資料，不過你可以使用 string 類別物件加以替換。

```
// #include <cstring>

// class definition
class BankAccount
{
private:
    char name[40];     // or std::string name;
    char acctnum[25];  // or std::string acctnum;
    double balance;
public:
    BankAccount(const char * client, const char * num, double bal = 0.0);
//or BankAccount(const std::string & client,
//               const std::string & num, double bal = 0.0);

    void show(void) const;
    void deposit(double cash);
    void withdraw(double cash);
};
```

6. 類別的建構函數的呼叫時機是，建立此類別的物件，或是明確的呼叫此建構函數時。當物件無效時，則呼叫類別解構函數。

7. 這裡有兩種解法（注意為了使用 strncpy() 必須匯入 cstring 或 string.h，不然就是匯入 string 來使用 string 類別）：

```
BankAccount::BankAccount(const char * client, const char * num, double bal)
{
    strncpy(name, client, 39);
    name[39] = '\0';
    strncpy(acctnum, num, 24);
    acctnum[24] = '\0';
    balance = bal;
}
```

或者是

```
BankAccount::BankAccount(const std::string & client,
                         const std::string & num, double bal)
{
    name = client;
    acctnum = num;
    balance = bal;
}
```

請注意，引數預設值要出現在函數原型中，而非在函數定義中。

8. 預設建構函數是一個沒有引數的函數，或是全部引數都有預設值的函數。這個預設建構函數的目的是，宣告物件時不用初始化之，包括你已經定義初始化的建構函數。它也使你可以宣告陣列。

9.
```
// stock3.h
#ifndef STOCK3_H_
#define STOCK3_H_

class Stock
{
private:
    std::string company;
    int shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();            // default constructor
    Stock(const std::string & co, int n, double pr);
    ~Stock() {}         // do-nothing destructor
    void buy(int num, double price);
    void sell(int num, double price);
    void update(double price);
    void show() const;
    const Stock & topval(const Stock & s) const;
    int numshares() const { return shares; }
    double shareval() const { return share_val; }
    double totalval() const { return total_val; }
    string co_name() const { return company; }
};
```

10. this 指標是類別成員函數可以使用的指標，this 指向的物件是呼叫此成員函數的物件。因此 this 是物件的位址，*this 即是物件本身。

第 11 章

1. 這是類別定義檔案的一個原型，以及成員函數檔案中的函式定義：

```
// prototype
Stonewt operator*(double mult);

//definition -- let constructor do the work
Stonewt Stonewt::operator*(double mult)
{
    return Stonewt(mult * pounds);
}
```

2. 成員函數屬於類別定義的一部份，只有物件能呼叫之。成員函數可以存取呼叫物件的成員，不必使用成員運算子。夥伴函數不屬於類別，所以可以用函數呼叫直接呼叫之。它不能直接存取類別成員，必須藉由引數傳入物件，然後利用成員運算子存取成員。例如，比較問題回顧 1 和問題回顧 4 的答案。

3. 它必須是夥伴函數才能存取私有成員，但不一定要夥伴函數才能存取公用成員。

4. 這是類別定義檔案的一個原型，以及成員函數檔案的函數定義：

```
// prototype
friend stonewt operator*(double mult, const Stonewt & s);

//definition -- let constructor do the work
Stonewt operator*(double mult, const Stonewt & s)
{
    return Stonewt(mult * s.pounds);
}
```

5. 下列 5 種運算子不能多載：

```
sizeof
.
.*
::
? :
```

6. 這些運算子必須定義為成員函數。

7. 以下是一種可能的函數原型和函數定義：

```
//prototype and inline definition
operator double () {return mag;}
```

注意，使用 magval() 成員函數比定義轉換函數更有意義。

第 12 章

1. a. 這語法是對的，但此建構函數沒有初始化 str 指標。應該將指標設為 NULL 或以 new [] 初始化指標。

   b. 此建構函數並未產生新字串；它只是複製原先字串的位址，應該使用 new [] 和 strcpy()。

   c. 拷貝字串時並未替字串配置記憶體。應用 new char[len + 1] 配置適量的記憶體。

2. 第一點，當物件消失時，物件的成員指標所指向的資料仍在記憶體中，但因為指標已經消失，所以空間仍被佔用但卻無法存取。修正方式是在解構函數中，刪除建構函數用 new 配置的記憶體。第二點，一旦解構函數清除了這種記憶體，如果程式用一個這種物件初始化另一物件，則最後可能刪除此記憶體兩次。這種問題是，因為用一個物件初始化另一個物件的預設方式是只複製指標值，而不是複製指標所指的內容，使得兩個指標指向相同的資料。解決方法是定義類別的複製建構函數，使初始化可以複製指標所指的資料。第三點，將一個物件指定給另一物件，也會造成兩個指標指向同一資料的相同情況。解決方法是多載指定運算子，使其複製資料而非複製指標。

3. C++ 自動提供以下成員函數：

   - 如果沒有定義建構函數，則產生預設建構函數。
   - 如果沒有定義複製建構函數，則產生複製建構函數。
   - 如果沒有定義指定運算子，則產生指定運算子。
   - 如果沒有定義解構函數，則產生預設解構函數。
   - 如果沒有定義取址運算子，則產生取址運算子。

   預設建構函數不作任何事情，但你可用此宣告陣列以及未初始化的物件。預設的複製建構函數和預設的指定運算子是，將成員逐一指定到另一物件。預設的解構函數不作任何事情。預設的取址運算子傳回呼叫物件的位址（也就是 this 指標之值）。

4. personality 成員應可宣告成字元陣列或 char 指標。另一方式是宣告成 String 物件，或者是 string 物件。在宣告時，成員函數沒有辦法設為是公有的。接著還會有一些小錯誤。以下是兩種可能的解法，粗體字是改變的地方（除了刪除以外）：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char personality[40];    // provide array size
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    friend ostream & operator<<(ostream & os, const nifty & n);
};    // note closing semicolon

nifty::nifty()
{
    personality[0] = '\0';
    talents = 0;
}

nifty::nifty(const char * s)
{
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

或另一解法：

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private: // optional
    char * personality;    // create a pointer
```

```
    int talents;
public: // needed
// methods
    nifty();
    nifty(const char * s);
    nifty(const nifty & n);
    ~nifty() { delete [] personality; }
    nifty & operator=(const nifty & n) const;
    friend ostream & operator<<(ostream & os, const nifty & n);
}; // note closing semicolon

nifty::nifty()
{
    personality = NULL;
    talents = 0;
}

nifty::nifty(const char * s)
{
    personality = new char [strlen(s) + 1];
    strcpy(personality, s);
    talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
    os << n.personality << '\n';
    os << n.talent << '\n';
    return os;
}
```

5. a.
```
Golfer nancy; // default constructor
Golfer lulu("Little Lulu"); // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer; // default constructor
Golfer next = lulu; // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker"; // Golfer(const char * name, int g)
*par = nancy; // default assignment operator
nancy = "Nancy Putter";// Golfer(const char * name, int g), then
                       // the default assignment operator
```

   請注意，某些編譯程式於敘述 #5 和敘述 #6，會額外呼叫預設的指定運算子。

   b. 類別應定義指定運算子，用以複製資料而非複製位址。

第 13 章

1. 基礎類別的公用成員會變成衍生類別的公用成員。基礎類別的保護成員會變成衍生類別的保護成員，基礎類別的私有成員可被繼承，但不能直接存取。請同時參考問題回顧 2 的答案，以瞭解例外情形。

2. 建構函數是不能繼承的，此外解構函數、指定運算子、和夥伴成員均不能被繼承。

3. 若回傳型態是 void，則你仍然可以使用單一指定，而不是串接指定：

```
baseDMA magazine("Pandering to Glitz", 1);
baseDMA gift1, gift2, gift3;
gift1 = magazine;          // ok
gift 2 = gift3 = gift1;    // no longer valid
```

   若此成員函數回傳物件，而不是 reference，則此成員函數的執行會比較慢，因為回傳敘述需要複製物件。

4. 建構函數的呼叫順序是依據類別衍生的順序，首先呼叫最早祖先的建構函數。解構函數的呼叫順序是相反的。

5. 是的，每個類別需要有自己的建構函數，如果衍生類別沒有新增成員，則建構函數可以只是一個空的主體，但一定要存在。

6. 只會呼叫衍生類別的成員函數，它取代基礎類別的定義。只有當衍生類別沒有重新定義成員函數，或是使用範疇運算子，才會呼叫基礎類別的成員函數。因此，會被重新定義的所有函數都應宣告為 virtual。

7. 若衍生類別的建構函數使用 new 或 new [] 運算子，來初始化類別成員中的指標，則應定義指定運算子。更一般性的說法是，若預設的指定對於衍生的類別成員是不正確的，就要定義指定運算子。

8. 是的，你可以將衍生類別物件的位址，指定給基礎類別的指標。唯有利用明確的型態轉換，才能將基礎類別物件的位址指定給衍生類別指標（向下轉型），而且使用這種指標並不一定安全。

9. 是的，你可以將衍生類別物件指定給基礎類別物件。任何衍生類別的新資料成員不會傳給基礎類別。程式會使用基礎類別的指定運算子。唯有衍生類別定義轉換運算子，程式才可以將基礎類別物件指定給衍生類別物件。所謂轉換運算子是以基礎類別的 reference，作為建構函數的唯一引數，或是定義以基礎類別為參數的指定運算子。

10. 它可以這樣做，因為 C++ 的基礎類別的 reference，可以參考從此基礎類別衍生的任何類別。

11. 傳遞物件值會呼叫複製建構函數。因為形式引數為基礎類別物件，所以會呼叫基礎類別的複製建構函數。複製建構函數的引數為基礎類別的 reference，所以此 reference 可以參考以引數傳入的衍生物件。最後的結果是建立一個新的基礎類別物件，其成員是衍生物件的基礎類別部分。

12. 傳遞物件的 reference 而不是物件值，可使函數利用虛擬函數的功能。此外，傳遞物件 reference 比較節省時間與記憶體，尤其是大物件。以值傳遞的優點是可以保護原始資料，但以 reference 傳遞加上 const 就能達到同樣的目的。

13. 如果 head() 為一般成員函數，則 ph->head() 會呼叫 Corporation::head()。如果 head() 為虛擬函數，則 ph->head() 會呼叫 PublicCorporation::head()。

14. 第一，這情況不適用 is-a 模式，所以公用繼承是不適當的。第二，House 中的 area() 定義會遮蔽 kitchen 的 area() 版本，因為這兩個成員函數有不同的簽名。

第 14 章

1.

class Bear	class PolarBear	Public，北極熊是一種熊
class Kitchen	class Home	Private，家有廚房
class Person	class Programmer	Public，程式設計師是一種人
class Person	class HorseAndJockey	Private，馬與騎師組包含一個人
class Person, class Automobile	class Driver	Person Public，因為司機是一個人；Automobile private，因為司機有汽車

2.
```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    fb.tell();
    cout << glip << endl;
}
```

3.
```
Gloam::Gloam(int g, const char * s)
             : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
             : glip(g), Frabjous(fr) { }
// note: the above uses the default Frabjous copy constructor
void Gloam::tell()
{
    Frabjous::tell();
    cout << glip << endl;
}
```

4.
```
class Stack<Worker *>
{
private:
    enum {MAX = 10};        // constant specific to class
    Worker * items[MAX];    // holds stack items
    int top;                // index for top stack item
public:
    Stack();
    Boolean isempty();
    Boolean isfull();
    Boolean push(const Worker * & item); // add item to stack
    Boolean pop(Worker * & item);        // pop top into item
};
```

5.
```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

範例程式 14.18 產生四個樣版：`ArrayTP<int, 10>`, `ArrayTP<double, 10>`, `ArrayTP<int,5>` 與 `Array< ArrayTP<int,5>, 10>`。

6. 假設有一類別繼承兩個有共同祖先的類別，則此類別最後會有二份這個祖先的成員。將此祖先類別變成虛擬基礎類別，就可解決這個問題。

第 15 章

1. a. 夥伴的宣告應如下：

```
friend class clasp;
```

b. 這需要向前宣告，如此編譯程式才得以解釋 `void snip(muff &)`：

```
class muff;     //forward declaration
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

c. 首先，`cuff` 類別必須宣告於 `muff` 類別之前，如此編譯程式才得以瞭解 `cuff::snip()`。其次，`muff` 必須使用向前宣告，編譯程式才得以瞭解 `snip(muff &)`。

```
class muff;     //forward declaration
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

2. 否，類別 A 有一夥伴屬於類別 B 的成員函數，所以 B 必須宣告於 A 宣告之前。但只有 A 向前宣告並不足夠，同時必須讓 A 知道 B 是個類別，但卻不能透露 B 的類別成員名稱。同理，如果 B 有一夥伴屬於 A 的成員函數，則 A 的完整宣告必須置於於 B 的宣告之前。這兩個需求是互斥的。

3. 存取類別的唯一方式是藉由它的公用介面，意思是對於 Sauce 物件你只能呼叫建構函數建立之，其它兩個成員（soy 和 sugar）預設是私有的。

4. 假設函數 `f1()` 呼叫函數 `f2()`。`f2` 中的 return 敘述，會使程式繼續執行 `f1()` 函數中呼叫 `f2()` 之後的敘述。`throw` 敘述會使程式根據目前的函數呼叫串列往回找，直到找到直接或間接包含 `f2()` 呼叫的 `try` 區塊。這個可能位於 `f1()` 或是呼叫 `f1()` 的函數中，以此類推。一旦找到，接著執行匹配的 `catch` 區塊，而不是執行函數呼叫後的第一個敘述。

5. 安排 catch 區塊的順序，應從最後的衍生類別安排到最開始的衍生類別。

6. 對於範例 #1，如果 pg 指向 Superb 物件或任何 Superb 的衍生物件，則 if 條件為 true。尤其是若 pg 指向 Magnificent 物件，則結果亦為 true。對於範例 #2，只有 pg 指向 Superb 物件，而非 Superb 的衍生物件，if 條件才為 true。

7. dynamic_cast 運算子只允許在類別階層架構中向上轉型，而 static_cast 運算子則可以向上轉型和向下轉型。static_cast 運算子也可以從列舉型態轉換成整數型態，反之亦然，以及各種數字型態之間的轉換。

第 16 章

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
    string st;        // a string object
public:
    RQ1() : st("") {}
    RQ1(const char * s) : st(s) {}
    ~RQ1() {};
// more stuff
};
```

不再需要明確的複製建構函數，解構函數，和指定運算子，因為 string 物件提供它自己的記憶體管理。

2. 你可以將 string 物件指定給另一個 string 物件。string 物件提供自己的記憶體管理，所以通常不需要擔心字串會超過其容量。

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia= new int[20];  // wrong, use with new, not new[]
auto_ptr<string>(new string);    // wrong, no name for pointer
int rigue = 7;
auto_ptr<int>(&rigue);           // wrong, memory not allocated by new
auto_ptr dbl (new double);       // wrong, omits <double>
```

5. 堆疊的 LIFO 觀念是在找到你要的資料之前，必須移除許多俱樂部。

6. 因為 set 只會儲存每個值的一份資料，如 5 個 5 分只會存成單一的 5。

7. 使用迭代器可在處理物件時，用類似指標的介面走訪資料，而不以陣列的方式（如在雙向鏈結串列中的資料）。

8. STL 的方法使 STL 函數可以用於一般陣列的指標，以及指向 STL 收納器類別的迭代器，而增加其一般性。

9. 你可以將 vector 物件指定給另一個 vector 物件。vector 管理它自己的記憶體，所以可以將資料項插入 vector，它會自己自動調整大小。利用成員函數 at() 可以取得自動的邊界檢查。

10. sort() 函數和 random_shuffle() 函數需要一個隨機存取的迭代器，而 list 物件恰好有一個雙向的迭代器。你可以用 list 樣版類別的 sort() 成員函數（請參考附錄 G）而非一般功能的函數來作排序，但是沒有一個成員函數等於 random_shuffle()。但是你可將 list 複製到 vector，對 vector 任意排列後，再將結果複製回 list。

第 17 章

1. iostream 檔案定義類別，常數，和一些處理輸出入的運作子。這些物件管理用於 I/O 的串流和緩衝區。這檔案同時建立連接，程式與標準輸出入串流的標準物件（cin，cout，cerr，clog 和寬字元的同等物件）。

2. 鍵盤輸入產生一連串字元，鍵入 **121**，產生 3 個字元，每個字元以 1 位元組的二進位碼表示。如果以 int 型態儲存值，則這 3 個字元轉換成值 121 的單一二進位表示式。

3. 系統預設將標準輸出和標準錯誤送至標準輸出裝置，一般是指螢幕。如果使作業系統將標準輸出重新導向至檔案，則標準輸出連到檔案而非螢幕；但是標準錯誤輸出仍連到螢幕。

4. ostream 類別為每個 C++ 基本型態定義 operator<<() 版本的函數，編譯程式將以下運算式

```
cout << spot
```

轉換成：

```
cout.operator<<(spot)
```

如此一來，編譯程式會將運算函數呼叫，與有相同之引數型態的函數原型產生匹配。

5. 你可以將回傳 ostream & 型態的輸出成員函數串接起來。這使得用物件呼叫成員函數會回傳此物件。然後，回傳的物件可以再呼叫序列中的下一個成員函數。

6.
```
//rq17-6.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    cout << "Enter an integer: ";
    int n;
    cin >> n;
    cout << setw(15) << "base ten" << setw(15)
           << "base sixteen" << setw(15) << "base eight" << "\n";
    cout.setf(ios::showbase); // or cout << showbase;
    cout << setw(15) << n << hex << setw(15) << n
           << oct << setw(15) << n << "\n";

    return 0;
}
```

7.
```
//rq17-7.cpp
#include <iostream>
#include <iomanip>

int main()
{
    using namespace std;
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::right, ios::adjustfield);
// or cout << showpoint << fixed << right;
    cout << "First format:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";

    return 0;
}
```

8. 輸出結果如下：

```
ct1 = 5; ct2 = 9
```

程式的第一部分會略過空白與換行字元，第二部分則不會。注意程式的第二部分於碰到第一個 q 後，開始讀取換行字元，而換行字元會算入所有的字元之內。

9. 若輸入行超過 80 個字元，則 ignore() 形式會有問題，因為它只略過前 80 個字元。

第 18 章

1.
```
class Z200
{
private:
    int j;
    char ch;
    double z;
public:
    Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
...
};

double x {8.8}; // or = {8.8}
std::string s {"What a bracing effect!"};
int k{99};
Z200 zip{200,'Z',0.67});
std::vector<int> ai {3, 9, 4, 7, 1};
```

2. r1(w)是有效的，並且參數 rx 引用 w。

   r1(w+1)是有效的，並且參數 rx 引用 w+1 的暫時值。

   r1(up(w))是有效的，並且參數 rx 引用由 up(w)函數所回傳的暫時值。

   一般而言，假使 lvalue 是傳送給一 const 的 lvalue reference 的參數，則此參數是被初值化為 lvalue。若一 rvalue 是被傳送給一函數，則 const 的 lvalue reference 參數將引用值的暫存複本。

   r2(w)是有效的，並且參數 rx 引用 w。

   r2(w+1)是錯誤的，因為 w+1 是 rvalue。

   r2(up(w))是錯誤的，因為 up(w)函數的回傳值是 rvalue。

   一般而言，假使 lvalue 是傳送給一 non-const 的 lvalue reference 的參數，則此參數是被初值化為 lvalue。但若是一 non-const 的 lvalue reference 的參數不可以接受一 rvalue 函數參數。

   r3(w)是錯誤的，因為 rvalue reference 不可以引用 lvalue，如 w。

   r3(w+1)是有效的，並且 rx 引用運算式 w+1 的暫時值。

   r3(up(w))是有效的，並且 rx 引用由 up(w)函數所回傳的暫時值。

3. a.
```
double & rx
const double & rx
const double & rx
```

non-const 的 lvalue reference 與 lvalue 參數 w 相匹配。其餘兩個參數是 rvalue，而且 const 的 lvalue reference 可以引用它們的複本。

b.
```
double & rx
double && rx
double && rx
```

lvalue reference 與 lvalue 參數 w 相匹配，而且 rvalue reference 與兩個 rvalue 參數相匹配。

c.
```
const double & rx
double && rx
double && rx
```

const 的 lvalue reference 與 lvalue 參數 w 相匹配，而且 rvalue reference 與兩個 rvalue 參數相匹配。

簡言之，non-const lvalue 參數與 lvalue 參數相匹配，non-const rvalue 參數與 rvalue 參數相匹配。const 的 lvalue 參數可以和 lvalue 或 rvalue 參數相匹配。若可以的話，編譯程式較喜歡前者。

4. 計有預設建構函數，複製建構函數，移動建構函數，解構函數，複製指定運算子以及移動指定運算子。因為編譯程式依據內文，自動提供這些函數的預設版本。

5. 當它將轉移資料的所有權取代複製時，此時移動建構函數可以被使用。但沒有一機制用來轉移一般標準陣列的所有權。假使 Fizzle 類別做用指標與動態記憶體配置的話，將可利用重新指定資料的位址給一新的指標，來轉移所有權。

6.
```
#include <iostream>
#include <algorithm>
template<typename T>
    void show2(double x, T fp) {std::cout << x << " -> " << fp(x) << '\n';}
int main()
{
    show2(18.0, [](double x){return 1.8*x + 32;});
    return 0;
}
```

7.
```
#include <iostream>
#include <array>
#include <algorithm>
const int Size = 5;
template<typename T>
  void sum(std::array<double,Size> a, T& fp);
int main()
{
    double total = 0.0;
   std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
```

```
   sum(temp_c, [&total](double w){total += w;});
   std::cout << "total: " << total << '\n';
   std::cin.get();
   return 0;
}
template<typename T>
  void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
        fp(*pt);
    }
}
```