8-1 「分數加減乘除」程式的 debug

丁培毅

8-1 「分數加減乘除」程式的 debug

目標：

丁培毅

8-1 「分數加減乘除」程式的 debug

目標：

丁培毅

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

8-1 「分數加減乘除」程式的 debug

目標：

丁培毅

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

   "Debugging is twice as hard as writing the code in the first place." - Brian W. Kernighan

8-1 「分數加減乘除」程式的 debug

丁培毅

目標：

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

   "Debugging is twice as hard as writing the code in the first place." - Brian W. Kernighan

2. 練習 **debug** 的基本程序

8-1 「分數加減乘除」程式的 debug

丁培毅

目標：

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

   "Debugging is twice as hard as writing the code in the first place." - Brian W. Kernighan

2. 練習 **debug** 的基本程序

   2.1 手動計算得到一些測試範例資料的完整轉變過程

8-1 「分數加減乘除」程式的 debug

丁培毅

目標：

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

   "Debugging is twice as hard as writing the code in the first place." - Brian W. Kernighan

2. 練習 **debug** 的基本程序

   2.1 手動計算得到一些測試範例資料的完整轉變過程

   2.2 要求程式輸出一些資料, 藉由這些資料來驗證程式中各個片段的執行結果是否符合預期

8-1 「分數加減乘除」程式的 debug

丁培毅

目標：

1. 調整心態: 程式寫完以後不可能第一次測試就發現所有的功能都是正確的, Debug 是正常程序中不可或缺的步驟

   "Debugging is twice as hard as writing the code in the first place." - Brian W. Kernighan

2. 練習 **debug** 的基本程序

   2.1 手動計算得到一些測試範例資料的完整轉變過程

   2.2 要求程式輸出一些資料, 藉由這些資料來驗證程式中各個片段的執行結果是否符合預期

   2.3 分析不符合預期的敘述, 找出修改的方法

8-1 分數的加減乘除

- 題目說明: 請撰寫一個程式輸入兩個普通分數 (e.g. 2/3, 7/2...), 以及運算符號, 程式會完成指定的運算並且用最簡分數列印運算的結果

8-1 分數的加減乘除

- 題目說明: 請撰寫一個程式輸入兩個普通分數 (e.g. 2/3, 7/2...), 以及運算符號, 程式會完成指定的運算並且用最簡分數列印運算的結果

- 範例輸入:

```
2/3 first operand
+
5/9 second operand
y
2  /   3
-
1  /6
y
   2/3
  *
9/  8
y
3/5
/
6/25
n
```

8-1 分數的加減乘除

- 題目說明: 請撰寫一個程式輸入兩個普通分數 (e.g. 2/3, 7/2...), 以及運算符號, 程式會完成指定的運算並且用最簡分數列印運算的結果

- 範例輸出:

  2/3 + 5/9 = 11/9

  2/3 - 1/6 = 1/2

  2/3 * 9/8 = 3/4

  3/5 / 6/25 = 5/2

- 範例輸入:

  2/3 first operand
  +
  5/9 second operand
  y
  2 / 3
  -
  1 /6
  y
  2/3
  *
  9/ 8
  y
  3/5
  /
  6/25
  n

基本工具的運用

- 程式語法有錯誤要借助編譯器給你的**錯誤訊息**以及**警告訊息**很快地更正過來

基本工具的運用

- 程式語法有錯誤要借助編譯器給你的**錯誤訊息**以及**警告訊息**很快地更正過來
  - 編譯器給你**錯誤訊息 (error)** 時, 語法還有錯誤, 編譯器不曉得你打算要怎麼做, 沒有辦法幫你轉換為可以執行的機器指令, 所以還沒有辦法執行, 沒有辦法測試, 訊息雖然是英文的, 但是其實它們很精確, 要練習看, 不要再用 google 翻譯, 因為翻譯過來的中文專有名詞不見了, 變成模稜兩可的, 但是可以用 google 搜尋定義、相關的解釋或是其他範例

基本工具的運用

- 程式語法有錯誤要借助編譯器給你的**錯誤訊息**以及**警告訊息**很快地更正過來
  - 編譯器給你**錯誤訊息 (error)** 時, 語法還有錯誤, 編譯器不曉得你打算要怎麼做, 沒有辦法幫你轉換為可以執行的機器指令, 所以還沒有辦法執行, 沒有辦法測試, 訊息雖然是英文的, 但是其實它們很精確, 要練習看, 不要再用 google 翻譯, 因為翻譯過來的中文專有名詞不見了, 變成模稜兩可的, 但是可以用 google 搜尋定義、相關的解釋或是其他範例
  - 編譯器給你**警告訊息 (warning)** 時, 你的程式語法裡有一些不太確定的東西, 或是有一些危險性, 編譯器雖然可以根據慣例猜測, 可以無視那個危險性, 但是希望你能夠用更明確的語法修改, 很多時候這些警告都會導致程式執行時結果錯誤

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)
- 那麼程式執行起來答案不正確該怎麼辦? 恐慌嗎? 崩潰嗎?

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)
- 那麼程式執行起來答案不正確該怎麼辦? 恐慌嗎? 崩潰嗎?
- 這叫做程式的 bug 嗎? 對, 有些是設計時邏輯上面的錯誤, 有些是語法的誤用

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)
- 那麼程式執行起來答案不正確該怎麼辦? 恐慌嗎? 崩潰嗎?
- 這叫做程式的 bug 嗎? 對, 有些是設計時邏輯上面的錯誤, 有些是語法的誤用
- 請不要把它看成一個黑盒子, 它是你生出來的, 雖然不用你負責, 但是它是可以拆開來一部分一部分檢視修改的「軟體」

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)
- 那麼程式執行起來答案不正確該怎麼辦? 恐慌嗎? 崩潰嗎?
- 這叫做程式的 bug 嗎? 對, 有些是設計時邏輯上面的錯誤, 有些是語法的誤用
- 請不要把它看成一個黑盒子, 它是你生出來的, 雖然不用你負責, 但是它是可以拆開來一部分一部分檢視修改的「軟體」
- 產生 bug 很不好嗎? 很丟臉嗎? 是不是腦袋有洞啊? 不是, 每一個剛設計好的程式都一定有 bug

心態的調整

- 程式開始執行以後答案不正確是非常非常正常的, 不需要覺得是一種打擊, 第一次執行就完全對了才是不正常的 (通常都代表這個程式是由已經測試正確的程式拷貝來的)
- 那麼程式執行起來答案不正確該怎麼辦? 恐慌嗎? 崩潰嗎?
- 這叫做程式的 bug 嗎? 對, 有些是設計時邏輯上面的錯誤, 有些是語法的誤用
- 請不要把它看成一個黑盒子, 它是你生出來的, 雖然不用你負責, 但是它是可以拆開來一部分一部分檢視修改的「軟體」
- 產生 bug 很不好嗎? 很丟臉嗎? 是不是腦袋有洞啊? 不是, 每一個剛設計好的程式都一定有 bug
- debug 這個步驟可以跳過去嗎? 不行, **debug 和 test** 是撰寫程式時正常的必經過程, 走過這個階段程式才算完成

除錯 (Debug)

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來
  - 準備程式的測試資料

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來
  - 準備程式的測試資料
  - 預期程式各個步驟針對這組測試資料時的輸出 (紙筆)

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來
  - 準備程式的測試資料
  - 預期程式各個步驟針對這組測試資料時的輸出 (紙筆)
  - 觀察程式整體表現, 哪一個輸出是錯誤的? 和上一步驟預期的不同

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來
  - 準備程式的測試資料
  - 預期程式各個步驟針對這組測試資料時的輸出 (紙筆)
  - 觀察程式整體表現, 哪一個輸出是錯誤的? 和上一步驟預期的不同
  - 一步一步驗證從開始處理測試資料到輸出錯誤的結果中間所有的步驟, 把每個步驟的輸出印出來和預期的比對

除錯 (Debug)

- 程式一設計完時需要對每一個動作的效果有所預期, 開始執行的時候, 需要在執行的過程中觀察它的表現, 仔細分析每一表現的原因, 一步一步調整程式的敘述使得程式具有預期的行為, debug 是和機器互動的程序, 不是用力把 bug 看出來
  - 準備程式的測試資料
  - 預期程式各個步驟針對這組測試資料時的輸出 (紙筆)
  - 觀察程式整體表現, 哪一個輸出是錯誤的? 和上一步驟預期的不同
  - 一步一步驗證從開始處理測試資料到輸出錯誤的結果中間所有的步驟, 把每個步驟的輸出印出來和預期的比對
  - 調整程式的敘述, 想辦法讓每一步驟都達成預期的效果

目標程式

- 這個程式包含 6 個輔助函式, 還有 main 函式

目標程式

- 這個程式包含 6 個輔助函式, 還有 main 函式

```
void scan_fractions(int*, int*); // 讀取分數
void get_operator(char*); // 讀取指定的運算符號
void add_fractions(int, int, int, int, int*, int*);  // 加, 減
void mult_fractions(int, int, int, int, int*, int*);  // 乘, 除
int find_gcd(int, int); // 最大公因數
void reduce_fraction(int*, int*, int*, int*); // 分數化簡

int main (void) {
    …
}
```

目標程式

- 這個程式包含 6 個輔助函式, 還有 main 函式

```
void scan_fractions(int*, int*); // 讀取分數
void get_operator(char*); // 讀取指定的運算符號
void add_fractions(int, int, int, int, int*, int*);  // 加, 減
void mult_fractions(int, int, int, int, int*, int*);  // 乘, 除
int find_gcd(int, int); // 最大公因數
void reduce_fraction(int*, int*, int*, int*); // 分數化簡

int main (void) {
    …
}
```

- 請下載 http://squall.cs.ntou.edu.tw/cprog/debug/prog1.cpp
- 這個程式目前由鍵盤輸入資料以後沒有任何輸出

步驟一

- 請注意觀察目前程式執行時的表現

步驟一

- 請注意觀察目前程式執行時的表現

輸入
1/2
+
3/4
可以一直打資料，完全沒有看到任何輸出

步驟一

- 請注意觀察目前程式執行時的表現

輸入
1/2
+
3/4
可以一直打資料, 完全沒有看到任何輸出

- 仔細觀察是 debug 過程中一定需要的, 你不能只知道沒有跑出答案, 沒有跑出答案還有分很多種, 你越仔細看到程式的細部表現, 越能縮短 debug 的時間

步驟二: 縮小問題、找出問題發生點

- 接下來需要根據程式的架構來分析 (這個程式用了很多函數, 架構非常模組化, 資料的流動很清楚, 是一個相對容易 debug 的程式)

輸入
1/2
+
3/4
還是完全沒有看到輸出

步驟二: 縮小問題、找出問題發生點

- 接下來需要根據程式的架構來分析 (這個程式用了很多函數, 架構非常模組化, 資料的流動很清楚, 是一個相對容易 debug 的程式)
- 輸入資料的程式在 19-21 列,
  輸出結果的程式在第 44 列,
  中間是 switch 敘述分派不同的運算

輸入
1/2
+
3/4
還是完全沒有看到輸出

步驟二: 縮小問題、找出問題發生點

- 接下來需要根據程式的架構來分析 (這個程式用了很多函數, 架構非常模組化, 資料的流動很清楚, 是一個相對容易 debug 的程式)
- 輸入資料的程式在 19-21 列,
  輸出結果的程式在第 44 列,
  中間是 switch 敘述分派不同的運算
- 讓我們先把 **22** 列到 **43** 列都註解掉
  去除中間這些運算以後程式變得很單純,
  沒有做什麼事情,
  這樣子可以讓我們的問題**簡化**一點, 先確定輸入是不是對的

輸入
1/2
+
3/4
還是完全沒有看到輸出

現在的程式相當於下面四列

現在的程式相當於下面四列

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator,
        op, numerator2, denominator2, nump, denomp);
```

現在的程式相當於下面四列

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator,
        op, numerator2, denominator2, nump, denomp);
```

如果 printf 沒有印出任何東西, 應該可以說是 **printf** 完全沒有執行到 (注意 printf 敘述通常有執行到就多多少少有印出東西來, 就算是資料錯誤或是格式轉換命令錯誤, 也會有印出一些東西, 完全沒有印出來 95% 是根本沒有執行到)

所以應該是 scan_fractions() 或是 get_operator() 的問題

現在的程式相當於下面四列

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator,
        op, numerator2, denominator2, nump, denomp);
```

如果 printf 沒有印出任何東西, 應該可以說是 **printf** 完全沒有執行到 (注意 printf 敘述通常有執行到就多多少少有印出東西來, 就算是資料錯誤或是格式轉換命令錯誤, 也會有印出一些東西, 完全沒有印出來 95% 是根本沒有執行到)

所以應該是 scan_fractions() 或是 get_operator() 的問題

請注意 **debug** 是一個很縝密的推理過程

現在的程式相當於下面四列

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator,
        op, numerator2, denominator2, nump, denomp);
```

如果 printf 沒有印出任何東西, 應該可以說是 **printf** 完全沒有執行到 (注意 printf 敘述通常有執行到就多多少少有印出東西來, 就算是資料錯誤或是格式轉換命令錯誤, 也會有印出一些東西, 完全沒有印出來 95% 是根本沒有執行到)

所以應該是 scan_fractions() 或是 get_operator() 的問題

請注意 **debug** 是一個很縝密的推理過程

首先需要根據觀察到的現象推論出問題到底在哪裡發生的

步驟三: 觀察細部表現

- 在 scan_fractions() 中增加列印敘述來檢查是不是有錯誤

步驟三: 觀察細部表現

- 在 scan_fractions() 中增加列印敘述來檢查是不是有錯誤

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
printf("num=%d denom=%d\n", *numerator, *denominator);
do { scanf(" %c", &discard); }  while(discard != '\n');
printf("num=%d denom=%d\n", *numerator, *denominator);
```

步驟三: 觀察細部表現

- 在 scan_fractions() 中增加列印敘述來檢查是不是有錯誤

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
printf("num=%d denom=%d\n", *numerator, *denominator);
do { scanf(" %c", &discard); }  while(discard != '\n');
printf("num=%d denom=%d\n", *numerator, *denominator);
```

第一個 printf 有順利印出結果, 但是第二個就沒有印出

所以問題幾乎已經確定就是兩個 **printf** 中間的 **do while** 迴圈吃掉了所有的輸入

所以第二個 printf 根本沒有執行到

如果這個時候還看不出來問題在哪裡, 繼續下面的測試

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

加上方括號可以比較容易看出來是這一列的輸出
這時候你如果輸入

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

加上方括號可以比較容易看出來是這一列的輸出
這時候你如果輸入

```
1/2 operand 1
+
3/4
```

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

加上方括號可以比較容易看出來是這一列的輸出
這時候你如果輸入

```
1/2 operand 1
+
3/4
```

會看到

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

加上方括號可以比較容易看出來是這一列的輸出

這時候你如果輸入

```
1/2 operand 1
+
3/4
```

會看到

```
1 / 2 operand 1
num=1 denom=2
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

步驟四：觀察更細部的表現

- 在迴圈內加上 printf 把「讀到的 discard 字元」列印出來

```
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while (discard != '\n');
```

加上方括號可以比較容易看出來是這一列的輸出

這時候你如果輸入

```
1/2 operand 1
+
3/4
```

會看到

```
1 / 2 operand 1
num=1 denom=2
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

輸入的 operand 1,\n,+,\n,3,/,

4,\n 字元通通被這個迴圈讀進來丟棄掉, 而且迴圈的執行條件永遠成立, 迴圈完全無法停止, 後面的程式完全執行不到

這個時候需要仔細想

scanf(" %c", &discard);

到底做了什麼事情, 為什麼得不到希望的效果?

這個時候需要仔細想

scanf(" %c", &discard);

到底做了什麼事情, 為什麼得不到希望的效果?

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
do { scanf("  %c", &discard); }  while(discard != '\n');
```

這個時候需要仔細想

scanf(" %c", &discard);

到底做了什麼事情, 為什麼得不到希望的效果?

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
do { scanf("  %c", &discard); }  while(discard != '\n');
```

輸入

1 / 2 operand 1 '\n'

時預期的效果是

這個時候需要仔細想

scanf(" %c", &discard);

到底做了什麼事情, 為什麼得不到希望的效果?

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
do { scanf("  %c", &discard); }  while(discard != '\n');
```

輸入

1 / 2 operand 1 '\n'

時預期的效果是

1 ==> *numerator
/ ==> slash
2 ==> *denominator
"operator 1 \n" 一個一個字元讀到 discard 變數中, 直到最後 '\n' 字元時迴圈結束

看到程式

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
printf("num=%d denom=%d\n", *numerator, *denominator);
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while(discard != '\n');
```

看到程式

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
printf("num=%d denom=%d\n", *numerator, *denominator);
do { scanf("  %c", &discard);
      printf("[%c]", discard);
}  while(discard != '\n');
```

印出 num=1 denom=2 時, 可以確定前半段是正確的, 後半段的迴圈並沒有如預期停在 **\n** 字元的地方而一直讀進所有的輸入字元

看到程式

```
nitems = scanf("%d %c%d", numerator , &slash , denominator);
printf("num=%d denom=%d\n", *numerator, *denominator);
do { scanf("  %c", &discard);
       printf("[%c]", discard);
}  while(discard != '\n');
```

印出 num=1 denom=2 時, 可以確定前半段是正確的, 後半段的迴圈並沒有如預期停在 **\n** 字元的地方而一直讀進所有的輸入字元

原因是 scanf 的格式轉換命令出的問題, 命令字串<space>%c 包含兩個命令 **<space>** 以及 **%c**

其中 <space> 命令會跳過輸入串流中所有的 white space (空格, \t, \n); %c 讀取一個字元

所以當 scanf() 函式看到如下的輸入

所以當 scanf() 函式看到如下的輸入

```
 operand 1\n
+\n
3/4\n
```

所以當 scanf() 函式看到如下的輸入

operand 1\n
+\n
3/4\n

時 discard 變數裡會得到下面方括號中的字元

所以當 scanf() 函式看到如下的輸入

```
 operand 1\n
+\n
3/4\n
```

時 discard 變數裡會得到下面方括號中的字元

```
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

所以當 scanf() 函式看到如下的輸入

```
 operand 1\n
+\n
3/4\n
```

時 discard 變數裡會得到下面方括號中的字元

```
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

其中並不包括 \n 字元, 因為被 <space> 命令跳過去了
所以更正的方法應該是刪除 <space> 命令成為

所以當 scanf() 函式看到如下的輸入

```
 operand 1\n
+\n
3/4\n
```

時 discard 變數裡會得到下面方括號中的字元

```
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

其中並不包括 \n 字元, 因為被 <space> 命令跳過去了
所以更正的方法應該是刪除 <space> 命令成為

```
scanf("%c", &discard);
```

所以當 scanf() 函式看到如下的輸入

```
 operand 1\n
+\n
3/4\n
```

時 discard 變數裡會得到下面方括號中的字元

```
[o][p][e][r][a][n][d][1]+
[+]3/4
[3][/][4]
```

其中並不包括 \n 字元, 因為被 <space> 命令跳過去了
所以更正的方法應該是刪除 <space> 命令成為

```
scanf("%c", &discard);
```

重新編譯執行, 得到

1 / 2 operand 1
num=1 denom=2
[ ][o][p][e][r][a][n][d][ ][1][
]+
3/4
num=3 denom=4
[
]1/2 + 3/4 = 1382800/1666606706

1 / 2 operand 1
num=1 denom=2
[ ][o][p][e][r][a][n][d][ ][1][
]+
3/4
num=3 denom=4
[
]1/2 + 3/4 = 1382800/1666606706

注意看到上面的 [
]
代表的就是讀到了 **\n**, 迴圈結束,

1 / 2 operand 1
num=1 denom=2
[ ][o][p][e][r][a][n][d][ ][1][
]+
3/4
num=3 denom=4
[
]1/2 + 3/4 = 1382800/1666606706

注意看到上面的 [
]
代表的就是讀到了 \n, 迴圈結束,

然後呼叫第二次的 scan_fractions(), 裡面印出
num=3 denom=4

步驟五

- 還原剛才在步驟二裡註解掉的 switch 邏輯
- 註解掉在步驟三步驟四裡加入的 printf 之後重新執行

步驟五

- 還原剛才在步驟二裡註解掉的 switch 邏輯
- 註解掉在步驟三步驟四裡加入的 printf 之後重新執行

```
1 /2
+
3/4
1/2 + 3/4 = 13441664/381243414
```

步驟五

- 還原剛才在步驟二裡註解掉的 switch 邏輯
- 註解掉在步驟三步驟四裡加入的 printf 之後重新執行

  1 /2
  +
  3/4
  1/2 + 3/4 = 13441664/381243414
- 這個加法的結果還是不對的, 再試一下減法

步驟五

- 還原剛才在步驟二裡註解掉的 switch 邏輯
- 註解掉在步驟三步驟四裡加入的 printf 之後重新執行

```
1 /2
+
3/4
1/2 + 3/4 = 13441664/381243414
```

- 這個加法的結果還是不對的, 再試一下減法

```
1/2
-
3/4
1/2 - 3/4 = 13441664/1844937886
```

步驟五

- 還原剛才在步驟二裡註解掉的 switch 邏輯
- 註解掉在步驟三步驟四裡加入的 printf 之後重新執行

```
1 /2
+
3/4
1/2 + 3/4 = 13441664/381243414
```

- 這個加法的結果還是不對的, 再試一下減法

```
1/2
-
3/4
1/2 - 3/4 = 13441664/1844937886
```

- 結果還是不對的, 但是分子是一樣的 (也許會有幫助)

看一下加法執行的內容, 目前相當於

看一下加法執行的內容, 目前相當於

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
add_fractions(numerator, denominator, numerator2, denominator2,
                    &num_ansp, &denom_ansp);
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator, op,
        numerator2, denominator2, nump, denomp);
```

看一下加法執行的內容, 目前相當於

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
add_fractions(numerator, denominator, numerator2, denominator2,
                    &num_ansp, &denom_ansp);
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator, op,
        numerator2, denominator2, nump, denomp);
```

前面幾個步驟裡已經確定 scan_fractions() 讀入正確的
numerator, denominator, numerator2, denominator2,

看一下加法執行的內容, 目前相當於

```
scan_fractions(&numerator, &denominator);
get_operator(&op);
scan_fractions(&numerator2, &denominator2);
add_fractions(numerator, denominator, numerator2, denominator2,
                    &num_ansp, &denom_ansp);
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
printf("%d/%d %c %d/%d = %d/%d\n", numerator, denominator, op,
        numerator2, denominator2, nump, denomp);
```

前面幾個步驟裡已經確定 scan_fractions() 讀入正確的 numerator, denominator, numerator2, denominator2,

所以現在應該要加上 printf 敘述查查看 **add_fractions()** 以及 **reduce_fraction()** 的結果是不是符合預期

重新執行一次
如果輸入

1/2
+
3/4

重新執行一次
如果輸入

1/2
+
3/4

預期

重新執行一次
如果輸入

1/2
+
3/4

預期

在執行完 add_fractions() 之後
num_ansp 應該要得到 1*4+3*2 = 10
denom_ansp 應該要得到 2*4 = 8

重新執行一次
如果輸入

1/2
+
3/4

預期

在執行完 add_fractions() 之後
num_ansp 應該要得到 1*4+3*2 = 10
denom_ansp 應該要得到 2*4 = 8

在執行完 reduce_fraction() 之後
nump 應該要是 5
denomp 應該要是 4

重新執行一次
如果輸入

1/2
+
3/4

預期

在執行完 add_fractions() 之後
num_ansp 應該要得到 1*4+3*2 = 10
denom_ansp 應該要得到 2*4 = 8

在執行完 reduce_fraction() 之後
nump 應該要是 5
denomp 應該要是 4

從前面執行結果可以看到顯然都是不如預期的
加上如下的 printf 敘述來列印 num_ansp 以及 denom_ansp

重新執行一次
如果輸入

1/2
+
3/4

預期

在執行完 add_fractions() 之後
num_ansp 應該要得到 1*4+3*2 = 10
denom_ansp 應該要得到 2*4 = 8

在執行完 reduce_fraction() 之後
nump 應該要是 5
denomp 應該要是 4

從前面執行結果可以看到顯然都是不如預期的
加上如下的 printf 敘述來列印 num_ansp 以及 denom_ansp

```
add_fractions(numerator, denominator, numerator2, denominator2,
                    &num_ansp, &denom_ansp);
printf("after add_fractions()  %d/%d\n", num_ansp, denom_ansp);
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

重新執行, 看到

重新執行, 看到

1/2
+
3/4
after add_fractions() 10/8
1/2 + 3/4 = 12851840/-1177390635

重新執行, 看到

1/2
+
3/4
after add_fractions() 10/8
1/2 + 3/4 = 12851840/-1177390635

這個時候可以推測 add_fractions() 應該沒有問題

問題可能就在 reduce_fraction() 裡面

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

```
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

```
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

看資料的流向可以推測 reduce_fraction() 應該是處理後面兩個參數 (**num_ansp**, **denom_ansp**) 裡的資料, 放到前面兩個參數 (**nump**, **denomp**) 裡面, 下面是 reduce_fraction() 函式的內容

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

```
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

看資料的流向可以推測 reduce_fraction() 應該是處理後面兩個參數 (**num_ansp**, **denom_ansp**) 裡的資料, 放到前面兩個參數 (**nump**, **denomp**) 裡面, 下面是 reduce_fraction() 函式的內容

```
void reduce_fraction(int *nump, int *denomp,
                     int *num_ansp, int *denom_ansp) {
    int q = find_gcd(*nump, *denomp);
    *num_ansp = *nump / q;
    *denom_ansp = *denomp / q;
}
```

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

```
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

看資料的流向可以推測 reduce_fraction() 應該是處理後面兩個參數 (**num_ansp**, **denom_ansp**) 裡的資料, 放到前面兩個參數 (**nump**, **denomp**) 裡面, 下面是 reduce_fraction() 函式的內容

```
void reduce_fraction(int *nump, int *denomp,
                     int *num_ansp, int *denom_ansp) {
    int q = find_gcd(*nump, *denomp);      反了, 反了!!
    *num_ansp = *nump / q;
    *denom_ansp = *denomp / q;
}
```

步驟六

剛才在步驟五裡面知道 add_fractions() 完成以後結果存放在 **num_ansp** 以及 **denom_ansp** 裡面, 接著是下面的呼叫 reduce_fraction() 敘述

```
reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp);
```

看資料的流向可以推測 reduce_fraction() 應該是處理後面兩個參數 (**num_ansp**, **denom_ansp**) 裡的資料, 放到前面兩個參數 (**nump**, **denomp**) 裡面, 下面是 reduce_fraction() 函式的內容

```
void reduce_fraction(int *nump, int *denomp,
                     int *num_ansp, int *denom_ansp) {
    int q = find_gcd(*nump, *denomp);
    *num_ansp = *nump / q;
    *denom_ansp = *denomp / q;
}
```

反了, 反了!!

怎麼輸入是 **nump, denomp**
輸出是 **num_ansp, denom_ansp**
呢???

這個函式計算前面兩個參數的最大公因數, 除掉以後放在後面兩個參數裡, 和剛才分析呼叫 reduce_fraction() 的方法不一致, 把輸入和輸出弄反了

這個函式計算前面兩個參數的最大公因數, 除掉以後放在後面兩個參數裡, 和剛才分析呼叫 reduce_fraction() 的方法不一致, 把輸入和輸出弄反了

reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp); 是把 (nump, denomp) 裡的資料化簡以後放在 (num_ansp, denom_ansp) 裡

這個函式計算前面兩個參數的最大公因數, 除掉以後放在後面兩個參數裡, 和剛才分析呼叫 reduce_fraction() 的方法不一致, 把輸入和輸出弄反了

reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp); 是把 (nump, denomp) 裡的資料化簡以後放在 (num_ansp, denom_ansp) 裡

因為 (nump, denomp) 裡是沒有意義的數值, 所以做完 reduce_fraction() 以後把原本在 (num_ansp, denom_ansp) 裡有意義的資料都毀了, 所以一直看到執行的結果很奇怪

這個函式計算前面兩個參數的最大公因數, 除掉以後放在後面兩個參數裡, 和剛才分析呼叫 reduce_fraction() 的方法不一致, 把輸入和輸出弄反了

reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp); 是把 (nump, denomp) 裡的資料化簡以後放在 (num_ansp, denom_ansp) 裡

因為 (nump, denomp) 裡是沒有意義的數值, 所以做完 reduce_fraction() 以後把原本在 (num_ansp, denom_ansp) 裡有意義的資料都毀了, 所以一直看到執行的結果很奇怪

修改的方法自然是「把參數 1,2 和參數 3,4 對調過來」

這個函式計算前面兩個參數的最大公因數, 除掉以後放在後面兩個參數裡, 和剛才分析呼叫 reduce_fraction() 的方法不一致, 把輸入和輸出弄反了

reduce_fraction(&nump, &denomp, &num_ansp, &denom_ansp); 是把 (nump, denomp) 裡的資料化簡以後放在 (num_ansp, denom_ansp) 裡

因為 (nump, denomp) 裡是沒有意義的數值, 所以做完 reduce_fraction() 以後把原本在 (num_ansp, denom_ansp) 裡有意義的資料都毀了, 所以一直看到執行的結果很奇怪

修改的方法自然是「把參數 1,2 和參數 3,4 對調過來」

```
reduce_fraction(&num_ansp, &denom_ansp, &nump, &denomp);
```

重新執行一次, 終於得到正確的結果了

重新執行一次, 終於得到正確的結果了

1/2
+
3/4
1/2 + 3/4 = 5/4

重新執行一次, 終於得到正確的結果了

1/2
+
3/4
1/2 + 3/4 = 5/4

不過這個函式的**參數型態**可以順道修一修, 只有輸出的參數 (num_ansp, denom_ansp) 需要傳記憶體位址, 輸入的參數 (nump, denomp) 傳數值進去就夠了

重新執行一次, 終於得到正確的結果了

1/2
+
3/4
1/2 + 3/4 = 5/4

不過這個函式的**參數型態**可以順道修一修, 只有輸出的參數 (num_ansp, denom_ansp) 需要傳記憶體位址, 輸入的參數 (nump, denomp) 傳數值進去就夠了

```
void reduce_fraction(int nump, int denomp,
                     int *num_ansp, int *denom_ansp) {
    int  q = find_gcd(nump, denomp);
    *num_ansp = nump / q;
    *denom_ansp = denomp / q;
}
```

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

依序測試
1/2
-
3/4

1/2
*
3/4

1/2
/
3/4

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

依序測試　　發現除法結果是不對的, 除法的程式碼如下:

```
1/2
-
3/4

1/2
*
3/4

1/2
/
3/4
```

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

依序測試

1/2
-
3/4

1/2
*
3/4

1/2
/
3/4

發現除法結果是不對的, 除法的程式碼如下:

```
mult_fractions(numerator, denominator,
                        numerator2, denominator2,
                        &num_ansp, &denom_ansp);
```

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

依序測試

1/2

-

3/4

1/2

*

3/4

1/2

/

3/4

發現除法結果是不對的, 除法的程式碼如下:

```
mult_fractions(numerator, denominator,
                        numerator2, denominator2,
                        &num_ansp, &denom_ansp);
```

看到程式想用乘法來完成 numerator/denominator 除以 numerator2/denominator2, 但是應該要改成乘以 denominator2/numerator2, 也就是

另外也發現原本程式處理 +,-,*,/ 的 switch 敘述中, 呼叫 reduce_fraction() 函式時參數是完全一樣的, 所以可以移到 switch 敘述之後

依序測試

1/2
-
3/4

發現除法結果是不對的, 除法的程式碼如下:

```
mult_fractions(numerator, denominator,
                        numerator2, denominator2,
                        &num_ansp, &denom_ansp);
```

1/2
*
3/4

看到程式想用乘法來完成 numerator/denominator 除以 numerator2/denominator2, 但是應該要改成乘以 denominator2/numerator2, 也就是

1/2
/
3/4

```
mult_fractions(numerator, denominator,
                        denominator2, numerator2,
                        &num_ansp, &denom_ansp);
```

- 呼, 終於完成 XD

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩
- 還好不是求助於別人!! 看到網路上好多人在學程式時都是自己捅出漏子, 還怪系統不好, 像被別人迫害一樣到處求解

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩
- 還好不是求助於別人!! 看到網路上好多人在學程式時都是自己捅出漏子, 還怪系統不好, 像被別人迫害一樣到處求解
    - scanf 的格式轉換命令 <space> 和 %c, 功能還沒有弄清楚就上場打仗了,沒有先在其他地方測試過

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩
- 還好不是求助於別人!! 看到網路上好多人在學程式時都是自己捅出漏子, 還怪系統不好, 像被別人迫害一樣到處求解
    - scanf 的格式轉換命令 <space> 和 %c, 功能還沒有弄清楚就上場打仗了,沒有先在其他地方測試過
    - 變數的名稱沒有代表它的意義的話就很容易用錯

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩
- 還好不是求助於別人!! 看到網路上好多人在學程式時都是自己捅出漏子, 還怪系統不好, 像被別人迫害一樣到處求解
    - scanf 的格式轉換命令 <space> 和 %c, 功能還沒有弄清楚就上場打仗了,沒有先在其他地方測試過
    - 變數的名稱沒有代表它的意義的話就很容易用錯
    - 參數是否傳遞數值還是位址, 好像沒有很清楚

- 呼, 終於完成 XD
- 好喔, 像個偵探, 只是目標不是別人捅的漏子, 是自己給自己找的麻煩
- 還好不是求助於別人!! 看到網路上好多人在學程式時都是自己捅出漏子, 還怪系統不好, 像被別人迫害一樣到處求解
  - scanf 的格式轉換命令 <space> 和 %c, 功能還沒有弄清楚就上場打仗了,沒有先在其他地方測試過
  - 變數的名稱沒有代表它的意義的話就很容易用錯
  - 參數是否傳遞數值還是位址, 好像沒有很清楚
- 程式寫得好, 沒有閱讀的陷阱, 容易看得懂, 事先設計好檢查點, debug 的時間就會很短; 程式沒有設計好, 各個部份的複雜度比較高, 就會讓 debug 的時間拉長 (通常多花一點時間把程式邏輯寫清楚, 架構寫清楚的話, 後續節省的 debug 時間一定划得來, 重要的是你撰寫程式的信心會越來越強)

- 這個課程以及實習的目的就是在練習程式撰寫與調整校正的過程, 只要遵循我們所強調的「 設計程序式程式的基本方法」, 自己設計出來的程式, 自己知道每一個敘述希望完成的目標, 知道變數裡的資料應該要如何轉變, 就可以一個敘述一個敘述驗證, 也就可以很快地得到有預期表現的程式

- 這個課程以及實習的目的就是在練習程式撰寫與調整校正的過程, 只要遵循我們所強調的「 設計程序式程式的基本方法」, 自己設計出來的程式, 自己知道每一個敘述希望完成的目標, 知道變數裡的資料應該要如何轉變, 就可以一個敘述一個敘述驗證, 也就可以很快地得到有預期表現的程式

- 千萬不要以為我們練習的目標是要讓程式一打完就是正確的... 如果你到達這個等級, 你一定是馬克佐伯格的取代者, 別懷疑, 未來的人類全靠你了

- 這個課程以及實習的目的就是在練習程式撰寫與調整校正的過程, 只要遵循我們所強調的「 設計程序式程式的基本方法」, 自己設計出來的程式, 自己知道每一個敘述希望完成的目標, 知道變數裡的資料應該要如何轉變, 就可以一個敘述一個敘述驗證, 也就可以很快地得到有預期表現的程式

- 千萬不要以為我們練習的目標是要讓程式一打完就是正確的... 如果你到達這個等級, 你一定是馬克佐伯格的取代者, 別懷疑, 未來的人類全靠你了

- 在這個時間點一定要和你溝通一下, 一直都有同學以為程式一打完就應該已經是對的, 也盡力去尋找達成這種效果的方法, 然後發現除了拷貝之外, 好像沒有辦法在偶爾的練習裡輕鬆地達成, 因此覺得很挫折, 覺得自己不適合這個科系

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：
- 程式碼對齊

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：
- 程式碼對齊
- 使用變數時好好地命名, 讓每一個變數的名稱能夠看出來它裡面存放資料的意義

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：
- 程式碼對齊
- 使用變數時好好地命名, 讓每一個變數的名稱能夠看出來它裡面存放資料的意義
- 一個變數只有單一的用途

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：

- 程式碼對齊
- 使用變數時好好地命名, 讓每一個變數的名稱能夠看出來它裡面存放資料的意義
- 一個變數只有單一的用途
- 不要使用全域變數

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：

- 程式碼對齊
- 使用變數時好好地命名, 讓每一個變數的名稱能夠看出來它裡面存放資料的意義
- 一個變數只有單一的用途
- 不要使用全域變數
- 不要使用 goto 敘述

- 有了這樣錯誤的認知, 所以也不去練習「設計、觀察、分析與調整校正」的標準方法....這種挫折感其實是完完全全不必要的 (不過話說如果真的只打算付出很少的時間, 想要學會怎樣設計程式、怎樣分析程式、怎樣修改程式、怎樣校正程式的表現幾乎是不太可能的, 有可能「不適合的感覺」是真的...)
- 有沒有方法可以減少程式出錯的可能性呢? 當然有, 寫程式是把自己的作法用程式表達出來, 正常人都是很容易出錯的, 在這個轉換的過程裡面是有很多思考陷阱的, 也有很多時候會產生閱讀與理解的陷阱, 但是也常常可以藉由一些經驗法則來避免, 例如：

- 程式碼對齊
- 使用變數時好好地命名, 讓每一個變數的名稱能夠看出來它裡面存放資料的意義
- 一個變數只有單一的用途
- 不要使用全域變數
- 不要使用 goto 敘述
- 盡量使用函式來組織程式, 函式的名稱仔細命名

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性
- 程式在設計時儘量不要和實際世界中系統運作模型差異太大

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性
- 程式在設計時儘量不要和實際世界中系統運作模型差異太大
- 減少隱含的假設

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性
- 程式在設計時儘量不要和實際世界中系統運作模型差異太大
- 減少隱含的假設
- 口語上怎麼描述的, 程式就寫成那樣, 不要刻意轉換

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性
- 程式在設計時儘量不要和實際世界中系統運作模型差異太大
- 減少隱含的假設
- 口語上怎麼描述的, 程式就寫成那樣, 不要刻意轉換
- 運用結構取代平行陣列, 不使用不熟悉無法掌握的語法

- 用函式來降低程式邏輯的複雜度, 減少迴圈和條件判斷的層數
- 函式裡面應該處理邏輯上面相關而且邏輯上同一層的事情, 每一個函式的長度不要超過 30 列, 每一個變數使用前需要初始化
- 儘量運用 const 來避免沒有留意到的變數修改
- 狀態變數該合併時要合併
- 運用陣列語法來取代指標增加程式的可讀性
- 使用一般性的資料結構來增加程式的可讀性
- 程式在設計時儘量不要和實際世界中系統運作模型差異太大
- 減少隱含的假設
- 口語上怎麼描述的, 程式就寫成那樣, 不要刻意轉換
- 運用結構取代平行陣列, 不使用不熟悉無法掌握的語法
- 基本原則是...寫出讓自己容易了解、容易解釋、可以預測、容易修改的程式, 而不只是正確運作的程式