基礎程式設計技巧(一)
程式與計算

許胖

板燒高中

July 7, 2015

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行爲
7. 其他運算子
8. 結論

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行為
7. 其他運算子
8. 結論

寫程式的差別

寫程式的差別

寫出一個完整的程式 ...

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，爲什麼會執行

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，爲什麼會執行
3. 什麼時候會什麼狀況，然後判斷出來、修正 (也就是 debug)

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，為什麼會執行
3. 什麼時候會什麼狀況，然後判斷出來、修正 (也就是 debug)
4. 用適當的工具解決問題

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，為什麼會執行
3. 什麼時候會什麼狀況，然後判斷出來、修正 (也就是 debug)
4. 用適當的工具解決問題
5. ... 族繁不及~~被宰~~備載

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，爲什麼會執行
3. 什麼時候會什麼狀況，然後判斷出來、修正 (也就是 debug)
4. 用適當的工具解決問題
5. ... 族繁不及~~被宰~~備載

- 以上就是培訓目標！

寫程式的差別

寫出一個完整的程式 ...

- 只要照著講義、照著書打一打，就可以動了。

寫「好」一個程式 ...

1. 要了解資料怎麼儲存在電腦中
2. 程式怎麼開始執行，爲什麼會執行
3. 什麼時候會什麼狀況，然後判斷出來、修正 (也就是 debug)
4. 用適當的工具解決問題
5. ... 族繁不及~~被宰~~備載

- 以上就是培訓目標！
- 就是要讓大家熟悉基本的 C++ 語法，以及學會基本的 coding 技巧。

關於許胖講義 ...

關於許胖講義 ...

給參與「演算法競賽」的人 ...

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

- 寫程式不是只有演算法比賽，生命也不是只有一個出口
- 越往這個領域深入，就會看到更多無盡的事物

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

- 寫程式不是只有演算法比賽，生命也不是只有一個出口
- 越往這個領域深入，就會看到更多無盡的事物
  - 寫遊戲引擎

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

- 寫程式不是只有演算法比賽，生命也不是只有一個出口
- 越往這個領域深入，就會看到更多無盡的事物
    - 寫遊戲引擎
    - 網頁設計

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

- 寫程式不是只有演算法比賽，生命也不是只有一個出口
- 越往這個領域深入，就會看到更多無盡的事物
  - 寫遊戲引擎
  - 網頁設計
  - 手機 App

關於許胖講義 ...

給參與「演算法競賽」的人 ...

1. 使用一個「有效」的方法解決問題
2. 不僅如此，還要知道不同工具使用上的優缺點
3. 手爆出很多 code，勇往直前
4. 進到 TOI 二階，保送大學

- 寫程式不是只有演算法比賽，生命也不是只有一個出口
- 越往這個領域深入，就會看到更多無盡的事物
  - 寫遊戲引擎
  - 網頁設計
  - 手機 App
  - 韌體 coding

關於這份投影片

目標

1. 知道 C++ 的語法皆爲「運算」

關於這份投影片

目標

1. 知道 C++ 的語法皆爲「運算」
2. 各運算子的用法及特性

目標

1. 知道 C++ 的語法皆爲「運算」
2. 各運算子的用法及特性
3. 注意未定義行爲

關於這份投影片

目標

1. 知道 C++ 的語法皆爲「運算」
2. 各運算子的用法及特性
3. 注意未定義行爲

XD

- 希望各位在之後的內容都要動手快樂寫程式 XD！

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行爲
7. 其他運算子
8. 結論

基本程式架構

C++ 基本架構

基本程式架構

C++ 基本架構

```
#include <iostream>
using namespace std;
int main() {
}
```

基本程式架構

C++ 基本架構

```
#include <iostream>
using namespace std;
int main() {
}
```

註

- 怎麼理解？

基本程式架構

C++ 基本架構

```
#include <iostream>
using namespace std;
int main() {
}
```

註

- 怎麼理解？ 不需要理解，我們先記起來。

基本程式架構

C++ 基本架構

```
#include <iostream>
using namespace std;
int main() {
}
```

註

- 怎麼理解？ 不需要理解，我們先記起來。
- 基本上程式的內容都寫在大括號中。

基本程式架構

C++ 基本架構

```
#include <iostream>
using namespace std;
int main() {
}
```

註

- 怎麼理解？ 不需要理解，我們先記起來。
- 基本上程式的內容都寫在大括號中。
- 裡面每個符號都要一樣 (分號也是)。

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「`cout << 1;`」，會發生什麼事？

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「`cout << 1;`」，會發生什麼事？
2. 還不清楚的話，可以在更下一行加上「`system("PAUSE");`」，在觀察看看。

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「`cout << 1;`」，會發生什麼事？
2. 還不清楚的話，可以在更下一行加上「`system("PAUSE");`」，在觀察看看。

註

- cout 是「輸出」符號，你要輸出的東西用「<<」串連。

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「`cout << 1;`」，會發生什麼事？
2. 還不清楚的話，可以在更下一行加上「`system("PAUSE");`」，在觀察看看。

註

- `cout` 是「輸出」符號，你要輸出的東西用「`<<`」串連。
- `system("PAUSE");` 代表「暫停」的意思。

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「cout << 1;」，會發生什麼事？
2. 還不清楚的話，可以在更下一行加上「system("PAUSE");」，在觀察看看。

註

- cout 是「輸出」符號，你要輸出的東西用「<<」串連。
- system("PAUSE"); 代表「暫停」的意思。
  - 因爲沒加上這行，程式就會直接執行結束。

程式的輸出

輸出

1. 試著在剛剛的大括號中打上「cout << 1;」，會發生什麼事？
2. 還不清楚的話，可以在更下一行加上「system("PAUSE");」，在觀察看看。

註

- cout 是「輸出」符號，你要輸出的東西用「<<」串連。
- system("PAUSE"); 代表「暫停」的意思。
  - 因爲沒加上這行，程式就會直接執行結束。
  - 加上這行，程式會在這裡「等你」。

注意事項

輸出

1. 如果改成「cout << 1」(去掉分號) 會發生什麼結果？

注意事項

輸出

1. 如果改成「cout << 1」(去掉分號) 會發生什麼結果？

註

- 「分號」對 C++ 而言代表「一個句子的結束」，因此當一行指令結束就要加分號。

注意事項

輸出

1. 如果改成「cout << 1」(去掉分號) 會發生什麼結果？
2. 試試看「cout << 1 << 2;」，和你所想的有何不同？

註

- 「分號」對 C++ 而言代表「一個句子的結束」，因此當一行指令結束就要加分號。

注意事項

輸出

1. 如果改成「cout << 1」(去掉分號) 會發生什麼結果？
2. 試試看「cout << 1 << 2;」，和你所想的有何不同？

註

- 「分號」對 C++ 而言代表「一個句子的結束」，因此當一行指令結束就要加分號。
- << 可以串很多東西一起輸出。

注意事項

輸出

1. 如果改成「cout << 1」(去掉分號) 會發生什麼結果？
2. 試試看「cout << 1 << 2;」，和你所想的有何不同？
3. 那麼「cout << 1 << "␣"<< 2;」呢？

註

- 「分號」對 C++ 而言代表「一個句子的結束」，因此當一行指令結束就要加分號。
- << 可以串很多東西一起輸出。
- "␣" 是雙引號中間夾著一個「空白」，要注意！

換行「符號」

換行

1. 試試看「`cout << 1 << 2 << endl;`」，和「`cout << 1 << 2;`」有什麼不同呢？

換行「符號」

換行

1. 試試看「`cout << 1 << 2 << endl;`」，和「`cout << 1 << 2;`」有什麼不同呢？
2. 如果看不出來，試試看「`cout << 1 << endl << 2;`」。

換行「符號」

換行

1. 試試看「`cout << 1 << 2 << endl;`」，和「`cout << 1 << 2;`」有什麼不同呢？
2. 如果看不出來，試試看「`cout << 1 << endl << 2;`」。

註

- 「`endl`」代表換行符號，輸出中很好用。

變數

變數

- 和數學「變數」的概念不太一樣

變數

變數

- 和數學「變數」的概念不太一樣
- 程式的變數像是「容器」，可以裝資料。

變數

變數

- 和數學「變數」的概念不太一樣
- 程式的變數像是「容器」，可以裝資料。
- C++ 裡，每個容器都要先講好用途，這個步驟叫做「宣告」。

變數

變數

- 和數學「變數」的概念不太一樣
- 程式的變數像是「容器」，可以裝資料。
- C++ 裡，每個容器都要先講好用途，這個步驟叫做「宣告」。

宣告變數

```
int x;
```

變數

變數

- 和數學「變數」的概念不太一樣
- 程式的變數像是「容器」，可以裝資料。
- C++ 裡，每個容器都要先講好用途，這個步驟叫做「宣告」。

宣告變數

```
int x;
```

註

- 宣告就是幫變數取名字，此例將變數取名爲「x」。

變數

- 和數學「變數」的概念不太一樣
- 程式的變數像是「容器」，可以裝資料。
- C++ 裡，每個容器都要先講好用途，這個步驟叫做「宣告」。

宣告變數

```
int x;
```

註

- 宣告就是幫變數取名字，此例將變數取名爲「x」。
- 「int」代表的意義是「整數」，規定變數 x 只能裝整數。

變數的功用

把數字裝到變數

```
#include <iostream>
using namespace std;
int main() {
  int x;                 // 宣告變數 x
  x = 5;                 // 把整數 5 裝進 x 裡面
  cout << x << endl;     // 印出變數 x 存的值
}
```

變數的功用

把數字裝到變數

```
#include <iostream>
using namespace std;
int main() {
  int x;                  // 宣告變數 x
  x = 5;                  // 把整數 5 裝進 x 裡面
  cout << x << endl;      // 印出變數 x 存的值
}
```

註

- 「int」宣告變數可以裝整數之外，還有很多不同的種類，以後會慢慢介紹。

變數的功用

把數字裝到變數

```
#include <iostream>
using namespace std;
int main() {
  int x;                    // 宣告變數 x
  x = 5;                    // 把整數 5 裝進 x 裡面
  cout << x << endl; // 印出變數 x 存的值
}
```

註

- 「int」宣告變數可以裝整數之外，還有很多不同的種類，以後會慢慢介紹。
- 「x = 5;」這行不要和數學中的「等於」搞混。

練習看看

練習

若把上個投影片「x = 5;」改成

1. 「x = 5.0;」會發生什麼事？

練習看看

練習

若把上個投影片「x = 5;」改成

1. 「x = 5.0;」會發生什麼事？
2. 「x = 0.5;」呢？

練習看看

練習

若把上個投影片「x = 5;」改成

1. 「x = 5.0;」會發生什麼事？
2. 「x = 0.5;」呢？
3. 那改成「5 = x;」呢？

練習看看

練習

若把上個投影片「`x = 5;`」改成

1. 「`x = 5.0;`」會發生什麼事？
2. 「`x = 0.5;`」呢？
3. 那改成「`5 = x;`」呢？

註

- 這些練習目的是要讓你了解問題出現時的現象，了解出問題的原因才有辦法 debug

練習看看

練習

若把上個投影片「x = 5;」改成

1. 「x = 5.0;」會發生什麼事？
2. 「x = 0.5;」呢？
3. 那改成「5 = x;」呢？

註

- 這些練習目的是要讓你了解問題出現時的現象，了解出問題的原因才有辦法 debug
- 為什麼會出現這些現象我們繼續下去就知道了

宣告多個變數

宣告兩個整數

- 可以寫成這樣：

```
int a;
int b;
```

宣告多個變數

宣告兩個整數

- 可以寫成這樣：

```
int a;
int b;
```

- 更可以簡化成這樣：

```
int a, b;
```

宣告多個變數

宣告兩個整數

- 可以寫成這樣：

```
int a;
int b;
```

- 更可以簡化成這樣：

```
int a, b;
```

宣告三個整數

```
int a, b, c;
```

如果容器不塞東西呢 ...

變數不塞整數進去

```
#include <iostream>
using namespace std;
int main() {
  int x;                   // 宣告變數 x
  cout << x << endl; // 印出變數 x 存的值
}
```

如果容器不塞東西呢 ...

變數不塞整數進去

```
#include <iostream>
using namespace std;
int main() {
  int x;                    // 宣告變數 x
  cout << x << endl;  // 印出變數 x 存的值
}
```

練習

1. 執行看看，發生什麼事？

如果容器不塞東西呢 ...

變數不塞整數進去

```
#include <iostream>
using namespace std;
int main() {
  int x;                    // 宣告變數 x
  cout << x << endl; // 印出變數 x 存的值
}
```

練習

1. 執行看看，發生什麼事？
2. 再執行幾次，又會發生什麼事呢？

初始化

初始化

- C++ 中，所有變數都要自己去初始化。

初始化

初始化

- C++ 中，所有變數都要自己去初始化。
  - 例如：`x = 5;`，把整數 5 丟給 x 等等。

初始化

初始化

- C++ 中，所有變數都要自己去初始化。
  - 例如：x = 5;，把整數 5 丟給 x 等等。
- 沒有初始化過的變數，裡面裝的資料是不確定的。

初始化

初始化

- C++ 中，所有變數都要自己去初始化。
  - 例如：`x = 5;`，把整數 5 丟給 x 等等。
- 沒有初始化過的變數，裡面裝的資料是不確定的。
  - 或許你很幸運看到 x 都是 0

初始化

初始化

- C++ 中，所有變數都要自己去初始化。
  - 例如：x = 5;，把整數 5 丟給 x 等等。
- 沒有初始化過的變數，裡面裝的資料是不確定的。
  - 或許你很幸運看到 x 都是 0
  - 但那只是恰巧而已。

程式的輸入

輸入

執行以下程式

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

會發生什麼事呢？

程式的輸入

輸入

執行以下程式

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

會發生什麼事呢？

練習

如果沒發生什麼事，試著輸入「1」再按 enter 鍵，會發生什麼事呢？

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

練習 (續)

- 如果輸入「5.0」再按 enter 鍵呢？

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

練習 (續)

- 如果輸入「5.0」再按 enter 鍵呢？
- 如果輸入「0.5」再按 enter 鍵呢？

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

練習 (續)

- 如果輸入「5.0」再按 enter 鍵呢？
- 如果輸入「0.5」再按 enter 鍵呢？
- 如果輸入「XD」再按 enter 鍵呢？

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

練習 (續)

- 如果輸入「5.0」再按 enter 鍵呢？
- 如果輸入「0.5」再按 enter 鍵呢？
- 如果輸入「XD」再按 enter 鍵呢？

多變數輸入

```
int x, y;
cin >> x >> y;
```

程式的輸入

輸入「符號」

- 「cin」代表輸入符號，可以輸入後面變數的資料。
  - 此例中，x 是整數，因此可以輸入一個整數。
  - cin 的 >> 不要和 cout 的 << 搞混。

練習 (續)

- 如果輸入「5.0」再按 enter 鍵呢？
- 如果輸入「0.5」再按 enter 鍵呢？
- 如果輸入「XD」再按 enter 鍵呢？

多變數輸入

```
int x, y;
cin >> x >> y;
```

- 不要在輸入中加入「endl」。

資料型態

資料型態

- 有裝整數的容器，那麼當然也可以宣告裝「小數點」的容器啦！

資料型態

資料型態

- 有裝整數的容器，那麼當然也可以宣告裝「小數點」的容器啦！
- 這些不同用途的容器我們稱爲「資料型態」。

資料型態

資料型態

- 有裝整數的容器，那麼當然也可以宣告裝「小數點」的容器啦！
- 這些不同用途的容器我們稱為「資料型態」。

關鍵字	意義	備註
bool	布林值	只有 true 和 false
int	整數	
long long	長整數	存比較大的整數，以後會介紹
double	浮點數	也就是小數點

Table: 資料型態

資料型態

資料型態

- 有裝整數的容器，那麼當然也可以宣告裝「小數點」的容器啦！
- 這些不同用途的容器我們稱爲「資料型態」。

關鍵字	意義	備註
bool	布林值	只有 true 和 false
int	整數	
long long	長整數	存比較大的整數，以後會介紹
double	浮點數	也就是小數點

Table: 資料型態

註

詳細內容之後再介紹，先來用看看這些東西。

布林值

布林值

- 一種資料型態，只拿來裝兩種數值：「`true`」和「`false`」。

布林值

布林值

- 一種資料型態，只拿來裝兩種數值：「true」和「false」。

宣告

```
bool b;
```

布林值

布林值

- 一種資料型態，只拿來裝兩種數值：「true」和「false」。

宣告

```
bool b;
```

注意

- 兩種不同的宣告不能用「逗號」隔開：

```
int a, bool b;
```

布林值

布林值

- 一種資料型態，只拿來裝兩種數值：「true」和「false」。

宣告

```
bool b;
```

注意

- 兩種不同的宣告不能用「逗號」隔開：

```
int a, bool b;
```

- 逗號有特殊意義，不要想成一般的「逗號」。

賦值

定義

- 將一個「數值」裝進一個變數中，稱爲賦值。

賦值

定義

- 將一個「數值」裝進一個變數中，稱為賦值。
- 例如，之前把整數 5 裝進整數變數 x 中：

```
int x;
x = 5;
```

賦值

定義

- 將一個「數值」裝進一個變數中，稱爲賦值。
- 例如，之前把整數 5 裝進整數變數 x 中：

```
int x;
x = 5;
```

賦值簡化

- 變數宣告和賦值可以寫在一起：

```
int x = 5; // 宣告一個整數變數 x 並且把 5 裝進去
```

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

1. `b = true;`

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

1. `b = true;`
2. `b = false;`

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

1. `b = true;`
2. `b = false;`
3. `b = 2;`

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

1. `b = true;`
2. `b = false;`
3. `b = 2;`
4. `b = 0;`

布林值

練習

```
bool b;
cout << b << endl;
```

對程式碼的 b 做以下賦值，會發生什麼事？

1. `b = true;`
2. `b = false;`
3. `b = 2;`
4. `b = 0;`
5. `b = -1;`

布林值的重要觀念

觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。

布林值的重要觀念

觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。
- 「0」會被當做「`false`」，印出時會印出「0」。

布林值的重要觀念

觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。
- 「0」會被當做「`false`」，印出時會印出「0」。

技巧

這個特性在之後會非常常用！大家要注意！

浮點數

- 先跳過 `long long`，先知道 `long long` 也是存整數就好。

浮點數

- 先跳過 `long long`，先知道 `long long` 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數

- 先跳過 long long，先知道 long long 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數宣告

```
double d;
```

浮點數

- 先跳過 long long，先知道 long long 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數宣告

```
double d;
```

賦值

- 把 1.0 丟給 d ⇒ `d = 1.0;`

浮點數

- 先跳過 long long，先知道 long long 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數宣告

```
double d;
```

賦值

- 把 1.0 丟給 d ⇒ `d = 1.0;`
- 把 0.5 丟給 d ⇒ `d = 0.5;`

浮點數

- 先跳過 long long，先知道 long long 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數宣告

```
double d;
```

賦值

- 把 1.0 丟給 d ⇒ d = 1.0;
- 把 0.5 丟給 d ⇒ d = 0.5;
  - 0.5 也可寫爲 d = .5;

浮點數

- 先跳過 long long，先知道 long long 也是存整數就好。
- 謎之音：「那幹嘛現在說= =」

浮點數宣告

```
double d;
```

賦值

- 把 1.0 丟給 d ⇒ d = 1.0;
- 把 0.5 丟給 d ⇒ d = 0.5;
  - 0.5 也可寫爲 d = .5;
- 18.23e5 ⇒ 代表 18.23×10^5 (科學記號)

大綱

1 簡介
2 程式架構
- 基本程式架構
- 輸出
- 變數
- 輸入
- 資料型態

3 算術運算子
- 運算性質
- 結合性與運算順序
- 整數除法與除零問題
- 應用：取餘數

4 比較和邏輯運算子
- 簡化規則
- 短路運算

5 位元運算子
- int 和 long long 的儲存形式
- 常用技巧：連續的 1
- 常用技巧：遮罩與指定位元
- Parity
- xor 性質

6 指定運算子
- 運算性質
- 未定義行爲

7 其他運算子
8 結論

算術運算子

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

算術運算子

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

註

- 不管運算順序和結合性，一般來說可以用五則運算來理解

算術運算子

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

註

- 不管運算順序和結合性，一般來說可以用五則運算來理解
- 只不過程式跟數學還是有差距 ...

算術運算子

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

註

- 不管運算順序和結合性，一般來說可以用五則運算來理解
- 只不過程式跟數學還是有差距 ...
  - 這個故事說來話長，我們先舉個簡單的例子吧！

舉個例子

$1 + 2 + 3 = ?$

舉個例子

$1 + 2 + 3 = ?$

- 答案：6。

舉個例子

$1 + 2 + 3 = ?$

- 答案：6。
- 爲什麼？(謎之音：「什麼爲什麼？」)

舉個例子

$1 + 2 + 3 = ?$

- 答案：6。
- 爲什麼？(謎之音：「什麼爲什麼？」)

定義

二元運算有一個運算子和兩個運算元，例如：

舉個例子

$1 + 2 + 3 = ?$

- 答案：6。
- 爲什麼？(謎之音：「什麼爲什麼？」)

定義

二元運算有一個運算子和兩個運算元，例如：

1. $1 + 2$：「+」稱爲「運算子」，「1」和「2」稱爲運算元 (我們常稱爲「被加數」和「加數」)。

舉個例子

$1+2+3=?$

- 答案：6。
- 爲什麼？(謎之音：「什麼爲什麼？」)

定義

二元運算有一個運算子和兩個運算元，例如：

1. $1+2$：「+」稱爲「運算子」，「1」和「2」稱爲運算元 (我們常稱爲「被加數」和「加數」)。
2. 我們可以知道「加減乘除餘」都是二元運算。

舉個例子

$1+2+3=?$

- 答案：6。
- 爲什麼？(謎之音：「什麼爲什麼？」)

定義

二元運算有一個運算子和兩個運算元，例如：

1. $1+2$：「+」稱爲「運算子」，「1」和「2」稱爲運算元 (我們常稱爲「被加數」和「加數」)。
2. 我們可以知道「加減乘除餘」都是二元運算。

- Well, 我們回到原來的問題 ...

回到原來的問題 ...

$1 + 2 + 3 = ?$

- 出現大麻煩啦！

回到原來的問題 ...

$1 + 2 + 3 = ?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1 + 2 + 3$」該怎麼辦呢？

回到原來的問題 ...

$1+2+3=?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1+2+3$」該怎麼辦呢？
- 解法：決定運算的方向。例如：

回到原來的問題 ...

$1+2+3=?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1+2+3$」該怎麼辦呢？
- 解法：決定運算的方向。例如：
  1. 先算 $1+2=3$，再算 $3+3=6$

回到原來的問題 ...

$1+2+3=?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1+2+3$」該怎麼辦呢？
- 解法：決定運算的方向。例如：
  1. 先算 $1+2=3$，再算 $3+3=6$
  2. 先算 $2+3=5$，再算 $1+5=6$

回到原來的問題 ...

$1 + 2 + 3 = ?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1 + 2 + 3$」該怎麼辦呢？
- 解法：決定運算的方向。例如：
  1. 先算 $1 + 2 = 3$，再算 $3 + 3 = 6$
  2. 先算 $2 + 3 = 5$，再算 $1 + 5 = 6$
- 謎之音：「那還不是一樣嘛？廢話= =」

回到原來的問題 ...

$1+2+3=?$

- 出現大麻煩啦！
  - 根據剛剛說的，加法只有兩個運算元，那麼「$1+2+3$」該怎麼辦呢？
- 解法：決定運算的方向。例如：
  1. 先算 $1+2=3$，再算 $3+3=6$
  2. 先算 $2+3=5$，再算 $1+5=6$
- 謎之音：「那還不是一樣嘛？廢話= =」

注意

決定運算方向對「計算機」而言意義重大！

再舉個例子 ...

$1 - 2 - 3 = ?$

再舉個例子 ...

$1-2-3=?$

- 我們直觀上會先算 $1-2=-1$，再算 $-1-3=-4$。

再舉個例子 ...

$1-2-3=?$

- 我們直觀上會先算 $1-2=-1$，再算 $-1-3=-4$。
- 因此 C++ 在設計上也會把加減乘除餘的結合性「設定」成從左到右算。

再舉個例子 ...

$1-2-3=?$

- 我們直觀上會先算 $1-2=-1$，再算 $-1-3=-4$。
- 因此 C++ 在設計上也會把加減乘除餘的結合性「設定」成從左到右算。

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

- 我們的運算規則：「先乘除餘，後加減」。

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

- 我們的運算規則：「先乘除餘，後加減」。
- 因此 C++ 發展出一套規則：運算順序

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

- 我們的運算規則：「先乘除餘，後加減」。
- 因此 C++ 發展出一套規則：運算順序
  - 運算順序小的優先運算

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

- 我們的運算規則：「先乘除餘，後加減」。
- 因此 C++ 發展出一套規則：運算順序
  - 運算順序小的優先運算
  - 若運算順序相同，則依照運算方向做計算。

萬一是四則運算呢？

$1 + 2 * 3 - 4 = ?$

- 我們的運算規則：「先乘除餘，後加減」。
- 因此 C++ 發展出一套規則：運算順序
  - 運算順序小的優先運算
  - 若運算順序相同，則依照運算方向做計算。

算術運算子	意義	運算順序	結合性
+	加法	6	左→右
-	減法	6	左→右
*	乘法	5	左→右
/	除法	5	左→右
%	取餘數	5	左→右

Table: 算術運算子

回到原來例子

$1 + 2 * 3 - 4 =?$

$1 + 2 * 3 - 4$

我們可以看到 $*$ 的運算順序最高

回到原來例子

$1 + 2 * 3 - 4 = ?$

$$
\begin{aligned}
& 1 + 2 * 3 - 4 && \text{我們可以看到 } * \text{ 的運算順序最高} \\
= & 1 + 6 - 4 && \text{加法和減法運算順序相同，依照結合性從左到右算}
\end{aligned}
$$

回到原來例子

$1 + 2 * 3 - 4 = ?$

$$
\begin{aligned}
& 1 + 2 * 3 - 4 && \text{我們可以看到 } * \text{ 的運算順序最高} \\
= & 1 + 6 - 4 && \text{加法和減法運算順序相同，依照結合性從左到右算} \\
= & 7 - 4 && \text{依照結合性從左到右算} \\
= & 3
\end{aligned}
$$

回到原來例子

$1+2*3-4=?$

$$
\begin{aligned}
&1+2*3-4 && \text{我們可以看到 } * \text{ 的運算順序最高}\\
=&1+6-4 && \text{加法和減法運算順序相同，依照結合性從左到右算}\\
=&7-4 && \text{依照結合性從左到右算}\\
=&3
\end{aligned}
$$

觀念

- C++ 的四則運算用優先順序和結合性來處理。

回到原來例子

$1 + 2 * 3 - 4 =?$

$$
\begin{aligned}
& 1 + 2 * 3 - 4 && \text{我們可以看到 } * \text{ 的運算順序最高} \\
= & 1 + 6 - 4 && \text{加法和減法運算順序相同，依照結合性從左到右算} \\
= & 7 - 4 && \text{依照結合性從左到右算} \\
= & 3
\end{aligned}
$$

觀念

- C++ 的四則運算用優先順序和結合性來處理。
- 這件事情非常重要，稍後就會知道爲什麼。

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1
2. `cout << 8.0 / 5.0 << endl;` 的結果？

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1
2. `cout << 8.0 / 5.0 << endl;` 的結果？Ans: 1.6

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1
2. `cout << 8.0 / 5.0 << endl;` 的結果？Ans: 1.6

註

- 在 8 / 5 中，8 和 5 被視爲 `int`，因此 C++ 會做「整數除法」。

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1
2. `cout << 8.0 / 5.0 << endl;` 的結果？Ans: 1.6

註

- 在 8 / 5 中，8 和 5 被視爲 `int`，因此 C++ 會做「整數除法」。
- 而在 8.0 / 5.0 中，8.0 和 5.0 被視爲浮點數 `double`，因此會做「浮點數除法」。

整數除法

整數除法

1. `cout << 8 / 5 << endl;` 的結果？Ans: 1
2. `cout << 8.0 / 5.0 << endl;` 的結果？Ans: 1.6

註

- 在 8 / 5 中，8 和 5 被視為 `int`，因此 C++ 會做「整數除法」。
- 而在 8.0 / 5.0 中，8.0 和 5.0 被視為浮點數 `double`，因此會做「浮點數除法」。

- 除法還有另外一個問題點 ...

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

```
cout << 1 / 0 << endl;
```

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

```
cout << 1 / 0 << endl;
```

註

如果無法編譯成功，那麼就宣告一個變數，把分母裝進去再試試看。

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

1. `cout << 1 / 0 << endl;`
2. `cout << 0 / 0 << endl;`

註

如果無法編譯成功，那麼就宣告一個變數，把分母裝進去再試試看。

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

1. `cout << 1 / 0 << endl;`
2. `cout << 0 / 0 << endl;`
3. `cout << 1.0 / 0.0 << endl;`

註

如果無法編譯成功，那麼就宣告一個變數，把分母裝進去再試試看。

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

1. `cout << 1 / 0 << endl;`
2. `cout << 0 / 0 << endl;`
3. `cout << 1.0 / 0.0 << endl;`
4. `cout << 0.0 / 0.0 << endl;`

註

如果無法編譯成功，那麼就宣告一個變數，把分母裝進去再試試看。

除以零

試試看

我們知道數學上是不能除以零的，那程式呢？

1. `cout << 1 / 0 << endl;`
2. `cout << 0 / 0 << endl;`
3. `cout << 1.0 / 0.0 << endl;`
4. `cout << 0.0 / 0.0 << endl;`

註

如果無法編譯成功，那麼就宣告一個變數，把分母裝進去再試試看。

注意

通常編譯可以過，但是在執行時會出些狀況，各位知道出了哪些狀況就好，不用了解太詳細。

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2
2. `cout << (-5) % 3 << endl;` 呢？

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2
2. `cout << (-5) % 3 << endl;` 呢？Ans:-2

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2
2. `cout << (-5) % 3 << endl;` 呢？Ans:-2

註

- 事情不該是這樣發展的啊！！！

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2
2. `cout << (-5) % 3 << endl;` 呢？Ans:-2

註

- 事情不該是這樣發展的啊！！！
- 謎之音：「應該結果是要 1 才對。」

取餘數

觀察現象

1. `cout << 5 % 3 << endl;` 會輸出什麼？Ans:2
2. `cout << (-5) % 3 << endl;` 呢？Ans:-2

註

- 事情不該是這樣發展的啊！！！
- 謎之音：「應該結果是要 1 才對。」
  - C++ 一個奇怪的特性 ...

解決辦法？

問題

要怎麼做出取餘數的效果呢？

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
    - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
    - 如果 $n < 0$

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m - 1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m - 1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m - 1$ 的數字

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字
   - 全都修成正值了！

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字
   - 全都修成正值了！但還差最後一步 ...

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字
   - 全都修成正值了！但還差最後一步 ...
4. 最後，再 mod m 一次，把所有數字修正回 0 到 $m-1$ 之間。

解決辦法？

問題

要怎麼做出取餘數的效果呢？

1. 假設 n 要 mod m ...
2. 首先，我們取 `n % m`
   - 如果 $n \geq 0$，會得到介於 0 到 $m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)$ 到 0 的數字
3. 接著加上 m
   - 如果 $n \geq 0$，會得到介於 m 到 $2m-1$ 的數字
   - 如果 $n < 0$，會得到介於 $-(m-1)+m=1$ 到 m 的數字
   - 全都修成正值了！但還差最後一步 ...
4. 最後，再 mod m 一次，把所有數字修正回 0 到 $m-1$ 之間。
   - 大功告成啦 `(n % m + m) % m`

練習題

UVa 10071 - Back to High School Physics

這題只要能夠讀懂題意就不難寫。如果不知道怎樣讀取多筆測資請先參考迴圈部分 (EOF 版)。

練習題

UVa 10071 - Back to High School Physics

這題只要能夠讀懂題意就不難寫。如果不知道怎樣讀取多筆測資請先參考迴圈部分 (EOF 版)。

UVa 10300 - Ecological Premium

一樣能讀懂題意就不難寫。

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行爲
7. 其他運算子
8. 結論

比較運算子

比較運算子	意義	運算順序	結合性
==	等於	9	左→右
!=	不等於	9	左→右
>	大於	8	左→右
<	小於	8	左→右
>=	不小於	8	左→右
<=	不大於	8	左→右

Table: 比較運算子

比較運算子

比較運算子	意義	運算順序	結合性
==	等於	9	左→右
!=	不等於	9	左→右
>	大於	8	左→右
<	小於	8	左→右
>=	不小於	8	左→右
<=	不大於	8	左→右

Table: 比較運算子

注意

- C++ 的等於寫作「==」，不要和賦值的「=」搞混。

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：

回傳值

例子

1. cout << (3 < 5) << endl;，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 true

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 `true`
  - 否則就是 `false`

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 `true`
  - 否則就是 `false`
- 這種概念我們稱爲「回傳值」

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 true
  - 否則就是 false
- 這種概念我們稱爲「回傳值」
  - 比較運算子的回傳值是布林值 bool

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 true
  - 否則就是 false
- 這種概念我們稱爲「回傳值」
  - 比較運算子的回傳值是布林值 bool
  - $3 < 5 \Rightarrow$ true

回傳值

例子

1. `cout << (3 < 5) << endl;`，會發生什麼事？

註

- 比較運算子也是二元運算，他會比較兩邊數字大小：
  - 如果正確，則爲 `true`
  - 否則就是 `false`
- 這種概念我們稱爲「回傳值」
  - 比較運算子的回傳值是布林值 `bool`
  - $3 < 5 \Rightarrow$ `true`
  - 因爲我們要輸出 `true`，根據 C++ 的規則，我們知道 `true` 代表非零，因此會印出一個非零的數字 (通常是 1)

運算簡化

例子

判斷整數 n % m 是否「不是 0」。

運算簡化

例子

判斷整數 n % m 是否「不是 0」。

判斷整除

```
n % m != 0
```

運算簡化

例子

判斷整數 n % m 是否「不是 0」。

判斷整除

`n % m != 0`

- 如果 n % m 的回傳值 $\neq 0 \Rightarrow$ `true`

運算簡化

例子

判斷整數 n % m 是否「不是 0」。

判斷整除

`n % m != 0`

- 如果 n % m 的回傳值 $\neq 0 \Rightarrow$ `true`
- 如果是 0，則爲 `false`

運算簡化

例子

判斷整數 n % m 是否「不是 0」。

判斷整除

```
n % m != 0
```

- 如果 n % m 的回傳值 $\neq 0 \Rightarrow$ `true`
- 如果是 0，則爲 `false`

簡化寫法

```
n % m
```

運算簡化

例子

判斷整數 `n % m` 是否「不是 0」。

判斷整除

`n % m != 0`

- 如果 `n % m` 的回傳值 $\neq 0 \Rightarrow$ `true`
- 如果是 0，則爲 `false`

簡化寫法

`n % m`

- 如果 `n % m` 的回傳值 $\neq 0$ ，可以被當做「`true`」

運算簡化

例子

判斷整數 `n % m` 是否「不是 0」。

判斷整除

`n % m != 0`

- 如果 `n % m` 的回傳值 $\neq 0 \Rightarrow$ `true`
- 如果是 0，則爲 `false`

簡化寫法

`n % m`

- 如果 `n % m` 的回傳值 $\neq 0$ ，可以被當做「`true`」
- 如果是 0，那麼就可以當做「`false`」

遙遠的記憶

布林值的重要觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。
- 「0」會被當做「`false`」，印出時會印出「0」。

遙遠的記憶

布林值的重要觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。
- 「0」會被當做「`false`」，印出時會印出「0」。

註

- 簡化的寫法大多時候可以取代原來一般寫法。

遙遠的記憶

布林值的重要觀念

- C++ 中，「非零整數」會被當做「`true`」，印出時也會印出一個非零整數 (通常是 1)。
- 「0」會被當做「`false`」，印出時會印出「0」。

註

- 簡化的寫法大多時候可以取代原來一般寫法。
- 通常比較運算子要和 `if`、`else` 配合。

邏輯運算子

邏輯運算子	意義	運算順序	結合性
&&	且	13	左→右
||	或	14	左→右
!	非	3	右→左

Table: 邏輯運算子

邏輯運算子

邏輯運算子	意義	運算順序	結合性
&&	且	13	左→右
||	或	14	左→右
!	非	3	右→左

Table: 邏輯運算子

作用

- 一般來説是連接比較運算子

邏輯運算子

邏輯運算子	意義	運算順序	結合性
&&	且	13	左→右
||	或	14	左→右
!	非	3	右→左

Table: 邏輯運算子

作用

- 一般來說是連接比較運算子
- 例如：`1 < x && x < 5`

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？

舉個例子

例子

判斷 `x` 是否介於 `a` 和 `b` 之間能不能寫成 `a <= x <= b;` 呢？Ans: 不行。

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 `a <= x <= b;` 呢？Ans: 不行。

用回傳值的觀點

- 我們知道 `<=` 運算子在列出很多個時，會由左到右算

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 `a <= x <= b;` 呢？Ans:不行。

用回傳值的觀點

- 我們知道 `<=` 運算子在列出很多個時，會由左到右算
- `a <= x` 先算出 `true` 或者是 `false`

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？Ans:不行。

用回傳值的觀點

- 我們知道 <= 運算子在列出很多個時，會由左到右算
- a <= x 先算出 true 或者是 false
- 假設 a=-4、b=-1、x=-2 (我們知道結果是 true)

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？Ans:不行。

用回傳值的觀點

- 我們知道 <= 運算子在列出很多個時，會由左到右算
- a <= x 先算出 true 或者是 false
- 假設 a=-4、b=-1、x=-2 (我們知道結果是 true)
  - a <= x <= b 先算 a <= x 得到 true

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？Ans: 不行。

用回傳值的觀點

- 我們知道 <= 運算子在列出很多個時，會由左到右算
- a <= x 先算出 true 或者是 false
- 假設 a=-4、b=-1、x=-2 (我們知道結果是 true)
  - a <= x <= b 先算 a <= x 得到 true
  - true <= b，因爲 true 通常是 1，但此時 b=-1，整句就會回傳 false

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 a <= x <= b; 呢？Ans:不行。

用回傳值的觀點

- 我們知道 <= 運算子在列出很多個時，會由左到右算
- a <= x 先算出 true 或者是 false
- 假設 a=-4、b=-1、x=-2 (我們知道結果是 true)
  - a <= x <= b 先算 a <= x 得到 true
  - true <= b，因爲 true 通常是 1，但此時 b=-1，整句就會回傳 false
  - 但事實上 x 是在 a 和 b 裡面。

舉個例子

例子

判斷 x 是否介於 a 和 b 之間能不能寫成 `a <= x <= b;` 呢？Ans: 不行。

用回傳值的觀點

- 我們知道 <= 運算子在列出很多個時，會由左到右算
- `a <= x` 先算出 `true` 或者是 `false`
- 假設 a=-4、b=-1、x=-2 (我們知道結果是 `true`)
  - `a <= x <= b` 先算 `a <= x` 得到 `true`
  - `true <= b`，因爲 `true` 通常是 1，但此時 b=-1，整句就會回傳 `false`
  - 但事實上 x 是在 a 和 b 裡面。
- `a <= x` 是 `false` 也會有同樣的問題。

練習題 (1)

UVa 10055 - Hashmat the brave warrior

取絕對值有兩種做法，一種是用 `if` 判斷；另一種是呼叫函數 abs() 就好了。abs() 函數被定義在 <cstdlib> 中，雖然沒有 include 在 Visual C++ 依然能編譯過，但是上傳時因爲編譯器的原因會導致編譯錯誤 (Compilation Error, CE)。

注意

另外要注意這一題的整數型態需用 `long long`，用 `int` 會造成「溢位現象」，這個原因會在後面說明。

練習題 (2)

UVa 11172 - Relational Operators

能夠理解題意就不難解決此道問題。

練習題 (2)

UVa 11172 - Relational Operators

能夠理解題意就不難解決此道問題。

UVa 11942 - Lumberjack Sequencing

依序給你一些木頭的長度，問你這些木頭是不是由長到短，或是由短到長排列。

短路運算

性質

- `A && B`

短路運算

性質

- A && B
  - && 運算子：只要 A 或 B 其中一個回傳 false，則整個運算式就會是 false

短路運算

性質

- `A && B`
  - `&&` 運算子：只要 A 或 B 其中一個回傳 `false`，則整個運算式就會是 `false`
  - C++ 設計上當 A 已經是 `false` (也就是確定整個運算式必爲 `false`)，則 C++ 會跳過 B

短路運算

性質

- A && B
  - && 運算子：只要 A 或 B 其中一個回傳 false，則整個運算式就會是 false
  - C++ 設計上當 A 已經是 false (也就是確定整個運算式必爲 false)，則 C++ 會跳過 B

範例

```
int i, j;
i = j = 0;
if ((i++ < 0) && (j++ > 0))
  cout << "XD" << endl; // 這行不會輸出
cout << i << "␣" << j << endl;
```

短路運算

性質

- `A || B`

短路運算

性質

- `A || B`
  - `||` 運算子：只要 A 或 B 其中一個回傳 `true`，則整個運算式就會是 `true`

短路運算

性質

- A || B
  - || 運算子：只要 A 或 B 其中一個回傳 true，則整個運算式就會是 true
  - C++ 設計上當 A 已經是 true (也就是確定整個運算式必為 true)，則 C++ 會跳過 B

短路運算

性質

- A || B
  - || 運算子：只要 A 或 B 其中一個回傳 true，則整個運算式就會是 true
  - C++ 設計上當 A 已經是 true (也就是確定整個運算式必為 true)，則 C++ 會跳過 B

範例

```
int i, j;
i = j = 0;
if ((i++ >= 0) || (j++ < 0))
  cout << "XD" << endl; // 會輸出 XD
cout << i << "␣" << j << endl;
```

大綱

1 簡介
2 程式架構
- 基本程式架構
- 輸出
- 變數
- 輸入
- 資料型態

3 算術運算子
- 運算性質
- 結合性與運算順序
- 整數除法與除零問題
- 應用：取餘數

4 比較和邏輯運算子
- 簡化規則
- 短路運算

5 位元運算子
- int 和 long long 的儲存形式
- 常用技巧：連續的 1
- 常用技巧：遮罩與指定位元
- Parity
- xor 性質

6 指定運算子
- 運算性質
- 未定義行爲

7 其他運算子
8 結論

基本觀念

觀念

- 位元 (bit, b)：計算機儲存資料的基本單位，只儲存 0 和 1

觀念

- 位元 (bit, b)：計算機儲存資料的基本單位，只儲存 0 和 1
- 位元組 (byte, B)：因爲位元很多，所以我們把 8 個位元「打包起來」，變成一個位元組

01001010

Table: 位元組

觀念

- 位元 (bit, b)：計算機儲存資料的基本單位，只儲存 0 和 1
- 位元組 (byte, B)：因爲位元很多，所以我們把 8 個位元「打包起來」，變成一個位元組

01001010

Table: 位元組

- 常見應用

觀念

- 位元 (bit, b)：計算機儲存資料的基本單位，只儲存 0 和 1
- 位元組 (byte, B)：因爲位元很多，所以我們把 8 個位元「打包起來」，變成一個位元組

01001010

Table: 位元組

- 常見應用
  - KB、MB、GB、TB、PB：資料大小

觀念

- 位元 (bit, b)：計算機儲存資料的基本單位，只儲存 0 和 1
- 位元組 (byte, B)：因爲位元很多，所以我們把 8 個位元「打包起來」，變成一個位元組

01001010

Table: 位元組

- 常見應用
  - KB、MB、GB、TB、PB：資料大小
  - Kbps、Mbps、Gbps：資料傳輸速度

int 表示法

int ...

- 有至少 2 個位元組

int 表示法

int ...

- 有至少 2 個位元組
- 謎之音：「蝦米？」

int 表示法

int ...

- 有至少 2 個位元組
- 謎之音：「蝦米？」不是 4 個位元組嘛！！！

int 表示法

int ...

- 有至少 2 個位元組
- 謎之音：「蝦米？」不是 4 個位元組嘛！！！
- 事實上當初定義時，`int` 只有「至少」2 位元組。

int 表示法

int ...

- 有至少 2 個位元組
- 謎之音：「蝦米？」不是 4 個位元組嘛！！！
- 事實上當初定義時，`int` 只有「至少」2 位元組。
- 現今大多是 4 位元組。

int 表示法

int ...

- 有至少 2 個位元組
- 謎之音：「蝦米？」不是 4 個位元組嘛！！！
- 事實上當初定義時，int 只有「至少」2 位元組。
- 現今大多是 4 位元組。

型態	長度
bool	1 位元組
int	2 或 4 位元組
long long	4 或 8 位元組
double	8 位元組

Table: 位元組長度

int 表示法

int 表示法

- 一般來說，`int` 由 4 個位元組組成

10100010	00110011	00100111	10101101

int 表示法

- 一般來説，int 由 4 個位元組組成

10100010	00110011	00100111	10101101

- 可以視爲一個長度是 32 的二進位數字，我們將位數依照高低編號

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

int 表示法

- 一般來說，`int` 由 4 個位元組組成

10100010	00110011	00100111	10101101

- 可以視為一個長度是 32 的二進位數字，我們將位數依照高低編號

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

- x_{31} 表示正負號

int 表示法

int 表示法

- 一般來說，`int` 由 4 個位元組組成

10100010	00110011	00100111	10101101

- 可以視爲一個長度是 32 的二進位數字，我們將位數依照高低編號

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

- x_{31} 表示正負號
  - 0 代表 `int` 是正數

int 表示法

- 一般來說，int 由 4 個位元組組成

10100010	00110011	00100111	10101101

- 可以視爲一個長度是 32 的二進位數字，我們將位數依照高低編號

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

- x_{31} 表示正負號
  - 0 代表 int 是正數
  - 1 代表 int 是負數

int 表示法

- 一般來説，int 由 4 個位元組組成

10100010	00110011	00100111	10101101

- 可以視爲一個長度是 32 的二進位數字，我們將位數依照高低編號

$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$

- x_{31} 表示正負號
  - 0 代表 int 是正數
  - 1 代表 int 是負數

註

int 的儲存方式很特別，要多花一些力氣説明。

規則

依照一般的二進位方式儲存。

int 存正數的情況

規則

依照一般的二進位方式儲存。

例如

- `int x = 1;`

00000000	00000000	00000000	00000001

int 存正數的情況

規則

依照一般的二進位方式儲存。

例如

- `int x = 1;`

00000000	00000000	00000000	00000001

- `int x = 255;`

00000000	00000000	00000000	11111111

int 存負數的情況

舉例

- `int x = -1;`

11111111	11111111	11111111	11111111

int 存負數的情況

舉例

- `int x = -1;`

11111111	11111111	11111111	11111111

- 謎之音：「根本黑魔法！」

int 存負數的情況

舉例

- `int x = -1;`

11111111	11111111	11111111	11111111

- 謎之音：「根本黑魔法！」

想法

- 我們知道 $(-1)+1=0$，那麼拿這種表示法加加看

	11111111	11111111	11111111	11111111
+	00000000	00000000	00000000	00000001
	100000000	00000000	00000000	00000000

int 存負數的情況

舉例

- `int x = -1;`

11111111	11111111	11111111	11111111

- 謎之音：「根本黑魔法！」

想法

- 我們知道 $(-1) + 1 = 0$，那麼拿這種表示法加加看

	11111111	11111111	11111111	11111111
+	00000000	00000000	00000000	00000001
	100000000	00000000	00000000	00000000

- 紅色的 1 因爲超過 32 位元，因此被捨棄，稱爲溢位

int 負數規則

練習

- `int x = -2;`

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

- `int x = -256;`

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

- `int x = -256;`

11111111	11111111	11111111	00000000

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

- `int x = -256;`

11111111	11111111	11111111	00000000

重點

- 這種表示法稱爲二補數 (2's complement)

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

- `int x = -256;`

11111111	11111111	11111111	00000000

重點

- 這種表示法稱爲二補數 (2's complement)
- 要想像負數 $-x$ 的表示法，訣竅是 $(-x) + x$ 會因爲溢位等於 0

int 負數規則

練習

- `int x = -2;`

11111111	11111111	11111111	11111110

- `int x = -256;`

11111111	11111111	11111111	00000000

重點

- 這種表示法稱爲二補數 (2's complement)
- 要想像負數 $-x$ 的表示法，訣竅是 $(-x)+x$ 會因爲溢位等於 0
- 記得 0 是全 0，-1 是全 1

位元運算子

位元運算子	意義	運算順序	結合性
<<	左移運算子	7	左→右
>>	右移運算子	7	左→右
&	位元 AND	10	左→右
^	位元 XOR	11	左→右
|	位元 OR	12	左→右
~	1's 補數	3	右→左

Table: 位元運算子

位元運算子

位元運算子	意義	運算順序	結合性
<<	左移運算子	7	左→右
>>	右移運算子	7	左→右
&	位元 AND	10	左→右
^	位元 XOR	11	左→右
|	位元 OR	12	左→右
~	1's 補數	3	右→左

Table: 位元運算子

注意

- 左移運算子和右移運算子不要和 cin 與 cout 的 <<、>> 混淆

左移和右移運算子

左移和右移

在位元操作上左移和右移 k 個位元。

左移和右移運算子

左移和右移

在位元操作上左移和右移 k 個位元。

舉例

- 2 << 2

左移和右移運算子

左移和右移

在位元操作上左移和右移 k 個位元。

舉例

- 2 << 2 $\Rightarrow$ 8

左移和右移運算子

左移和右移

在位元操作上左移和右移 k 個位元。

舉例

- 2 << 2 $\Rightarrow$ 8

00000000	00000000	00000000	00000010

左移和右移運算子

左移和右移

在位元操作上左移和右移 k 個位元。

舉例

- 2 << 2 $\Rightarrow$ 8

00000000	00000000	00000000	00000010

00000000	00000000	00000000	00001000

左移和右移運算子 (2)

再來個例子

- `5 >> 1`

左移和右移運算子 (2)

再來個例子

- 5 >> 1 $\Rightarrow$ 2

左移和右移運算子 (2)

再來個例子

- 5 >> 1 $\Rightarrow$ 2

00000000	00000000	00000000	00000101

左移和右移運算子 (2)

再來個例子

- 5 >> 1 $\Rightarrow$ 2

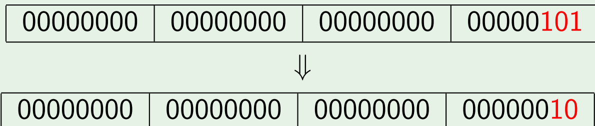
00000000	00000000	00000000	00000101

00000000	00000000	00000000	00000010

左移和右移運算子 (2)

再來個例子

- 5 >> 1 ⇒ 2



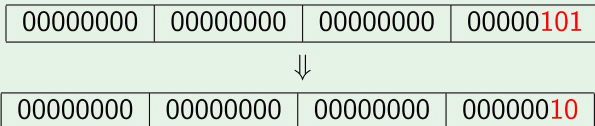
注意

- 不管是左移還是右移，移出去的位元會被捨棄。

左移和右移運算子 (2)

再來個例子

- 5 >> 1 ⇒ 2



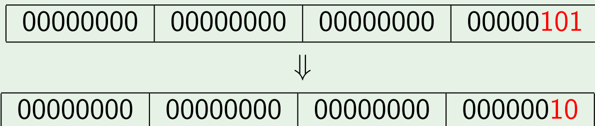
注意

- 不管是左移還是右移，移出去的位元會被捨棄。
- 之前提到 x_{31} 決定正負號，在左移右移會影響到 x_{31} 時會比較複雜，例如

左移和右移運算子 (2)

再來個例子

- `5 >> 1` ⇒ 2



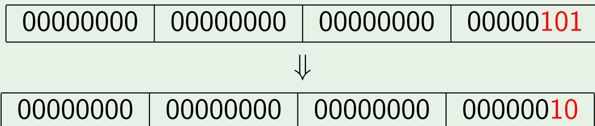
注意

- 不管是左移還是右移，移出去的位元會被捨棄。
- 之前提到 x_{31} 決定正負號，在左移右移會影響到 x_{31} 時會比較複雜，例如
  - `2147483647 << 1`

左移和右移運算子 (2)

再來個例子

- `5 >> 1` ⇒ `2`



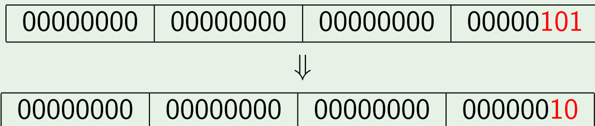
注意

- 不管是左移還是右移，移出去的位元會被捨棄。
- 之前提到 x_{31} 決定正負號，在左移右移會影響到 x_{31} 時會比較複雜，例如
  - `2147483647 << 1`
  - `-5 >> 1`

左移和右移運算子 (2)

再來個例子

- 5 >> 1 ⇒ 2



注意

- 不管是左移還是右移，移出去的位元會被捨棄。
- 之前提到 x_{31} 決定正負號，在左移右移會影響到 x_{31} 時會比較複雜，例如
  - 2147483647 << 1
  - -5 >> 1
  - (2147483647 << 1) >> 1

左移和右移運算子 (3)

觀察

- `a << k` 會得到什麼數字呢？

左移和右移運算子 (3)

觀察

- a << k 會得到什麼數字呢？
- 那麼 a >> k 呢？

觀察

- `a << k` 會得到什麼數字呢？
- 那麼 `a >> k` 呢？

結論

- 一般來說 `a << k` 會得到 $a \times 2^k$，`a >> k` 會得到 $a/2^k$

左移和右移運算子 (3)

觀察

- a << k 會得到什麼數字呢？
- 那麼 a >> k 呢？

結論

- 一般來説 a << k 會得到 $a \times 2^k$，a >> k 會得到 $a/2^k$
- 有些情況比較複雜，大家看看就好，起碼對這些運算「有感覺」。

位元運算子 (2)

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

&	1	0
1	1	0
0	0	0

Table: and 運算子

位元運算子 (2)

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

&	1	0
1	1	0
0	0	0

Table: and 運算子

^	1	0
1	0	1
0	1	0

Table: xor 運算子

位元運算子 (2)

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

&	1	0
1	1	0
0	0	0

Table: and 運算子

^	1	0
1	0	1
0	1	0

Table: xor 運算子

|	1	0
1	1	1
0	1	0

Table: or 運算子

位元運算子 (2)

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

&	1	0
1	1	0
0	0	0

Table: and 運算子

^	1	0
1	0	1
0	1	0

Table: xor 運算子

|	1	0
1	1	1
0	1	0

Table: or 運算子

觀察

- and、or 運算子類似之前的邏輯運算子，不同在於這是位元運算。

位元運算子 (2)

位元運算子

對於兩個位元 x 和 y，遵守以下運算規則：

&	1	0
1	1	0
0	0	0

Table: and 運算子

^	1	0
1	0	1
0	1	0

Table: xor 運算子

|	1	0
1	1	1
0	1	0

Table: or 運算子

觀察

- and、or 運算子類似之前的邏輯運算子，不同在於這是位元運算。
- xor 很特別，可以記為不同數字為 1，相同為 0。

舉例

- 5 & 3

舉例

- 5 & 3 $\Rightarrow$ 1

舉例

- 5 & 3 $\Rightarrow$ 1

結果

	00000000	00000000	00000000	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

舉例

- 5 & 3 ⇒ 1

結果

	00000000	00000000	00000000	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

- 5 | 3

舉例

- 5 & 3 ⇒ 1

結果

	00000000	00000000	00000000	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

- 5 | 3 ⇒ 7

舉例

- 5 & 3 ⇒ 1

結果

	00000000	00000000	00000000	00000101
&	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000001

- 5 | 3 ⇒ 7

結果

	00000000	00000000	00000000	00000101
|	00000000	00000000	00000000	00000011
	00000000	00000000	00000000	00000111

位元運算子 (3)

補數運算子

對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

Table: and 運算子

位元運算子 (3)

補數運算子

對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

Table: and 運算子

說明

- 就是把 1 變爲 0，把 0 變爲 1 (相當於邏輯運算子的 !)

位元運算子 (3)

補數運算子

對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

Table: and 運算子

說明

- 就是把 1 變爲 0，把 0 變爲 1 (相當於邏輯運算子的 !)
- 又稱爲 1's 補數

位元運算子 (3)

補數運算子

對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

Table: and 運算子

說明

- 就是把 1 變為 0，把 0 變為 1 (相當於邏輯運算子的 !)
- 又稱為 1's 補數
- ~0

位元運算子 (3)

補數運算子

對於兩個位元 x 和 y，遵守以下運算規則：

~	1	0
	0	1

Table: and 運算子

說明

- 就是把 1 變為 0，把 0 變為 1 (相當於邏輯運算子的 !)
- 又稱為 1's 補數
- ~0 $\Rightarrow$ -1

一元運算子

一元運算子

一元運算子就是只有一個運算元的運算子。

一元運算子

一元運算子就是只有一個運算元的運算子。

運算子	意義	運算順序	結合性
+	正號	3	右→左
-	負號	3	右→左

一元運算子

一元運算子

一元運算子就是只有一個運算元的運算子。

運算子	意義	運算順序	結合性
+	正號	3	右→左
-	負號	3	右→左

舉例

~~3 會先算右邊的 ~3，得到 -4，接著 -4 再和左邊的負號運算子「運算」，回傳結果爲 3。

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

舉例

- 3 個 1

00000000	00000000	00000000	00000111

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

舉例

- 3 個 1

00000000	00000000	00000000	00000111

- 5 個 1

00000000	00000000	00000000	00011111

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

觀察

- 可以很容易發現，k 個 1 恰好是 $2^k - 1$。

問題

要怎樣產生 2 進位下連續 k 個 1？

觀察

- 可以很容易發現，k 個 1 恰好是 $2^k - 1$。
- 前提是不牽扯到正負號 x_{31}

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

觀察

- 可以很容易發現，k 個 1 恰好是 $2^k - 1$。
- 前提是不牽扯到正負號 x_{31}

結論

- `(1 << k) - 1`

常用技巧：連續的 1

問題

要怎樣產生 2 進位下連續 k 個 1？

觀察

- 可以很容易發現，k 個 1 恰好是 $2^k - 1$。
- 前提是不牽扯到正負號 x_{31}

結論

- `(1 << k) - 1`
- 注意減號和左移運算子的優先順序。

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？ (假設 $a < b$)

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？ (假設 $a < b$)

舉例

- x_0 到 x_2

00000000	00000000	00000000	00000111

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？ (假設 $a < b$)

舉例

- x_0 到 $x_2 \Rightarrow$ 恰好是 3 個 1 的情形

00000000	00000000	00000000	00000111

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？ (假設 $a < b$)

舉例

- x_0 到 $x_2 \Rightarrow$ 恰好是 3 個 1 的情形

00000000	00000000	00000000	00000111

- x_3 到 x_7

00000000	00000000	00000000	11111000

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？ (假設 $a < b$)

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？(假設 $a < b$)

結論

- 觀察之後，可以發現是 $2^{b+1} - 2^a$

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？(假設 $a < b$)

結論

- 觀察之後，可以發現是 $2^{b+1} - 2^a$
- 該怎麼實作就從之前取 k 個 1 的方法去擴展就可以得到。

常用技巧：連續的 1 (加強版)

問題 (加強版)

要怎樣產生 2 進位下 x_a 到 x_b 都是 1？(假設 $a < b$)

結論

- 觀察之後，可以發現是 $2^{b+1} - 2^a$
- 該怎麼實作就從之前取 k 個 1 的方法去擴展就可以得到。
- 記得熟悉位元運算，有時候就會有題目會用到。

位元技巧：取負數

問題

給你一個正數 x，問如何不用負號的情況下求出 -x 呢？

位元技巧：取負數

問題

給你一個正數 x，問如何不用負號的情況下求出 -x 呢？

提示

比較 -x 和 ~x 的不同。

位元技巧：取負數

問題

給你一個正數 x，問如何不用負號的情況下求出 -x 呢？

提示

比較 -x 和 ~x 的不同。

註

這個例子只是展現位元運算有時候很神奇，這個方法很多時候並不常用。

更多性質

位元運算的性質

再看看位元運算的性質：

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

|	x
1	1
0	x

Table: or 運算子

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

|	x
1	1
0	x

Table: or 運算子

觀察

x 是變數時 ... (可能是 0 或 1)

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

|	x
1	1
0	x

Table: or 運算子

觀察

x 是變數時 ... (可能是 0 或 1)

- x & 0 永遠是 0

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

|	x
1	1
0	x

Table: or 運算子

觀察

x 是變數時 ... (可能是 0 或 1)

- x & 0 永遠是 0
- x | 1 永遠是 1

更多性質

位元運算的性質

再看看位元運算的性質：

&	x
1	x
0	0

Table: and 運算子

|	x
1	1
0	x

Table: or 運算子

觀察

x 是變數時 ... (可能是 0 或 1)

- x & 0 永遠是 0
- x | 1 永遠是 1 (這些性質很有用途！)

常用技巧：遮罩

問題

要知道 x_0 是 1 還是 0，要怎麼做呢？

常用技巧：遮罩

問題

要知道 x_0 是 1 還是 0，要怎麼做呢？

做法

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
&	00000000	00000000	00000000	00000001
	00000000	00000000	00000000	0000000x_0

常用技巧：遮罩

問題

要知道 x_0 是 1 還是 0，要怎麼做呢？

做法

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
&	00000000	00000000	00000000	00000001
	00000000	00000000	00000000	0000000x_0

還記得剛剛位元運算的性質嗎？

常用技巧：遮罩

問題

要知道 x_0 是 1 還是 0，要怎麼做呢？

做法

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
&	00000000	00000000	00000000	00000001
	00000000	00000000	00000000	0000000x_0

還記得剛剛位元運算的性質嗎？

推廣版

- 要知道 x_i 是 1 還是 0 要怎麼做？

常用技巧：遮罩

問題

要知道 x_0 是 1 還是 0，要怎麼做呢？

做法

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
&	00000000	00000000	00000000	00000001
	00000000	00000000	00000000	0000000x_0

還記得剛剛位元運算的性質嗎？

推廣版

- 要知道 x_i 是 1 還是 0 要怎麼做？
- 如果我們要取出 x_a 到 x_b 的位元，要怎麼做呢？

常用技巧：指定位元

問題

要如何把一個整數 x 當中，x_a 的位元「變成」1？

常用技巧：指定位元

問題

要如何把一個整數 x 當中，x_a 的位元「變成」1？

觀察

將 x_0 改爲 1

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
|	00000000	00000000	00000000	00000001
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{1}1$

常用技巧：指定位元

問題

要如何把一個整數 x 當中，x_a 的位元「變成」1？

觀察

將 x_0 改爲 1

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
|	00000000	00000000	00000000	00000001
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_1 1$

註

利用剛剛提到的性質：1 和任意位元 or 起來都是 1。

觀察 (續)

將 x_2 改為 1

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7\cdots x_3x_2x_1x_0$
|	00000000	00000000	00000000	00000100
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7\cdots x_3 1 x_1x_0$

常用技巧：指定位元

觀察 (續)

將 x_2 改為 1

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}\cdots x_{3}x_{2}x_{1}x_{0}$
|	00000000	00000000	00000000	00000100
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}\cdots x_{3}1x_{1}x_{0}$

結論

可以套用之前連續 1 的技巧，就可以任意指定一些位元為 1。

常用技巧：指定位元

另一個問題

要如何把一個整數 x 當中，x_a 的位元「變成」0？

常用技巧：指定位元

另一個問題

要如何把一個整數 x 當中，x_a 的位元「變成」0？

觀察

將 x_0 改為 0

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
&	11111111	11111111	11111111	11111110
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_1 0$

常用技巧：指定位元

另一個問題

要如何把一個整數 x 當中，x_a 的位元「變成」0？

觀察

將 x_0 改爲 0

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
&	11111111	11111111	11111111	11111110
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{1}0$

結論

- 同樣也是利用位元運算的性質，和剛剛指定 1 相似。

常用技巧：指定位元

另一個問題

要如何把一個整數 x 當中，x_a 的位元「變成」0？

觀察

將 x_0 改為 0

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_0$
&	11111111	11111111	11111111	11111110
	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_8$	$x_7x_6\cdots x_1$0

結論

- 同樣也是利用位元運算的性質，和剛剛指定 1 相似。
- 求出此常數可利用「補數」來求出。

位元技巧：取 2^k 餘數

- 取 2 的餘數

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3`

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 $x_1 x_0$
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

優點

- 和「%」相比速度較快。

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 $x_1 x_0$
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

優點

- 和「%」相比速度較快。
- 在負數下也沒有問題。

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

優點

- 和「%」相比速度較快。
- 在負數下也沒有問題。

缺點

- 不易閱讀。

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

優點

- 和「%」相比速度較快。
- 在負數下也沒有問題。

缺點

- 不易閱讀。
- 只能取特定餘數。

位元技巧：取 2^k 餘數

- 取 2 的餘數
  - 因爲餘數只有 0、1 兩種，恰好是看 x_0
  - `x % 2` ⇒ `x & 1`
- 取 4 的餘數
  - 餘數只有 0(00)、1(01)、2(10)、3(11) 四種，恰好是看 x_1x_0
  - `x % 4` ⇒ `x & 3` ⇒ `x & ((1 << 2) - 1)`
- 取 2^k 的餘數 ⇒ `x & ((1 << k) - 1)`

優點

- 和「%」相比速度較快。
- 在負數下也沒有問題。

缺點

- 不易閱讀。
- 只能取特定餘數。
- 要注意運算順序！

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5)

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5) $\Rightarrow$ 2

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5) $\Rightarrow$ 2

00000000	00000000	00000000	00000101

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5) $\Rightarrow$ 2

00000000	00000000	00000000	00000101

- Parity(255)

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5) $\Rightarrow$ 2

00000000	00000000	00000000	00000101

- Parity(255) $\Rightarrow$ 8

Parity

Parity 問題

給妳一個正整數 x，問在 2 進位下有幾個 1？

範例

- Parity(5) $\Rightarrow$ 2

00000000	00000000	00000000	00000101

- Parity(255) $\Rightarrow$ 8

00000000	00000000	00000000	11111111

Parity

普通寫法

一個一個計算：

```
for (; x; x /= 2) {
  if (x % 2 != 0)
    cnt++;
}
```

Parity

普通寫法

一個一個計算：

```
for (; x; x /= 2) {
  if (x % 2 != 0)
    cnt++;
}
```

位元運算寫法

```
for ( ; x; x >>= 1) { // 右移代替除法
  if (x & 1) // 省略「!= 0」，同時把除法改成位元運算
    cnt++;
}
```

Parity

究極 Parity

檢查 Parity 是否爲奇數：

```
unsigned int v; // 32-bit word
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x11111111U) * 0x11111111U;
(v >> 28) & 1;
```

Parity

究極 Parity

檢查 Parity 是否爲奇數：

```
unsigned int v; // 32-bit word
v ^= v >> 1;
v ^= v >> 2;
v = (v & 0x11111111U) * 0x11111111U;
(v >> 28) & 1;
```

註

看看就好，不要刻意去記這些炫砲技能。

xor 性質

給一個整數 x，x ^ x 恆爲 0。

xor 性質

給一個整數 x，x ^ x 恆為 0。

解說

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
^	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
	00000000	00000000	00000000	00000000

xor 性質

給一個整數 x，x ^ x 恆爲 0。

解說

	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
^	$x_{31}x_{30}\cdots x_{24}$	$x_{23}x_{22}\cdots x_{16}$	$x_{15}x_{14}\cdots x_{8}$	$x_{7}x_{6}\cdots x_{0}$
	00000000	00000000	00000000	00000000

註

xor 運算的性質是「同爲 0 或同爲 1 xor 起來就是 0」。

位元技巧：交換兩數

問題

交換兩個 `int` x 和 y 的值。

位元技巧：交換兩數

問題

交換兩個 `int` x 和 y 的值。

swap 版

```
swap(x, y);
```

位元技巧：交換兩數

問題

交換兩個 `int` x 和 y 的值。

swap 版

```
swap(x, y);
```

變數版

```
int tmp = x;
x = y;
y = tmp;
```

位元技巧：交換兩數

問題

交換兩個 `int` x 和 y 的值。

swap 版

```
swap(x, y);
```

變數版

```
int tmp = x;
x = y;
y = tmp;
```

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y
第一行後	x xor y	y

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y
第一行後	x xor y	y
第二行後	x xor y	y xor x xor y

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y
第一行後	x xor y	y
第二行後	x xor y	x

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y
第一行後	x xor y	y
第二行後	x xor y	x
第三行後	x xor y xor x	x

位元技巧：交換兩數

位元運算版

```
x ^= y;
y ^= x;
x ^= y;
```

	x	y
原來的值	x	y
第一行後	x xor y	y
第二行後	x xor y	x
第三行後	y	x

UVa 10469 - To Carry or not to Carry

這題算是位元運算的基本應用。

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行爲
7. 其他運算子
8. 結論

指定運算子

運算子	意義	運算順序	結合性
=	賦值	16	右→左

Table: 指定運算子

複合指定運算子

運算子	意義	運算順序	結合性
+=	加法賦值	16	右→左
−=	加法賦值	16	右→左
*=	乘法賦值	16	右→左
/=	除法賦值	16	右→左
%=	取餘賦值	16	右→左

複合指定運算子

運算子	意義	運算順序	結合性
+=	加法賦值	16	右→左
-=	加法賦值	16	右→左
*=	乘法賦值	16	右→左
/=	除法賦值	16	右→左
%=	取餘賦值	16	右→左

意義

這些複合指定運算子代表的意義爲：

- `x += a` ⇒ `x = x + a`
- `x -= a` ⇒ `x = x - a`
- `x *= a` ⇒ `x = x * a`
- `x /= a` ⇒ `x = x / a`
- `x %= a` ⇒ `x = x % a`
- 不難理解。

複合指定運算子

運算子	意義	運算順序	結合性
<<=	左移賦值	16	右→左
>>=	右移賦值	16	右→左
&=	位元 AND 賦值	16	右→左
^=	位元 XOR 賦值	16	右→左
|=	位元 OR 賦值	16	右→左

複合指定運算子

運算子	意義	運算順序	結合性
<<=	左移賦值	16	右→左
>>=	右移賦值	16	右→左
&=	位元 AND 賦值	16	右→左
^=	位元 XOR 賦值	16	右→左
|=	位元 OR 賦值	16	右→左

意義

這些複合指定運算子代表的意義為：

- `x <<= a` ⇒ `x = x << a`
- `x >>= a` ⇒ `x = x >> a`
- `x &= a` ⇒ `x = x & a`
- `x ^= a` ⇒ `x = x ^ a`
- `x |= a` ⇒ `x = x | a`
- 以此類推。

複合指定運算子

運算子	意義	運算順序	結合性
++	字尾遞增	2	左→右
--	字尾遞減	2	左→右
++	字首遞增	3	左→右
--	字首遞減	3	左→右

複合指定運算子

運算子	意義	運算順序	結合性
++	字尾遞增	2	左→右
--	字尾遞減	2	左→右
++	字首遞增	3	左→右
--	字首遞減	3	左→右

註

- 字尾系列寫做「i++」、「j--」。

複合指定運算子

運算子	意義	運算順序	結合性
++	字尾遞增	2	左→右
--	字尾遞減	2	左→右
++	字首遞增	3	左→右
--	字首遞減	3	左→右

註

- 字尾系列寫做「i++」、「j--」。
- 字頭系列寫做「++i」、「--j」。

複合指定運算子

運算子	意義	運算順序	結合性
++	字尾遞增	2	左→右
--	字尾遞減	2	左→右
++	字首遞增	3	左→右
--	字首遞減	3	左→右

註

- 字尾系列寫做「i++」、「j--」。
- 字頭系列寫做「++i」、「--j」。
- 不管是字首還是字尾，代表的意義都是 i = i + 1 和 j = j - 1

字首系列 vs 字尾系列

試試看

- `cout << i++ << endl;`
- `cout << ++i << endl;`
- `i++; cout << i << endl;`
- `++i; cout << i << endl;`

比較這四者之間有何不同？

字首系列 vs 字尾系列

試試看

- `cout << i++ << endl;`
- `cout << ++i << endl;`
- `i++; cout << i << endl;`
- `++i; cout << i << endl;`

比較這四者之間有何不同？

字首系列

- 會先做運算，再回傳

字首系列 vs 字尾系列

試試看

- `cout << i++ << endl;`
- `cout << ++i << endl;`
- `i++; cout << i << endl;`
- `++i; cout << i << endl;`

比較這四者之間有何不同？

字首系列

- 會先做運算，再回傳
- 回傳值是運算後的值

字首系列 vs 字尾系列

試試看

- `cout << i++ << endl;`
- `cout << ++i << endl;`
- `i++; cout << i << endl;`
- `++i; cout << i << endl;`

比較這四者之間有何不同？

字首系列

- 會先做運算，再回傳
- 回傳值是運算後的值

字尾系列

- 會先回傳，再做運算

字首系列 vs 字尾系列

試試看

- `cout << i++ << endl;`
- `cout << ++i << endl;`
- `i++; cout << i << endl;`
- `++i; cout << i << endl;`

比較這四者之間有何不同？

字首系列

- 會先做運算，再回傳
- 回傳值是運算後的值

字尾系列

- 會先回傳，再做運算
- 回傳值是運算前的值

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

註

- 答案：沒有人知道！

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

註

- 答案：沒有人知道！
- 在不同的編譯器會有不同的結果。

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

註

- 答案：沒有人知道！
- 在不同的編譯器會有不同的結果。
- 大多數是因爲在同一行之內改同一變數一次以上。

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

註

- 答案：**沒有人知道！**
- 在不同的編譯器會有不同的結果。
- 大多數是因爲在同一行之內改同一變數**一次**以上。

其他例子

- `i = ++i + 1;`

未定義行爲

例子

```
int i = 0;
cout << i++ + ++i << endl;
```

答案是多少？

註

- 答案：沒有人知道！
- 在不同的編譯器會有不同的結果。
- 大多數是因爲在同一行之內改同一變數一次以上。

其他例子

- `i = ++i + 1;`
- `i++*++i+i--*--i`

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行為
7. 其他運算子
8. 結論

其他運算子

運算子	意義	運算順序	結合性
`sizeof`	求記憶體大小	3	右→左
`(type)`	強制轉型	3	右→左
`,`	逗號	18	左→右

其他運算子

運算子	意義	運算順序	結合性
`sizeof`	求記憶體大小	3	右→左
(type)	強制轉型	3	右→左
,	逗號	18	左→右

觀念

- 萬物對計算機而言皆是「運算」。

其他運算子

運算子	意義	運算順序	結合性
`sizeof`	求記憶體大小	3	右→左
(type)	強制轉型	3	右→左
,	逗號	18	左→右

觀念

- 萬物對計算機而言皆是「運算」。
- 既然是運算，就有「結合性」和「運算順序」。

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- `sizeof(int)`

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- `sizeof(int)` 在筆者的機器上會是 4 位元組

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- `sizeof(int)` 在筆者的機器上會是 4 位元組
- `sizeof(double)`

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- `sizeof(int)` 在筆者的機器上會是 4 位元組
- `sizeof(double)` 在筆者的機器上會是 8 位元組

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- sizeof(int) 在筆者的機器上會是 4 位元組
- sizeof(double) 在筆者的機器上會是 8 位元組
-
```
bool b = true;
cout << sizeof b << endl;
```

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- sizeof(int) 在筆者的機器上會是 4 位元組
- sizeof(double) 在筆者的機器上會是 8 位元組
- ```
  bool b = true;
  cout << sizeof b << endl;
  ```
  在筆者的機器上會是 1 位元組
  ```

sizeof 運算子

用途

可以知道某個資料型態或變數所使用的位元組數。

例子

- sizeof(int) 在筆者的機器上會是 4 位元組
- sizeof(double) 在筆者的機器上會是 8 位元組
- ```
  bool b = true;
  cout << sizeof b << endl;
  ```
  在筆者的機器上會是 1 位元組

注意

每個人的機器會出現不同的結果，像是之前提到有些機器的 int 會是 2 個位元組。
  ```

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

例子

- `int` 變數 x 轉爲 `double`

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

例子

- int 變數 x 轉爲 double $\rightarrow$ (double) x 或者 double(x)

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

例子

- `int` 變數 x 轉爲 `double` → `(double) x` 或者 `double(x)`
- `double` 常數轉爲 `int`

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

例子

- `int` 變數 x 轉爲 `double` → `(double) x` 或者 `double(x)`
- `double` 常數轉爲 `int` → `(int) 5.14` 或者 `int(5.14)`

型別轉換

(type) 運算子

C++ 有資料型態，若型態間需要強制轉換就要使用這個運算子

例子

- `int` 變數 x 轉爲 `double` → `(double) x` 或者 `double(x)`
- `double` 常數轉爲 `int` → `(int) 5.14` 或者 `int(5.14)`

註

我們說過資料型態代表容器可以裝的資料類型不同，因此我們之後會遇到需要「改變資料類型」的狀況，那時需要做型別轉換。

逗號運算子

意義

- 最常被人誤解的運算子

逗號運算子

意義

- 最常被人誤解的運算子、運算子

逗號運算子

意義

- 最常被人誤解的運算子、運算子、運算子！(因爲很重要所以要說三次)

逗號運算子

意義

- 最常被人誤解的運算子、運算子、運算子！(因爲很重要所以要說三次)
- 逗號運算子可以分隔兩個運算式，回傳值是右邊運算式的回傳值。

逗號運算子

意義

- 最常被人誤解的運算子、運算子、運算子！(因爲很重要所以要說三次)
- 逗號運算子可以分隔兩個運算式，回傳值是右邊運算式的回傳值。

實例

用迴圈讀入 n，直到 $n = 0$ 停止：

逗號運算子

意義

- 最常被人誤解的運算子、運算子、運算子！(因爲很重要所以要說三次)
- 逗號運算子可以分隔兩個運算式，回傳值是右邊運算式的回傳值。

實例

用迴圈讀入 n，直到 $n = 0$ 停止：

```
int n;
while (cin >> n, n) {
}
```

大綱

1. 簡介
2. 程式架構
   - 基本程式架構
   - 輸出
   - 變數
   - 輸入
   - 資料型態
3. 算術運算子
   - 運算性質
   - 結合性與運算順序
   - 整數除法與除零問題
   - 應用：取餘數
4. 比較和邏輯運算子
   - 簡化規則
   - 短路運算
5. 位元運算子
   - int 和 long long 的儲存形式
   - 常用技巧：連續的 1
   - 常用技巧：遮罩與指定位元
   - Parity
   - xor 性質
6. 指定運算子
   - 運算性質
   - 未定義行爲
7. 其他運算子
8. 結論

小結

重點整理

1. 句子結尾是分號「;」。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
4. 除以零會遇到的現象。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
4. 除以零會遇到的現象。
5. 「零」代表 `false`，「非零」代表 `true`。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
4. 除以零會遇到的現象。
5. 「零」代表 `false`，「非零」代表 `true`。
6. 邏輯運算子是短路運算。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
4. 除以零會遇到的現象。
5. 「零」代表 `false`，「非零」代表 `true`。
6. 邏輯運算子是短路運算。
7. `int` 和 `long long` 如何儲存，以及位元運算技巧。

小結

重點整理

1. 句子結尾是分號「;」。
2. 初始化的重要性。
3. C++ 運算子依照運算順序和結合性做運算，大約了解運算的優先順序。
4. 除以零會遇到的現象。
5. 「零」代表 `false`，「非零」代表 `true`。
6. 邏輯運算子是短路運算。
7. `int` 和 `long long` 如何儲存，以及位元運算技巧。
8. 注意未定義行爲。

運算子小結

運算優先順序

一元運算子 → 算術運算子 → 比較運算子 → 邏輯運算子 → 位元運算子 → 指定運算子、複合指定運算子 → 逗號運算子

運算子小結

運算優先順序

一元運算子 → 算術運算子 → 比較運算子 → 邏輯運算子 → 位元運算子 → 指定運算子、複合指定運算子 → 逗號運算子

觀念

- 萬一忘記順序怎麼辦呢？

運算子小結

運算優先順序

一元運算子 → 算術運算子 → 比較運算子 → 邏輯運算子 → 位元運算子 → 指定運算子、複合指定運算子 → 逗號運算子

觀念

- 萬一忘記順序怎麼辦呢？
- 當然是把括號括好啦～運算順序只要知道大概，這不是必背的東西，我們的目的是「寫出好程式」而非在運算順序上多作著墨！