**Software Testing Final Project**

* 為路邊撿來的(?) C++ unsigned 大數上 test

**Interface**

* C++ template，用 nontype parameter BitLength 決定大數的儲存空間大小，如果運算結果超過這個大小還是會「overflow」
* 例子:
* LargeInteger<6400> L1 // 可儲存 0 - 2^6400-1

**Constructors:**

// Constructors.

// Constructors.

LargeInteger( );

LargeInteger(const LargeInteger<BitLength>& RHS);

LargeInteger(LargeInteger<BitLength>&& RHS);

LargeInteger(QWORD Number);

// Construct from other bit length.

template<size\_t RHSBitLength>

explicit LargeInteger(const LargeInteger<RHSBitLength>& RHS);

* default ctor
* copy, move ctor with same BitLength
* copy from differnt BitLength, called RHSBitLength
* construct from an QWORD value, i.e. unsigned long long

**Destructor.**

~LargeInteger();

**Bitwise operators.**

// Bitwise operators.

LargeInteger<BitLength>& operator>>=(int NumberToShift);

LargeInteger<BitLength>& operator<<=(int NumberToShift);

LargeInteger<BitLength>& operator&= (const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator|= (const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator^= (const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength> operator~ ( ) const;

LargeInteger<BitLength> operator>>(int NumberToShift) const;

LargeInteger<BitLength> operator<<(int NumberToShift) const;

**Comparison operators**

// Comparison operators.

bool operator==(const LargeInteger<BitLength>& RHS) const;

bool operator!=(const LargeInteger<BitLength>& RHS) const;

bool operator>=(const LargeInteger<BitLength>& RHS) const;

bool operator<=(const LargeInteger<BitLength>& RHS) const;

bool operator> (const LargeInteger<BitLength>& RHS) const;

bool operator< (const LargeInteger<BitLength>& RHS) const;

**Conversions**

// Convertions.

explicit operator bool () const;

explicit operator QWORD() const;

explicit operator DWORD() const;

* 把大數轉成一些 builtin type

**Assignment operators**

// Assignment operators.

LargeInteger<BitLength>& operator=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator=(LargeInteger<BitLength>&& RHS);

LargeInteger<BitLength>& operator=(QWORD Number);

**Arithmetic operators.**

// Arithmetic operators.

LargeInteger<BitLength>& operator+=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator-=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator\*=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator/=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength>& operator%=(const LargeInteger<BitLength>& RHS);

LargeInteger<BitLength> operator+ ( ) const;

LargeInteger<BitLength> operator- ( ) const;

LargeInteger<BitLength> operator++(int);

LargeInteger<BitLength>& operator++( );

LargeInteger<BitLength> operator--(int);

LargeInteger<BitLength>& operator--( );

**Other member functions.**

**I/O data without casting.**

// I/O data without casting.

void GetRawData (void\* OutputDataBuffer) const;

void SetRawData (const void\* InputDataBuffer);

* 此大數實作的 IO 不 depend on iostream，純粹用 binary data(這裡用void\* 表示)接收或設定大數內容
  + 很多專案其實也不 depend on iostream，有些甚至會自己實作 IO library 或者 string 等等拿來做基建設的 class，例如 LLVM

**I/O via a string.**

// I/O via a string.

const char\* ConvertToString (unsigned int Radix) const;

void SetFromString (const char\* StrNumber,

unsigned int Radix);

* 從 C style string 接收或設定大數內容，還要給定 radix(幾進位)

**Presented in byte**

// Presented in byte.

size\_t GetBufferSize ( void) const;

**Private members**

QWORD\* NumberCell;

static const int QWORDLength =(BitLength>0)?

((BitLength/64)+((BitLength%64)?1:0)):

1;

* NumberCell 指向大數儲存的空間，QWORDLength 代表大數儲存空間的大小(以 8byte 為單位)

**Output functions used by the public function ConvertToString**

// Output functions used by the public function ConvertToString.

void RegularConvertToString (unsigned int Radix , char\* StrToWrite) const;

void PowerOfTwoConvertToString(unsigned int Radix , char\* StrToWrite) const;

**Input functions used by the public function SetFromString**

// Input functions used by the public function SetFromString.

void RegularSetFromString (const char\* StrNumber, unsigned int Radix);

void PowerOfTwoSetFromString (const char\* StrNumber, unsigned int Radix);

**Friend declaration**

// Friend declaration.

template<size\_t OtherBitLength>

friend class LargeInteger;

**Global Arithmetic operator declaration.**

// Opertor+

template<size\_t Length>

LargeInteger<Length> operator+(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator+(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator+(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator+(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator-

template<size\_t Length>

LargeInteger<Length> operator-(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length> operator-(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator-(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator-(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator\*

template<size\_t Length>

LargeInteger<Length> operator\*(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator\*(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator\*(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator\*(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator/

template<size\_t Length>

LargeInteger<Length> operator/(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length> operator/(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator/(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator/(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator%

template<size\_t Length>

LargeInteger<Length> operator%(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length> operator%(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator%(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator%(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

**Bitwise global operator declaration.**

// Operator&

template<size\_t Length>

LargeInteger<Length> operator&(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator&(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator&(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator&(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator|

template<size\_t Length>

LargeInteger<Length> operator|(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator|(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator|(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator|(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

// Operator^

template<size\_t Length>

LargeInteger<Length> operator^(const LargeInteger<Length>& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator^(const LargeInteger<Length>& LHS, LargeInteger<Length>&& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator^(LargeInteger<Length>&& LHS, const LargeInteger<Length>& RHS);

template<size\_t Length>

LargeInteger<Length>&& operator^(LargeInteger<Length>&& LHS, LargeInteger<Length>&& RHS);

**Test**

* 測試之前先在 API 上面加入兩個 getter，把 private member 傳出來:

// Test Utilities

const QWORD\* GetInternalBuffer() const{

return NumberCell;

}

int GetInternalBufferLength() const{

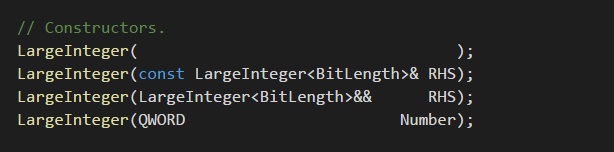
return QWORDLength;

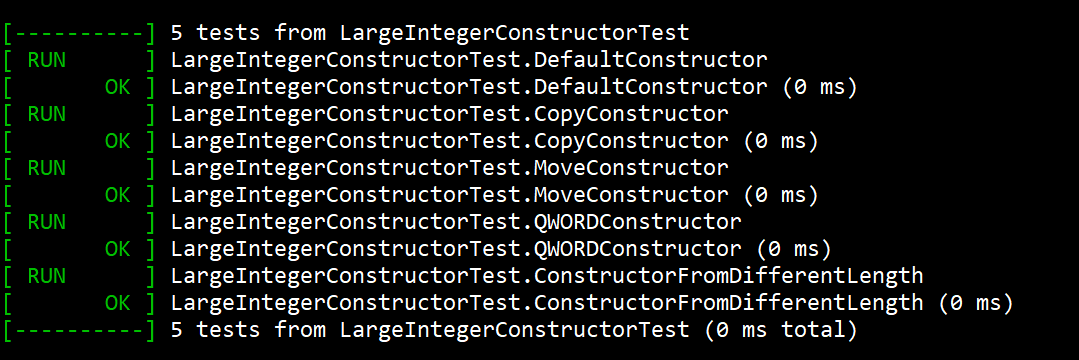
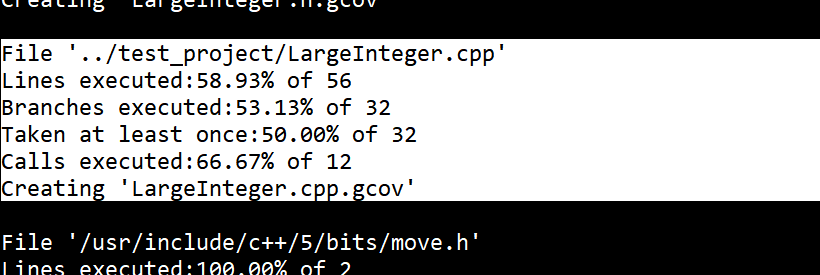
}

* 一個是存放大數 binary 的 buffer，一個是 buffer size

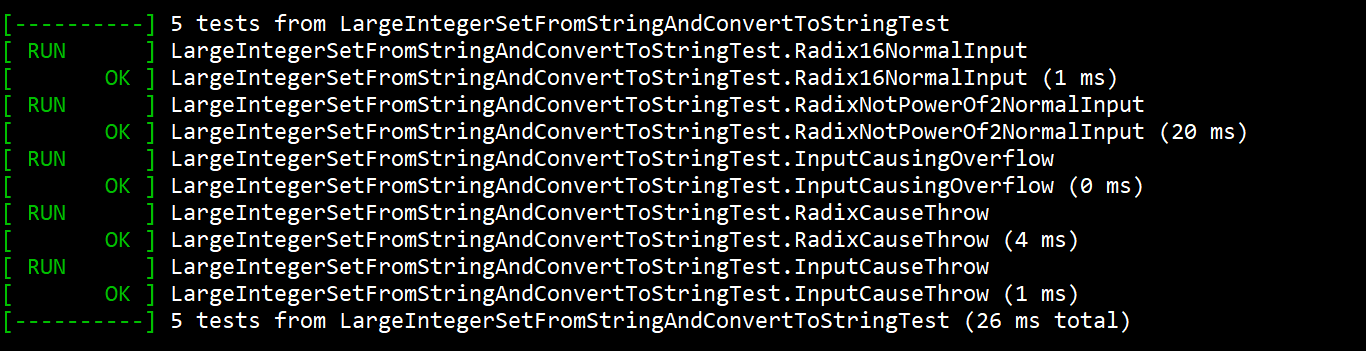
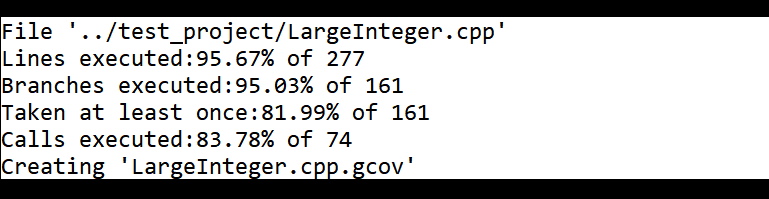
**Constructors:**

* ctors 總共用一個 test case 測試，叫做 LargeIntegerConstructorTest
* 除此之外還為它們寫一個 test fixture，可以達成 code sharing

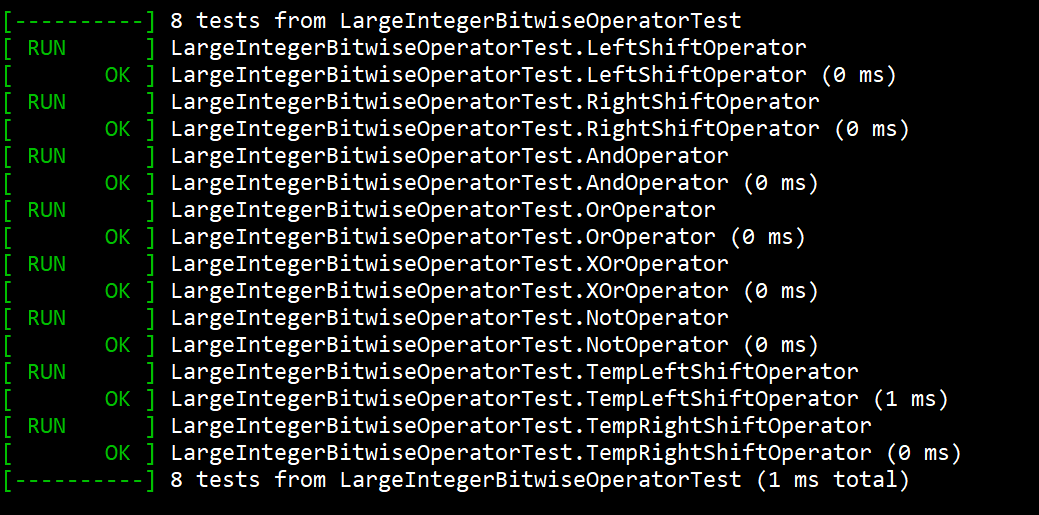
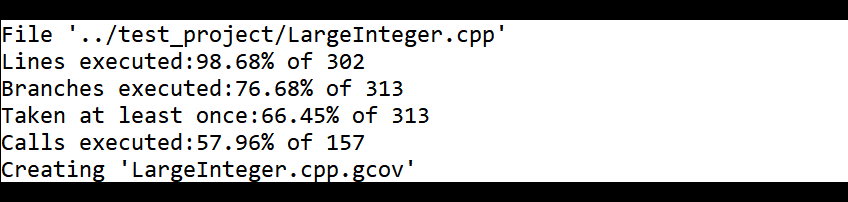


* 總之是測這四個 ctor
  + default: 看要到的空間是否初始化為 0
  + copy: 看新物件的空間跟舊物件的空間是否不同，並且內容是否一致
  + move: 看新物件的空間是否等於舊物件的空間，並且 move 之後舊物件指向 nullptr
  + ctor from different size:
    - 新物件如果比較大，則 leading bits 要填 0
    - 如果比較小就要把 leading bits 刪除
* test:
* 
* 

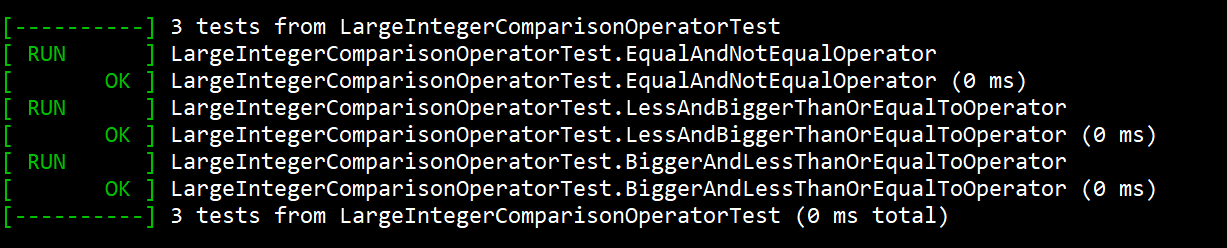
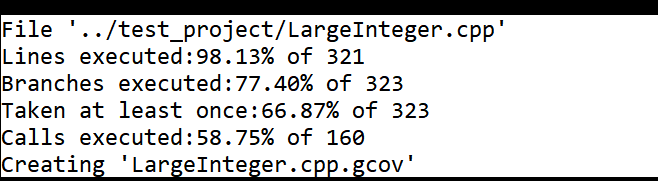
**Set And Convert to String:**

* 接收或設定一個字串，並且要給多少進位:
* 例子:
* string str1 = "0101010101010101";
* LargeInteger<6400> L;
* int radix = 2;
* L.SetFromString(str1.c\_str(), radix);
* cout << L.ConvertToString(radix); // output 0101010101010101
* 我丟入各種字串跟進位；確認輸出字串是否正確，以及當有非法字元或者進位時，是否會 throw exception
* 分成很多 test:
  + Radix16NormalInput
  + RadixNotPowerOf2NormalInput
  + InputCausingOverflow
  + RadixCauseThrow
  + InputCauseThrow
* test:
* 
* 

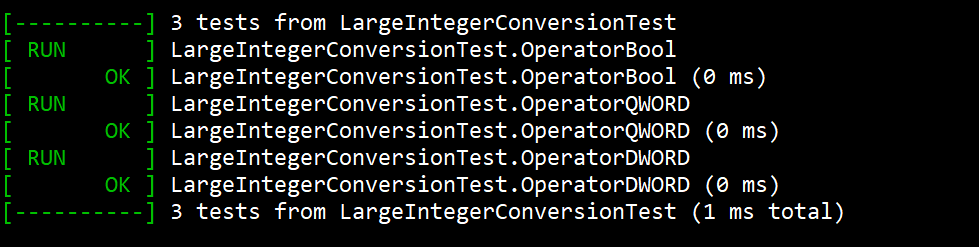
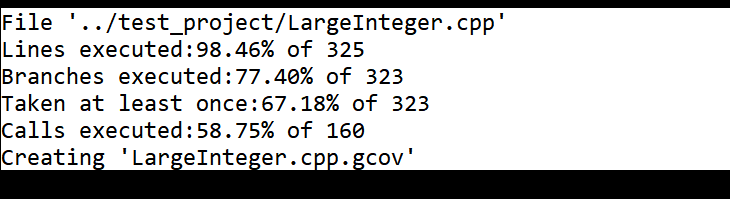
**Bitwise Operators:**

* 總之就是給很多 tests 跟邊界條件看 output 是否正確:
* test:
  + LeftShiftOperator
  + RightShiftOperator
  + AndOperator
  + OrOperator
  + XOrOperator
  + NotOperator
  + TempLeftShiftOperator
  + TempRightShiftOperator
* 
* 
* 這裡會發生一個奇怪的現象：test code 變多，branch coverage 卻變少了；原因是因為我要測試的 code 本身是 template，我只要生出不同的 instantiation，並且用到特定的 member function，compiler 就會把這個 member 跟這個 member 呼叫到的其他 member 都生出來，在這裡我是用到了 LargeInteger<6401>::SetFromString 跟 LargeInteger<6401>::ConvertToString，然後這兩個會呼叫到不少 member function，這些 member function 我就沒有額外測試，所以 branch coverage 會降低；但我認為測試他們沒差，因為我原本使用的 LargeInteger<6400> 對應的 member function 我都有測到。另外也可以看到，隨著 test code 變多，要被測試的程式碼的數量也變更多了，原因也是因為原本的程式碼是 template… 測越多才會生越多 code

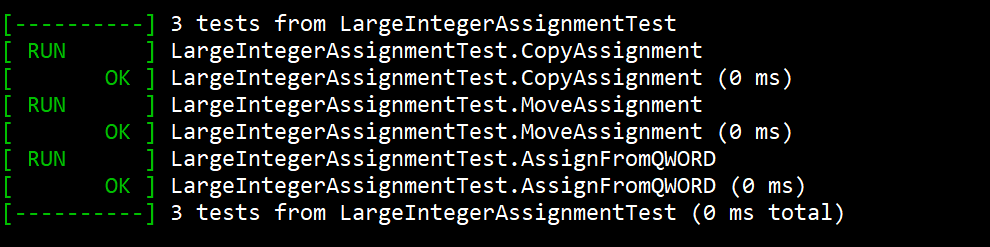
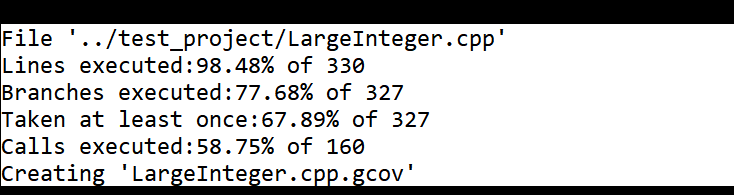
**Comparison Operators:**

* 總之就是給很多 tests 跟邊界條件比較結果是否正確:
* test:
  + EqualAndNotEqualOperator
  + LessAndBiggerThanOrEqualToOperator
  + BiggerAndLessThanOrEqualToOperator
* 
* 

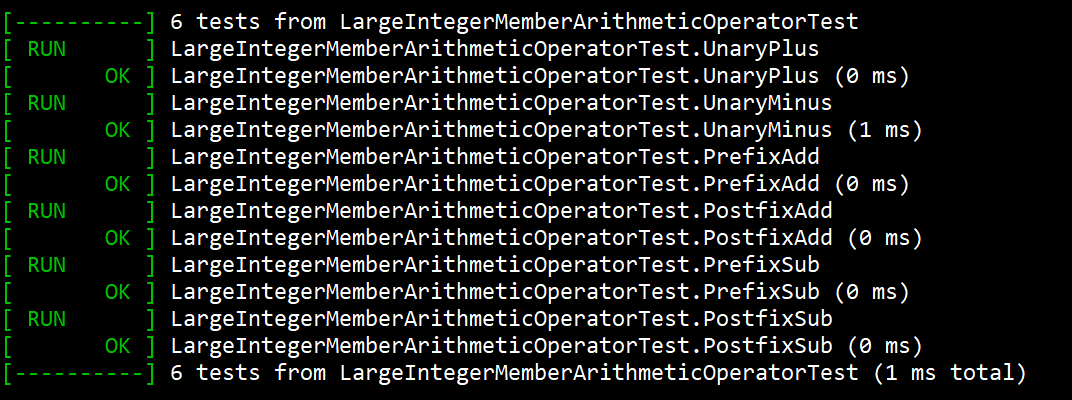
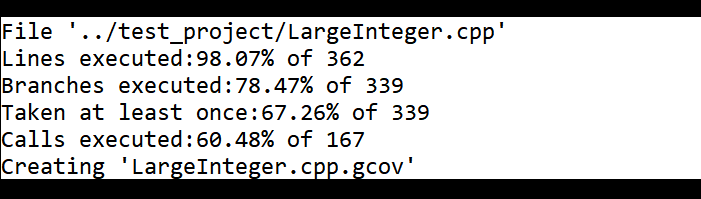
**Conversion Operators:**

* 看轉換出來的 bool, unsigned 跟 long long unsigned 是否正確
* test:
  + OperatorBool
  + OperatorQWORD
  + OperatorDWORD
* 
* 

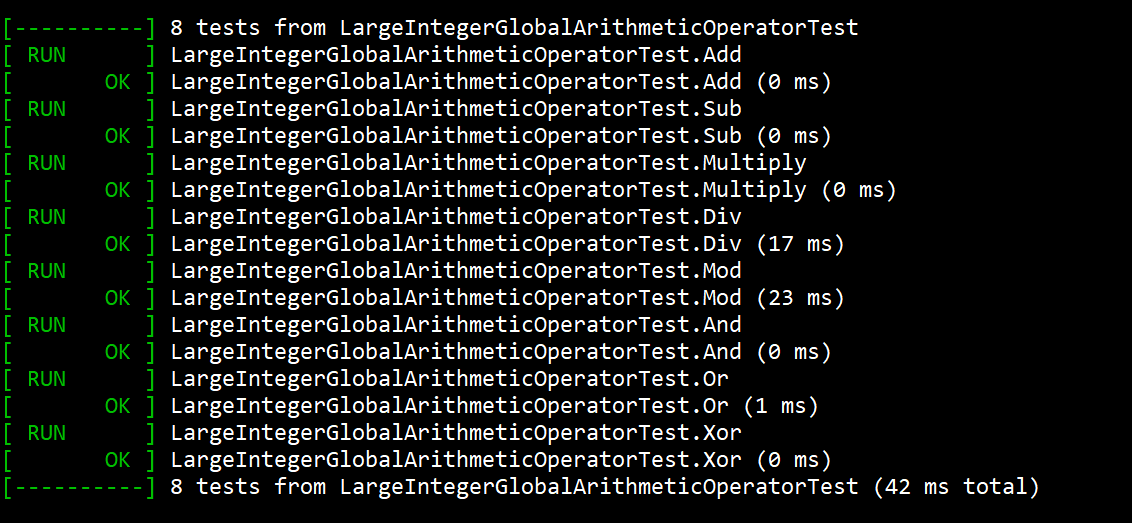
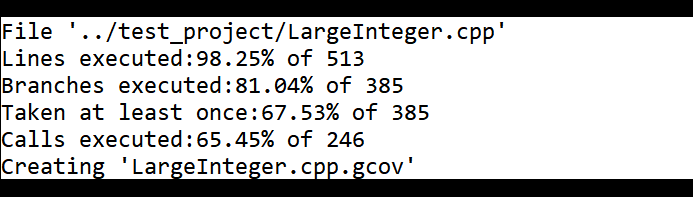
**Assignment Operators:**

* 看 copy assign 跟 assign from QWORD 之後，原物件是否等於被 assign 的物件
* 看 move assign 之後，原物件的儲存空間是否被 reset 成 NULL
* test:
  + CopyAssignment
  + MoveAssignment
  + AssignFromQWORD
* 
* 

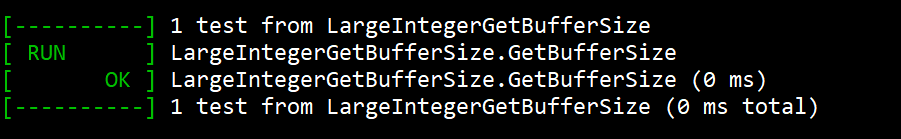
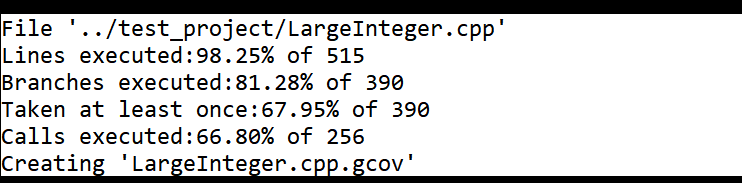
**Arithmetic “member” Operators:**

* 總之就是生一堆測資看運算結果是否正確:
* test:
  + UnaryPlus
  + UnaryMinus
  + PrefixAdd
  + PostfixAdd
  + PrefixSub
  + PostfixSub
* 
* 

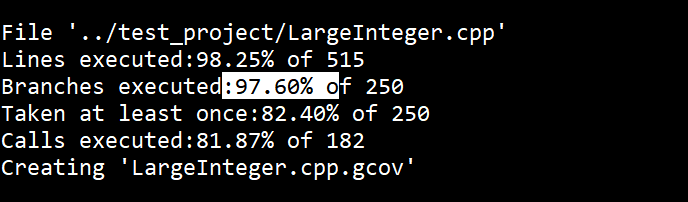
**Arithmetic “global” Operators:**

* 總之就是生一堆測資看運算結果是否正確:
  + 還有 global 有支援 move，要檢查 move 的結果
* test:
  + Add
  + Sub
  + Multiply
  + Div
  + Mod
  + And
  + Or
  + Xor
* 
* 
* 測完這個之後 test code 直接從 316 行變 513 行了

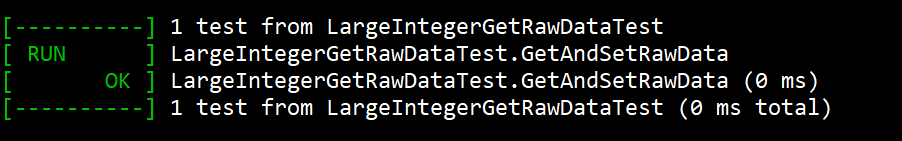
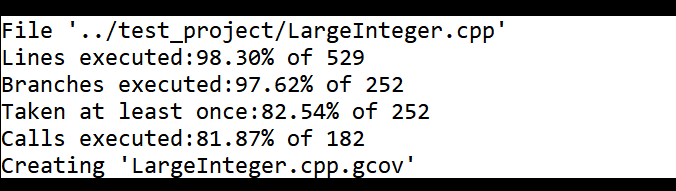
**test for Member function GetBufferSize:**

* 只是看特定 BitLength 的 LargeInteger 的 buffer 大小是否正確
* test:
  + GetBufferSize
* 
* 

**Summary**

* 總之我會因為上述的 template 問題導致 branch coverage 不高；如果把會些會導致生出那種版本的 test code 註解掉，coverage 就會升高了:
* 
* branch 數量變少，coverage 也變高了

**test for Set and Get Raw Data:**

* 忘了測這個 API…
  + 他不會對 buffer 做 boundary check，是個有點糟的 API
* test:
  + GetAndSetRawData
* 
* 

**google test output:**

* 總結 Google test 的 output:  
  