C/C++ Sequence Point 问题介绍

引子

C / C++中的序列点问题是一个非常常见的问题，如果不注意，则非常容易在日常的开发中遇到类似的问题。本文的由来也是由于偶然发现在 ullib/dict 下广为使用的签名函数的 bug。
先来看几个例子，其中 i、j、k、crc 等变量均为 int 类型：

```
1. printf("%d, %d\n", i++, i++);
2. i = ++i + 1;
3. a[i] = i++;
4. bar(i++, bar(++i, 1));
5. i = j = 1;
6. (i += 2) = 2;
7. i = ++j;
8. itemp = ((unsigned char)str[i] * Mod_Prime_List_1[0xFF & (i++)] + itemp);    /* ullib dict 下的代码段 */
9. crc = (crc>>8) ^ CRCTable[(buffer[i] ^ crc) & 0xff];    /* 常见 crc 算法实现 */
10. (i = j) + (i = k);
11. *( *p = 2, p) = 7;    //p 为指向有效地址空间的 int 指针类型。
12. j = i ? ++i : --i;
```

先公布答案，上面的例子中只有第 9、第 11 和第 12 个是符合标准（C 和 C++两种标准）的写法，而第 5 和第 7 个看似非常正确并广为使用的例子，事实上只符合 C 标准中的定义，在 ISO C++ 2003 的标准中这两个例子是 undefined behavior（很让人 Orz），至于第 6 个不能被 C 编译器编译，只能在 C++编译器下通过编译（但这并不意味着这是符合标准的写法），但结果依然是未定义的（更加让人 囧 rz）。
注：由于撰写本文的时间比较仓促，很多地方直接引用了标准的说法，但尽可能多的给出了翻译和自己的理解，保证在阅读过程中能尽量少的去看那些晦涩的 language law。对于很多词没有使用惯用的翻译（比如 Side effects 没有采用通常的翻译解释为“副作用”，因为个人认为这样会带来不必要的误解），因此对很多特有词没有进行翻译。
另外对于一些阅读起来可能有难度的部分，给出了★的标志。

什么是 Side effects？

在搞清楚什么是 Sequence Point 之前，我们需要先了解什么叫 Side effects。
在 C99 对 Side effects 定义如下：
Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. [1]
C++2003 有类似定义如下：

Accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment.[2]

C99 和 C++2003 对于这个问题的定义几乎一致，指对数据或者文件等对象进行修改的操作，被称为 side effects，它会导致程序执行环境发生状态转移。对表达式的求值有可能导致 Side effects。

需要说明的是，一般不把对 cpu register 数据的修改看作发生一次 side effects，因为 cpu 在进行计算操作的时候必须将数据 load 到 register 中，而不能直接根据内存单元地址进行计算，而 side effects 是指对内存、文件等相对于 cpu 是外存设备的数据的修改。

相对应的，对于声明为 volatile-qualified 的数据类型，无论是读写，都认为有一次强制的内存读写发生，具体可以参阅后面对 volatile-qualified 类型的专门描述。

什么是 Sequence Point?

C99 与 C++ 2003 对此有相同的定义：

At certain specified points in the execution sequence called sequencepoints, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.[1][2]

中文含义是：Sequence Point 是在执行中一个确定的时间点，这些时间点被特别规定，在这些时间点以前的 evaluations 中，所有的 Side effects 已经完成，而此时间点以后的 evaluations 的 Side effects 还没有开始。

C99（Annex C）对 Sequence Point 规定的时间点列表为：

— The call to a function, after the arguments have been evaluated (6.5.2.2).

— The end of the first operand of the following operators: logical AND && (6.5.13); logical OR || (6.5.14); conditional ? (6.5.15); comma , (6.5.17).

— The end of a full declarator: declarators (6.7.5);

— The end of a full expression: an initializer (6.7.8); the expression in an expressionstatement (6.8.3); the controlling expression of a selection statement (if or switch)(6.8.4); the controlling expression of a while or do statement (6.8.5); each of the expressions of a for statement (6.8.5.3); the expression in a return statement(6.8.6.4).

— Immediately before a library function returns (7.1.4).

— After the actions associated with each formatted input/output function conversion specifier (7.19.6, 7.24.2).

— Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call (7.20.5).

C++ 2003 中对此列表的描述与 C99 基本一致。

另外，在参考文献 3 中提到：

需要特别说明的是，由于 operator||、operator&&以及 operator,可以重载，当它们使用函数语义的时候并不提供 built-in operators 所规定的那几个序列点，而仅仅只是在函数的所有参数求值后有一个序列点，此外函数语义也不支持||、&&的短路语义，这些变化很有可能会导致难以发觉的错误，因此一般不建议重载这几个

运算符。

常见的 Sequence Point 问题

在 C99 和 C++ 2003 中有如下规定：

Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.[1][2]

从上述规定及 Sequence Point 的定义可以看出，在连续两个 Sequence Point 之间，如果某个对象需要被写入，则在这两个 Sequence Point 之间的操作中，对该对象的访问只应局限于用于计算将要写入的值，而不能对这个对象产生多次写请求。一般情况下，连续的两个 Sequence Point 是指一个 full expression（当然也有其它情况，如上面引用的 C99 的 Annex C）。

举例来说，有如下代码段：

```
int i;       /*  ①  */
i = 0;       /*  ②  */
```

在②句，这是一个标准定义的 full expression，所以在①结尾“；”和②结尾“；”位置有一个 Sequence Point，在两个 Sequence Point 中，产生一次 Side effects，所以这条语句是 well-defined 的。

而对于引子中的例子：

```
2. i = ++i + 1;
3. a[i] = i++;
```

很明显，他们都在两个 Sequence Point 之间尝试对 i 进行了两次赋值，例如在 2 中，一次发生在“++i”，一次发生在“=”上。这会造成 undefined behavior。

The order of evaluation 带来的问题

C99 中规定：

1. Except as specified later (for the function-call (), &&, ||, ?:, and comma operators), the order of evaluationof subexpressions and the order in which side effects take place are both unspecified.

2. The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

C++ 2003 中有类似规定：

Except where noted, the order of evaluation of operands of individual operators and subexpressions of individual expressions, and the order in which side effects take place, is unspecified.

这些规定，都明确的说明了在对于表达式、函数实参的求值上，其求值的顺序以及 Side effect 的顺序都是 undefined。

例如如下代码段：

```
int i, j, *p;
p = &i;
j = 0;
*p = j++;     /*  ①  */
```

根据标准规定，在①处，进行“j++”和对*p 的 Side effects 的顺序是由编译器自定义的。
理解了这个，就不难理解引子中如下代码是如何发生了问题：

```
8. itemp = ((unsigned char)str[i] * Mod_Prime_List_1[0xFF & (i++)] + itemp);
10. (i = j) + (i = k);
```

其中第 8 个是现在 ullib 下签名函数的一段核心实现，事实上，基于上述标准的规定，所以，“i”和“i++”是两个独立的 subexpression，那么他们的先后顺序是 undefined 的，所以第一个 i 在计算时，可能使用老的值，也可能使用“i++”后发生了 Side effects 的值。
第 10 个问题与此类似，可能是 j 也可能是 k 的值，这依赖于 Side effects 的顺序。

而在函数传参的过程中依然存在 Sequence Point 的问题，C99 规定：
The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.
结合上述 C99 的 Annex C，可以很容易理解引子中另外两个例子的错误：

```
1. printf(“%d, %d\n”, i++, i++);
4. bar(i++, bar(++i, 1));
```

在很多无聊的考试会出现这种类型的题目，不同的编译器都会给出不同的结果。比如第一个问题在使用 gcc 3.x 的测试中可以看到，在开优化和不开优化的情况下会出下截然不同的结果。

从另一个角度看 Sequence Point（★）

现在，我们使用 Abstract syntax tree[5]来分析 Sequence Point 问题。
具体的关于 Abstract syntax tree 的生成方法见参考文献 5。
1、代码： `x += x * x;`

```
Abstract syntax tree:
          +=  [x]
         /   \
        /     \
      (x)      *
              / \
             (x) (x)
```

从下往上看，圆括号代表读，方括号代表写，可以看出写发生在所有读之后，并且只有一次，所以是 well-defined。
2、代码：

```
int x[2], *y;
y=x;
```

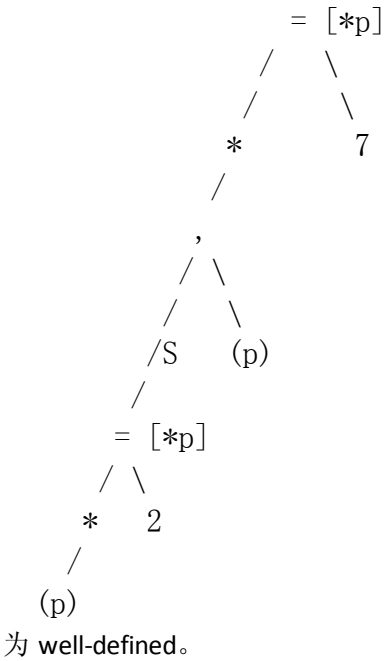
```
    *y = f(y++);
Abstract syntax tree:
          = [*y]
        /    \
       /      \
      *       (S)
     /        /\
    /        /  \
  (y)       f   ++ [y]
                   \
                   (y)
```

产生了多次对 y 的 Side effects，所以是 undefined。

3、例子 11

11. *(*p = 2, p) = 7; //p 为指向有效地址空间的 int 指针类型。

Abstract syntax tree:



为 well-defined。

Signal 对 Sequence Point 的影响（★）

C99 中对此的定义：

When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

简言之就是执行顺序被信号中断的情况下，上一个 Sequence Point 之前的结果是可信赖的，而从上一个到后面紧接着的这一个是可能发生问题的。

请注意，这里用的是 **abstract machine**，并且由于这条规定与 C99 的 7.14.1.1p5 冲突，被参考文献 6 严重鄙视，而 C++ 2003 则对此做出了 concrete machine 的规定，尽管这依然于

7.14.1.1p5 冲突。

Volatile-qulified 类型变量与 Sequence Point（★）

在参考文献 3 中提到了这个问题，并引入了 ctrlz 与 RoachCock 两位大牛对这个问题的看法，我个人更加赞同 RoachCock 的观点，即：两个 Sequence Point 之间多次读取同一个 volatile-qualified 类型变量是合法的，但不能被优化为读取一次，因为标准规定：

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3.

一段在嵌入式开发中很常见的代码示例如下：

```
extern volatile int i;
if (i != i) {   // 探测很短的时间内 i 是否发生了变化
   // ...
}
```

如果 i != i 被优化为只读一次，则结果恒为 false，

关于这个问题，一个可以肯定的结论是：

对于 volatile-qualified 类型的变量在跨序列点之后必须要重新读取，volatile 就是用来阻止编译器做出跨序列点的过激优化的，而对于 non-volatile-qualified 类型的跨序列点多次读取则可能被优化成只读一次（直到某个语句或者函数对该变量发生了修改，在此之前编译器可以假定 non-volatile-qualified 类型的变量是不会变化的，因为目前的 C/C++抽象机器模型是单线程的）。[3]

C++ 2003 中的一些不太好的新规定（★）

最后来看例子中的三个让人意想不到的问题：

```
5. i = j = 1;
6. (i += 2) = 2;
7. i = ++j;
```

在 C 语言中，assignment operators 的结果是 non-lvalue，而 C++2003 确将 assignment operators 的结果改成了 lvalue，这导致了很多习惯性的代码成为了非标准的代码，如上述例子 5 和 7，也使得 6 成为可以编译通过的代码，但是结果确是 undefined 的。

在例子 5 中，j=1 的结果是 lvalue，由于这是一个 subexpression，则 i 到底是读取 j 的老值还是赋了 1 的值，这是 undefined 的。

在例子 6 中，由于(i += 2)的结果是 lvalue，则可以继续对其进行赋值，但是由于这里产生了两次 Side effects，所以行为依然是未定义的。

在例子 7 中，由于 C++ 2003 规定前置“＋＋”使得结果成为 lvalue，则在两个 Sequence Point 之间再次发生两次 Side effects，所以行为依然是未定义的。

一个不知道是好还是坏的结果是，上面的这三个例子，尤其是广为流传的 5 和 7，在现在的

通用 C++编译器中均没有处理，它们完全无视了 C＋＋标准对此的规定，在测试的 gcc 2.9.6、gcc 3.x，icc 10.x 和 vc2005 上，均没有给出警告（打开所有警告的情况下），而是继续按照 C 的标准进行了编译。
就这个问题，Andrew Koenig 大牛向 C＋＋标准委员会提交了新的建议，要求为 assignment operators 增加新的 Sequence Point，但似乎到目前为止还没有结果，大家可以在 http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html 看到这份提议，名字为“Sequence points and lvalue-returning operators”。

a[i] = i++的另一个问题

那么，如果有一句是“a[i] = i++;”，虽然不知道 a[]的哪个分量会被改写，但 i 是不是会加 1 呢？
答案是否定的，由于 Sequence Point 会导致这条语句的未定义行为。

编译器报警

虽然编译器并非完全的支持了各种标准，但是尽量的使用编译器给我们的选项，打开报警是非常有帮助的。
一般出现 Sequence Point 的时候，cc（或 gcc 编译 c 代码）会告知“warning: operation on `i' may be undefined”类似的报警。
在实际的使用中发现，cc 编译器对警告的处理比 g++更好，但是 cc 只是一个 C 编译器，不能编译 C++代码，g++对一些常见的 Sequence Point 问题视而不见。

总结

Sequence Point 是一个看上去很简单，但其实比较并非想象那么简单的问题，要想深入的搞清楚其中来龙去脉的细节，需要仔细阅读标准并对编译器的实现有一些基本的了解。

不过幸运的是，大多数时候我们并不需要了解这些细节，如果单纯为了避免序列点问题，一般说来，只需要记住我们不要在一个语句中多次对同一个变量赋值和读取，再加上编译器给出的警告，就可以有效的避免各种问题了（个人觉得能理解引子中提出的问题就足够了）。

由于时间仓促，可能其中部分叙述不太准确，欢迎大家发邮件来讨论，我的邮件地址：mailto:jingmi@baidu.com。

附录 1

参考文献：

1.C99 标准

2.C++ 2003 标准

3.http://www.newsmth.net/bbscon.php?bid=335&id=177091

4.http://c-faq.com/expr/seqpoints.html

5.http://www.open-std.org/jtc1/sc22/wg14/www/docs/n926.htm

6.http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1944.htm

7.http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html

8.http://c-faq.com/expr/evalorder4.html