链滴

C/C++ 中的序列点

作者：localvar

原文链接：https://ld246.com/article/1521536325345

来源网站：链滴

原文链接

0. 什么是副作用（side effects）

C99定义如下：

Accessing a volatile object, modifying an object, modifying a file, or calling a function that do s any of those operations are all side effects, which are changes in the state of the execution nvironment.

C++2003定义如下：

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

可以看出C99和C++2003对副作用的定义基本类似，一个程序可以看作一个状态机，在任意一个时刻序的状态包含了它的所有对象内容以及它的所有文件内容（标准输入输出也是文件），副作用会导致态的跳转。

一个变量一旦被声明为volatile-qualified类型，则表示该变量的值可能会被程序之外的事件改变，每读取出来的值只在读取那一刻有效，之后如果再用到该变量的值必须重新读取，不能沿用上一次的值因此读取volatile-qualified类型的变量也被认为是有副作用，而不仅仅是改写。

注，一般不认为程序的状态包含了CPU寄存器的内容，除非该寄存器代表了一个变量，例如:

```
void foo() {
  register int i = 0;  // 变量i被直接放入寄存器中，本文中被称为寄存器变量
                       // 注，register只是一个建议，不一定确实放入寄存器中
                       // 而且没有register关键字的auto变量也可能放入寄存器
                       // 这里只是用来示例，假设i确实放入了寄存器中
  i = 1;               // 寄存器内容改变，对应了程序状态的改变，该语句有副作用
  i + 1;               // 编译时该语句一般有警告：“warning: expression has no effect”
                       // CPU如果执行这个语句，也肯定会改变某个寄存器的值，但是程序状态
                       // 并未改变，除了代表i的寄存器，程序状态不包含其他寄存器的内容，
                       // 因此该语句没有任何副作用
}
```

特别的，C99和C++2003都指出，no effect的expression允许不被执行：

An actual implementation need not evaluate part of an expression if it can deduce that its val e is not used and that no needed side effects are produced (including any caused by calling a unction or accessing a volatile object).

1. 什么是序列点（sequence points）

C99和C++2003对序列点的定义相同：

At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall ha e taken place.

中文表述为，序列点是一些被特别规定的位置，要求在该位置前的evaluations所包含的一切副作用

此处均已完成，而在该位置之后的evaluations所包含的任何副作用都还没有开始。

例如C/C++都规定完整表达式（full-expression）后有一个序列点：

```
extern int i, j;
i = 0;
j = i;
```

上面的代码中i = 0以及j = i都是一个完整表达式，;说明了表达式的结束，因此在;处有一个序列点，照序列点的定义，要求在i = 0之后j = i之前的那个序列点上对i = 0的求值以及副作用全部结束（0被入i中），而j = i的任何副作用都还没有开始。由于j = i的副作用是把i的值赋给j，而i = 0的副作用是把赋值为0，如果i = 0的副作用发生在j = i之后，就会导致赋值后j的值是i的旧值，这显然是不对的。

由序列点以及副作用的定义很容易看出，在一个序列点上，所有可能影响程序状态的动作均已完成，这样能否推断出在一个序列点上一个程序的状态应该是确定的呢？！答案是不一定，这取决于我们代的写法。但是，如果在一个序列点上程序的状态不能被确定，那么标准规定这样的程序是undefined ehavior，稍后会解释这个问题。

2. 表达式求值（evaluation of expressions）与副作用发生的相顺序

C99和C++2003都规定：

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

也就是说，C/C++都指出一般情况下在表达式求值过程中的操作数求值顺序以及副作用发生顺序是未明的（unspecified）。为什么C/C++不详细定义这些顺序呢？原因是因为C/C++都是极端追求效率语言，不规定这些顺序，是为了允许编译器有更大的优化余地，例如：

```
extern int *p;
extern int i;
*p = i++;    // 表达式(1)
```

根据前述规定，在表达式(1)中到底是*p先被求值还是i++先被求值是由编译器决定的；两次副作用（对p赋值以及i++）发生的顺序是由编译器决定的；甚至连子表达式i++的求值（就是初始时i的值）以副作用（将i增加1）都不需要同步发生，编译器可以先用初始时i的值（即子表达式i++的值）对*p赋，然后再将i增加1，这样就把子表达式i++的整个计算过程分成了两个不相邻的步骤。而且通常编译都是这么实现的，原因在于i++的求值过程同*p = i++是有区别的，对于单独的表达式i++，执行顺一般是（假设不考虑inc指令）：先将i加载到某个寄存器A（如果i是寄存器变量则此步骤可以跳过）将寄存器A的值加1、将寄存器A的新值写回i的地址；对于*p = i++，如果要先完整的计算子表达式i++由于i++表达式的值是i的旧值，因此还需要一个额外的寄存器B以及一条额外的指令来辅助*p = i++执行，但是如果我们先将加载到A的值写回到*p，然后再执行对i增加1的指令，则只需要一个寄存器可，这种做法在很多平台都有重要意义，因为寄存器的数目往往是有限的，特别是假如有人写出如下语句：

```
extern int i, j, k, x;
x = (i++) + (j++) + (k++);
```

编译器可以先计算(i++) + (j++) + (k++)的值，然后再对i、j、k各自加1，最后将i、j、k、x写回内，这比每次完整的执行完++语义效率要高。

3. 序列点对副作用的限制

C99和C++2003都有类似的如下规定：

Between the previous and next sequence point a scalar object shall have its stored value modi ied at most once by the evaluation of an expression. Furthermore, the prior value shall be acc ssed only to determine the value to be stored. The requirements of this paragraph shall be m t for each allowable ordering of the subexpressions of a full expression; otherwise the behavio is undefined.

也就是说，在相邻的两个序列点之间，一个对象只允许被修改一次，而且如果一个对象被修改则在这个序列点之间对该变量的读取的唯一目的只能是为了确定该对象的新值（例如i++，需要先读取i的值确定i的新值是旧值+1）。特别的，标准要求任意可能的执行顺序都必须满足该条件，否则代码将是undefined behavior。

之所以序列点会对副作用有如此的限制，就是因为C/C++标准没有规定子表达式求值以及副作用发生间的顺序，例如：

```
extern int i, a[];
extern int foo(int, int);
i = ++i + 1;    // 该表达式对i所做的两次修改都需要写回对象，i的最终值取决
                // 于到底哪次写回最后发生，如果赋值动作最后写回，则i的值
                // 是i的旧值加，如果++i动作最后写回，则i的值是旧值加，
                // 因此该表达式的行为是undefined
a[i++] = i;     // 如果=左边的表达式先求值并且i++的副作用被完成，则右边的
                // 值是i的旧值加，如果i++的副作用最后完成，则右边的值是i
                // 的旧值，这也导致了不确定的结果，因此该表达式的行为将是
                // undefined
foo(foo(0, i++), i++); // 对于函数调用而言，标准没有规定函数参数的求值
                // 顺序，但是标准规定所有参数求值完毕进入函数体
                // 执行之前有一个序列点，因此这个表达式有两种执
                // 行方式，一种是先求值外层foo调用的i++然后求值
                // foo(0, i++)，然后进入到foo(0, i++)执行，这之
                // 前有个序列点，这种执行方式还是在两个相邻序列
                // 点之间修改了i两次，undefined
                // 另一种执行方式是先求值foo(0, i++)，由于这里
                // 有一个序列点，随后的第二个i++求值是在新序列
                // 点之后，因此不算是两个相邻的序列点之间修改i两次
                // 但是，前面已经指出标准规定任意可能的执行路径
                // 都必须满足条件才是定义好的行为，这种代码仍然
                // 是undefined
```

前面我提到在一个序列点上程序的状态不一定是确定的，原因就在于相邻的两个序列点之间可能会发多个副作用，这些副作用的发生顺序是未指定的，如果多于一个的副作用用于修改同一个对象，例如例代码i = ++i + 1;，则程序的结果是依赖于副作用发生顺序的；另外，如果某个表达式既修改了某对象又需要读取该对象的值，且读取对象的值并不用于确定对象新值，则读取和修改两个动作的先后序也会导致程序的状态不能唯一确定。

所幸的是，“在相邻的两个序列点之间，一个对象只允许被修改一次，而且如果一个对象被修改则在两个序列点之间只能为了确定该对象的新值而读一次”这一强制规定保证了符合要求的程序在任何一序列点位置上其状态都可以确定下来。

注，由于对于UDT类型存在operator重载，函数语义会提供新的序列点，因此某些对于built-in类型是ndefined behavior的表达式对于UDT确可能是良好定义的，例如：

```
i = i++; // 如果i是built-in类型对象，则该表达式在两个相邻的序列点之间对
         // i修改了两次，undefined
```

```
// 如果i是UDT类型该表达式也许是i.operator=(i.operator++(int))，
// 函数参数求值完毕后会有一个序列点，因此该表达式并没有在两个
// 相邻的序列点之间修改i两次，OK
```

由此可见，常见的问题如printf("%d, %d", i++, i++)这种写法是错误的，这类问题作为笔试题或者面题是没有任何意义的。

类似的问题同样发生在cout << i++ << i++这种写法上，如果overload resolution选择成员函数operator<<，则等价于(cout.operator<<(i++)).operator<<(i++)，否则等价于operator<<(operator<<(cout, i++), i++)，如果i是built-in类型对象，这种写法跟foo(foo(0, i++), i++)的问题一致，都是定义行为，因为存在某条执行路径使得i会在两个相邻的序列点之间被修改两次；如果i是UDT则该写是良好定义的，跟i = i++一样，但是这种写法也是不推荐的，因为标准对于函数参数的求值顺序是unpecified，因此哪个i++先计算是不能预计的，这仍旧会带来移植性的问题，这种写法应该避免。

4. 编译器的跨序列点优化

根据前述讨论可知，在同一个表达式内对于同一个变量i，允许的行为是：

A. 不读取，改写一次，例如：

```
i = 0;
```

B. 读取一次或者多次，改写一次，但所有读取仅仅用于决定改写后的新值，例如：

```
i = i + 1; // 读取一次，改写一次
i = i & (i - 1);
```

C. 不改写，读取一次或者多次，例如：

```
j = i & (i - 1);
```

对于情况B和C，编译器是有一定的优化权利的，它可以只读取一次变量的值然后直接使用该值多次。

但是，当该变量是volatile-qualified类型时编译器允许的行为究竟如何目前还没有找到明确的答案，ctlz认为如果在两个相邻序列点之间读取同一个volatile-qualified类型对象多次仍旧是undefined behavor，原因在于该读取动作有副作用且该副作用等价于修改该对象，RoachCock的意见是两个相邻的序点之间读取同一个volatile-qualified类型应该是合法的，但是不能被优化成只读一次。一段在嵌入式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，故RoachCock认为编译器不能够对volatile-qualifie类型的变量做出只读一次的优化。ctrlz则认为这段代码本身是不正确的，应该改写成：

```
int j = i;
if (j != i) { // 将对volatile-qualified类型变量的多次读取用序列点隔开
  // ...
}
```

虽然尚不能确定volatile-qualified类型的变量在相邻两个序列点之间读取多次行为是否合法以及将如优化（不管怎么样，对于volatile-qualified类型这种代码应该尽量避免），但是可以肯定的是，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

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类型的变量不会变化的，因为目前的C/C++抽象机器模型是单线程的），例如：

```
bool flag = true;
void foo() {
    while (flag) { // (2)
        // ...
    }
}
```

如果编译器探测到foo()没有任何语句（包括foo()调用过的函数）对flag有过修改，则也许会把(2)优成只在进入foo()的时候读一次flag的值而不是每次循环都读一次，这种跨序列点的优化很有可能导致循环。但是这种代码在多线程编程中很常见，虽然foo()没有修改过flag，也许在另一个线程的某个函调用中会修改flag以终止循环，为了避免这种跨序列点优化带来到错误，应该把flag声明为volatile bo l，C++2003对volatile的说明如下：

> Note: volatile is a hint to the implementation to avoid aggressive optimization involving the bject because the value of the object might be changed by means undetectable by an imple entation. See 1.9 for detailed semantics. In general, the semantics of volatile are intended to e the same in C++ as they are in C.

5. C99定义的序列点列表

- The call to a function, after the arguments have been evaluated.
- The end of the first operand of the following operators:
  - logical AND && ;
  - logical OR || ;
  - conditional ?: ;
  - comma , .
- The end of a full declarator:
  - declarators;
- The end of a full expression:
  - an initializer;
  - the expression in an expression statement;
  - the controlling expression of a selection statement (if or switch);
  - the controlling expression of a while or do statement;
  - each of the expressions of a for statement;
  - the expression in a return statement.
- Immediately before a library function returns.
- After the actions associated with each formatted input/output function conversion specifier.
- Immediately before and immediately after each call to a comparison function, and also bet een any call to a comparison function and any movement of the objects passed as arguments to that call.

6. C++2003定义的序列点列表

所有C99定义的序列点同样是C++2003所定义的序列点。此外，C99只是规定库函数返回之后有一个列点，并没有规定普通函数返回之后有一个序列点，而C++2003则特别指出，进入函数（function-entry）和退出函数（function-exit）各有一个序列点，即拷贝一个函数的返回值之后同样存在一个序点。

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，因此一不建议重载这几个运算符。

7. C++2003中两处关于lvalue的修改对序列点的影响

在C语言中，assignment operators的结果是non-lvalue，C++2003则将assignment operators的果改成了lvalue，目前尚不清楚这一改动对于built-in类型有何意义，但是它却导致了很多在合法的C码在目前的C++中是undefined behavior，例如：

```
extern int i;
extern int j;
i = j = 1;
```

由于j = 1的结果是lvalue，该结果作为给i赋值的右操作数，需要一个lvalue-to-rvalue conversion，个conversion代表了一个读取语义，因此i = j = 1就是先将1赋值给j，然后读取j的值赋值给i，这个为是undefined，因为标准规定两个相邻序列点之间的读取只能用于决定修改对象的新值，而不能发在修改之后再读取。

由于C++2003规定assignment operators的结果是lvalue，因此下列在C99中非法的代码在C++200中却是可以通过编译的：

```
extern int i;
(i += 1) += 2;
```

显然按照C++2003的规定这个代码的行为是undefined，它在两个相邻的序列点之间修改了i两次。

类似的问题同样发生在built-in类型的前缀++/--operators上，C++2003将前缀++/--的结果从rvalu修改为lvalue，这甚至导致了下列代码也是undefined behavior：

```
extern int i;
extern int j;
i = ++j;
```

同样是因为lvalue作为assignment operator的右操作数需要一个左值转换，该转换导致了一个读取作且这个读取动作发生在修改对象之后。

C++的这一改动显然是考虑不周的，导致了很多C语言的习惯写法都成了undefined behavior，因此ndrew Koenig在1999年的时候就向C++标准委员会提交了一个建议要求为assignment operators加新的序列点，但是到目前为止C++标准委员会都还没有就该问题达成一致意见。