УДК 004

ТОЧКИ СЛЕДОВАНИЯ В С

Волков Виталий Александрович
студент

Митьков Станислав Александрович
студент

Мордовский государственный университет им. Н.П. Огарева, Саранск

author@apriori-journal.ru

Аннотация. В данной статье рассматриваются точки следования в языке С. Их список, особенности поведения и эффективность в программах.

Ключевые слова: точка следования; С; компилятор; операторы; выражения; конструктор.

SEQUENCE POINTS IN C

Volkov Vitaly Alexandrovich
student

Mit'kov Stanislav Aleksandrovich
student

Ogarev Mordovian State University, Saransk

Abstract. This article discusses the sequence points in C, behaviors and efficiency programs.

Key words: sequence point; C; the compiler; operators; expressions; constructor.

Программа на С состоит из 2-х типов утверждений:

- выражения;
- определения и объявления типов, имен функций, констант и т.д.

На С выражения имеют побочные эффекты при выполнении и поэтому определяют поведение самой программы. Точки следования в свою очередь тесно связаны с вычислениями выражений.

В природе существует ряд побочных эффектов при выполнении С программы:

1. Модификация объектов, т.е., внесение изменений в некоторую ячейку памяти или регистр.
2. Обращение к объектам, объявленным как volatile.
3. Вызов функций, которые производят побочные эффекты, такие как файловые вводы или выводы, к примеру.
4. Вызов функции, выполняющей любое действие из них.

Итак, основным действием С программ является модификация ячеек и вызов системных функций [4].

Последовательность вычисления выражений

Компилятор способен генерировать исполняемые инструкции, вычисляющие все выражения. Все выражения вычисляются в том порядке, указанном в исходном коде. Мы утверждаем «почти», потому что компиляторы имеют некоторую свободу при изменении порядка вычисления выражений, если они видят оптимизацию. Тем не менее, масштабы таких изменений ограничены точками следования. Каждая из точек следования это точка в последовательности выражений, в которой компиляторы закончили вычисления предыдущих выражений и не начали вычисление последующих выражений.

Точки следования это места, где программисты знают, какое выражение (или подвыражение) уже вычислено, а какое выражение (или подвыражение) до сих пор не рассчитано. Другими словами, точка следования это та точка в программе, в которой мы знаем, на каком этапе при выполнении программы присутствуем. Между всеми точками следования неизвестно ничего о порядке вычисления выражения и подвыражения. К удивлению в C программах присутствует очень незначительное количество точек следования, что говорит, что мы не знаем, какое выражение вычислено, а какое нет [1].

Простор при оптимизации

Причина неопределенности в порядке вычислений между всеми точками следования заключается в эффективности. Компилятору дана свобода при вычислении выражения в том порядке, который наиболее эффективен для платформы и процессора. Это значит, что проблема последовательности, о которой говорилось выше, является, как правило, проблемой переносимости. Проблема неопределенности в порядке вычислений может не беспокоить, пока вы применяете один компилятор для одной платформы. Но вы можете столкнуться с ней после установки другой версии компилятора и перекомпиляции своего исходного кода. Или же при переносе программы на другие платформы.

Список точек следования:

- в конце полных выражений;
- перед выполнением выражения в теле функции и после вычисления каждого аргумента в ее вызове;
- до выполнения любых выражений вне функций и после копирования возвращаемых значений;
- после вычисления первых выражений в a?b:c, a&&b, a||b или a,b;
- после инициализации базового класса и членов в списке инициализации конструкторов [3].

Давайте рассмотрим каждую точку следования:

Окончание полного выражения – это точка с запятой. В условном выражении if(q==0 && w==f(e,r,t)) окончание полного выражения это закрывающая скобка.

Точки следования как до, так и после вызова функций означают, что выполнение вызываемых функций и вызывающий контекст согласованы: выполнен ряд операторов до вызова функции, вычислен ряд аргументов функции, а потом выполняется тело самой функции. Точно так же при возврате из функции: каждый оператор тела функции выполнен, вычислено каждое возвращаемое выражение, а потом программное управление передается в контекст вызова, начиная с оператора после вызова функции.

Операторы. Точки следования в таких логических выражениях, как && и ||, тернарном операторе ? и операторе 'запятой' означают, что левосторонний операнд вычисляется раньше правостороннего. Только эти операнды в C++ устанавливают точки следования.

Заметим, что оператор 'запятую' часто путают с запятой в качестве разделителя, как, например, в списке аргументов функции. В основном в программе запятая используется в качестве разделителя. Оператор 'запятая' используется редко, в основном в циклах for как в примере for(i=0, j=0; i<100||j<200;++i,++j). В списке аргументов функции f(++i,++j) запятая является просто разделителем между аргументами функций и не предполагает какого-либо порядка вычисления аргументов функции.

Просто для демонстрации того, как все может быть запутанным, рассмотрим следующий пример. Пусть f - функция с одним аргументом и мы передаем в нее (++i,++j), как здесь f((++i,++j)). В этом случае запятая будет не разделителем между аргументами функции, а оператором ‘запятой‘. Тогда ++i будет вычисляться раньше ++j. Результат вычисления будет передаваться в качестве аргумента в функцию f. И тут возникает вопрос: что является результатом оператора ‘запятой‘? Результат вычисления правого операнда!

Не менее запутанным является такое выражение array[++i, ++j]. Помните, что на элементы в двухмерном массиве ссылаются как array[i, j].

Поэтому запятая в array[++i,++j] является оператором, а не разделителем.

Список инициализации для конструктора. Точка следования существует между инициализацией всех базовых классов и членов в списке конструктора, так как порядок всех инициализаций четко определен. Обратите внимание, что данный порядок определяет не порядок, в котором базовый класс и члены появляются в списках инициализации, а порядок, в котором перечислены при определении класса. К примеру:

```
class Array
{
  private:
    int* data;
    unsigned size;
    int lBound, hBound;
   public:
    Array(int low, int high) : size(high-low+1),
      lBound(low), hBound(high), data(new int[size]) {}
};
```

В этом примере порядок инициализации следуюший: data(new int(size)), потом size(high-low+1), потом lBound(low), потом hBound(high), что в этом случае приведет к проблеме, так как size используется при инициализации разумными значениями. Вот вам и еще одна ловушка в C.

```
x = f() + g() + h();
```

На первый взгляд все прозрачно. 3 функции будут вызваться в неопределенном порядке, будут подсчитаны суммы их значений, затем присваивание. Но, что если функции обращаются к общей глобальной или же статической переменной для чтения и модификаций? И вновь проблемы точек следования [2].

Интересный случай:

```
f( new C(i++), new V(i) );
```

Тут можно наблюдать много происходящих вещей: произведение побочных эффектов модификации переменных i, выделение памяти для C и V и конструирование объектов. Мы не имеем понятия, как данные побочные эффекты протекают. Но знаем, что i++ вычисляется до вызова C, и смело предполагаем, что исполняющие системы будут выделять память до ее попытки инициализироваться через вызов конструктора. Мы не имеет понятия, будет ли i инкрементироваться прежде, чем передана самому конструктору V.

Именно данное отсутствие определенности проблематично при возникновении исключений, которые генерируются операторами new или конструкторами. При исключении мы не узнаем, что произошло и как правильно обработать все это.

Проблемы в вышеприведенном примере были следующими: множественный доступ к переменной, также и при модификации. Мы просто пытались показать много нюансов в одном выражении. Правило гласит, нужно избегать сложных выражений. Каждая проблема, обсуждаемая в статье, может быть предотвращена, если разложить сложное выражение на ряд простых. К примеру, вместо:

```
z[q]=q++ + 1;
```

можно было написать

```
z[q]=q + 1;
q++;
```

Деля выражение на простые, мы тем самым вводим новые точки следования и как результат имеем строго определенный порядок вычислений. Это же касается и последнего примера:

```
1f( new X(i++), new Y(i) );
```

можно написать

```
X* xptr = new X(i++);
Y* yptr = new Y(i);
f( xptr, yptr );
```

Если нам удастся поймать исключение Y exception, то мы сможем твердить, что X создан и есть больше шансов обработать данное исключение правильно. Или же нам можно было заключить все эти операторы в самостоятельные try-блоки, если нужно было обработать исключения bad_alloc от всех вызовов оператора new.

Список использованных источников

1. Александров Э.Э., Афонин В.В. Введение в программирование на языке С: учеб. пособие. Саранск: Изд-во Мордов. ун-та, 2009. 316 с.
2. Александров Э.Э., Афонин В.В. Программирование на языке С в Microsoft Visual Studio 2010: учеб. пособие. Саранск: Изд-во Мордов. ун-та, 2010. 424 с.
3. Афонин В.В., Федосин С.А. О структурировании лабораторно-практических занятий при изучении дисциплин программирования // Образовательные технологии и общество. 2014. Т. 17. № 4. С. 497-506.
4. Александров Э.Э., Афонин В.В. Программирование на языке С в Microsoft Visual Studio 2010.[Электронный ресурс]. Режим доступа: http://www.intuit.ru/department/pl/prcmsvs2010 (дата обращения: 15.10.2015).