сишные трюки
(0xF выпуск)

крис касперски ака мыщъх, a.k.a. souriz, a.k.a. nezumi, a.k.a. elraton, no-email

ну вот, наконец, мы добрались и до вопросов стиля программирования и вытащили из копилки три любопытных трюка, способствующих ясности листинга и страхующих программиста от совершения неявных ошибок (особенно при работе в коллективной среде). впрочем, удобство это вопрос привычки и предмет непрекращающихся священных войн, граничащих с корпоративными стандартами, диктующими свои правила оформления листинга, так что прежде чем применять мыщъхиные трюки на практике проконсультируйтесь с адвокатом. шутка.

совет 1 — рекурсия вместо циклов

Как известно, большую часть процессорного времени программа проводит в циклах и потому их оптимизация может дать весьма позитивный результат. Рассмотрим простейший цикл вида:

```
for(i = 0; i < n; i++) result *= n;
```

Листинг 1 пример типичного цикла for

А теперь попробуем извратиться, заменив цикл на рекурсивный вызов функции, в результате чего у нас получится следующий код:

```
foo(int n, int result)
{
        if(n) return foo(n - 1, result * n);
        return result;
}
```

Листинг 2 рекурсия, заменяющая цикл

На первый взгляд кажется, что рекурсивный вариант будет ужасно тормозить, поскольку накладные расходы на передачу аргументов и вызов функции чрезвычайно велики, к тому же возникает прямая угроза исчерпания стека при большом количестве итераций. На самом деле компиляторы уже давно научились избавляться от хвостовой рекурсии, трансформируя ее в цикл, что подтверждаются дизассемблерным листингом, приведенном ниже (***примечание****: хвостовой рекурсией — tail recursion — называется такой тип рекурсии, при котором вызов рекурсивной функции следует непосредственно за оператором return*):

```
.text:0000006C loc_6C:                                 ; CODE XREF: _foo+12↓j
.text:0000006C                 mov     edx, ecx
.text:0000006E                 imul    eax, edx
.text:00000071                 dec     ecx
.text:00000072                 jnz     short loc_6C
```

Листинг 3 фрагмент дизассемблерного листинга, демонстрирующий как компилятор Microsoft Visual C++ 6.0 сумел избавиться от рекурсии

Сгенерированный компилятором код "оптимизированного" цикла полностью идентичен своему не оптимизированному собрату. И в чем же тогда состоит обещанный выигрыш?! В данном случае — ни в чем, однако, некоторые алгоритмы в рекурсивной форме выглядят более естественно и наглядно. Следовательно, их отладка упрощается, а вероятность совершения ошибок — уменьшается. Многие руководства по оптимизации настоятельно рекомендуют избавляться от ресурсоемкой рекурсии еще на стадии кодирования, однако, в случае хвостовой рекурсии компилятор все сделает за нас сам!!!

Однако, следует помнить, что данный трюк не распространяется на остальные виды рекурсии и с оптимизацией рекурсивного вычисления чисел Фиббоначи компилятор уже не справляется:

```
fib(int n)
```

```
{
        if (n < 2) return 1;
        return fib(n - 1) + fib(n - 2);
}
```

Листинг 4 рекурсивное вычислений чисел Фиббоначи

В дизассемблерном листинге мы отчетливо видим два вызова функции fib (см. листинг 5), что приводит к огромному падению производительности, совершенно не компенсируемому улучшением читабельности и наглядности алгоритма.

```
.text:00000030 fib             proc near               ; CODE XREF: _fib+16↓p
.text:00000030                                         ; _fib+1F↓p ...
…
.text:00000041
.text:00000041 loc_41:                                 ; CODE XREF: _fib+8↑j
.text:00000041                 lea     eax, [esi-2]
.text:00000044                 push    edi
.text:00000045                 push    eax
.text:00000046                 call    _fib
.text:0000004B                 dec     esi
.text:0000004C                 mov     edi, eax
.text:0000004E                 push    esi
.text:0000004F                 call    _fib
.text:00000054                 add     esp, 8
.text:00000057                 add     eax, edi
.text:00000059                 pop     edi
.text:0000005A                 pop     esi
.text:0000005B                 retn
.text:0000005B fib             endp
```

Листинг 5 фрагмент дизассемблерного листинга с не-хвостовой рекурсией

совет 2 — сокрытие ветвления в логических операторах

Язык Си (как и остальные языки высокого уровня) _всегда_ оптимизирует выполнение логических операторов (даже если все опции оптимизации выключены). Если выражение foo не равно нулю, то вычисления выражения bar в конструкции (foo || bar) никогда _не_ выполняется. Соответственно, если foo равно нулю, то в конструкции (foo && bar) вычислять bar нет никакой необходимости, более того, если бы такое вычисление выполнялось, то привычная конструкция (m && n/m) привела бы к развалу программы при m равном нулю.

Отсюда, ветвление вида "if (foo==0) bar;" можно заменить на аналогичное ему выражение "(foo || bar)":

```
if (foo==0) my_func(x,y,z);
```

Листинг 6 классический вариант с ветвлением

```
foo || my_func(x,y,z);
```

Листинг 7 оптимизированный вариант

И хотя ветвления на самом деле никуда не делись (компилятор послушно вставит их в код в том же самом количестве, что и раньше), данный трюк можно использовать например, чтобы прищемить членов жури на олимпиадных и конкурсных задачах в стиле "отсортировать m чисел, используя не более k сравнений".

Другое (более существенное) преимущество — выражение, в отличие от явных ветвлений, может использоваться где угодно, существенно упрощая программирование и повышая наглядность листинга:

```
for (;;)
{
        some_func(i,j,k);
        if (foo==0) my_func(x,y,z);
}
```

Листинг 8 не оптимизированный вариант

Если заменить ветвления на "логику", весь цикл укладывается в одну строку без всяких фигурных скобок:

```
for (;;) some_func(i,j,k), (foo || my_func(x,y,z));
```

Листинг 9 оптимизированный по наглядности вариант

И хотя некоторые могут заявить, что не оптимизированный вариант более нагляден, это не так. Наглядность на 90% вопрос привычки, однако, скорость чтения (и восприятия) листинга обратно пропорциональна его размеру и потому компактный стиль программирования намного более предпочтителен.

совет 3 — опасайтесь операторов "--" и "++"

За постфиксными (и в чуть меньшей степени за постфиксными) операторами "--"/"++" закрепилась дурная слава "небезопасных" и приводящих к неопределенному поведению (по-английски undefined behavior или, сокращенно, ub), ссылками на которое пестрит текст Стандарта и многочисленных руководств по Си/Си++.

Практически все знают, что результат вычисления функции foo(x++, x++) зависит не только от значения переменной x, но и особенностей используемого транслятора, ведь порядок вычисления аргументов отдан на откуп реализаторам и компиляторы могут вычислять их в произвольном порядке. Отсюда и ub.

Считается, что если префиксный/постфиксный оператор встречается до точки следования всего один раз, то такая конструкция безопасна, однако, это идеализированное утверждение, не учитывающее суровых реалий программисткой жизни.

Вопрос на засыпку. Является ли следующая конструкция (не)безопасной и почему: foo(++i, ++j). На первый взгляд здесь все законно и никакого ub не возникает. В случае, если foo является функцией, это действительно так, ну а если это макрос следующего вида:

```
#define max(i,j) ((i) < (j) ? (i) : (j))
```

Листинг 10 классический макрос, таящий в себе _огромную_ опасность

На выходе препроцессора мы получим следующий код:

```
((++i) < (++j) ? (++j) : (++i))
```

Листинг 11 результат обработки макроса max препроцессором

Теперь уже и слепой увидит, что переменные инкрементируются _дважды_ и мы получаем довольно неожиданный результат. Но ведь не будешь же каждый листинг прогонять через препроцессор! Более того, определения, бывшие ранее макросами, в следующей версии сторонней библиотеки могут превратиться в функции, равно как и наоборот!

Поэтому, вместо конструкции "foo(++i, ++j)" настоятельно рекомендуется использовать "(foo((i+1), (j+1)), i++, j++)", которая хоть и проигрывает в компактности/производительности, зато полностью безопасна. Кстати, обратите внимание на два обстоятельства.

Первое — переменные разделяются не точной с запятой, а просто запятой, что делает эту запись единым выражением. Если бы мы использовали точку с запятой, то при замене "for(;;) foo(++i, ++j)" на "for(;;) foo((i+1); (j+1)); i++; j++;" пришлось бы использовать фигурные скобки, о которых легко забыть, особенно если for находится на одной строке, а foo – на другой. К тому же в выражениях if/else "внеплановые" фигурные скобки порождают неприятные побочны эффекты — а зачем они нам? Правда… операторы, разделенные запятой значительно хуже поддаются оптимизации, но это уже издержки которые приходится платить за безопасность и универсальность.

Второе — круглые скобки вокруг (i+1) и (j+1) формально не обязательны, но! если foo – макрос, разработчик которого забыл заключить аргументы в скобки, то при его обработке препроцессором мы можем огрести по полной программе, получив совсем не тот результат, который ожидали. Опять-таки, с формальной точки зрения ответственность лежит на разработчике макроса, но в реальной жизни мы вынуждены предугадывать возможные ошибки своих коллег, предпринимая превентивные меры по их устранению, особенно при аудите кода. Всякая замена потенциально опасных конструкций на безопасные должна быть полностью эквивалентной в смысле побочных эффектов, в противном случае последствия такого аудита могут оказаться фатальными.