сишные трюки
(7й выпуск)

трюк 1: массив, начинающийся в единицы

В Си массивы начинаются с нуля, а во многих других языках — с единицы, что сильно напрягает при переходе с одного языка на другой и еще больше — при переносе программ. Приходится вносить множество правок в самых разных местах и потом долго отлаживать программу. Да и сами сишники далеко не в восторге от того, что индекс последнего элемента массива размером N равен N-1. Это служит не только источником досадных ошибок, но в некоторых случаях снижает производительность.

Существует хитрый хак, позволяющий создавать массивы, начинающиеся с единицы, или, строго говоря, вообще с _любого_ числа. Это невероятно полезно в тех случаях, когда нам нужно создать массив, проиндексированный k, k+1,,,k+n (например, возрастом человека от 16 лет до 100).

А делается это приблизительно так:

```
// задаем размер массива, его тип и индекс первого элемента
#define SIZE_OF_ARRAY                0x1000
#define TYPE_OF_ARRAY                int
#define INDEX_OF_FIRST_ELEMENT       1

// объявляем "сырую" переменную-указатель
TYPE_OF_ARRAY *p;

// выделяем блок памяти требуемого размера
p = (TYPE_OF_ARRAY*)malloc(SIZE_OF_ARRAY*sizeof(TYPE_OF_ARRAY));

// проверка на успешность выделения памяти
if (!p) /* обработка ситуации ошибки выделения */

// сдвигаем указатель на нужное кол-во позиций,
// так, чтобы индекс первого элемента стал равен INDEX_OF_FIRST_ELEMENT
p -= INDEX_OF_FIRST_ELEMENT;
…

// работаем с массивом

…
// возвращаем указатель на место
p += INDEX_OF_FIRST_ELEMENT;

// освобождаем блок памяти
free(p);
```

Листинг 1 создание массива, начинающегося с произвольного элемента на хипе

Аналогичным путем можно обрабатывать и локальные массивы (в этом случае наша задача даже упрощается, поскольку не нужно заботиться об освобождении памяти), которая при выходе из функции освободится сама:

```
// задаем размер массива, его тип и индекс первого элемента
#define SIZE_OF_ARRAY                0x1000
#define TYPE_OF_ARRAY                int
#define INDEX_OF_FIRST_ELEMENT       1

// выделяем блок памяти требуемого размера
TYPE_OF_ARRAY raw[SIZE_OF_ARRAY];
TYPE_OF_ARRAY *p = raw;

// сдвигаем указатель на нужное кол-во позиций,
// так, чтобы индекс первого элемента стал равен INDEX_OF_FIRST_ELEMENT
p -= INDEX_OF_FIRST_ELEMENT;
```

Листинг 2 создание массива, начинающегося с произвольного элемента на стеке

трюк 2: марафон битовых и логических операций

В некоторых руководствах и конференциях можно прочесть, что выражение вида (a || b || c) _практически_ всегда быстрее, чем (a | b | c). Так ли это? И если так, то почему? Попробуем разобраться. Поскольку все современные компиляторы поддерживают "быстрые булевы операции", они вычисляют значение сложного выражения лишь до первого "срабатывания" ~~ложи~~. То есть, если a =! 0, то оставшуюся часть выражения можно не проверять, поскольку уже и так все ясно.

Напротив, битовое выражение (a | b | c) требует вычисления всех переменных, что на первый взгляд выглядит более похабным и менее производительным. Но это только на первый взгляд. Все зависит от того насколько часто "быстрая булева оптимизация" прерывает вычисление выражения до его завершения. Если все переменные _преимущество_ равны нулю, то ни о каком выигрыше говорить не приходится, а вот код получается намного более громоздким, что легко подтверждается дизассемблером:

```
.text:00000020          mov     eax, [esp+arg_0]
.text:00000024          test    eax, eax
.text:00000026          jnz     short locret_34
.text:00000028          mov     ecx, [esp+arg_4]
.text:0000002C          test    ecx, ecx
.text:0000002E          jnz     short locret_34
.text:00000030          mov     ecx, [esp+arg_8]
```

Листинг 3 результат дизассемблирования bar(int a, int b, int c){if (a || b || c) return a;}

Что мы видим?! Условные переходы. То есть ветвления. А процессоры с конвейерной архитектурой (типа Pentium Pro+) условных переходов не любят, особенно если они выполняются в цикле. Как следствие — вместо обещанного выигрыша мы получаем _падение_ производительности.

```
.text:00000000          mov     eax, [esp+arg_0]
.text:00000004          mov     edx, [esp+arg_4]
.text:00000008          mov     ecx, eax
.text:0000000A          or      ecx, edx
.text:0000000C          mov     edx, [esp+arg_8]
.text:00000010          or      ecx, edx
.text:00000012          retn
```

Листинг 4 результат дизассемблирования foo(int a,int b,int c){if (a | b | c) return a;}

А вот в выражении (a | b | c) никаких условных переходов нет и оно выполняется предельно быстро!

Поэтому, используете (a || … || x) только при большом количеств элементов (намного больше трех) и только если одна или несколько переменных гораздо чаще равны TRUE, чем FALSE (при этом они должны стоять первыми слева). Тоже самое относится и к &&, лишь с той оговоркой, что переменные, преимущественно равные FLASE следует выдвигать вперед.

трюк 3: оценка качества генератора случайных чисел

Генератор случайных чисел — используется не только в криптоалгоритмах, но и в более "приземленных" программах. Например, в играх. И очень часто нам важно знать, насколько "случаен" выдаваемый им результат (допустим, мы моделируем казино в поисках беспроигрышной стратегии).

Причем, кроме "общей" вероятности, представляет интерес выяснить степень распределения вероятности по каждому из битов. Некоторые генераторы страдают хронической предсказуемостью определенных бит (как правило, младших или старших).

Следующая программа как раз и позволяет оценить насколько предсказуем тот или иной бит:

```
#define N 10000
unsigned int buf[sizeof(int)*8];

main()
{
        int x=0;
        unsigned int a, l;
```

```
        srand((unsigned)time(NULL));
        for (l = 0; l < sizeof(int)*8; l++)
        {
                for (a = 0,x = 0; a < N; a++)
                {
                        buf[l] += ((rand()>>l) & 1);
                }
        }

        for (a = 0; a < sizeof(int)*8; a++)
                printf("%02d:%02d.%02d\t",a,10000*buf[a]/N/100,10000*buf[a]/N%100);
        printf("\n");
}
```

Листинг 5 тестовая программа

Результат прогона на MS VC 6 показывает, что биты от 0 до 14 генерируются довольно-таки качественно (с вероятностью 50/50 и погрешностью порядка 1%), а вот начиная с 15 бита мы имеем сплошной облом!!! То есть, по сути rand() возвращает 14 битный результат, не дотягивающий даже до WORD!

```
00:50.24        01:50.23        02:49.84        03:49.22        04:49.81
05:49.58        06:49.84        07:49.77        08:50.24        09:50.43
10:50.62        11:50.10        12:49.89        13:49.49        14:50.90
15:00.00        16:00.00        17:00.00        18:00.00        19:00.00
20:00.00        21:00.00        22:00.00        23:00.00        24:00.00
25:00.00        26:00.00        27:00.00        28:00.00        29:00.00
30:00.00        31:00.00
```

Листинг 6 распределение вероятности по битам в штатной функции rand() от MS VC (номер бита: вероятность выпада 1 или 0).

А вот gcc 3.3.4 (другие версии не тестировались) дает совсем другой результат, задействовав 31 бит, и на десятые доли процента обгоняющий MS VC по качеству. В некоторых приложениях эта разница оказывается более, чем критична! Так что gcc рулит!

```
00:50.71        01:50.23        02:49.54        03:50.11        04:49.42
05:50.87        06:49.87        07:49.53        08:50.16        09:50.38
10:50.60        11:49.60        12:50.53        13:49.75        14:49.98
15:50.07        16:49.34        17:49.54        18:49.74        19:49.45
20:50.40        21:50.62        22:49.70        23:49.56        24:49.40
25:50.38        26:49.73        27:50.23        28:49.90        29:49.41
30:49.75        31:00.00
```

Листинг 7 распределение вероятности по битам в штатной функции rand() от gcc

трюк 4: самое быстрое сравнение памяти

Большинство реализаций функции memcmp() так или иначе сводятся к оптимизированному ассемблерному алгоритму:

```
while ( --count && *(char *)buf1 == *(char *)buf2 )
{
        buf1 = (char *) buf1 + 1;
        buf2 = (char *) buf2 + 1;
}
```

Листинг 8 канонический алгоритм сравнения двух блоков памяти

А как на счет того, чтобы ускорить его раза эдак в два, причем без использования SSE, pre-fetch'a и прочих подобных расширений? Поверьте мне, парни, это возможно! Достаточно разбить блоки памяти на кусочки по от 128 байт до ~4 Кбайт (в зависимости от размера самих блоков), и сравнивать не сами блоки, а контрольные суммы "кусочков". Если функции расчета контрольных сумм разместить в отдельных потоках, то на многоядерных и HT-процессорах они будет выполняться _параллельно_ в результате чего производительность практически удвоится. Но даже на однопроцессорных машинах мы получим определенных выигрыш, поскольку контроллеры памяти оптимизированы под работу с одним потоком памяти и попеременно обращение к двум ячейкам, находящихся в различных DRAM-банкам в некоторых случаях вызывает значительную деградацию производительности (подробнее об этом можно прочитать

в моей книге "техника оптимизации программ", электронная версия которой доступна на http://kpnc.opennnet.ru и ftp://nezumi.org.ru).

Единственная проблема заключается в том, что идентичность CRC еще не гарантирует идентичности блоков! То есть, данный алгоритм никогда не выдает "ложных негативных" результатов, но с определенной (правда, очень невысокой) степенью вероятностью способен на "ложный позитив". Поэтому, его следует применять для сравнения тех блоков, о которых заранее известно, что они скорее всего различны и мы просто хотим убедиться в этом, поскольку, если CRC-алгоритм не обнаружил сравнений, для 100% уверенности необходимо инициировать стандартный алгоритм сравнения. Впрочем, при дроблении сравниваемых блоков на кусочки порядка 128 байт, вероятность коллизий CRC32 (и тем более CRC64) насколько мала, что ей можно полностью пренебречь (если, конечно, речь идет о непредумышленной "подделке" блоков злобствующими хакерами).