сишные трюки от мыщъха

крис касперски ака мыщъх, no-email

мыщъх продолжает делиться трюками и хитростями эффективного программирования на си. сегодня мы рассмотрим: строки, указатели, циклы, память и многое другие аспекты практического программирования, которые наверняка вызовут дикий "вой" у всех теоретиков от языка, но… они работают и это главное!

строки

борьба с инвариантами: самой распространенной ошибкой, в разы снижающей производительность, является присутствие функций-инвариантов в теле цикла. Вот классический пример:

```
for(a = 0, x = 0; a < strlen(s); a++)
{
        x += s[a];
}
```

Листинг 1 не оптимизированный вариант с инвариантом в теле цикла

С точки зрения программиста очевидно, что функция strelen _не_ модифицирует строку s, а потому может быть вычислена лишь однажды. Только вот компилирует этого не знает, придерживаясь принципа: все, что может быть передано по ссылке, _может_ быть изменено, поэтому strlen(s) заново вычисляется на _каждой_ итерации цикла, что при длинных строках снижает производительность более чем на порядок!

Исправленный вариант выглядит так:

```
n = strlen(s);
for(a = 0, x = 0; a < n; a++)
{
        x += s[a];
}
```

Листинг 2 оптимизированный вариант с выносом инварианта

выравнивание строк: наиболее эффективно обрабатываются строки, начинающиеся с адреса, кратного четырем. Именно так компилятор размещает их в стеке и статической памяти. отсюда функция strlen(s) выполняются эффективно, а вот strlen(s+1) — не очень. Тоже самое относится и ко всем остальным функциям. Поэтому, всегда стремитесь выравнивать строки, когда это только возможно. Скажем, "strcpy(s, "bytes "); strcat(s, very_long_string);" выполняется неэффективно, но если переписать код так: "strcpy(s, "bytes: "); strcat(s, very_long_string);", то скорость его выполнения значительно возрастет, за счет того, что адрес конца строки s станет кратен 4 байтам.

правильный выбор функций: при работе с относительно короткими строками замена strlen(s) на strchr(s, 0) может дать до 5-7% ускорения, а вот замена нескольких strcat'ов на последовательность вызов нестандартной функцией stpcpy (которая тем не менее присутствует _во всех_ современных компиляторах), дает выигрыш уже в разы!

указатели

Компиляторы стремятся размещать переменные в регистрах, избегая "дорогостоящих" операций обращения к памяти, однако не всегда это у них получается, особенно при работе с указателями, поскольку, в общем случае компилятор не может быть уверен, что два _различных_ указатели не адресуют _одну_ _и_ _туже_ ячейку памяти.

Вот, например:

```
f(char *x, int *dst, int n)
{
        int i; for (i = 0; i < n; i++) *dst += x[i];
```

```
}
```

Листинг 3 пример с лишними обращениями к памяти, от которых можно избавиться вручную

Компилятор не может поместить переменную dst в регистр, поскольку если ячейки *x и *dst частично или полностью перекрываются, модификация ячейки *dst приводит к неожиданному изменению *x! Бред, конечно, но Стандарт таких трюков не запрещает, а оптимизатор не имеет права отступать от Стандарта, поэтому обращения к памяти происходят на _каждой_ итерации, а это весьма "дорогостоящая" в плане процессорных тактов операция!

Переписанный код выглядит так:

```
f(char *x, int *dst, int n)
{
        int i,t =0;
        for (i=0;i<n;i++) t+=x[i];     // сохранение суммы во временной переменной
        *dst+=t;                       // запись конечного результата в память
}
```

Листинг 4 оптимизированный вариант

операции разыменования: префиксы намного выгоднее по сравнению с постфиксов при разыменовании, в результате чего код while(*++p) существенно эффективнее чем while(*p++), во всяком случае на платформе x86 и, по всей видимости x86-64 (к сожалению, в силу отсутствия железа проверить возможности не было). Однако, операциями разыменования смешивать с постфиксами и префиксами следует _крайней_ осторожно, иначе можно получить очень неожиданный результат (см. "неудачный выбор приоритетов в Си"). При работе на x86 (с распространёнными компиляторами) использование индексов эффективнее сдвига указателей. Не всегда, но очень часто. То есть, код типа "for(a=0;a<len;a++) *dst++ = *src++;" гораздо сложнее оптимизируется, чем "for(a=0;a<len;a++) dst[a] = src[a];", хотя качественный оптимизатор в обоих случаях должен сгенерировать идентичный машинный код.

неудачный выбор приоритетов в Си: вопреки "здравому смыслу" конструкция типа *p[a]++ увеличивает отнюдь _не_ содержимое ячейки, на которую указывает *(p+a), а значение самого указателя p! Для достижения ожидаемого результата необходимо либо явно навязать наше намерение компилятору путем расстановки скобок: "(*p)[a]++;", либо же вовсе отказаться от оператора "++", заменив его оператором "+=" и тогда наш код будет выглядеть так: "*p[a]+=1;"

Представляется интересным докопаться до _сути_ происходящего. Ведь основное кредо Си - краткость. Чего стоит один неявный int, который попил много крови разработчикам компиляторов. И тут... вдруг сталкиваешься с таким расточительством! Ведь, чтобы использовать '*' надо ставить скобки, а это - целых два нажатия на клаву. Зачем? Может быть, есть такие ситуации, где именно такой расклад приоритетов дает выигрыш? Вообще: о чем думали в этот момент разработчики языка? В доступных мне книжках никаких вразумительных объяснений ситуации я так и не нашел.

...прозрение наступило внезапно и причина, как выяснилась, оказалась даже не в самом языке, а... в особенностях косвенной автоинкрементной/авто-декрементной адресации процессора PDP-11, из которого, собственно, и вырос Си. Команда типа "MOV @(p)+, xxx" пересылает содержимое **p в xxx и затем увеличивает значение p. Да! Именно p, а отнюдь не ячейки, на которую **p ссылается!!!

Так стоит ли удивляться тому, что люди, взращенные на идеологии PDP-11, перенесли ее поведение и на разрабатываемый ими язык? И, кстати, о птичках. Система адресации PDP-11 _намного_ мощнее, удобней и элегантнее, того уродства, что реализовано в x86...

Хотите испытать свой компилятор? Нет проблем! Вот довольно познавательный листинг!

```
main()
{
        char    buf; char* p_buf[2]; char **p;
        #define INIT buf=0x66; *p_buf=&buf;   *(p_buf+1)=&buf; p=&p_buf;

        INIT;
        printf("char **p;\n");
        printf("p = %p; *p = %p; **p = %x\n\n",p, *p, **p);
```

```
        *p[0]++;                printf("*p[0]++;\n");
        printf("p = %p; *p = %p; **p = %x\n",p, *p, **p);
        printf("смотрите, увеличилось _не_ содержимое **p,\n");
        printf("а указатель, на который ссылается *p!\n");
        printf("т.е. мы получили _совсем_ не то, что хотели!\n\n");

        INIT;
        (*p)[0]++;              printf("(*p)[0]++;\n");
        printf("p = %p; *p = %p; **p = %x;\n",p, *p, **p);
        printf("хорошо, заключаем *p в скобки, тем самым явно\n");
        printf("навязывая компилятору последовательность действий\n\n");

        INIT;
        *p[0]+=1;               printf("*p[0]+=1;\n");
        printf("p = %p; *p = %p; **p = %x;\n",p, *p, **p);
        printf("забавно, но замена оператора ++ на оператор +=\n");
        printf("эту проблему как рукой снимает!\n");
}
```

Листинг 5 пример, демонстрирующий специфику приоритетов операций разыменования в Си

регистровые ре-ассоциации

Для преодоления катастрофической нехватки регистров, некоторые компиляторы стремятся совмещать счетчик цикла с указателем на обрабатываемые данные. Код вида "`for (i = 0; i < n; i++) n+=a[i];`" трансформируется оптимизатором в "`for (p= a; p < &a[n]; p++) n+=*p;`" Экономия налицо! Вместо четырех переменных после преобразования остались всего лишь три!

Впервые (насколько мне известно) эта техника использовалась в компиляторах фирмы Hewlett-Packard, где она фигурировала под термином *register reassociation*. А что же конкуренты?! Возьмем следующий код (кстати, выдранный из документации на HP компилятор):

```
int a[10][20][30];
void example (void)
{
        int i, j, k;
        for (k = 0; k < 10; k++)
                for (j = 0; j < 10;j++)
                        for (i = 0; i < 10; i++)
                                a[i][j][k] = 1;
}
```

Листинг 6 неоптимизированный кандидат на регистровую ре-ассоциацию

Грамотный оптимизатор должен переписать его так:

```
int a[10][20][30];
void example (void)
{
        int i, j, k;
        register int (*p)[20][30];
        for (k = 0; k < 10; k++)
                for (j = 0; j < 10; j++)
                        for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10; i++)
        *(p++[0][0]) = 1;
}
```

Листинг 7 оптимизированный вариант — счетчик цикла совмещен с указателем на массив

Эксперимент показывает, что ни Microsoft Visual C++, ни GCC не выполняют регистровых реассоциаций ни в сложных, ни даже в простейших случаях. С приведенным примером справился один лишь Intel C++, да и то лишь частично, поэтому, в критических к производительности случаях, оптимизировать код необходимо вручную.