си'шные трюки
VI выпуск

крис касперски ака мыщъх, no-email

реализация ROL/ROR

Огромным недостатком языка Си (за который бы голову оторвать его создателям) является отсутствие операторов циклического битового сдвига, которые присутствуют практически на всех процессорах и без которых не обходится ни подсчет CRC32, ни вычисление корректирующих кодов Рида-Соломона, ни куча других подобных алгоритмов.

Некоторые программисты используют ассемблерные вставки, прибегая к непосредственному вызову команд процессора (на x86 это ROL/ROR – циклический битовый сдвиг влево и вправо соответственно), однако, такое решение делает программу непереносимой, причем, совершенно неоправданно непереносимой, поскольку, циклический сдвиг элементарно реализуется стандартными средствами Си.

Существует множество способов сделать это, вот только один из них. Не самый быстрый, требующий двух дополнительных переменных (хотя, в принципе, можно обойтись и одной), зато наглядный и универсальный, то есть работающий с переменными любой разрядности:

```
ROL(int a, int n)
{
        int t1, t2;
        n = n % (sizeof(a)*8);        // нормализуем n
        t1 = a << n;                  // двигаем a вправо на n бит, теряя старшие биты
        t2 = a >> (sizeof(a)*8 - n);  // перегоняем старшие биты в младшие
        return t1 | t2;               // объединяем старшие и младшие биты
}
```

Листинг 1 реализация циклического сдвига влево на Си (здесь: "a" – переменная, которую нужно сдвигать, "n" – на сколько разрядов сдвигать);

```
ROR(int a, int n)
{
        int t1, t2;
        n = n % (sizeof(a)*8);        // нормализуем n
        t1 = a >> n;                  // двигаем a влево на n бит, теряя младшие биты
        t2 = a << (sizeof(a)*8 - n);  // перегоняем младшие биты в старшие
        return t1 | t2;               // объединяем старшие и младшие биты
}
```

Листинг 2 реализация циклического сдвига влево на Си

```
#define VAL 0x12345678
printf("%Xh %Xh %Xh\n",VAL,ROL(VAL,8),ROR(VAL,8));
…
$12345678h 34567812h 78123456h
```

Листинг 3 испытание обоих функций на прочность

Данный алгоритм автоматически определяет разрядность аргумента a, поэтому функии ROL/ROR можно безболезненно преобразовать в макросы. Также, можно реализовать универсальную функцию/макрос ROx, где направление сдвига задается значением аргумента n, если он положительный — сдвигаем a вправо, вызывая ROL, если n отрицательный — сдвигаем a влево, вызывая ROR и передавая -1*n.

бумеранг всегда возвращается

Имея в своем распоряжении функцию циклического сдвига, мы можем выполнять эффективный разворот, отображение и другие типы трансформаций геометрических фигур. Главное — выбрать правильную схему отображения байт сдвигаемой переменной на байты двумерного массива с геометрической фигурой внутри.

В самом деле, и разворот, и трансформация по сути своей являются сдвиговыми операциями, что приведенный ниже пример, собственно говоря, и подтверждает:

```
int x;
// кодируем "пропеллер" в двухмерном массиве,
// развернутым в линейный массив, представленный двойным словом
//
// ** <- фигура типа
.. *  <- "бумеранг"
union{int ou; char uo[4];} xxx; xxx.ou = 0x2A2A2A20;

for (x=0;x<8;x++)
{
        // выбираем схему отображения,
        // реализующую разворот
        printf("%c%c\n%c%c\n\n",xxx.uo[0],xxx.uo[3],xxx.uo[1],xxx.uo[2]);
        xxx.ou=ROL(xxx.ou,8);
}
```

Листинг 4 циклический сдвиг, вращающий "бумеранг"

Откомпилировав программу и запустив ее на выполнение, мы получим следующий результат, для экономии бумаги записанный в одну строку:

```
 * ->  ** -> ** -> *  ->  * -> ** -> ** -> *
**      *    *     **    **     *    *     **
```

Листинг 5 "полет бумеранга"

Как видно, "бумеранг" исправно вращается против часовой стрелки (или по часовой стрелке, если используется функция ROR), но стоит нам выбрать другую схему отображения, как вместо вращения мы получим зеркальное отображение. Для этого в нашей программе достаточно изменить всего одну строку:

```
        printf("%c%c\n%c%c\n\n",xxx.uo[0], xxx.uo[1], xxx.uo[2], xxx.uo[3]);
```

Листинг 6 схема отображения, обеспечивающая зеркальное отражение

```
 * -> *  -> ** -> ** ->  *  -> * -> ** -> **
**    **     *    *     **     **    *    *
```

Листинг 7 демонстрация зеркального отображения

Теперь каждая последующая трансформация "бумеранга" представляет собой зеркального отображение предыдущей!

Слегка усовершенствовав функцию циклического разворота, мы сможем трансформировать не только массивы 2x2, но и гораздо большие фигуры.

вещественная арифметика в целых числах

Прошло то время, когда математический сопроцессор считался предметом гордости его владельца. Теперь — это неотъемлемая часть процессора, однако, скорость сложения/вычитания вещественных чисел заметно отстает от целых и потому, в приложениях, критичных к производительности, использования целочисленных переменных является более предпочтительным. А как быть, если нам нужна дробная часть? Очень просто — умножаем целое число на 10*N (где N – количество знаков после запятой) и дальше работаем с ним как обычно, а на финальном этапе вычислений делим результат на 10*N.

Однако, используя этот прием, следует помнить о том, что операции умножения/деления на 10 нельзя реализовать через битовые сдвиги, а операция целочисленного деления на x86 выполняется крайней медленно и неэффективно. Намного менее эффективно, чем операция вещественного деления! Поэтому, преобразование типов съедает львиную долю выигрыша от производительности и оправдывает себя лишь в случае действительно громоздких вычислений, когда временем, потраченном на преобразование, можно пренебречь. Но и в этом случае, в целочисленных вычислениях не должно присутствовать инструкций деления, в противном случае лучше все-таки использовать вещественную арифметику и не выпендриваться.

Разумеется, под "делением" (равно, как и взятием остатка) имеется ввиду "деление на число не являющееся степенью двойки". Кстати говоря, существуют формулы быстрого деления/взятия остатка на любую константу, и некоторые компиляторы (в том числе и Microsoft Visual C++) используют их, придавая производительности неожиданное подкрепление. Так что вопрос о том, какую арифметику лучше всего использовать — остается открытым. Это зависит и от типа процессора, и от интеллектуальности компилятора и от многих других вещей, поэтому, прежде, чем приступать к оптимизации, следует внимательно изучить конкретную оперативную обстановку.

кэширование вычислений

Если "тяжеловесная" функция многократно вызывается с одними и теми же аргументами, лучшим способом оптимизации будет запоминание однажды вычисленного результата и возвращение его в готовом виде, в случае совпадения аргументов. В особенности это касается операций поиска в файле/базе данных. Идентичные запросы здесь — обычное дело и базы данных давно научились кэшировать их. Так почему бы не применить этот подход и к поиску в файле?

С математическими функциями все обстоит еще интереснее. Как известно, существуют два основных подхода — вычисления на лету и предвычисленные таблицы (часто комбинируемые с различными алгоритмами интерполяции). Проблема в том, что мы наперед не знаем какой из двух подходов окажется эффективнее. Вычисления требуют времени, предвычисленные таблицы — памяти, а обращение к памяти (особенно вытесненной на диск) — операция не из "дешевых", однако, кэширование вычислений позволяет решить эту проблему, автоматически адаптируясь под конкретную ситуацию, и особенно хорошо себя проявляя в распределенных системах, где мы можем одновременно начать поиск с поиском уже вычисленного значения. А там уже кого опередит. Если ответ вернется раньше, чем вычисления будут закончены — прерываем их!

Даже на однопроцессорных машинах с поддержкой Hyper Threading уже наблюдается ощутимый прирост производительности! И чем больше процессоров — тем значительнее выигрыш. Единственное, о чем следует позаботиться — чтобы два и более процессоров не выполняли одних и тех же вычислений одновременно. То есть, получив запрос на наличие уже вычисленных значений, функция, не должна выполнять их сама, даже если они отсутствуют в ее кэше, поскольку их _уже_ выполняет кто-то другой. Соответственно, если функция уже приступила к вычислениям, одна должна вернуть сообщение "данных еще нет, но скоро будут", чтобы та, другая, функция не начинала вычисления с нуля.