записки хакера (разное)

крис касперски ака мыщъх, no email

сортировка списков

Списки — популярные и во многом интересные структуры данных. Они естественным образом поддерживают быструю вставку и удаление элементов, но ни в одном известном мне руководстве не затрагиваются вопросы их сортировки. Ну да, конечно, в библиотеке STL есть класс list с готовым методом stort. Пользуйся – не хочу! Но универсальное решение по определению не есть хорошее. Это раз. Не все все программисты пользуются STL — это два. Наконец, в Си ничего подобного и в помине нет – это три. К тому же было бы полным непрофессионализмом взвывать list::sort, не задумываясь о том, как он работает. А правда, как? Давайте, не подглядывая в исходный код STL, попытаемся реализовать сортировку список самостоятельно.

Начнем с того, что отсортировать список тривиальным вызовом qsort не удастся, поскольку она рассчитывает, что следующий сортируемый элемент расположен непосредственно за концом предыдущего. При обработке массивов все так и происходит, но списки ведут себя иначе. Малого того, что элементы списка могут размещаться в памяти в каком угодно порядке, нет никаких гарантий, что за концом какого-то элемента находится действительный элемент!

Рассмотрим следующий пример:

```
struct LIST *last_record = 0;
strict LIST *list = (struct LIST*) malloc(sizeof(struct LIST));

for(a=0;a<N; a++)
{
        list->val = a;
        list->next_record = (struct LIST*) malloc(sizeof(struct LIST));
        list = list->next_record;
} list->val=a; list->next_record = 0;
```

Листинг 1 кандидат на сортировку

Допустим, нам необходимо отсортировать список так, чтобы последующий элемент имел большее или такое же значение val. Как это сделать? Алгоритм сортировки прост как косяк: перегоняем списки в массив, сортируем его как обычно, затем либо уничтожаем несортированный список и создаем новый, либо "натягиваем" отсортированный массив на уже существующий "скелет", т.е. используем уже выделенные блоки памяти. Естественно, последний способ работает _намного_ быстрее! Демонстрационная программа для сортировки прилагается.

двух маршрутные списки

В тех случаях, когда требуется осуществить ту или иную выборку элементов из списка для их последующей обработки (проще говоря _фильтрацию_), то можно пойти двумя путями:

1) вернуть отфильтрованные элементы в отдельном списке (что имеет тот недостаток, что понапрасну расходует память, а при "многокаскадной" фильтрации мы рискуем окончательно запутаться в куче списков);
2) помимо поля `struct LIST *next_record` добавить еще одно поле "`struct LIST *next_filtred__record`" В таком случае мы будем иметь дело всего лишь с _одним_ списком, содержащим как оригинальные, так и отфильтрованные данные.

Естественно, еще потребуется поле "`struct LIST* first_filtred_record`", содержащее указатель на первый отфильтрованный элемент списка. Оно необходимо затем, что первый элемент оригинального списка не обязательно будет совпадать с первым отфильтрованным элементом. К тому же "двух маршрутные" списки естественным образом поддерживают "каскадную" фильтрацию. В самом деле, мы сканируем список, перескакивая по

полям "next_filtred_record", и, если следующий элемент не проходит через фильтра, то предыдущая ссылка перескакивает наперед. Так же, очевидно, что нам потребуется два набора функций для работы с двумя маршрутами.

обработка ошибки выделения памяти

Постоянная проверка успешности выполнения интенсивно используемых функций во-первых, слишком утомительна, во-вторых, загромождает исходный текст, и, в-третьих, приводит к неоправданному увеличению объема откомпилированного кода программы.

```
char *p;
p = malloc(BLOCK_SIZE);
if (p==0)
{
        fprintf(stderr,"-ERR: недостаточно памяти для продолжения операци\n");
        _exit();
}
```

Листинг 2 пример неудачной проверки успешности выделения памяти

Кстати говоря, следующий код: "p = malloc(x); if (!p) return 0;" даже хуже, чем отсутствие проверки вообще, т. к. при обращении к нулевому указателю Windows хоть выругается, указав на место сбоя, а такая кривая проверка просто тихо кончит программу...

Решение заключается в создании "оберток" для интенсивно используемых функций, проверяющих успешность завершения вызываемой ими функции и при необходимости рапортующих об ошибке с завершением программы или передающих управление соответствующему обработчику данной аварийной ситуации.

```
void* my_malloc(int x)
{
        int *z;
        z=malloc(x);
        if (!z) GlobalError_and_save
}
```

Листинг 3 корректная "обертка" вокруг функции malloc()

Между прочим, виртуальная память не безгранична и иногда она неожиданно кончается. Попытка выделения нового блока посредством malloc дает ошибку. В этой ситуации очень важно корректно сохранить все не сохраненные данные и выйти. А как быть если для сохранения требуется определенное кол-во памяти? Да очень просто! при старте программы выделяем malloc'ом столько памяти, сколько ее может потребоваться для аварийного сохранения. если нас обломают на память, то просто не запускаемся а с ругательством сваливаем. затем, при нехватке памяти просто сохраняемся (необходимая память у нас есть) и все!!!