сишные трюки
(1Ch выпуск)

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

доброкачественная инициализация переменных — явление достаточно редкое, можно даже сказать уникальное. анализ открытых исходных текстов и дизассемблирование закрытых двоичных модулей показывает, что большинство программистов не имеют никакого представления о внутренней кухне компилятора, совершая грубые ляпы, о которых мы сейчас и поговорим!

#1 – стек, статика и динамика

Чем отличаются локальные автоматические переменные от локальных статических/глобальных? Да много чем отличаются… так сразу на вскидку и не скажешь, а без учебника и не вспомнишь, что статические/глобальные переменные инициализируются "самостоятельно" в отличии от… Незнание (игнорирование) этой особенности приводит к двум распространенным ошибкам.

Код вида int x = 0; foo(){… return x;} выглядит ужасно непрофессионально (x гарантированно обнуляется компилятором путем помещения его в секцию данных, тогда как явное присвоение нуля — выполняется уже в реальном времени, напрягая процессор лишними машинными командами). Неприятно, конечно (явное свидетельство того, что программист умных книжек не читает), но все-таки не смертельно.

А вот другая ошибка, совершаемая уже теми, кто учебники все-таки читает, но не дочитывает — "foo(){ static int x;… return x;}". Что здесь неправильно?! Ведь переменная x _гарантированно_ равняется нулю и инициализировать ее "вручную" необязательно! Да, верно, x будет равна нулю, но… лишь при первом выполнении функции. При всех последующих в ней останется знание, которое она имела на момент выхода из функции foo(), что рискует развалить всю программу, а потому инициализировать статические переменные все-таки нужно, если, конечно, они не задействованы для умышленного сохранения значений для использования их в последующих вызовах функции.

Явное присвоение значений — довольно расточительная операция в плане процессорных тактов и объема кода, особенно если переменных много. Намного выгоднее передать функции memset указатель на начало блока статических (глобальных) переменных и проинициализировать их одним махом. Даже при десятке переменных экономия становится очевидной и хорошо заметной. Следуя духу и этикету языка, следовало бы загнать все переменные в единую структуру, чтобы обеспечить гарантированный порядок их размещения в памяти, однако, работать со структурами жутко неудобно, да и не нужно. Статические и глобальные переменные размещаются в памяти в порядке их объявления в программе и потому нам достаточно всего лишь получить указатель на первую и последнюю переменные.

С локальными автоматическими переменными этот трюк, увы, не работает. В общем случае они размещаются в порядке обратном _обращению_ к ним (т.е. как только компилятор встречает обращение к локальной переменной он забрасывает ее на верхушку стека и потому последняя используемая переменная оказывается первой в стековом кадре), однако, из этого правила существует множество исключений.

Оптимизирующие компиляторы стремятся выкинуть максимум локальных переменных, загнав их в регистры или вычисляя эффективные значения еще на стадии компиляции. А последние версии GCC и MS VC в добавок к этому бьют стековый фрейм на две части, складируя в одну буфера, а в другую скалярные переменные и указатели для затрудняя атак на переполнение. Как следствие мы уже не можем инициализировать локальные переменные через memset. То есть еще как можем! Достаточно поместить их в структуру! Неудобно, конечно, но… на какие жертвы не пойдешь ради оптимизации! Только в этом случае она будет называться "пессимизацией", поскольку компилятор не может оптимизировать члены структуры так же свободно, как обычные локальные переменные.

Выход?! Некоторые компиляторы поддерживают нестандартный ключ, предписывающий выполнять инициализацию стекового кадра при его открытии. На первый взгляд очень полезная штука, но пользоваться ей категорически не рекомендуется (поэтому мыщъх даже не будет говорить что это за ключ такой и кто его поддерживает), поскольку в этом случае весь стековый фрейм инициализируется целиком, даже если содержит массивы, явно

инициализируемые по ходу программы ненулевыми значениями. Но это мелочь. Подумаешь, двойное обращение к памяти! Гораздо хуже, когда программист, закладывающийся на то, что инициализацию локальных переменных выполнит компилятор, публикует код своей программы или использует его фрагменты в другом проекте, забыв о том, что _там_ локальные переменные уже не инициализируются!!!

Вывод: без особой нужды массивы лучше в стеке не размещать. Используйте для этого статическую память или кучу. Первая инициализируется при загрузке исполняемого файла в память, вторую программист инициализирует явно, когда это действительно необходимо (на самом деле, менеджер кучи, встроенный в операционную систему, всегда выполняет инициализацию блоков памяти перед их отдачей прикладной программе, однако, при переходе на прикладной уровень — уровень библиотек и RTL, — мы, в общем случае, не можем сказать, выполняется ли автоматическая инициализация или нет, а потому лучше не рисковать, особенно, если программу планируется переносить на другие платформы или компилировать более чем одним компилятором).

#2 – строки и массивы

А вот другая популярная ошибка, ставшая неофициальным стандартом де-факто и встречающая практически повсеместно:

```
foo()
{
        char s[]="hello, sailor!\n";
        …
        bar(s);
}
```

Листинг 1 классическая ошибка использования локальных буферов

Что не так?! Вполне приличный код! А если подумать?! Компилятор размещает строку "hello, sailor!\n" в секции данных (хотя тут возможны вариации), что происходит на стадии компиляции, а затем копирует ее в локальный буфер при каждом вызове функции уже на стадии исполнения! Таким образом, мы получаем двойной перерасход памяти и довольно ощутимые тормоза, которые в данном случае ничем не оправданы, поскольку, функция bar не изменяет строку s, а потому перед "char s[]" необходимо поставить "static" или вынести s в глобальные переменные.

Впрочем, это мелочи жизни. Настоящие проблемы начинаются когда программист (причем, совершенно вменяемый трезвый и совсем не обкуренный) пишет код вида:

```
foo()
{
        int matrix[100][100]={{1,2,3},{4,5,6},{7,8,9}};
        …
}
```

Листинг 2 ужас летящий на крыльях ночи

А здесь что не так?! Программист создает законных двухмерный массив, инициализируя малую его часть (очевидно, что остальные ячейки предполагается заполнить по ходу выполнения функции foo). На самом деле, согласно Стандарту, здесь инициализируется _весь_ массив, причем, ненулевые ячейки компиляторы инициализируют индивидуально, расходуя на каждую из них по меньшей мере одну машинную инструкцию, но это в идеале, а на практике, компилятору MS VC необходимо 27 команд чтобы справится с вышеприведенным массивом, что не есть хорошо, особенно, если функция foo вызывается больше одного раза. К тому же, стек не резиновый и обычно (читай — по умолчанию) потоку достается порядка 1 Мбайта.

За бездумное размещение массивов в стеке — уже давно пора расстреливать. Ключевое слово "static" размещенное перед "int matrix" сокращает потребности памяти и увеличивает скорость выполнение программы в несколько раз! А как быть, если статический массив нас ну никак не устраивает? Допустим, массив должен инициализироваться при каждом вхождении в функцию. Нет ничего проще!!! Размещаем исходный массив в глобальной или статической переменной, а при каждом вхождении в функцию копируем его во временный буфер, выделяемый из пула динамической памяти. Копирование, осуществляемое посредством memcpy, _намного_ быстрее поэлементной инициализации! (Напоминаю, что статические массивы инициализируются на стадии компиляции, не транжиря процессорное время).

```
foo()
{
        static int _matrix[100][100]={{1,2,3},{4,5,6},{7,8,9}};
        int (*matrix)[100][100];
        matrix= (int(*)[100][100]) malloc(sizeof(matrix));
        if (!matrix) return -1; else memcpy(matrix, _matrix, sizeof(matrix));
        …
        free(matrix);
}
```

Листинг 3 оптимизированный вариант работы с массивом

Конечно, куча не лишена недостатков. Выделение динамической памяти занимает больше времени, чем стековой, динамические блоки необходимо освобождать и еще обрабатывать ситуацию с нехваткой памяти, проверяя успешность завершения malloc, однако: а) динамической памяти _как_ _правило_ имеется в избытке и отсутствие проверки в общем-то не фатально; б) никто не гарантирует успешность выделения стековой памяти, а универсального способа проверки (работающего во всех системах) нет и остается только молится, надеясь на то, что стека все-таки хватит. Короче, существует тысяча и одна причина для отказа от использования стековой памяти при работе со строками и массивами.

#3 – массивы, начинающиеся не с нуля

В прошлых выпусках "трюков" мы уже рассматривали способы как организовать массив, начинающийся, например, с единицы, что особенно удобно при переносе программ с Паскаля и Фортрана на Си, однако, те способы не работали с Си++ и к тому же не позволяли создать массив, начинающиеся с произвольного индекса, например, с 0x69.

Предложенный ниже прием полностью совмести с Си++, однако работает не на всех платформах. Руководящая идея проста как два весла: получаем указатель на массив и уменьшаем его на величину начального индекса, например, мы можем создать массив 6...9:

```
foo()
{
        static int x_array[9 - 6];
        int *p_array = x_array - 6;
        …
        return 0;
}
```

Листинг 4 создание массива p_array[6..9]

Аналогичный трюк работает и со стековыми массивами (хотя, как мы уже говорили выше, в стеке массивы лучше не размещать), и с динамическими (нужно только не забывать увеличивать указатель на массив при освобождении памяти, что не украшает программу и чревато появлением ошибок, но… за любые удобства в этом мире приходится чем-то платить.

Другое существенное ограничение заключается в том, что при вычитании начального индекса из указателя мы рискуем нарваться на "заворот". Но это навряд ли. Во всех современных операционных системах и стек, и куча, и секция данных лежат довольно далеко от нулевого адреса, однако, создавать массив типа 666666...666669 уже опасно. На одной системе (или даже версии системы) это может сработать, а на другой — уже нет.