сишные трюки
(12h выпуск)

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

кто-то в шутку сказал, что программисты в среднем тратят 10% времени на написание программы и 90% — на ее отладку. разумеется, это преувеличение и правильно спроектированная программа должна отлаживать себя сама или по крайней мере автоматизировать этот процесс. сегодняшний выпуск трюков, как вы уже догадались, посвящен магии отладки.

трюк 1: обрамление отладочного кода

Достаточно многие программисты используют для "обрамления" отладочного кода директивы условной трансляции (пример использования которых приведен в листинге 1), в результате чего отладочный код автоматически удаляется из release-версии продукта.

```
#define _DEBUG_                                                 // debug info is enabled
…
#ifdef _DEBUG_
        pritnf("output debug info\n");
#endif
```

Листинг 1 распространенный, но неудобный способ "обрамления" отладочного кода

Однако, это не самый продвинутый вариант и при желании его можно существенно оптимизировать, заменив директиву препроцессора "#ifdef" на оператор "if(0)" (см. листинг 2):

```
#define _DEBUG_ 1                                               // debug info is enabled
…
if(_DEBUG_)
{
        pritnf("output debug info\n");
}
```

Листинг 2 оптимизированный способ "обрамления" отладочного кода

Если _DEBUG_ == 0, то выражение "if(_DEBUG_)" превращается в "мертвый код", автоматически детектируемый и удаляемый практически всеми оптимизирующими компиляторами.

Кстати говоря, оператор "if(0)" выгодно использовать для временного отключения части кода, что обычно делается с помощью комментариев. Однако, при многократном включении/отключении большого количества строк, приходится тратить кучу времени на их комментирование, вставляя оператор "//" в начало каждой строки. Теоретически, весь блок кода можно отключить с помощью оператора "/* - - - */", но воспользоваться этой теорией удается далеко не всегда. Увы! Язык Си/Си++ не поддерживает вложенных комментариев последнего типа и если они уже встречаются в отключаемом коде, мы получаем сообщение об ошибке.

С другой стороны, код, отключенный посредством комментариев, в продвинутых средах разработки отмечается другим цветом (например, серым), а потому намного более нагляден, чем оператор "if(0)", который никак не выделяется в листинге и потому однажды отключенный код рискует отправиться в забвение и чтобы этого не произошло рекомендуется использовать директиву "#pragma message", выводящую сообщение при компиляции о том, что такой-то участок кода временно отключен.

трюк 2: условные точки останова — своими руками

Практически все современные отладчики поддерживают условные точки останова, однако, их возможности довольно ограничены. В частности, мы не можем вызывать API-функции и потому даже такая простая задача как остановить отладчик в определенном потоке превращается в головоломку, для решения которой приходится прибегать к анализу регистра FS и прочим шаманским трюкам.

Лишь немногие отладчики позволяют загружать условные точки останова из текстового файла, который легко редактировать в своем любимом IDE с отступами, переносами строки и

прочими атрибутами форматирования, а без форматирование мало-мальски сложное условие останова становится практически нечитаемым и его приходится отлаживать вместе с отлаживаемой программой. Вот такая, значит, рекурсия получается.

Между тем, если мы не хачим двоичный файл, то намного удобнее внедрять точку останова непосредственно в сам исходный текст! На x86 платформе для этого достаточно вызывать ассемблерную инструкцию int 0x3. Естественно, это решение не универсально и к тому же системно зависимо, однако, системно зависимый код можно вынести в макрос/отдельную функцию.

"Ручные" точки останова сохраняются вместе с самой программой, что "отвязывает" нас от отладчика и мы можем попеременно использовать soft-ice, OllyDebugger и Microsoft Visual C++, например. Кстати говоря, даже если на целевой машине никакой отладчик вообще не установлен, точки останова, внедренные в программу, приведут к вызову Доктора Ватсона. Это, конечно, не отладчик, но все же лучше чем совсем ничего.

```
#define BREAK1_ENABLED 1
#define BREAK1_TEXT "arg1 and arg2 are equal"

#define break_in __asm int 0x3

foo(int arg1, int arg2)
{
        #ifdef BREAK1_ENABLED
                if (arg1 == arg2) break_in;
        #pragma message("BREAKPOINT:" BREAK1_TEXT __FILE__)
        #endif
}
```

Листинг 3 пример использования "рукотворных" условных точек останова

трюк 3: мистическое исчезновение ошибок

Некоторые виды ошибок мистическим образом исчезают при запуске программы под отладчиком и можно дебажить программу хоть до посинения, но так и не получить никакого результата.

На самом деле, прикладная программа практически не имеет никаких шансов определить — находится ли она под отладкой или нет. Исключение составляют специальные анти-хакерские приемы и пошаговое исполнение + ошибки синхронизации.

Более вероятная причина исчезновения ошибок заключается в том, что вместе с генерацией отладочной информации компилятор отрубает оптимизатор и выполняет ряд дополнительных действий, изменяющих логику поведения программы (например, инициализирует переменные).

Чтобы не спугнуть ошибки, необходимо отлаживать release-версию программы. Вот так прямо в ассемблерных кодах и отлаживать. А как быть, если мы хотим подняться на уровень исходных текстов?! К сожалению, в общем случае это невозможно. Но тут есть одна хитрость, существенно упрощающая нам жизнь.

Используя предопределенный макрос __LINE__ мы без труда заставим компилятор генерировать информацию о номерах строк, автоматически внедряемых в программу. Конечно, это совсем не тоже, что отладка на уровне исходных текстов, но все-таки какая-то зацепка уже есть. Правильно расставив директивы __LINE__ и используя их в дальнейшем в качестве своеобразных "вешков", мы легко сореентируемся — в какой части программы сейчас находится (правда, при этом следует помнить, что компилятор может переупорядочивать машинные команды по своему усмотрению и потому номера строк, определенные при помощи __LINE__ не всегда соответствуют действительности и могут отличаться на несколько строк).

Самое замечательное, что эта задача поддается автоматизации. Не составит большого труда написать плагин для OllyDebugger, распознающий внедренные номера строк и выводящий соответствующий фрагмент исходного текста на экран.

Рассмотрим следующий пример (см. листинг 4):

```
// макрос для внедрения номеров строк
#define XX dbgline(__LINE__);

// служебная функция для внедрения номеров строк
static dbgline(int line)
```

```
{
        char buf[1024];
        sprintf(buf,"%x\n",line);
        OutputDebugString(buf);
}

main()
{
        XX              // вывести номер строки [в данном случае == 15]
        printf("hello, world!\n");
        XX              // вывести номер строки [в данном случае == 17]

}
```

Листинг 4 простейший пример программы, автоматически внедряющий номера строк исходного текста в свою release-версию

Мы определяем макрос XX, вызывающий функцию dbgline() и передающий ей номер строки в качестве аргумента, что приводит к генерации следующего машинного кода: PUSH __LINE__/CALL dbgline(), который можно найти и автоматически, используя __LINE__ в качестве опорной метки (естественно, если программа занимает более одного файла, необходимо воспользоваться макросом __FILE__, который здесь не показан для упрощения).

А чтобы оптимизирующий компилятор не заинлайнил dbgline, мы объявляем ее как static. API-функция OutputDebugString() не является обязательной и просто вываливает номера строк, отображаемых отладчиком в специальном окне. Это на тот случай, если мы совсем не разбираемся в ассемблере. Кстати, дизассемблерный листинг приведенной программы выглядит так:

```
.text:00401000 _mai    proc    near
.text:00401000         push    ebp
.text:00401001         mov     ebp, esp
.text:00401003         push    15                      ; номер текущей строки
.text:00401005         call    sub_401026              ; gdbline
.text:0040100A         add     esp, 4
.text:0040100D         push    offset aHelloWorld ; "hello, world!\n"
.text:00401012         call    _printf
.text:00401017         add     esp, 4
.text:0040101A         push    17                      ; номер текущей строки
.text:0040101C         call    sub_401026              ; gdbline
.text:00401021         add     esp, 4
.text:00401024         pop     ebp
.text:00401025         retn
.text:00401025 _main   endp
```

Листинг 5 дизассемблерный листинг нашей программы