Gem #120: Создание скриптов GDB- Часть 2

Автор: Jean-Charles Delay, AdaCore

Краткое содержание: В предыдущем Gem #119 было показано, какие возможности скриптов присутствуют в GDB на основе его языка написания макро-кода, с помощью которого можно настраивать отладчик через файл `gdbinit`. В Gem #120 будут обсуждаться более продвинутые возможности скриптов GDB.

Давайте начнём…

Регистры

GDB предоставляет вам ряд предопределенных переменных. Мы уже видели некоторые из них в предыдущем Gem #119 (`$argc, $arg0, ...`), но доступно гораздо больше. Среди них есть переменные для доступа к значениям машинных регистров. См. Список ниже для полного перечисления их в случае x86:

```
$eax
$ebx
$ecx
$edx
$eflags
$esi
$edi
$esp
$ebp
$eip
$cs
$ds
$es
$fs
$gs
$ss
```

Вооружившись ими, и используя язык макрокодирования, который мы видели в предыдущем Gem, мы можем легко написать некоторые функции для удобного отображения переменных. Например, давайте напишем функцию, которая отображает значения флага в `$eflags`:

```
define eflags
    printf "     OF <%d>  DF <%d>  IF <%d>  TF <%d>",\
           (($eflags >> 0xB) & 1), (($eflags >> 0xA) & 1), \
           (($eflags >> 9) & 1), (($eflags >> 8) & 1)
    printf "  SF <%d>  ZF <%d>  AF <%d>  PF <%d>  CF <%d>
",\
           (($eflags >> 7) & 1), (($eflags >> 6) & 1),\
           (($eflags >> 4) & 1), (($eflags >> 2) & 1), ($eflags & 1)
    printf "     ID <%d>  VIP <%d> VIF <%d> AC <%d>",\
           (($eflags >> 0x15) & 1), (($eflags >> 0x14) & 1), \
           (($eflags >> 0x13) & 1), (($eflags >> 0x12) & 1)
    printf "  VM <%d>  RF <%d>  NT <%d>  IOPL <%d>
",\
           (($eflags >> 0x11) & 1), (($eflags >> 0x10) & 1),\
           (($eflags >> 0xE) & 1), (($eflags >> 0xC) & 3)
end
```

```
document eflags
Print eflags register.
end
```

С этим макросом команда `eflags` дает следующий вывод:

```
gdb$ eflags
  OF <0>  DF <0>  IF <1>  TF <0>  SF <1>  ZF <0>  AF <0>  PF <0>  CF <0>
  ID <1>  VIP <0> VIF <0> AC <0>  VM <0>  RF <0>  NT <0>  IOPL <0>
```

Это выводит значение каждого флага, но довольно многословно. Давайте напишем функцию, которая будет печатать значения флага, представляя каждое из них буквой, и печатать его в верхнем регистре, если флаг установлен, или в нижнем регистре, если это не так.

```
define flags
    if (($eflags >> 0xB) & 1)
        printf "O "
    else
        printf "o "
    end
    if (($eflags >> 0xA) & 1)
        printf "D "
    else
        printf "d "
    end
    if (($eflags >> 9) & 1)
        printf "I "
    else
        printf "i "
    end
    if (($eflags >> 8) & 1)
        printf "T "
    else
        printf "t "
    end
    ...
    if ($eflags & 1)
        printf "C "
    else
        printf "c "
    end
    printf "
"
end
document flags
Print flags register.
end
```

Вывод этой функции, например:

```
o d I t S z a p c
```

Конечно, вы можете создать столько функций, сколько вам нужно, и настроить их так, как вам нравится.

Улучшенная отладка

Когда вы отлаживаете запущенный процесс, часто желательно получить общее представление о контексте процесса и делать это для каждого шага в программе. Полезные команды контекста процесса, которые вы создали до сих пор, могут вызываться автоматически каждый раз, когда вы прерываете свою работу при отладке вашей программы. Для этого вы можете использовать специфичный для GDB макрос, называемый `hook-stop`.

Функция ловушки `hook-stop` - это специальное определение, которое GDB вызывает при каждом событии точки останова. Это означает, что вы можете использовать его для вызова пользовательских функций каждый раз, когда останавливается GDB (после точки останова, после каждого `next/nexti` и т. д.).Единственное, что вам нужно сделать, это определить и заставить делать то, что вы хотите.

Например:

```
define hook-stop
  eflags
end
document hook-stop
/!\ For internal use only - do not call manually /!\
end
```

Как только функция ловушки `hook-stop` определена пользователем, функция `eflags`, которая выводит значение каждого флага регистра, будет вызываться каждый раз, когда GDB достигнет точки останова.

Подсказки, связанные с Ada, при макро-кодировании

Помните, что синтаксис сценариев макросов GDB несколько варьируется в зависимости от текущего набора языков. Язык обычно устанавливается на «c» при отладке программ на C и «ada» при отладке программ на Ada. Но синтаксис "c" и синтаксис "ada" различаются. Например, назначение в режиме «c» выполняется с помощью:

```
set var = 1
```

тогда как в режиме" ada " это выглядит так:

```
set var := 1
```

Как видите, GDB адаптирует свой синтаксис к текущему языку, так что пользователь с одинаковым соглашением пишет, разрабатывает ли он или отлаживает. Эта особенность имеет свои плюсы и минусы, но это не тема этого Gem #120. Единственное, что вам нужно знать, это то, что из-за этого большинство макросов, которые мы определили до сих пор, не будут работать при отладке программ Ada.

Однако есть обходной путь. Внутри каждого определения макроса вы можете вручную установить текущий язык. Поэтому следующий макрос:

```
define hexdump
  if $argc != 1
    help hexdump
  else
    Do the work ...
  end
end
```

можно переписать следующим образом:

```
define hexdump
  set language c
  if $argc != 1
    help hexdump
  else
    Do the work ...
  end
  set language auto
end
```

Это позволяет избежать любых проблем, связанных с текущим используемым языком, и, поскольку мы устанавливаем язык обратно на «auto», после вызова этого макроса не будет никаких побочных эффектов.

Остерегаться

Сценарии GDB отличаются от большинства языков программирования, в частности тем, что отсутствует понятие области видимости: каждая команда оказывает глобальное влияние на среду GDB. Скажем, например, что у вас есть два макроса macroA и macroB, и что первый вызывает второй:

```
define macroA
  set language c              <= (1)
  ... do some stuff ...
  macroB
  ... do some stuff ...
  set language auto
end


define macroB
  set language c              <= (2)
  ... do some stuff ...
  set language auto           <= (3)
end
```

Приведенный выше пример будет работать как шарм в режиме «c», но посмотрите, что происходит в режиме «ada»:

(1) Вы вводите macroA, устанавливаете язык на «c»

(2) Вы вводите macroB, устанавливаете язык на «c»

(3) Вы оставляете macroB, устанавливаете язык обратно на «auto» (что означает «ada»).

Вернувшись из macroB, язык устанавливается на «ада». Если вторая часть macroA использует специфичные для C функции, такие как оператор сравнения != (Вместо /= для ada), выполнение макроса завершится неудачно.

По этой причине обязательно используйте языковой режим «c» при выполнении макросов и всегда восстанавливайте его до «auto» непосредственно перед выходом из «контекста».

Идти дальше

Сценарии являются мощной функцией отладчика GNU и предлагают значительную помощь при отладке программ с использованием командной строки. Это также может быть очень полезно при использовании GPS, поскольку GPS получает файл `.gdbinit`, так что все предопределенные макросы доступны из интерфейса командной строки GPS. Это позволяет объединить возможности текстовой и графической отладки.

Связанный со статьёй текст программы

Attached Files отсутствуют

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Сведения об авторе отсутствуют.

Last Updated: 10/26/2017
Posted on: 3/12/2012

Обсуждение…