Tutorial inicial para o GDB

Disciplina: MC102
Turmas: W / Y
Professor: Ricardo Torres
Monitores: Raoni Fassina
Ricardo Prates

- Índice

  1. Introdução
  2. Compilação para Debug
  3. Comandos básicos
  4. Exemplo prático
  5. Comandos intermediários
  6. Exercício

1. Introdução

Este tutorial foi escrito com o intuito de auxiliar os alunos iniciantes na prática da programação a utilizarem uma ferramenta de debug. O software em questão é o GDB (GNU Debugger) e pode ser utilizado num terminal (konsole, xterm, gnome-terminal) ou integrado no editor de texto Emacs.

Um debugger é um programa que supervisiona a execução do nosso programa para que tenhamos a oportunidade de verificar como ele está funcionando. Através de um debugger podemos executar o nosso programa linha por linha, bem como descobrir qual o valor das variáveis em cada instante da execução. Há algumas funcionalidades específicas que nos ajudam a ir direto para uma linha determinada ou ainda mudar o valor de uma variável forçosamente e observar os resultados.

Apesar do compilador ser bastante informativo a respeito de erros de sintaxe em nossos códigos, ele não é capaz de "adivinhar" que um certo loop não vai parar nunca ou que estamos mudando os valores das variáveis erradas. Assim, quando o nosso programa não funciona bem, pode ser bastante cansativo e demorado encontrar os erros apenas lendo o código novamente. Neste ponto é que o debugger se faz bastante útil.

2. Compilação para Debug

Para que o debugger possa controlar a execução do nosso programa, ele precisa que o compilador insira algumas informações especiais (por exemplo, uma tabela de símbolos) ao traduzir o código fonte para executável. No caso do gcc, utilizamos o flag de compilação "-g" para solicitarmos a inclusão destas informações no executável. Observem o comando de exemplo:

```
[ra123456@host ~]$ gcc basico.c -o exemplo -g
```

Este comando informa ao gcc que o arquivo fonte "basico.c" na pasta atual deve ser compilado num executável de nome "exemplo" a ser colocado também na pasta atual e por fim, que a instrumentação de debug deve ser adicionada a esse executável.

Para iniciar o gdb num terminal, digite:

```
[ra123456@host ~]$ gdb exemplo
```

Se tudo estiver correto, o gdb informará na tela uma série de mensagens a respeito da licença de utilização e então apresentará o prompt "(gdb)" indicando que está pronto para receber um comando.

```
GNU gdb Red Hat Linux (6.6-16.fc7rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)
```

Lembrem-se, ao tentar debugar um programa que não foi compilado com a opção "-g", o GDB irá acusar um erro (os símbolos estão faltando) na sua inicialização:

```
(...)
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb)
```

3. Comandos básicos

Os três primeiros comandos mais simples a serem aprendidos são o run, kill e quit.

run: indica ao gdb que a execução do seu programa deve ser iniciada. Quando o gdb é aberto, o seu programa ainda não está executando até que você indique isso.

kill: este comando força o término da execução do seu programa. O gdb vai perguntar se você tem certeza de que quer fazer isso.

quit: para sair em definitivo do gdb, utilize o comando quit. Se o programa ainda estiver em execução, o gdb irá perguntar se você tem certeza de que quer sair.

Se o comando "run" for executado imediatamente após a inicialização do gdb, o programa será executado até o fim, sem ser interrompido, como se o gdb não estivesse supervisionando-o.

Isso é útil nos casos em que o programa termina abruptamente, pois o gdb pode nos dar algumas informações a respeito da causa do término. Contudo, na maioria das vezes estaremos interessados em observar o programa sendo executado aos poucos. Para isso, usaremos os seguintes comandos:

break <lugar>: insere um 'breakpoint' (ponto de parada) no programa. Pode-se usar 'break funcao' para parar no início de uma função específica, ou 'break n', para parar numa linha "n" específica.

step <n>: executa a linha atual e passa para a próxima linha. Se uma função for encontrada, cada linha dela será executada também. Pode ser acompanhado de um argumento "n" que indica quantas linhas devem ser executadas (o padrão é 1).

next <n>: funciona como step, exceto que as chamadas de função são tratadas como se fossem um comando só (não 'entra' na chamada de função). Pode ser acompanhado de um argumento "n" que indica quantas linhas devem ser executadas (o padrão é 1).

Com estes comandos podemos controlar a execução do programa. Para códigos muito pequenos, como o do programa de exemplo "basico.c", normalmente vamos incluir um breakpoint na função "main" se não tivermos idéia de por onde começar o debug.

O último comando básico que precisamos para fazer o debug da maioria dos problemas simples é o de impressão:

> display <expressão>: imprime o valor atual da variável passada como argumento. Se uma expressão for passada como argumento, o resultado será impresso. Se não houver argumentos, todos os resultados impressos com display serão sumarizados. As expressões serão armazenadas na lista de exibição.
>
> undisplay <n>: remove a entrada "n" da lista de exibição.
>
> print <expressão>: imprime o valor de uma variável ou expressão sem adicioná-los à lista de exibição.

Lembrem-se de que variáveis que ainda não foram inicializadas terão valores esquisitos, por causa do "lixo" que estava na memória. Podemos agora observar um pequeno exemplo ilustrativo para treinarmos os comandos aprendidos.

4. Exemplo prático

Vamos compilar o programa de exemplo "basico.c" com instrumentação para debug.

```
[ra123456@host ~]$ gcc basico.c -o exemplo -g
```

Podemos executar o programa inicialmente para observar o seu resultado:

```
[ra123456@host ~]$ ./exemplo
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9)
(1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (1, 9)
(2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9)
(3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8) (3, 9)
(4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8) (4, 9)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7) (5, 8) (5, 9)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7) (6, 8) (6, 9)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 7) (7, 8) (7, 9)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8) (8, 9)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (9, 9)
[ra123456@host ~]$
```

Vamos inspecionar o funcionamento do programa utilizando o gdb:

```
[ra123456@host ~]$ gdb exemplo
```

Após a inicialização, vamos inserir o primeiro breakpoint na função main:

```
(gdb) break main
Breakpoint 1 at 0x8048405: file basico.c, line 6.
```

O gdb indica qual o endereço da função na memória em hexadecimal (0x8048405), o arquivo fonte no qual ela foi escrita (basico.c) e a primeira linha da função que contém algum comando (linha 6). Vamos executar o programa com "run":

```
(gdb) run
Starting program: /home/cc2008/ra123456/tutorial_gdb/exemplo
```

```
Breakpoint 1, main () at basico.c:6
6         for( i = 0; i < 10; i++ ) {
```

Agora o gdb indica que atingimos um breakpoint localizado na linha 6 do arquivo basico.c. O número da linha e o comando ao qual ela corresponde são apresentados. Notem que esta linha será a próxima do programa (ela ainda NÃO foi executada).

Vamos executar apenas esta linha, com o comando step:

```
(gdb) step
7           for( j = 0; j < 10; j++ ) {
```

A inicialização i=0 da linha 6 foi feita. O teste i<10 foi verdadeiro e, portanto, estamos agora na linha 7 do programa. Podemos conferir o valor da variável "i" com o comando display:

```
(gdb) display i
1: i = 0
```

Cada vez que o comando display é executado, um número de contagem é exibido para ordenar o resultado. Vamos solicitar a impressão de uma expressão simples:

```
(gdb) display i + 4
2: i + 4 = 4
```

Agora podemos solicitar a sumarização dos resultados impressos até agora:

```
(gdb) display
2: i + 4 = 4
1: i = 0
```

Notem que este é um recurso muito útil para acompanharmos o progresso dos valores enquanto o programa é executado. Vamos prosseguir:

```
(gdb) step
8             printf( "(%d, %d) ", i, j );
2: i + 4 = 4
1: i = 0
```

Cada vez que o gdb imprimir seu status, a lista de exibição será impressa também. O printf que será executado deve imprimir um par ordenado "(i, j)" na tela. Vamos adicionar a variável "j" na lista e remover a expressão i + 4 (índice 2).

```
(gdb) display j
3: j = 0
(gdb) undisplay 2
```

Agora podemos executar a próxima linha (printf) e acompanharmos os resultados:

```
(gdb) step
7           for( j = 0; j < 10; j++ ) {
3: j = 0
1: i = 0
```

Aconteceram algumas coisas notáveis: (1) após o printf, retornamos à linha 7, porque temos que executar o comando de repetição (i++) e testar se a condição de repetição (j < 10) ainda é válida; (2) apesar de um printf ter ocorrido, nada foi impresso na tela. Isso ocorre porque há um buffer de impressão, que acumula vários caracteres até que surja uma quebra de linha (um caractere "\n").

Vamos prosseguir com mais um step:

```
(gdb) step
8             printf( "(%d, %d) ", i, j );
3: j = 1
1: i = 0
```

O comando de repetição fez j se tornar 1, a condição de repetição foi satisfeita e portanto vamos novamente para a linha do printf. Para evitarmos repetir cada um dos printfs, vamos avançar um pouco com o step:

```
(gdb) step 16
8             printf( "(%d, %d) ", i, j );
3: j = 9
1: i = 0
```

Executamos 16 instruções, ou seja, 8 "printf" e 8 vezes a repetição do "for". Neste momento, o j vale 9. Vejamos as próximas instruções executadas:

```
(gdb) step
7           for( j = 0; j < 10; j++ ) {
3: j = 9
1: i = 0
```

Lembre-se que a linha indicada pelo gdb é a próxima a ser executada. Portanto, o j será incrementado em 1 e então a condição j < 10 será testada (e vai falhar). Assim, o programa não vai mais repetir este "for" e passará para o próximo comando.

```
(gdb) step
10          printf( "\n" );
3: j = 10
1: i = 0
```

Observem que o j vale 10 (como previsto) e não repetimos o "for". Agora estamos prestes a observar a impressão de uma quebra de linha:

```
(gdb) step
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9)
6         for( i = 0; i < 10; i++ ) {
3: j = 10
1: i = 0
```

A impressão da quebra de linha fez com que o buffer fosse transferido para a tela. Todos os pares ordenados que haviam sido "impressos" estão agora evidentes. Eles representam os valores que as variáveis i e j assumiram até agora.

Voltamos à linha 6 para a repetição do loop mais externo. Observem o que acontecerá a seguir:

```
(gdb) step
7           for( j = 0; j < 10; j++ ) {
3: j = 10
1: i = 1
```

A repetição do loop foi executada e a condição de repetição (i < 10) foi satisfeita. Notem que o i passou a valer 1, mas o j ainda vale 10 porque a linha 7 do programa ainda não foi executada.

```
(gdb) step
8             printf( "(%d, %d) ", i, j );
3: j = 0
1: i = 1
```

Agora a inicialização foi bem-sucedida, o teste de repetição estava válido (j < 10 com j valendo 0) e entramos novamente no loop mais interno. Podemos perceber

que agora acontecerá algo parecido com o que tivemos antes:

```
(gdb) step 18
8             printf( "(%d, %d) ", i, j );
3: j = 9
1: i = 1
```

Com o comando acima pulamos as 18 próximas instruções (9 printf e 9 repetições do "for").

```
(gdb) step
7           for( j = 0; j < 10; j++ ) {
3: j = 9
1: i = 1
```

A repetição do "for" acima vai falhar porque j não terá mais um valor menor que 10.

```
(gdb) step
10          printf( "\n" );
3: j = 10
1: i = 1
```

Esta quebra de linha fará com que o buffer de impressão seja colocado na tela:

```
(gdb) step
(1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (1, 8) (1, 9)
6         for( i = 0; i < 10; i++ ) {
3: j = 10
1: i = 1
```

Estamos de novo na linha 6, para tentar repetir o loop mais externo. Como sabemos que a repetição será bem-sucedida, podemos saltar 23 instruções (1 repetição do "for" externo + 10 repetições do "for" interno bem-sucedidas + 10 "printf" de pares ordenados + 1 repetição do for interno que falha + 1 "printf" que imprime uma quebra de linha).

```
(gdb) step 23
(2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9)
6         for( i = 0; i < 10; i++ ) {
3: j = 10
1: i = 2
```

Percebam que o loop mais externo engloba um ciclo de repetição de 23 instruções. Podemos acompanhar as voltas do loop através da observação da variável i. Se continuarmos saltando mais 23 instruções, teremos outra volta deste loop:

```
(gdb) step 23
(3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (3, 5) (3, 6) (3, 7) (3, 8) (3, 9)
6         for( i = 0; i < 10; i++ ) {
3: j = 10
1: i = 3
```

Vamos agora saltar 6 voltas completas do loop externo, ou seja 6 x 23 = 138 instruções:

```
(gdb) step 138
(4, 0) (4, 1) (4, 2) (4, 3) (4, 4) (4, 5) (4, 6) (4, 7) (4, 8) (4, 9)
(5, 0) (5, 1) (5, 2) (5, 3) (5, 4) (5, 5) (5, 6) (5, 7) (5, 8) (5, 9)
(6, 0) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7) (6, 8) (6, 9)
(7, 0) (7, 1) (7, 2) (7, 3) (7, 4) (7, 5) (7, 6) (7, 7) (7, 8) (7, 9)
(8, 0) (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8) (8, 9)
(9, 0) (9, 1) (9, 2) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (9, 9)
6         for( i = 0; i < 10; i++ ) {
3: j = 10
```

```
1: i = 9
```

Percebam que este é um momento interessante, porque a variável "i" será incrementada e finalmente o teste i < 10 vai falhar:

```
(gdb) step
13        return 0;
3: j = 10
1: i = 10
```

Prosseguimos para a próxima linha após o loop externo, que é o retorno do programa.

```
(gdb) step
15      }
3: j = 10
1: i = 10
```

A partir de agora o gdb irá revelar um pouco do mecanismo que envolve o valor de retorno do programa para quem o chamou:

```
(gdb) step
0x00126f70 in __libc_start_main () from /lib/libc.so.6
(gdb) step
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
```

Pronto! A execução do nosso programa foi finalizada com sucesso! Na próxima seção, aprenderemos alguns comandos que nos permitem obter mais controle sobre a execução do programa para que não tenhamos que calcular quantas instruções precisamos saltar com o "step" quando desejamos chegar a um certo ponto da execução. Para finalizarmos o gdb, usamos o quit:

```
(gdb) quit
```

5. Comandos intermediários

Normalmente, quando queremos que o programa siga o seu curso até chegar a um ponto em que temos interesse, utilizamos breakpoints e watchpoints ao invés de calcularmos quantas instruções precisamos saltar. Eis os comandos que manipulam tais pontos:

continue: executa o programa até que algum breakpoint seja alcançado.

break <lugar> if <condicao>: podemos inserir breakpoints condicionais ao colocarmos a cláusula "if <condicao>". Isso significa que o breakpoint só será considerado caso a condição seja verdadeira.

watch <expressão>: insere a expressão na "lista de vigília". Se o valor desta expressão mudar, a execução é interrompida.

info breakpoints: exibe os breakpoints e watchpoints inseridos até agora, juntamente com o seu índice.

info watchpoints: o mesmo que "info breakpoints"

delete <n>: remove o breakpoint/watchpoint de índice "n" da lista.

Podemos treinar alguns destes comandos no exemplo anterior (basico.c). Execute

os comandos na ordem dada e observe os resultados:

Primeira sugestão:

```
[ra123456@host ~]$ gdb exemplo
(gdb) break main
(gdb) break 7 if i==6
(gdb) run
(gdb) display i
(gdb) display j
(gdb) continue
(gdb) continue
(gdb) quit
```

Segunda sugestão:

```
[ra123456@host ~]$ gdb exemplo
(gdb) break main
(gdb) run
(gdb) display i
(gdb) watch i
(gdb) continue
(gdb) continue
(gdb) continue
...
(gdb) quit
```

6. Exercício

Comando extra para este exercício:

display/FMT: o comando display pode ser utilizado seguido de uma barra e um formato FMT. Este formato pode ser composto de um número de repetição e uma letra de formato. O número de repetição é usado para indicar quantos itens seguidos da memória devem ser impressos e a letra de formato define como interpretar os dados.

letras de formato: d - decimal f - ponto flutuante c - caractere s - string

Exemplo: Suponha um array de inteiros (int linha[5] = {1,2,3,4,5};). Para imprimir os números do array, utiliza-se o comando:

```
(gdb) display/5d linha
```

Este comando produz o seguinte resultado:

```
1: /d linha = {1, 2, 3, 4, 5}
```

Enunciado do exercício:

O programa intermediario.c está com o seu funcionamento incorreto. O vetor de números {5,3,7,6,1,2,10,9,4,8} deveria ser ordenado crescentemente e então impresso na tela, porém o resultado está longe do esperado.

Compile o programa com a flag de debug, execute-o e observe o resultado. Então, utilize o gdb para tentar descobrir onde está o erro e em seguida corrija-o.

Elabore uma breve explicação de quais comandos você utilizou e como descobriu o

```
erro.
```