Depuração de programas

Marcelo Ferreira Siqueira

1 Nota sobre autoria

O texto que segue abaixo foi retirado, quase inteiramente, das notas de aula do professor Fábio Henrique Viduani Martinez, do Departamento de Computação e Estatística (DCT) da Universidade Federal de Mato Grosso do Sul (UFMS). As modificações foram praticamente marginais.

2 Introdução

À medida que temos resolvido problemas maiores e mais complexos, natural e conseqüentemente nossos programas têm ganhado complexidade e extensão. Uso correto de estruturas de programação, identificadores de variáveis significativos, indentação, comentários, tudo isso tem sido usado na tentativa de escrever programas corretamente. Ou seja, nosso desejo é sempre escrever programas eficientes, coesos e corretos.

Infelizmente, nem sempre isso é possível. Por falta de atenção, de experiência, de disciplina, ou mesmo pela complexidade do problema que precisamos resolver, não é incomum o desenvolvimento de um programa que contém erros. Nesta aula, aprenderemos a usar um poderoso depurador de programas interativo, chamado GDB, como aliado na construção de programas corretos. Veremos algumas de suas principais características, talvez as mais úteis. Interessados e interessadas em mais informações e características avançadas desta ferramenta devem consultar a página e o manual da mesma.

3 Depurador GDB

Um **depurador** é um programa que permite a um programador visualizar o que acontece no interior de um outro programa durante sua execução. Neste sentido, podemos dizer que um depurador é uma poderosa ferramenta de auxílio à programação. O depurador GDB, do Projeto GNU, projetado inicialmente por Richard Stallman[1], é usado com freqüência para depurar programas compilados com o compilador GCC, também do projeto GNU. Para auxiliar um programador na procura por erros em um programa, o GDB pode agir de uma das quatro principais formas abaixo:

- iniciar a execução de um programa, especificando qualquer coisa que possa afetar seu comportamento;

[1] Richard Matthew Stallman (1953−), nascido nos Estados Unidos, físico, ativista político e ativista de software.

- interromper a execução de um programa sob condições estabelecidas;
- examinar o que aconteceu quando a execução do programa foi interrompida;
- modificar algo no programa, permitindo que o programador corrija um erro e possa continuar a investigar outros erros no mesmo programa.

O depurador GDB pode ser usado para depurar programas escritos nas linguagens de programação C, C++, Objective-C, Modula-2, Pascal e Fortran.

GDB é um *software livre*, protegido pela Licença Pública Geral (GPL) da GNU. A GPL dá a seu usuário a liberdade para copiar ou adaptar um programa licenciado, mas qualquer pessoa que faz uma cópia também deve ter a liberdade de modificar aquela cópia – o que significa que deve ter acesso ao código fonte –, e a liberdade de distribuir essas cópias. Empresas de software típicas usam o *copyright* para limitar a liberdade dos usuários; a Fundação para o Software Livre (*Free Software Foundation*) usa a GPL para preservar essas liberdades. Fundamentalmente, a Licença Pública Geral (GPL) é uma licença que diz que todas as pessoas têm essas liberdades e que ninguém pode restringi-las.

Para executar um programa com auxílio do depurador GDB, o programa fonte na linguagem C deve ser compilado com o compilador GCC usando a opção `-g`. Essa diretiva de compilação faz com que o compilador adicione informações extras no programa executável.

4 Primeiro contato

De uma janela de comandos, você pode chamar o depurador GDB, como a seguir:

```
$ gdb
GNU gdb 6.1-20040303 (Apple version gdb-437) (Sun Dec 25 08:31:29 GMT 2005)
Copyright 2004 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 "powerpc-apple-darwin".
(gdb)
```

Em geral, quando queremos depurar um programa, digitamos o comando `gdb` *`programa`* em uma janela de comandos, onde *`programa`* é um programa executável compilado e gerado pelo compilador `gcc` usando a diretiva de compilação `-g`. Por exemplo, se queremos depurar o programa `consecutivos.cpp`, devemos primeiro compilar o programa com a diretiva `-g`,

```
g++ -Wall -ansi -pedantic -g consecutivos.cpp -o consecutivos
```

e, em seguida, executar o depurador GDB com o programa `consecutivos` como argumento:

```
$ gdb consecutivos
GNU gdb 6.1-20040303 (Apple version gdb-437) (Sun Dec 25 08:31:29 GMT 2005)
Copyright 2004 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 "powerpc-apple-darwin"...
Reading symbols for shared libraries ....
done

(gdb)
```

A linha de comando

```
$ gdb consecutivos
```

faz a chamada do depurador GDB com a carga do programa executável `consecutivos`, que foi previamente gerado pelo processo de compilação, usando o compilador GCC com a diretiva de compilação `-g`. A opção `-g` obriga o GCC a inserir a tabela de símbolos[2] do processo de compilação no executável.

Para sair do GDB e retornar à janela de comandos basta digitar o comando `quit`, como a seguir:

```
(gdb) quit
$
```

5 Sintaxe dos comandos do GDB

Um comando do GDB é composto por uma linha de entrada, contendo o nome do comando seguido de zero ou mais argumentos. O nome de um comando do GDB pode ser truncado, caso essa abreviação não seja ambígüa. O usuário também pode usar a tecla `Tab` para fazer com que o GDB preencha o resto do nome do comando ou para mostrar as alternativas disponíveis, caso exista mais que uma possibilidade. Uma linha em branco como entrada significa a solicitação para que o GDB repita o último comando fornecido pelo usuário.

Podemos solicitar ao GDB informações sobre os seus comandos usando o comando `help`. Através do comando `help`, que pode ser abreviado por `h`, sem argumentos, obtemos uma pequena lista das classes de comandos do GDB:

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
```

[2] Você verá a definição de tabela de símbolos na disciplina Compiladores.

```
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Usando o nome de uma das classes gerais de ajuda como um argumento do comando `help`, podemos obter uma lista dos comandos desta classe. Por exemplo,

```
(gdb) help status
Status inquiries.

List of commands:

info -- Generic command for showing things about the program being debugged
macro -- Prefix for commands dealing with C preprocessor macros
show -- Generic command for showing things about the debugger

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Também podemos usar o comando `help` diretamente com o nome de um comando que queremos ajuda. Por exemplo,

```
(gdb) help help
Print list of commands.
(gdb)
```

Para colocar um programa executável sob execução no GDB é necessário usar o comando `run`, que pode ser abreviado por `r`. Este programa executável deve ter sido usado como um parâmetro na chamada do depurador ou então deve ser carregado no GDB usando o comando `file` ou `exec-file`.

6 Pontos de parada

Uma das principais vantagens no uso de depuradores de programas é a possibilidade de interromper a execução de um programa antes de seu término. Sob o GDB, um programa pode ser interrompido intencionalmente por diversas razões. Neste ponto de interrupção, podemos examinar e modificar o conteúdo de variáveis, criar outros pontos de parada, ou remover antigos, e continuar a execução do programa. Usualmente, as mensagens emitidas pelo GDB fornecem explicações satisfatórias sobre o estado do programa, mas podemos também solicitar essas informações explicitamente a qualquer momento, através do comando `info program`. Esse comando apresenta o estado do programa, sua identificação de processo, se está sob execução ou não e, neste último caso, os motivos pelos quais sua execução foi interrompida.

Um **ponto de parada** interrompe a execução de um programa sempre que um determinado ponto do código fonte é alcançado e/ou uma determinada condição é verificada. No GDB existem três tipos de pontos de parada: *breakpoints*, *watchpoints* e *catchpoints*. Nesta aula, aprenderemos a usar dois deles (os dois primeiros).

O GDB atribui um número para cada ponto de parada quando da sua criação. Estes números são números inteiros sucessivos começando com 1. Em diversos comandos para controle de

pontos de parada, podemos usar seu número seqüencial para referenciá-lo. Assim que é criado, um ponto de parada está habilitado. No entanto, durante a execução de um programa, podemos desabilitá-lo de acordo com nossa conveniência.

Um *breakpoint* é um ponto de parada que interrompe a execução de um programa em um dado ponto especificado do código fonte. Seu uso mais freqüente é com o argumento do ponto de execução como sendo o número da linha do código fonte. Por exemplo,

```
breakpoint 13
```

estabelece um ponto de parada na linha 13 do programa. O ponto de parada interromperá o programa antes da execução do código localizado na linha especificada.

O comando `breakpoint` pode ser abreviado por `br`.

Outra forma de uso de um *breakpoint* é através de sua associação a uma expressão lógica, estabelecendo um ponto de parada em uma dada localização do código fonte, caso uma determinada condição seja satisfeita. Por exemplo,

```
breakpoint 22 if  ant == prox
```

estabelece um ponto de parada na linha 22, caso a expressão lógica descrita após a palavra reservada `if` seja avaliada com valor verdadeiro. O ponto de parada interromperá então o programa antes da execução do código na linha 22. Além disso, um *breakpoint* também pode ser criado através do comando

```
breakpoint identificador
```

onde o argumento `identificador` é o nome de uma função.

Podemos também usar um ponto de parada na execução de um programa que suspende sua execução sempre que o valor de uma determinada expressão se modifica, sem ter de predizer um ponto particular no código fonte onde isso acontece. Esse ponto de parada é chamado de *watchpoint* do GDB. Por exemplo,

```
watch  iguais
```

estabelece um ponto de parada que irá suspender a execução do programa quando o valor da expressão descrita – que no caso contém apenas a variável `iguais` – for modificado.

O comando `watch` pode ser abreviado por `wa`.

Como já mencionado, o GDB atribui um número seqüencial, iniciando com 1, para cada ponto de parada, quando da sua criação. Podemos imprimir uma tabela com informações básicas sobre os pontos de parada associados ao programa em execução com o comando `info breakpoints [`*n*`]`, que pode ser abreviado por `info break [`*n*`]`. O argumento opcional *n* mostra as informações apenas do ponto de parada especificado. Por exemplo,

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483f5 in main at consecutivos.c:13
        breakpoint already hit 1 time
2   breakpoint     keep y   0x0804848e in main at consecutivos.c:22
        stop only if ant == prox
3   hw watchpoint  keep y              iguais

(gdb)
```

Muitas vezes, queremos eliminar um ponto de parada que já cumpriu o seu papel na depuração do programa e não queremos mais que o programa seja interrompido naquele ponto. Podemos usar os comandos `clear` ou `delete` para remover um ou mais pontos de parada. Os comandos `clear` e `delete`, com um argumento numérico, removem um ponto de parada especificado. Por exemplo, os comandos `clear 1` e `delete 2` removem os pontos de parada identificados pelos números 1 e 2, respectivamente. A diferença entre estes comandos se dá quando são usados sem argumentos: o comando `clear` remove o ponto de parada da linha de código em execução, se existir algum, e o comando `delete` sem argumentos remove todos os pontos de parada existentes no programa.

Ao invés de remover definitivamente um ponto de parada, pode ser útil apenas desabilitá-lo temporariamente e, se for necessário, reabilitá-lo posteriormente. O comando `disable` desabilita todos os pontos de parada de um programa. Podemos desabilitar um ponto de parada específico através do uso de seu número como argumento do comando `disable`. Por exemplo, `disable 3` desabilita o ponto de parada de número 3. Podemos também reabilitar um ponto de parada através do comando `enable`, que reabilita todos os pontos de parada desabilitados, ou `enable` *n*, que reabilita o ponto de parada de número *n*.

Após verificar o que aconteceu com o programa em um dado ponto de parada, podemos retomar a execução do programa a partir deste ponto de dois modos diferentes: através do comando `continue`, que pode ser abreviado por `c`, ou através do comando `step`, que pode ser abreviado por `s`. No primeiro comando, a execução é retomada e segue até o final do programa ou até um outro ponto de parada ser encontrado. No segundo, apenas a próxima linha do programa é executada e o controle do programa é novamente devolvido ao GDB. O comando `step` pode vir seguido por um argumento numérico, indicando quantas linhas do código fonte queremos progredir na sua execução. Por exemplo, `step 5` indica que queremos executar as próximas 5 linhas do código. Caso um ponto de parada seja encontrado antes disso, o programa é interrompido e o controle repassado ao GDB.

Comumente, adicionamos um ponto de parada no início de uma função onde acreditamos que exista um problema, um erro, executamos o programa até que ele atinja este ponto de parada e então usamos o comando `step` nesta área suspeita, examinando o conteúdo das variáveis envolvidas, até que o problema se revele.

7 Programa fonte

O GDB pode imprimir partes do código fonte do programa sendo depurado. Quando um programa tem sua execução interrompida por algum evento, o GDB mostra automaticamente o conteúdo da linha do código fonte onde o programa parou. Para ver outras porções do código, podemos executar comandos específicos do GDB.

O comando `list` mostra ao programador algumas linhas do código fonte carregado previamente. Em geral, mostra 10 linhas em torno do ponto onde a execução do programa se encontra. Podemos explicitamente solicitar ao GDB que mostre um intervalo de linhas do código fonte, como por exemplo, `list 1,24`. Neste caso, as primeiras 24 linhas do código fonte do programa carregado no GDB serão mostradas.

```
(gdb) list 1,24
1       /**************************************************************************
2        *
```

```
3         * consecutivos.cpp
4         *
5         * Verifica se um dado número inteiro positivo tem dois dígitos
6         * consecutivos iguais.
7         *
8         ************************************************************************/
9
10        #include <iostream>
11
12        int main(void)
13        {
14          using std::cout ;
15          using std::endl ;
16          using std::cin ;
17
18          int num ;
19          int ant ;
20          int prox ;
21          bool iguais ;
22
23          cout << endl << "Informe um número: " ;
24          cin >> num ;
(gdb)
```

8 Verificação de dados

Uma forma usual de examinar informações em nosso programa é através do comando `print`, que pode ser abreviado por `p`. Este comando avalia um expressão e mostra seu resultado. Por exemplo, `print 2*a+3` mostra o resultado da avaliação da expressão `2*a+3`, supondo que `a` é uma variável de um dos tipos básicos numéricos do nosso programa. Sem argumento, esse comando mostra o último resultado apresentado pelo GDB.

Uma facilidade que o GDB nos oferece é mostrar o valor de uma expressão freqüentemente. Dessa forma, podemos verificar como essa expressão se comporta durante a execução do programa. Podemos criar uma **lista de impressão automática** que o GDB mostra cada vez que o programa tem sua execução interrompida. Podemos usar o comando `display` para adicionar uma expressão à lista de impressão automática de um programa. Por exemplo, `display iguais` adiciona a variável `iguais` a esta lista e faz com que o GDB imprima o conteúdo desta variável sempre que a execução do programa for interrompida por algum evento. Aos elementos inseridos nesta lista são atribuídos números inteiros consecutivos, iniciando com 1. Para remover uma expressão desta lista, basta executar o comando `delete display` *n*, onde *n* é o número da expressão que desejamos remover. Para visualizar todas as expressões nesta lista é necessário executar o comando `info display`.

Todas as variáveis internas de uma função, e seus conteúdos, podem ser visualizados através do comando

```
info locals
```

9 Alteração de dados durante a execução

Podemos ainda alterar conteúdos de variáveis durante a execução de um programa no GDB. Em qualquer momento da execução, podemos usar o comando `set var` para alterar o conteúdo de uma variável. Por exemplo, `set var x=2` modifica o conteúdo da variável `x` para 2. O lado direito após o símbolo = no comando `set var` pode conter uma constante, uma variável ou uma expressão válida. Esta característica do GDB permite que um programador possa alterar os conteúdos de variáveis durante a execução de um programa e, com isso, possa controlar seu fluxo de execução e corrigir possíveis erros.

10 Resumo dos comandos

Comando	Significado
PROGRAMA FONTE	
list [*n*]	Mostra linhas em torno da linha *n* ou as próximas 10 linhas, se não especificada
list *m*,*n*	Mostra linhas entre *m* e *n*
VARIÁVEIS E EXPRESSÕES	
print *expr*	Imprime *expr*
display *expr*	Adiciona *expr* à lista de impressão automática
info display	Mostra a lista de impressão automática
delete display *n*	Remove a expressão de número *n* da lista de impressão automática
info locals	Mostra o conteúdo de todas as variáveis locais na função corrente
set var *var=expr*	Atribui *expr* para a variável *var*
PONTOS DE PARADA	
break *n*	Estabelece um *breakpoint* na linha *n*
break *n* if *expr*	Estabelece um *breakpoint* na linha *n* se o valor de *expr* for verdadeiro
break *func*	Estabelece um *breakpoint* no início da função *func*
watch *expr*	Estabelece um *watchpoint* na expressão *expr*
clear [*n*]	Remove o ponto de parada na linha *n* ou na próxima linha, se não especificada
delete [*n*]	Remove o ponto de parada de número *n* ou todos os pontos de parada, se não especificado
enable [*n*]	Habilita todos os pontos de parada suspensos ou o ponto de parada *n*
disable [*n*]	Desabilita todos os pontos de parada, ou o ponto de parada *n*
info break	Mostra todos os pontos de parada
EXECUÇÃO DO PROGRAMA	
file [*prog*]	Carrega o programa executável *prog* e sua tabela de símbolos ou libera o GDB da execução corrente, se não especificado
exec-file [*prog*]	Carrega o programa executável *prog* ou libera o GDB da execução corrente, se não especificado
run	Inicia a execução do programa
continue	Continua a execução do programa
step [*n*]	Executa a próxima linha do programa ou as próximas *n* linhas
info program	Mostra o estado da execução do programa
quit	Encerra a execução do GDB
AJUDA	
help	Mostra as classes de ajuda de comandos do GDB
help *classe*	Mostra uma ajuda sobre a *classe* de comandos
help *comando*	Mostra uma ajuda sobre o *comando*

11 Exemplos de execução

Vamos usar o GDB com o programa `consecutivos`, que verifica se um dado número inteiro positivo n possui dois dígitos consecutivos iguais.

Programa 11.1: Código fonte do programa `consecutivos`.

```
/*********************************************************************
*
* consecutivos.cpp
*
* Verifica se um dado número inteiro positivo tem dois dígitos
* consecutivos iguais.
*
*********************************************************************/

#include <iostream>

int main(void)
{
  using std::cout ;
  using std::endl ;
  using std::cin ;

  int num ;
  int ant ;
  int prox ;
  bool iguais ;

  cout << endl << "Informe um número: " ;
  cin >> num ;

  prox = num % 10 ;
  iguais = false ;
  while( ( num != 0 ) && ( !iguais ) ) {
    num = num / 10 ;
    ant = prox ;
    prox = num % 10 ;
    if ( ant == prox ) {
      iguais = true ;
    }
  }

  if ( iguais ) {
    cout << "Número tem dois dígitos consecutivos iguais." << endl << endl ;
  }
  else {
    cout << "Número não tem dois dígitos consecutivos iguais." << endl << endl ;
  }

  return 0;
}
```

Iniciamos o processo compilando o programa 11.1 com a diretiva `-g`:

```
$ gcc -Wall -ansi -pedantic -g consecutivos.cpp -o consecutivos
```

Depois, carregamos o GDB com o programa executável gerado:

```
$ gdb consecutivos
GNU gdb 6.1-20040303 (Apple version gdb-437) (Sun Dec 25 08:31:29 GMT 2005)
Copyright 2004 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 "powerpc-apple-darwin"...
Reading symbols for shared libraries ....
done

(gdb)
```

Para verificar se o programa foi de fato carregado na chamada do GDB, podemos usar o comando `list` para ver seu código fonte ou então o comando `info program`, para visualizar o estado do programa:

```
(gdb) list 10,45
10      #include <iostream>
11
12      int main(void)
13      {
14        using std::cout ;
15        using std::endl ;
16        using std::cin ;
17
18        int num ;
19        int ant ;
20        int prox ;
21        bool iguais ;
22
23        cout << endl << "Informe um número: " ;
24        cin >> num ;
25
26        prox = num % 10 ;
27        iguais = false ;
28        while( ( num != 0 ) && ( !iguais ) ) {
29          num = num / 10 ;
30          ant = prox ;
31          prox = num % 10 ;
32          if ( ant == prox ) {
33            iguais = true ;
34          }
35        }
36
37        if ( iguais ) {
38          cout << "Número tem dois dígitos consecutivos iguais." << endl
                                                                  << endl ;
39        }
40        else {
41          cout << "Número não tem dois dígitos consecutivos iguais." << endl
                                                                    << endl ;
42        }
43
44        return 0;
45      }
(gdb)
```

ou

```
(gdb) info program
The program being debugged is not being run.
(gdb)
```

O programa pode ser executado neste momento, através do comando `run`:

```
(gdb) run
Starting program: /Users/marcelos/workdir/ensino/2009/lab-aed1-1/supl/consecutivos
Reading symbols for shared libraries . done

Informe um número: 12233
Número tem dois dígitos consecutivos iguais.


Program exited normally.
(gdb)
```

A execução acima foi realizada *dentro* do GDB, sem a adição de qualquer ponto de parada. Como o programa ainda continua carregado no GDB, podemos executá-lo quantas vezes quisermos. Antes de iniciar a próxima execução, vamos adicionar alguns pontos de parada no programa:

```
(gdb) break 23
Breakpoint 1 at 0x28d4: file consecutivos.cpp, line 23.
(gdb)
```

Note que não iniciamos a execução do programa ainda. Então,

```
Starting program: /Users/marcelos/workdir/ensino/2009/lab-aed1-1/supl/consecutivos
Reading symbols for shared libraries . done

Breakpoint 1, main () at consecutivos.cpp:23
23        cout << endl << "Informe um número: " ;
(gdb)
```

O ponto de parada estabelecido na linha 23 do programa foi encontrado e o GDB interrompeu a execução do programa neste ponto. Podemos visualizar todas as variáveis, e seus conteúdos, dentro da função `main` usando o comando `info locals`:

```
(gdb) info locals
num = 0
ant = 0
prox = 0
iguais = 2413851396
(gdb)
```

Observe que todas as variáveis declaradas constam da relação apresentada pelo GDB e que o conteúdo de uma delas é um valor "estranho", já que nenhuma dessas variáveis foi inicializada até a linha 23 do programa.

Vamos adicionar mais pontos de parada neste programa, um deles um *breakpoint* associado a uma expressão lógica e o outro um *watchpoint* associado a uma variável:

```
(gdb) break 32 if ant == prox
Breakpoint 2 at 0x2994: file consecutivos.cpp, line 32.
(gdb) watch iguais
Hardware watchpoint 3: iguais
(gdb)
```

Agora, podemos visualizar todos os pontos de parada associados a este programa com o comando `info break`:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x000028d4 in main at consecutivos.cpp:23
        breakpoint already hit 1 time
2   breakpoint     keep y   0x00002994 in main at consecutivos.cpp:32
        stop only if ant == prox
3   hw watchpoint  keep y              iguais
(gdb)
```

Podemos continuar a execução do programa usando o comando `cont`:

```
(gdb) cont
Continuing.
Hardware watchpoint 3: iguais

Old value = 2413851396
New value = false
main () at consecutivos.cpp:28
28        while( ( num != 0 ) && ( !iguais ) ) {
```

Observe que na linha 27 do programa a variável `iguais` é inicializada, substituindo então um valor não válido, um lixo que `iguais` continha após sua declaração, pelo valor `false`. Assim, o GDB interrompe a execução do programa nesta linha, devido ao *watchpoint* de número 3, mostra a mudança de valores que ocorreu nesta variável e mostra também a próxima linha a ser executada.

Vamos então continuar a execução do programa:

```
Continuing.

Breakpoint 2, main () at consecutivos.cpp:32
32          if ( ant == prox ) {
(gdb)
```

O GDB interrompeu a execução do programa devido ao *breakpoint* 2, na linha 32 do programa. Neste ponto, deve ter ocorrido o evento em que os conteúdos das variáveis `ant` e `prox` coincidem. Para verificar se este evento de fato ocorreu, podemos usar o comando `print` para verificar os conteúdos dessas variáveis:

```
(gdb) print ant
$1 = 5
(gdb) print prox
$2 = 5
(gdb)
```

Vamos continuar a execução deste programa:

```
Continuing.
Hardware watchpoint 3: iguais

Old value = false
New value = true
main () at consecutivos.cpp:28
28        while( ( num != 0 ) && ( !iguais ) ) {
(gdb)
```

Observe que o programa foi interrompido pelo ponto de parada de número 3 (*watchpoint*), já que o conteúdo da variável `iguais` foi modificado de `false` para `true`, ou seja, de 'falso' para 'verdadeiro'. Isso ocorreu na linha 33 do programa, última linha do bloco de comandos da estrutura de repetição `while`. Assim, a próxima linha a ser executada é a linha 28, contendo o comando `while` e sua expressão lógica associada, como mostrado pelo GDB.

Vamos dar uma olhada no conteúdo das variáveis da função `main`:

```
(gdb) info locals
num = 5
ant = 5
prox = 5
iguais = true
(gdb)
```

O programa então está quase no fim. Vamos continuar sua execução:

```
(gdb) cont
Continuing.
Hardware watchpoint 3 deleted because the program has left the block
in which its expression is valid.

Watchpoint 3 deleted because the program has left the block in
which its expression is valid.
0x94c6bc80 in std::operator<< <std::char_traits<char> > ()
(gdb)
```

Vamos para mais um exemplo. Agora, vamos executar um programa sob o controle do GDB que não soluciona o problema associado. Ou seja, o programa executável contém algum erro que, à primeira vista, não conseguimos corrigir.

O programa `soma-sufixos.cpp` abaixo calcula a soma dos elementos de todos os "sufixos" de um dado vetor de números inteiros. O programa resolve este problema de forma "inocente", usando uma função, denominada `soma_sufixos`, com complexidade de tempo quadrática no número de elementos do vetor de entrada.

```
/*********************************************************************
*
* soma-sufixos.cpp
*
* Calcula a  soma dos "sufixos"  de um dado  vetor com até  10 números
* inteiros usando um algoritmo  de complexidade quadrática (o mesmo da
* avaliação 1 de AED1).
*
*********************************************************************/

#include <iostream>

void soma_sufixos( int , int[] , int[] ) ;

int main()
{
  using std::cout ;
  using std::endl ;
  using std::cin ;

  int a[ 10 ] ;
```

```
  int n ;

  // Entrada de dados
  cout << endl << "Informe o tamanho do vetor: " ;
  cin >> n ;

  int i ;
  for( i = 0 ; i < n ; i++ ) {
    cout << endl << "Informe o elemento de índice " << i << " do vetor: " ;
    cin >> a[ i ] ;
  }

  // Cálculo da soma de sufixos
  int b[ 10 ] ;
  soma_sufixos( n , a , b ) ;

  // Saída de dados
  for( i = n - 1 ; i >= 0 ; i-- ) {
    cout << endl
 << "A soma dos elementos de índice "
 << i
 << " até "
 << n - 1
 << " do vetor de entrada é: "
 << b[ i ]
        << endl ;
  }

  cout << endl ;

  return 0;
}


void soma_sufixos( int n , int a[] , int b[] )
{
  int i ;
  for ( i = n - 1 ; i >= 0 ; i-- ) {
    int j ;
    int soma = 0 ;
    for ( j = i ; j < n ; j-- ) {
      soma += a[ j ] ;
    }
    b[ i ] = soma ;
  }
}
```

Em uma tentativa de execução do programa `soma-sufixos`, ocorre o seguinte erro:

```
$ soma-sufixos

Informe o tamanho do vetor: 6

Informe o elemento de índice 0 do vetor: 1

Informe o elemento de índice 1 do vetor: 2

Informe o elemento de índice 2 do vetor: 3
```

```
Informe o elemento de índice 3 do vetor: 4

Informe o elemento de índice 4 do vetor: 5

Informe o elemento de índice 5 do vetor: 6
Segmentation fault
$
```

A mensagem `Segmentation fault` (em português, `Falha de segmentação`) não nos dá nenhuma dica sobre o *local* em que o erro ocorreu. E se já nos debruçamos sobre o código fonte e não conseguimos encontrá-lo, a melhor idéia é usar um depurador para nos ajudar. Neste exemplo, mostraremos o uso do GDB sem muitas interrupções no texto, a menos que necessárias. Então, segue uma depuração do programa:

```
$ gcc -Wall -ansi -pedantic -g soma_sufixos.cpp -o soma_sufixos
$ gdb soma-sufixos
GNU gdb 6.1-20040303 (Apple version gdb-437) (Sun Dec 25 08:31:29 GMT 2005)
Copyright 2004 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 "powerpc-apple-darwin"...
Reading symbols for shared libraries ....
done

(gdb) list 15,36
15      int main()
16      {
17        using std::cout ;
18        using std::endl ;
19        using std::cin ;
20
21        int a[ 10 ] ;
22        int n ;
23
24        // Entrada de dados
25        cout << endl << "Informe o tamanho do vetor: " ;
26        cin >> n ;
27
28        int i ;
29        for( i = 0 ; i < n ; i++ ) {
30          cout << endl << "Informe o elemento de índice " << i << " do vetor: " ;
31          cin >> a[ i ] ;
32        }
33
34        // Cálculo da soma de sufixos
35        int b[ 10 ] ;
36        soma_sufixos( n , a , b ) ;
(gdb) break soma_sufixos
Breakpoint 1 at 0x2790: file soma-sufixos.cpp, line 59.
(gdb) run
Starting program: /Users/marcelos/workdir/ensino/2009/lab-aed1-1/supl/soma-sufixos
Reading symbols for shared libraries . done

Informe o tamanho do vetor: 6

Informe o elemento de índice 0 do vetor: 1
```

```
Informe o elemento de índice 1 do vetor: 2

Informe o elemento de índice 2 do vetor: 3

Informe o elemento de índice 3 do vetor: 4

Informe o elemento de índice 4 do vetor: 5

Informe o elemento de índice 5 do vetor: 6

Breakpoint 1, soma_sufixos (n=6, a=0xbffff1d0, b=0xbffff1f8) at soma-sufixos.cpp:59
59        for ( i = n - 1 ; i >= 0 ; i-- ) {
(gdb) list 56,67
56      void soma_sufixos( int n , int a[] , int b[] )
57      {
58        int i ;
59        for ( i = n - 1 ; i >= 0 ; i-- ) {
60          int j ;
61          int soma = 0 ;
62          for ( j = i ; j < n ; j-- ) {
63            soma += a[ j ] ;
64          }
65          b[ i ] = soma ;
66        }
67      }
(gdb) step 3
63            soma += a[ j ] ;
(gdb) info locals
j = 5
soma = 0
i = 5
(gdb) display a[j]
1: a[j] = 6
(gdb) display i
2: i = 5
(gdb) display j
3: j = 5
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 5
2: i = 5
1: a[j] = 6
(gdb) step
63            soma += a[ j ] ;
3: j = 4
2: i = 5
1: a[j] = 5
```

Neste ponto, podemos notar algo estranho: o valor da variável `j` é menor do que o valor da variável `i`. Se o programa está prestes a executar a linha 63, isso não deveria ocorrer, pois o laço `for` da linha 62 deve somar os elementos do vetor `a` com índices que variam de `i` até `n-1`. O que fez com que `j` se tornasse menor do que `i` durante a execução do laço? Bem, o valor da variável `j` é modificado apenas na linha 62 do programa

```
62          for ( j = i ; j < n ; j-- ) {
```

Agora, podemos perceber que a variável `j` está sendo *decrementada* ao invés de ser *incrementada*. Certamente, esta não era nossa intenção, pois o fato de `j` ser decrementada faz com que o

teste de parada do laço jamais seja satisfeito. Em outras palavras, nosso programa deveria ter entrada em *loop*, como costumamos falar em linguagem "computês". Mas, por que o programa terminou com aquela mensagem que vimos? Para saber mais sobre isso, vamos continuar executando o programa com o comando `step`, que executa uma linha por vez:

```
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 4
2: i = 5
1: a[j] = 5
(gdb) step
63            soma += a[ j ] ;
3: j = 3
2: i = 5
1: a[j] = 4
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 3
2: i = 5
1: a[j] = 4
(gdb) step
63            soma += a[ j ] ;
3: j = 2
2: i = 5
1: a[j] = 3
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 2
2: i = 5
1: a[j] = 3
(gdb) step
63            soma += a[ j ] ;
3: j = 1
2: i = 5
1: a[j] = 2
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 1
2: i = 5
1: a[j] = 2
(gdb) step
63            soma += a[ j ] ;
3: j = 0
2: i = 5
1: a[j] = 1
(gdb) step
62          for ( j = i ; j < n ; j-- ) {
3: j = 0
2: i = 5
1: a[j] = 1
(gdb) step
63            soma += a[ j ] ;
3: j = -1
2: i = 5
1: a[j] = 6
```

Como podemos verificar acima, o valor de `j` é -1 depois que executamos `step` pela última vez. Mas, não existe elemento no vetor `a` com índice igual a -1. Na verdade, `a[-1]` é o

conteúdo da posição de memória imediatamente anterior à posição de memória do elemento `a[0]` de `a`. Se esta posição não pertencer ao nosso programa, o sistema operacional não permitirá que o programa a acesse e, por isso, nosso programa causará a tal *falha de segmentação*.

É interessante notar que esta falha pode "demorar" a ocorrer; isto é, pode ser que o laço seja repetido muitas vezes antes da falha de segmentação ser comunicada ao nosso programa. Uma forma fácil de verificar isso é usando o comando `cont` para executar o programa até a falha ocorrer:

```
(gdb) cont
Continuing.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xbf7ffffc
0x000027c8 in soma_sufixos (n=6, a=0xbffff1d0, b=0xbffff1f8) at soma-sufixos.cpp:63
63            soma += a[ j ] ;
3: j = -2096245
2: i = 5
1: a[j] = Cannot access memory at address 0xbf7ffffc
Disabling display 1 to avoid infinite recursion.
(gdb)
```

Note que a falha de segmentação só ocorreu depois que, na *minha* execução, o valor de `j` se tornou -2096245. É muito provável que você obtenha um valor diferente quando executar este exemplo em seu computador e até mesmo valores distintos duas ou mais execuções do programa no mesmo computador.