6.12 Depuradores de Alto Nível

Depuradores são programas que dão suporte à tarefa de depuração de outros programas. Existem depuradores de **baixo** e **alto níveis**. Os mais difíceis de ser utilizados são os depuradores de baixo nível que requerem conhecimento de assembly e de como o compilador traduz o código-fonte do programa analisado.

Um depurador de alto nível fornece o meio mais eficiente para determinar a causa de um erro. Mas, o tempo despendido aprendendo a usar um depurador de alto nível deve ser levado em consideração. Muitos depuradores requerem um tempo de aprendizagem considerável.

A maioria dos IDEs vem acompanhada de depuradores de alto nível que são razoavelmente fáceis de utilizar. Apesar de diferirem em algumas facilidades adicionais, estes depuradores têm basicamente o mesmo funcionamento que será descrito a seguir.

Basicamente, depuradores de alto nível permitem que o programador coloque **pontos de parada** (***breakpoints***) ao longo do programa. Estes pontos de parada são utilizados em instruções (i.e., não se pode colocar um ponto de parada numa declaração de tipo, por exemplo). Quando um programa contendo *breakpoints* é executado sob a supervisão de um depurador, a execução do programa pára imediatamente antes da execução da instrução contendo o primeiro *breakpoint*. Neste instante, o programador tem opções de examinar valores de variáveis locais e globais, avaliar expressões, etc.

Enquanto o programa está temporariamente parado num *breakpoint*, o programador pode continuar a execução do programa normalmente até que o próximo *breakpoint* seja atingido ou executar o programa instrução a instrução. Para executar esta última opção, o programador tem duas alternativas. Uma delas é usualmente denominada *step* (ou *step over*) e a outra é (usualmente) denominada *step into*. Ambas as alternativas executam uma única instrução e param a execução do programa antes da próxima instrução. A diferença entre *step* (*over*) e *step into* é que *step* trata chamadas de funções como se fossem instruções indivisíveis, enquanto que *step into* considera uma função como um conjunto de instruções. Portanto, quando a opção *step into* é utilizada imediatamente antes da chamada de uma função, a execução do programa pára imediatamente antes da primeira instrução no corpo da função. Em contraste, quando a opção *step* é utilizada imediatamente antes da chamada de uma função, a execução do programa pára imediatamente antes da instrução seguindo a chamada da função. Quando a próxima instrução a ser executada não é uma chamada de função, não existe diferença entre *step* e *step into*.

Tipicamente, uma sessão de depuração utilizando um depurador de alto nível consiste nas seguintes atividades:

1. Instalar *breakpoints* em locais onde deseja-se que o programa suspenda a execução.

2. Executar o programa passo a passo interrompendo a execução do mesmo em instruções de interesse.
3. Examinar os conteúdos de algumas variáveis de interesse e verificar se estes conteúdos correspondem às expectativas.
4. Modificar os conteúdos de algumas variáveis e repetir os passos precedentes para verificar como o programa se comporta com estas modificações.

Um bom depurador oferece várias opções para executar cada uma das atividades enumeradas acima.

6.13 Depuração Usando gdb

Proficiência em depuração deve fazer parte do conjunto de habilidades de qualquer bom programador. Aqui, será apresentado um exemplo de uma sessão de depuração utilizando o depurador gdb que serve como introdução a esta atividade essencial. Idealmente, esta sessão de depuração deve ser acompanhada num ambiente de laboratório Linux, uma vez que o depurador gdb e o compilador gcc fazem parte de virtualmente qualquer distribuição de Linux. Mas, usuários de Windows também podem usar essas ferramentas de desenvolvimento que acompanham o compilador gcc.

6.13.1 Preparação

Na sessão de depuração ser apresentada aqui, serão utilizados dois arquivos que compõem um programa multiarquivo. Este programa contém bugs intencionais e o estilo utilizado em sua escrita não deve ser imitado. Diferentemente de um programa normal, este programa foi cuidadosamente escrito para produzir erros de modo a demonstrar certas características do gdb. Numa situação normal, você poderá encontrar estes erros facilmente sem o auxílio de nenhum depurador.

A sessão de depuração foi executada utilizando gdb versão 6.4.90-debian na distribuição Linux Ubuntu 6.10 e o programa foi compilador utilizando gcc versão 4.1.2. Todavia, mesmo que você utilize exatamente estes mesmos programas, você poderá obter resultados ligeiramente diferentes daqueles apresentados aqui.

O que o programa usado aqui faz é encontrar o máximo divisor comum (MDC) de dois números inteiros introduzidos pelo usuário. Os conteúdos dos arquivos que compõem o programa são os seguintes[1]:

[1] As linhas desses arquivos foram numeradas para facilitar o acompanhamento da lição.

Arquivo `mdc.c`:

```
 1. /****
 2.  *
 3.  * mdc(): Retorna o mdc de dois numeros inteiros.
 4.  *
 5.  * ALERTA: Este programa contem bugs intencionais!
 6.  *
 7.  ****/
 8. int mdc(int x, int  y)
 9. {
10.    int divisor;
11.
12.    if ( (x < 0) || (y < 0) )
13.       return 0;
14.
15.    divisor = (y < x) ? y : x;
16.
17.    while ( (x%divisor) || (y%divisor) )
18.       divisor--;
19.
20.    return divisor;
21. }
```

Arquivo `main.c`:

```
 1. /****
 2.  *
 3.  * Titulo: depurar
 4.  *
 5.  * Descricao: Este programna foi desenvolvido apenas
 6.  *            para demonstrar o uso do depurador gdb
 7.  *
 8.  * Entrada: Dois valores inteiros
 9.  * Saida: O MDC dos numeros introduzidos
10.  *
11.  * ALERTA: Este programa contem bugs intencionais!
12.  *
13.  * Compilacao com gcc: gcc -g main.c mdc.c -o depurar
14.  *
15.  ****/
16.
17.
18. #include <stdio.h>
19.
20. extern int mdc(int x, int  y);
21.
22. int a, b;
23.
24. int main(void)
25. {
26.    int MDC;
27.
28.    printf("Introduza dois inteiros:");
29.    scanf("%d %d", &a, &b);
```

```
30.
31.     MDC = mdc(a, b);
32.
33.     printf("\nO MDC e': %d\n", MDC);
34.
35.     return 0;
36. }
```

O programa apresentado calcula o MDC de dois números utilizando o seguinte raciocínio:

- O MDC é no máximo igual ao menor dos dois números. Portanto, faz-se o valor inicial da variável `divisor` igual ao menor deles.
- Então, enquanto a variável `divisor` não divide ambos os números, reduz-se seu valor de uma unidade.
- Eventualmente, a variável `divisor` atingirá o valor 1 que é divisor de qualquer número inteiro positivo.

Para acompanhar esta lição sobre o uso do depurador gdb, siga o seguinte procedimento:

1. Faça download do arquivo `depurar.zip` que se encontra no site do livro na internet (v. **Prefácio**). Se, eventualmente, não conseguir fazer download desse arquivo, copie os conteúdos dos arquivos apresentados para dois arquivos com as respectivas denominações.
2. Crie um diretório denominado, por exemplo, `depurar`.
3. Navegue até este diretório.
4. Compile este programa com o compilador gcc, de modo que sejam geradas informações de depuração (v. abaixo).

Para que um programa possa ser depurado usando um depurador, ele deve ter incluído em seu código executável informações que não estão presentes quando o programa é compilado normalmente. A razão para a não inclusão destas informações num programa compilado normalmente é simples: elas fazem com que o programa ocupe mais espaço em memória e o torna mais lento. Portanto, deve-se compilar um programa com essas informações apenas durante a fase de testes e depuração do programa. Passada esta fase, o programa deve ser recompilado normalmente.

Para incluir informações de depuração num programa compilado com o gcc, utilize a opção `-g`. No caso do programa a ser utilizado nesta lição, você deve compilá-lo utilizando o comando:

```
gcc -g main.c mdc.c -o depurar
```

Após compilar o programa desse modo, ele estará pronto para ser examinado com o auxílio de um depurador. Para ver como o programa funciona antes de começar a depurá-lo, digite, no prompt do Linux:

```
./depurar
```

Quando solicitado, introduza dois números, como, por exemplo, 9 e 10. Após introduzir estes números você deparar-se-á com o primeiro erro do programa:

```
Segmentation fault (core dumped)
```

Esta mensagem é emitida pelo sistema operacional Linux e informa que o programa foi abortado devido a uma falha conhecida no mundo Linux como *Segmentation fault*. Este é um erro de execução (v. **Seção 6.10.3**) e existem várias transgressões de um programa que podem gerá-lo, como, por exemplo, tentativa de acesso a uma região de memória proibida. Entre parênteses, o sistema informa ainda que o sistema pode ter feito um despejo de memória (*core dump*, em inglês). Este despejo de memória representa o estado do programa quando o mesmo foi abortado e contém, entre outras coisas, o conteúdo dos registradores e da pilha de execução. Quando este despejo é salvo num arquivo[2], este arquivo é denominado *core* e reside no diretório onde o programa está sendo executado.

Um arquivo *core* contém informações suficientes para reconstituir o estado do programa no instante em que ele foi abortado e é utilizado para depuração do programa que lhe deu origem (v. **Seção 6.13.8**).

Pode parecer paradoxal à primeira vista, mas nenhum depurador localiza ou remove erros de programas. Isto é, na realidade, o que estas ferramentas de programação fazem é auxiliar o programador a localizar e corrigir erros no programa. Como a tarefa de depuração em si pode ser uma tarefa bem complexa (muitas vezes, bem mais complexa do que a própria tarefa de programação), o programa a ser utilizado como objeto de depuração é um programa extremamente simples. Portanto, a tarefa principal aqui não será remover erros, mas sim executar tarefas típicas de depuração, sem a preocupação de encontrar e corrigir erros para que sua atenção não seja desviada do objetivo principal de aprender a utilizar um depurador de alto nível.

---

[2] Este arquivo, denominado *arquivo core* ou *arquivo post-mortem*, pode não ser criado se o sistema que você estiver utilizando não estiver configurado para tal. Esta configuração depende da interface de comandos (*shell*) que você estiver utilizando. Se a *shell* utilizada for *sh* ou *bash* , utilize o comando `ulimit -Sc 200`, onde o número corresponde ao tamanho máximo do arquivo gerado; se você estiver utilizando a *shell csh* ou *tcsh* utilize o comando `limit coredumpsize 200k`.

6.13.2 Executando gdb

Para executar gdb com a finalidade de depurar um programa, navegue até o diretório[3] onde se encontra o programa e utilize o seguinte comando:

```
gdb nome-do-programa
```

No caso presente, digite o seguinte comando no prompt do Linux:

```
gdb depurar
```

Quando este comando é executado, o gdb inicia apresentando uma mensagem de abertura com informações sobre versão, licença, etc. Em seguida, aparece o prompt do gdb:

```
(gdb)
```

6.13.3 Obtendo Ajuda sobre Comandos

Para obter ajuda online sobre os comandos do gdb, digite `h` no prompt do gdb[4]:

```
(gdb) h
```

Observe que o que você obtém é uma lista de classe de comandos e não uma descrição dos comandos em si. Para facilitar o uso da ajuda online, estes comandos são divididos em categorias. Para examinar os comandos em cada categoria, digite `h` seguido do nome da categoria. Uma categoria de comandos essencial é denominada *running* e contém uma lista de todos os comandos necessários para executar o programa sob a supervisão do gdb. Digite o seguinte comando no prompt do gdb para tomar conhecimento desses comandos:

```
(gdb) h running
```

Outra importante categoria de comandos é denominada *breakpoints* e contém uma lista de todos os comandos necessários para instalar diversos tipos de pontos de parada permitidos pelo gdb. Para conhecer estes comandos digite:

```
(gdb) h breakpoints
```

---

[3] O gdb permite que se especifiquem diretórios onde se encontram arquivos necessários para realização de uma sessão de depuração.

[4] Todos os comandos digitados no prompt do gdb devem ser terminados com `[ENTER]`, como ocorre no Linux. Esta informação será considerada redundante daqui por diante e não será mais mencionada.

Não se aflija por deparar-se com tantos comandos, alguns dos quais totalmente obscuros para você. Logo, você perceberá que apenas alguns poucos comandos são utilizados com freqüência.

Apesar de o gdb incorporar várias características da *shell bash* de sistemas operacionais da família Unix, ao contrário de qualquer *shell* destes sistemas, o interpretador de comandos do gdb não faz distinção entre maiúsculas e minúsculas. Por exemplo, os comandos `HELP`, `Help` e `help` possuem exatamente o mesmo significado para o gdb.

6.13.4 Executando um Programa

Para executar um programa sob a supervisão do gdb, utilize o comando `run` (abreviadamente, `r`). Se seu programa precisar de argumentos de linha de comando (v. **Seção 8.5**), estes argumentos devem seguir o comando `r`.

Digite `r` no prompt do gdb para iniciar a execução do programa `depurar`. Após executar este passo, você deverá ver o seguinte na parte inferior do console:

```
(gdb) r
Starting program: /home/ulysses/Debug/depurar
Introduza dois inteiros:
```

Note que o programa `depurar` começou a ser executado. Basicamente, a única diferença entre esta execução e aquela anterior é que, agora, o programa está sendo executado sob a supervisão direta do gdb, e não do sistema operacional.

Continuando, digite dois números inteiros como antes e veja o que acontece. Você deverá obter algo semelhante ao seguinte na parte inferior da tela:

```
Program received signal SIGSEGV, Segmentation fault.
0xb7e669ea in _IO_vfscanf () from /lib/tls/i686/cmov/libc.so.6
(gdb)
```

Observe que, novamente, o programa foi abortado. Mas, agora, o gdb apresenta informações adicionais que não aparecem quando o programa é executado diretamente no sistema operacional. Essas informações, no entanto, são bastante obscuras para um iniciante e serão brevemente decifradas aqui.

A primeira linha apresentada pelo gdb:

```
Program received signal SIGSEGV, Segmentation fault.
```

indica exatamente aquilo que o sistema operacional apresenta; i.e., o tipo de erro que causou o aborto do programa. A única diferença é que, aqui, o gdb é mais preciso e

informa que o programa recebeu um sinal, representado pela macro **SIGSEGV**, que não foi capturado pelo programa[5].

A segunda linha apresentada pelo gdb parece ainda mais enigmática:

```
0xb7e669ea in _IO_vfscanf () from /lib/tls/i686/cmov/libc.so.6
```

Brevemente, esta linha informa o endereço em notação hexadecimal da instrução que causou o aborto do programa. Além disso, o gdb informa a função que contém esta instrução e onde se encontra esta função. Apesar de parecer obscura, esta informação contém uma dica importante sobre a instrução que causou o erro. Isto é, ela informa que o erro foi causado por uma função denominada `_IO_vfscanf()`. Como não existe nenhuma função com esta denominação no programa sendo depurado, esta função só pode ter sido chamada por uma das funções de biblioteca utilizada pelo programa. Como o nome da função contém *scanf* em sua composição, é bem provável que ela tenha sido invocada por alguma chamada da função **scanf()** no programa.

6.13.5 Situando-se com Backtrace e List

O comando `backtrace` (abreviadamente, `bt`) apresenta a seqüência de funções chamadas da mais recente para a menos recente e permite que o programador visualize o fluxo de execução do programa. Portanto, digite `bt` para obter um resumo dessa seqüência de chamadas de funções:

```
(gdb) bt
#0  0xb7e669ea in _IO_vfscanf () from /lib/tls/i686/cmov/libc.so.6
#1  0xb7e6ebbb in scanf () from /lib/tls/i686/cmov/libc.so.6
#2  0x08048400 in main () at main.c:29
(gdb)
```

O resultado da execução deste comando confirma a suspeita levantada na execução do comando `run`. Ou seja, realmente, foi a função **scanf()**, chamada pela função **main()**, que chamou a função `_IO_vfscanf()` que casou o erro fatal.

Neste ponto, já se tem um suspeito para a falha do programa que é a função **scanf()**. Mas, é muito pouco provável que esta função ou qualquer outra função da biblioteca padrão de C contenha bugs. Esse argumento é defensável porque funções de biblioteca são utilizadas por tantos programadores, com tanta freqüência e durante tanto tempo que qualquer eventual bug que tenha surgido numa tal função de biblioteca já teria sido detectado e corrigido. Isso torna-se ainda mais claro quando se considera a trivialidade

[5] Tratamento de sinais é abordado no **Volume II**, mas este conhecimento não é essencial no presente contexto.

do programa em questão e o uso comum da função **scanf()**. Portanto, quando o gdb indica a ocorrência de um erro numa função de biblioteca, o mais provável é que esta função tenha sido chamada incorretamente. Isto é, mais precisamente, o mais provável é que se tenha chamado a função utilizando parâmetros incorretos.

Pode-se confirmar esta nova suspeita abrindo-se o arquivo que contém a chamada da função **scanf()** e examinando-se como foi feita esta chamada. Mas, resultado similar pode ser obtido sem precisar abandonar o gdb usando o comando `list` (abreviadamente, `l`) que lista linhas de um arquivo-fonte. Este comando pode ser utilizado de diversas maneiras, mas as mais comuns são as seguintes:

`l` *`nome-do-arquivo`*`:`*`linha`*
`l` *`linha`*
`l` *`nome-da-função`*`:`*`linha`*

Nos dois últimos casos, as linhas de programa apresentadas são do arquivo corrente, que é o arquivo que contém a próxima instrução do programa a ser executada.

Examine novamente o resultado da execução do comando `bt` acima. Note que o gdb informa que a chamada suspeita da função **scanf()** encontra-se na linha 29 do arquivo `main.c`. Portanto, emitindo-se o comando:

```
l main.c:29
```

pode-se obter a informação desejada, que deve ser semelhante ao seguinte:

```
(gdb) l main.c:29
24      int main(void)
25      {
26         int MDC;
27
28         printf("Introduza dois inteiros:");
29         scanf("%d %d", a, b);
30
31         MDC = mdc(a, b);
32
33         printf("\nO MDC e': %d", MDC);
(gdb)
```

Observe que, quando executa o comando `list`, o gdb também apresenta linhas circunvizinhas (o que, aliás, é uma boa característica). Se você desejar ver outras linhas além da última apresentada, simplesmente digite `[ENTER]` no prompt do gdb.

Examinando a linha 29 na listagem apresentada pelo gdb, pode-se facilmente identificar o erro que causou o aborto do programa. Este é um erro cometido com freqüência por iniciantes em C, mas que, muitas vezes, por descuido, também acomete programadores experientes.

Notou o erro? Simplesmente utilizaram-se variáveis como argumentos da função **scanf()** ao invés dos seus endereços. Portanto, deve-se corrigir este erro, recompilar o programa e ver se ele funciona com esta alteração.

Os comandos `where` (abreviadamente, `whe`) e `info stack` (abreviadamente, `info s`) também permitem que o programador obtenha o caminho percorrido pelo fluxo de execução até o ponto onde ele se encontra. Esses comandos funcionam exatamente do mesmo modo que `backtrace`.

6.13.6 Encerrando a Execução de um Programa

Como o erro que causou o aborto do programa já foi descoberto, pode-se encerrar a execução do programa no gdb. Para fazer isto, utiliza-se o comando `kill` (abreviadamente, `k`). Digite `k` no prompt do gdb e você obterá como resposta:

```
Kill the program being debugged? (y or n)
```

Ou seja, o gdb está solicitando que você confirme ou não se realmente deseja encerrar a execução do programa. Como é isso mesmo que você deseja, responda "y".

Note que, se você respondeu "y" como recomendado, o prompt do gdb aparece novamente. Isto significa que a execução do programa foi encerrada, mas o gdb continua sendo executado. Você pode finalizar o gdb digitando `quit` (abreviadamente, `q`).

Numa situação real de depuração, seria melhor deixar o gdb em execução enquanto o programa está sendo corrigido, pois, talvez, o programa sob escrutínio ainda contenha erros mesmo após as correções serem efetuadas. Entretanto, no contexto desta demonstração de uso do gdb, encerre-o, pois, logo mais, será apresentado um novo modo de executá-lo.

6.13.7 Corrigindo um Erro

Uma vez identificada a origem de um determinado erro de programação, corrigi-lo é relativamente fácil. No caso do programa sendo depurado, a correção consiste em simplesmente substituir a instrução:

```
scanf("%d %d", a, b);
```

por:

```
scanf("%d %d", &a, &b);
```

Para efetuar esta alteração, abra o arquivo `main.c` em seu editor de programas favorito e faça a devida alteração. Em seguida, recompile o programa e execute-o, conforme descrito anteriormente.

Quando solicitado, introduza dois números inteiros, como por exemplo:

```
Introduza dois inteiros:20 15
```

Neste caso, o programa responde:

```
O MDC e': 5
```

Este resultado apresentado pelo programa é auspicioso, pois o máximo divisor comum de `20` e `15` é realmente `5`. Assim, o programa parece que está funcionando perfeitamente bem.

Mas, como funcionar uma vez não significa funcionar sempre, execute novamente o programa e, desta vez, quando solicitado, introduza 0 como valor do primeiro inteiro e você obterá como resultado algo como:

```
Floating point exception (core dumped)
```

Isto quer dizer que o programa novamente foi abortado e foi criado um arquivo *core* (se seu sistema estiver configurado para isto).

Se o gdb estiver em execução, enquanto você corrige e recompila o programa, você pode executá-lo sem problemas no depurador, pois ele é capaz de identificar que o programa foi alterado e utilizar as novas versões dos arquivos correspondentes. No entanto, você não conseguirá reconstruir o programa se o mesmo estiver em execução, pois o *linker* irá reclamar que o programa está em uso e, portanto, não poderá ser alterado (i.e., o arquivo executável não poderá ser substituído).

Sempre que corrigir um erro, convença-se de que realmente entendeu a causa o por que a correção adotada resolve o problema. Certifique-se ainda que a correção não trará efeitos danosos para outras partes do programa.

6.13.8 Depuração Post-mortem: Examinando um Arquivo Core

Como foi descoberto um novo problema no programa sendo testado, é uma boa idéia examiná-lo novamente com o gdb[6]. A título de ilustração, será mostrado aqui como o gdb pode ser utilizado para examinar arquivos *core*. Mas, antes de prosseguir, certifique-se, utilizando o comando `ls` do Linux, de realmente um arquivo core foi criado no diretório onde o programa está sendo executado[7]. Se este arquivo não foi gerado pelo sistema operacional, você não poderá acompanhar este segmento da lição.

Na linha de comando do Linux invoque o gdb do seguinte modo:

```
gdb depurar core
```

Este comando de execução do gdb é semelhante àquele apresentado antes, mas, agora, adicionalmente, solicita-se ao gdb que examine o arquivo *core* que se encontra no mesmo diretório do arquivo executável `depurar`. O resultado da execução desse comando deve ser semelhante ao seguinte:

```
Reading symbols from /lib/tls/i686/cmov/libc.so.6...done.
Loaded symbols for /lib/tls/i686/cmov/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./depurar'.
Program terminated with signal 8, Arithmetic exception.
#0  0x08048451 in mdc (x=20, y=0) at mdc.c:17
17          while ( (x%divisor) || (y%divisor) )
(gdb)
```

Observe que, mesmo sem executar o programa, o gdb é capaz de apontar a causa do erro que provocou o aborto do programa apenas examinando o arquivo core gerado pelo Linux[8]. Mas, um arquivo *core* só faz sentido se for utilizado com a versão exata do programa que causou sua geração. Ou seja, se após a geração de um arquivo *core*, o programa que deu origem a este arquivo for recompilado, esse arquivo *core* não poderá ser utilizado com a nova versão do programa.

A três últimas linhas apresentadas por gdb oferecem pistas sobre o que causou a morte do programa. A primeira dessas linhas:

---

[6] Lembre-se que aqui se está fazendo uma simulação de uma sessão de depuração utilizando um programa contendo bugs que um programador com experiência mediana em C facilmente encontraria sem precisar do gdb. Mas, o objetivo aqui não é encontrar os erros do programa, mas sim mostrar como utilizar o gdb para esta tarefa. No mundo real, você só deverá utilizar o gdb quando outras alternativas mais simples de detectar e corrigir erros estiverem esgotadas.

[7] Veja comentário a respeito da geração de arquivos *core* apresentado na **Seção 6.13.1**.

[8] Este tipo de depuração, que difere daquela apresentada anteriormente por não ser em tempo real, é denominada depuração *post-mortem* (i.e., póstuma), pois utiliza um arquivo gerado em decorrência do aborto (i.e., *morte*) do programa.

```
Program terminated with signal 8, Arithmetic exception.
```

informa que o programa foi abortado devido a uma operação aritmética inválida.

A linha seguinte:

```
#0  0x08048451 in mdc (x=20, y=0) at mdc.c:17
```

informa que a instrução que causou o problema encontra-se na linha 17 da função `mdc()`, que foi chamada com os argumentos `x` e `y` assumindo os valores `20` e `0`, respectivamente.

Finalmente, a última linha apresentada pelo gdb mostra exatamente qual foi a instrução que causou o problema:

```
17          while ( (x%divisor) || (y%divisor) )
```

Neste ponto você já deve ter percebido que este erro é muito fácil de descobrir. Mas, mesmo que você encontre o erro imediatamente, continue fingindo que o erro ainda não foi encontrado, de modo que se possam explorar mais algumas facilidades oferecidas pelo gdb.

6.13.9 Instalando e Removendo Breakpoints

Suponha que, neste instante, tudo que se sabe a respeito do erro que causou o aborto do programa seja que ele foi causado pela função `mdc()` e que talvez esta função possa ter recebido parâmetros indevidos. Então, pode-se executar o programa até logo antes de esta função ser chamada e examinar os valores dos parâmetros passados para ela. Para isto utiliza-se um ponto de parada na instrução onde é feita a chamada da função.

Para instalar um *breakpoint* numa dada instrução, utiliza-se o comando `break` (abreviadamente, `b`). Este comando pode assumir diversos formatos, mas o mais simples deles é:

```
b local
```

onde *`local`* pode ser especificado de várias maneiras, tais como:

- Um número de linha no arquivo corrente ou num arquivo especificado
- Um nome de função
- Um endereço de instrução (por exemplo, o endereço `0x08048451` que aparece no último conjunto de mensagens apresentado acima pelo gdb)

Em depuração prática, tipicamente, utilizam-se apenas uns poucos *breakpoints* de cada vez.

Para praticar o uso de *breakpoints*, instale um *breakpoint* na instrução que faz a chamada da função `mdc()` utilizando os seguintes passos:

1. Para identificar o número da linha onde se encontra a instrução na qual você deseja instalar o *breakpoint*, digite o seguinte comando:

   ```
   l main
   ```

2. O comando `list` mostra que a instrução desejada encontra-se na linha 31. Portanto, para instalar um *breakpoint* nesta linha digite o seguinte comando[9]:

   ```
   b main.c:31
   ```

   Quando este comando é executado, o gdb responde com a seguinte mensagem:

   ```
   Breakpoint 1 at 0x80483cd: file main.c, line 31.
   ```

   Note que o gdb atribui um número seqüencial a cada *breakpoint* instalado. Neste caso, o número atribuído foi 1, visto que este é o primeiro *breakpoint* instalado nesta sessão de depuração. É uma boa idéia memorizar ou tomar nota do número atribuído a cada *breakpoint*, pois este número pode ser necessário posteriormente se for desejado desativar um dado *breakpoint*.

Após instalar o *breakpoint*, digite `r` para colocar o programa em funcionamento (se você estiver seguindo exatamente todos os passos anteriores, o programa está carregado, mas ainda não está em execução). Feito isto, o programa começa a ser executado como antes e pára à espera da introdução dos dois números inteiros. Mas, antes de continuar a execução do programa, aprenda um pouco mais sobre *breakpoint* nos parágrafos seguintes.

Tendo um *breakpoint* instalado, como foi feito acima, a execução do programa será interrompida sempre que a instrução associada ao *breakpoint* estiver prestes a ser a executada. Como esta é apenas uma sessão de treinamento no uso do gdb, remova o *breakpoint* que foi instalado há pouco, pois, em seguida, será apresentado um outro tipo *breakpoint*.

---

[9] A razão pela qual é necessário especificar o nome do arquivo e não apenas o número da linha é o fato de o arquivo `main.c` não ser o arquivo corrente. Sabe-se isso porque a última informação apresentada pelo gdb refere-se ao arquivo `mdc.c`, que contém a instrução que causou o aborto do programa.

Para remover um *breakpoint*, utiliza-se o comando `delete` (abreviadamente, `d`) ou `clear` (abreviadamente, `cl`). Mas, estes comandos não são equivalentes. O comando `clear` é muito mais poderoso, pois permite remover vários *breakpoints* de uma única vez. Como, na prática, não é recomendável ter muitos *breakpoints* ativos numa mesma sessão de depuração, o comando `delete` é mais utilizado. Este comando tem a seguinte sintaxe[10]:

```
d número-do-breakpoint
```

onde *`número-do-breakpoint`* é o número de ordem que o gdb atribui ao *breakpoint*. Na presente sessão de depuração, apenas um *breakpoint* foi instalado e ele possui numeração 1. Mas, se for necessário saber quais são os *breakpoints* instalados numa dada sessão de depuração e que números lhes são atribuídos, pode-se utilizar o comando `info` (abreviadamente, `i`), como mostrado a seguir[11]:

```
i b
```

Quando você digita este comando, o gdb responde com:

```
Breakpoint 1 at 0x80483cd: file main.c, line 31.

Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483cd in main at main.c:31
```

Ou seja, o gdb informa que atualmente existe apenas um *breakpoint* instalado e número deste *breakpoint* é 1. Portanto, para remover este *breakpoint*, digite o comando:

```
d 1
```

Feito isso, o gdb atende a este comando silenciosamente à la Unix. Se você não acredita que o *breakpoint* foi realmente removido, digite novamente o comando:

```
i b
```

e o gdb responderá:

```
No breakpoints or watchpoints.
```

---

[10] Se o número do *breakpoint* não for especificado, o gdb interpretará o comando como se fosse desejado remover todos os *breakpoints* instalados. Neste caso, o gdb solicita confirmação.

[11] O comando `info` não serve apenas para *breakpoints*. Este comando é muito útil pois ele permite a obtenção de informações sobre muitos outros aspectos associados à sessão corrente de depuração ou ao ambiente no qual o programa está sendo executado. Você pode obter maiores informações sobre este comando digitando `help info` (ou, simplesmente, `h i`) no prompt do gdb.

Um tipo de *breakpoint* bem interessante é o condicional que, diferentemente daquele apresentado há pouco, interrompe a execução do programa apenas quando uma determinada condição é satisfeita. Este tipo de *breakpoint* assume o seguinte formato:

`b` *`local`* `if` *`condição`*

onde *`local`* é especificado conforme visto antes e *`condição`* é uma expressão cujo resultado de sua avaliação determina se o *breakpoint* será acionado ou não. Mais precisamente, quando o resultado da avaliação desta expressão for diferente de zero, o *breakpoint* será acionado; caso contrário, ele não o será.

Instale novamente um *breakpoint* na instrução 31 do arquivo `main.c`, mas, desta vez, utilize o comando:

```
b main.c:31 if a == 0
```

O gdb responde com:

```
Breakpoint 2 at 0x80483cd: file main.c, line 31.
```

Observe que mesmo que, no momento, se tenha apenas um *breakpoint* (supondo que o anterior foi removido), o gdb numera este novo *breakpoint* com 2. Isto significa que, numa sessão de depuração, o gdb não reinicia a numeração de *breakpoints*.

Este *breakpoint* é semelhante àquele instalado anteriormente, mas, agora, o programa será interrompido apenas quando o valor da variável `a`, que representa o primeiro valor introduzido pelo usuário, for igual a zero.

Lembrando que o programa sendo depurado funciona bem quando os valores introduzidos são diferentes de 0, mas foi abortado quando o primeiro valor introduzido foi igual a zero, a escolha de um *breakpoint* condicional foi uma boa idéia. Entretanto, se você executar o programa novamente e introduzir zero como valor para o segundo argumento, você verá que o programa também é abortado. Assim, uma melhor escolha para a instalação desse *breakpoint* seria:

```
b main.c:31 if a == 0 || b == 0
```

Se você digitar este comando, obterá do gdb a seguinte resposta:

```
Note: breakpoint 2 also set at pc 0x80483cd.
Breakpoint 3 at 0x80483cd: file main.c, line 31.
```

Esta informação quer dizer que existem dois *breakpoints* associados à mesma instrução 31 do arquivo `main.c`. Faz sentido ter mais de um *breakpoint* condicional associado a

uma mesma instrução, mas, neste caso, o *breakpoint* 2 é redundante pois sua condição também é levada em consideração no *breakpoint* 3. Portanto, remova o *breakpoint* 2 digitando o comando[12]:

```
d 2
```

Agora digite o comando `r` para iniciar a execução do programa e, quando instado, introduza os valores `20` e `10` como dados de entrada para o programa. Assim, você obterá como resultado o seguinte:

```
Introduza dois inteiros: 20 10

O MDC e': 10

Program exited normally.
```

Conforme esperado o *breakpoint* instalado não teve nenhum efeito, pois os valores introduzidos para o programa foram ambos diferentes de zero. Agora, execute novamente o programa e introduza os valores 10 e 0 quando solicitado. Você deverá, então obter como resposta o seguinte:

```
Breakpoint 3, main () at main.c:31
31          MDC = mdc(a, b);
```

Neste instante, o *breakpoint* instalado foi realmente ativado, uma vez que um dos valores introduzidos foi igual a zero, o que tornou válida a condição de parada.

Na prática, a instalação de *breakpoints* em locais adequados depende da intuição do programador. Como ponto de partida, pode-se adotar o método de busca binária descrito na **Seção 6.11.3**. Mas, o mais importante é ter sempre em mente que um *breakpoint* deve ser colocado num ponto que permita obter informações importantes sobre o estado do programa.

6.13.10 Examinando Valores de Variáveis: Comando Print

Quando o programa está parado num *breakpoint* várias ações podem ser executadas. Por exemplo, pode-se examinar o valor de uma variável ou expressão utilizando o comando `print` (abreviadamente, `p`), cujo formato é:

`p` *`expressão`*

[12] Na realidade não faz diferença se você apaga ou não este *breakpoint*, mas tenha paciência. Isto é apenas um treinamento com o objetivo de torná-lo *íntimo* do gdb.

onde *`expressão`* é uma expressão que pode conter variáveis válidas no contexto corrente (i.e., dentro do escopo de todas essas variáveis). No contexto que corresponde à execução da função **main()**, existem três variáveis válidas: `a`, `b` e `MDC`. Para examinar o valor corrente da variável a, digite o comando:

```
p a
```

Se você continua seguindo exatamente as instruções até aqui, deverá obter a seguinte mensagem do gdb:

```
$1 = 10
```

O número à esquerda precedido por `$` é um valor sequencialmente atribuído a cada expressão apresentada pelo gdb e pode ser utilizado na formação de expressões subseqüentes. Por exemplo, se desejar obter o valor da expressão `a + b`, você pode digitar o comando:

```
p a + b
```

ou ainda:

```
p $1 + b
```

Já que `$1` representa o valor de `a`. Em qualquer dos casos, a resposta do gdb é:

```
$2 = 10
```

Utilizar os valores das expressões armazenadas pelo gdb só é vantajoso quando o número precedido por `$` representa uma expressão complexa, o que não é o caso aqui. É importante notar ainda que `$1` não representa a variável `a` em si, mas sim o valor associado a ela quando a mesma foi avaliada. Portanto, mesmo que o valor da variável `a` seja alterado, o valor de `$1` será sempre o mesmo.

6.13.11 Controlando a Execução de um Programa

Estando estacionada numa dada instrução, a execução do programa pode prosseguir de diversas maneiras, de acordo com os comandos apresentados na **Tabela 1**.

COMANDO	ABREVIAÇÃO	O PROGRAMA É EXECUTADO ATÉ...

COMANDO	ABREVIAÇÃO	O PROGRAMA É EXECUTADO ATÉ...
`continue`	`c`	o próximo *breakpoint*, se algum for encontrado durante a execução, ou até o final se nenhum *breakpoint* for encontrado
`finish`	`fin`	o final da função sendo executada
`next`	`n`	a próxima instrução, tratando chamada de função como uma instrução indivisível; portanto, o programa pára na próxima instrução após a chamada da função.
`step`	`s`	a próxima instrução, tratando chamada de função como um conjunto de instruções; portanto, o programa pára na primeira instrução da função.
`jump`	`j`	a linha especificada

Tabela 1: Principais Comandos de Execução do Depurador gdb

Conforme mostra a **Tabela 1**, a diferença entre os comandos `next` e `step` é relevante apenas quando a próxima instrução a ser executada trata-se de uma *chamada de função para a qual existem informações de depuração disponíveis*. Quando este é o caso, o comando `next` trata a próxima instrução como uma instrução elementar (i.e., indivisível) da linguagem C, enquanto que a opção `step` implica executar passo a passo as instruções da função chamada. Se a próxima instrução não contém nenhuma chamada de função com informações de depuração, não existe diferença entre estas duas opções.

Na prática, quando a próxima instrução é uma chamada de função, utiliza-se `next` quando não se suspeita que a referida função esteja causando problemas e utiliza-se `step` quando a suspeita existe.

Um aspecto importante relacionado ao uso dos comandos `next` e `step` diz respeito a chamadas de funções de biblioteca. Normalmente, na prática, você não deve inspecionar o interior destas funções (a não ser, é claro, que você esteja desenvolvendo uma biblioteca) pelas razões expostas a seguir:

- É raro encontrar bugs em tais funções pelas razões expostas na **Seção 6.13.5**. Mas, se realmente você suspeita que uma função de biblioteca

contém bugs, é mais fácil pesquisar na internet em *chats*, FAQs, etc. se outros programadores já tiveram o mesmo problema e qual é a solução proposta para o mesmo.

- Talvez, você não entenda completamente a codificação de uma tal função. Assim, como poderá depurá-la?
- Provavelmente, a biblioteca não foi compilada com informações de depuração (por exemplo, `-g`, se a biblioteca foi compilada com gcc) ou, talvez, o código-fonte nem esteja disponível. Portanto, mesmo que você esteja convencido que deve depurar uma função de biblioteca, talvez isso não seja possível.

Continuando a sessão de depuração, neste instante, o programa encontra-se estacionado na linha 31 do arquivo `main.c`. Ou seja, a próxima instrução a ser executada é:

```
MDC = mdc(a, b);
```

Como esta instrução contém uma chamada de função, os comandos `next` e `step` agem de forma diferente. Experimente utilizar o comando `next` digitando o comando:

```
n
```

Como resultado da execução deste comando, você obterá:

```
Program received signal SIGFPE, Arithmetic exception.
0x08048451 in mdc (x=10, y=0) at mdc.c:17
17          while ( (x%divisor) || (y%divisor) )
```

Isto significa que, mais uma vez, o programa foi abortado. Aliás, isso já era esperado, pois já se sabia da execução anterior que a função `mdc()` causa o aborto do programa quando um dos argumentos recebidos por ela é igual a zero.

Agora, execute novamente o programa digitando o comando `r` no prompt do gdb. Então, o gdb responde como:

```
The program being debugged has been started already.
Start it from the beginning? (y or n)
```

Isto é, o gdb informa que o programa ainda não foi encerrado. Pode parecer estranho que o gdb tenha informado que o programa tenha sido abortado e agora está informando que o programa ainda está em execução. Mas, a verdade é que a informação anterior do gdb indica que o programa *será* abortado e não que já *tenha sido* abortado. Nenhuma outra instrução escrita em C no programa-fonte será executada, mas existem instruções em linguagem de máquina no programa executável que serão executadas antes de o programa ser finalmente encerrado. Para confirmar este arrazoado, digite `n` para que o

gdb não reinicie o programa e, em seguida, no prompt do gdb digite `c` para continuar a execução do programa e observe o que ocorre.

Como se sabe que o programa será abortado de qualquer modo, responda "y" para reiniciar a execução do programa. Então, quando instado pelo programa introduza, novamente, os valores `10` e `0`.

A execução do programa novamente pára na instrução da linha 31 do arquivo `main.c` como antes. Isto quer dizer que os *breakpoints* são mantidos durante várias execuções de um programa durante uma mesma sessão de depuração. Outras configurações criadas para o programa sendo depurado pelo gdb também são mantidas. Por exemplo, os valores identificados por `$1` e `$2` da última execução do programa continuam ativos. Entretanto, esses valores não são mantidos quando se reinicia o próprio depurador.

O programa encontra-se neste momento estacionado na linha 31 do arquivo `main.c`, de modo que a próxima instrução a ser executada é:

```
MDC = mdc(a, b);
```

Agora, ao invés de utilizar o comando `next`, como na última execução do programa, utilize o comando `step` digitando:

```
s
```

Então, o gdb responde com:

```
mdc (x=10, y=0) at mdc.c:12
12          if ( (x < 0) || (y < 0) )
```

Isto mostra que a função `mdc()` foi chamada com os parâmetros `x` e `y` assumindo, respectivamente, os valores `10` e `0`. O gdb informa ainda que o programa está prestes a executar a instrução:

```
if ( (x < 0) || (y < 0) )
```

que faz parte da função `mdc()`.

Digite `s` ou `n` no prompt do gdb e o depurador lhe informará qual será a próxima instrução a ser executada:

```
15          divisor = (y < x) ? y : x;
```

Digite `s` mais uma vez no prompt do gdb e você obterá:

```
17          while ( (x%divisor) || (y%divisor) )
```

Agora, examine o valor da variável `divisor` digitando o comando:

```
p divisor
```

Então, você obterá:

```
$4 = 0
```

Se você ainda não havia descoberto o erro que causa o aborto do programa, provavelmente, neste momento, será capaz de descobri-lo. A variável `divisor`, utilizada como denominador nas expressões `x%divisor` e `y%divisor`, vale zero neste instante e, por isso, o programa executa uma operação ilegal e é abortado.

Numa situação real, seus próximos passos deveriam ser os seguintes:

1. Encerrar a execução do programa
2. Corrigir o erro num editor de programas
3. Recompilar o programa
4. Testar novamente o programa para verificar se a alteração efetuada surte efeito

Entretanto, como aqui não se está lidando com uma situação realística, serão apresentadas mais algumas características úteis do gdb.

Uma das facilidades mais interessantes do gdb é que ele permite que se altere o valor de uma variável durante a execução de um programa por meio do comando `set`, que possui o seguinte formato:

`set` *`atribuição`*

onde *`atribuição`* é uma atribuição escrita segundo as regras da linguagem C.

Atribua o valor `12` à variável `divisor` utilizando o comando `set` do seguinte modo:

```
set divisor = 12
```

Mais uma vez, o gdb atende silenciosamente a este comando à la Unix[13].

[13] Se estiver desconfiado, digite `p divisor` para examinar o valor atual da variável `divisor`.

6.13.12 Selecionando um Contexto: Comando Frame

Se você estiver seguindo todas recomendações até aqui, a execução do programa encontra-se no interior da função `mdc()`. Suponha, então, que você deseje examinar o valor de uma variável fora do escopo desta função, mas que seja local a uma outra função que faz parte da seqüência de funções chamadas até atingir a função `mdc()`.

A pilha de execução (*stack*) de um programa é dividida em blocos contíguos em memória denominados ***stack frames*** (ou, simplesmente, ***frames***). A cada chamada de função, é criado um *stack frame* para a chamada contendo: o endereço da instrução que fez a chamada, cópias dos parâmetros reais utilizados na chamada e as variáveis locais de duração automática da função. Quando a função retorna, o espaço alocado em memória para o *stack frame* da chamada é liberado. Em qualquer instante, a pilha de execução contém todos os *frames* associados a funções correntemente em execução. Para saber quais são estas funções você utiliza o comando `backtrace`, conforme foi visto na **Seção 6.13.5**. Portanto, digitando o comando `bt` neste instante você obtém:

```
#0  mdc (x=2, y=0) at mdc.c:15
#1  0x080483e4 in main () at main.c:31
```

O gdb informa que, atualmente, existem dois *frames* armazenados na pilha de execução. O *frame* numerado com 0 é aquele correntemente sendo executado e o *frame* 1 foi o que provocou a execução do *frame* 0. Ou seja, traduzindo, a função correntemente executada é a função `mdc()`, que constitui o *frame* 0, e quem chamou esta função foi a função **main()**, que constitui o *frame* 1.

Assim, se você desejar examinar o valor de uma variável que não faz parte do *frame* corrente, mas que faz parte de outro *frame* armazenado na pilha de execução, você precisa informar o gdb em qual *frame* está o escopo desta variável. Isto pode ser realizado utilizando o comando `frame` (abreviadamente, `f`).

Antes de praticar o uso do comando `frame`, digite o seguinte comando no prompt do gdb:

```
p MDC
```

e o gdb responderá com:

```
No symbol "MDC" in current context.
```

Isto significa que o gdb desconhece a variável `MDC` no contexto (i.e., *frame*) sendo executado.

Agora, utilize o comando `frame` para indicar em que *frame* a variável faz sentido. Primeiro, digite o comando:

```
f 1
```

e o gdb responderá assim:

```
#1  0x080483e4 in main () at main.c:31
31          MDC = mdc(a, b);
```

Em seguida, digite o mesmo comando que você havia introduzido antes:

```
p MDC
```

e a resposta do gdb agora será algo como[14]:

```
$3 = -1208769904
```

A resposta do gdb é esquisita apenas pelo fato de a variável `MDC` não ter sido iniciada, mas, pelo menos agora, o gdb reconhece a existência da variável.

6.13.13 Examinando Valores de Variáveis: Comando Display

Se você estiver seguindo todas recomendações até aqui, a execução do programa encontra-se parada na instrução:

```
while ( (x%divisor) || (y%divisor) )
```

Se você digitar `s` para continuar a executar o programa até a próxima instrução, verá que a próxima instrução a ser executada é o corpo do laço **while** conforme informa o gdb:

```
18              divisor--;
```

Agora, suponha que você deseja monitorar os valores assumidos pela variável `divisor` por meio da apresentação de seu valor a cada execução do corpo do laço. Utilizando o comando `print` visto anteriormente, você teria que digitar:

```
p divisor
```

a cada execução do corpo do laço. Nesta situação, é mais confortável utilizar o comando `display` (abreviadamente, `disp`).

[14] Provavelmente, o resultado que você obterá não será exatamente este, pois a variável `MDC` não foi iniciada.

O comando `display` apresenta o valor de uma variável sempre que a execução do programa pára. Experimente-o com a variável `divisor` digitando, no prompt do gdb, o comando:

```
disp divisor
```

Tendo digitado este comando, o gdb responde com:

```
1: divisor = 12
```

Digite `s` para executar mais uma vez o corpo do laço **while** e você obterá:

```
17          while ( (x%divisor) || (y%divisor) )
1: divisor = 11
```

Observe que o gdb informa o novo valor assumido pela variável `divisor`.

Suponha que você já esteja satisfeito com a apresentação, decorrente do uso do comando `display`, de diversos valores de uma variável. Então, você pode desativar esta exibição de valores utilizando o comando `undisplay` (abreviadamente, `undisp`), como, por exemplo:

```
undisp divisor
```

6.13.14 Outros Comando do gdb

Com a apresentação do comando `undisp`, encerra-se este breve treinamento sobre o uso do depurador gdb. Existem outros inúmeros comandos do gdb que não foram apresentados aqui, conforme você deve ter verificado quando utilizou o comando `help` na **Seção 6.13.3**.

Do mesmo modo que se argumenta que o comando mais importante do sistema operacional Unix (ou Linux) é o comando `man`, pode-se sustentar que o comando mais importante do gdb é o comando `help`, pois, utilizando estes comandos, podem-se obter informações sobre os demais.

Antes de encerrar esta seção, é importante chamar atenção para o fato de o gdb utilizar várias características interessantes da interface de comandos (*shell*) *bash*. Duas destas características permitem uma significativa economia de esforços:

- O uso da tecla `[TAB]` para completar comandos.
- A manutenção de um histórico de comandos que podem ser recuperados utilizado as setas para cima e para baixo do teclado.

6.13.15 O Depurador gdb como Ferramenta de Aprendizagem

Sem dúvida, o depurador gdb é um programa de excelente qualidade que pode intimidar à primeira vista, mas que, de fato, é bem fácil de usar depois que se adquire alguma prática. O que muitos não sabem é que o gdb pode ser utilizado como uma ferramenta para aprendizagem. Isto é, mesmo que você não tenha nenhum problema de depuração em mãos, utilizando o gdb, você também pode aprender muito sobre programação, em particular, e sobre computadores e sistemas operacionais, em geral.

Para utilizar o gdb como ferramenta didática que irá ajudá-lo em seu crescimento como programador, simplesmente utilize-o e aprofunde sua curiosidade sobre os mais diversos comandos disponíveis.

Encontrar e corrigir erros de programação também constituem uma excelente oportunidade de aprendizagem. Aproveite esta oportunidade para tentar entender por que o erro foi cometido e adote medidas que minimizem a possibilidade de o mesmo erro ser novamente cometido em futuros programas.

Você pode praticar a lição apresentada aqui novamente de várias maneiras de acordo com sua criatividade e curiosidade. Também, para adquirir proficiência no uso do depurador gdb, habitue-se a utilizá-lo para observar o funcionamento de seus programas (mesmo quando estes não contêm bugs). Assim procedendo, você estará adquirindo não apenas experiência no uso de depuradores de alto nível como também entenderá melhor como o compilador traduz programas e como estes são executados.