CAPÍTULO 6

LEGIBILIDADE E DEPURAÇÃO DE PROGRAMAS

6.1 Introdução

Linguagens de alto nível foram criadas com a expectativa de que programas escritos usando-as apresentassem as seguintes propriedades:

- **Legibilidade**. Esta propriedade é associada à facilidade de leitura e compreensão do programa.
- **Manutenibilidade**. Um programa que apresenta boa manutenibilidade é um programa fácil de ser mantido (i.e., modificado).
- **Portabilidade**. Esta propriedade corresponde à facilidade com que um programa escrito numa dada plataforma pode ser compilado numa outra plataforma sem ser necessário alterar seu código-fonte.

Mas o simples fato de escrever um programa numa linguagem de alto nível não garante que ele possua essas propriedades. Isto é, a satisfação delas depende principalmente do próprio programador.

A maioria dos livros de programação dedica-se à tarefa de ensinar a **sintaxe** (i.e, o formato) e a **semântica** (i.e., o funcionamento) das construções de uma dada linguagem de programação. Isto é, estes livros preocupam-se principalmente em ensinar o que ***pode*** ser feito, pouco se preocupando com aquilo que ***deve*** ser feito com as construções da linguagem. Alguns

poucos livros destinam-se a ensinar a **pragmática** das construções de uma linguagem de programação[1]. Na mesma linha destes últimos livros estão os **manuais de estilo**, documentos facilmente encontrados na internet que contêm recomendações sobre como programar de modo a atender critérios de legibilidade, manutenibilidade e portabilidade.

Parte do conteúdo deste capítulo lida com legibilidade e pode ser vista como um mini manual de estilo de programação em C. A outra parte do capítulo lida com um tópico de natureza essencialmente prática: depuração de programas.

6.2 Uso de Comentários

O objetivo principal de comentários é explicar o programa para outras pessoas e para você mesmo quando for lê-lo algum tempo após sua escrita. Comentários podem também ser utilizados para teste e depuração de programas, conforme será analisado na **Seção 6.11.3**. Aqui, serão apresentados alguns conselhos sobre ***quando***, ***como*** e ***onde*** comentar um programa a fim de clarificá-lo.

O melhor momento para comentar um trecho de um programa é exatamente quando este trecho está sendo escrito. Isto aplica-se especialmente àqueles trechos de programa contendo sutilezas, inspirações momentâneas ou coisas do gênero, que o próprio programador terá dificuldade em entender algum tempo depois. Alguns comentários passíveis de ser esquecidos, tal como a data de início da escrita do programa, também devem ser acrescentados no momento da escrita do programa. Outros comentários podem ser acrescentados quando o programa estiver pronto, embora o ideal continue sendo comentar todo o programa à medida que o mesmo é escrito.

Comentários devem ser claros e dirigidos para programadores com alguma experiência na linguagem. Eles não têm que ser didáticos como

[1] Um exemplo notável é o livro *Code Complete*, de Steve McConnell, cuja referência é apresentada na **Bibliografia**.

alguns comentários apresentados aqui e em outros textos de ensino de programação. Estes comentários didáticos são úteis nestes textos que têm exatamente o objetivo de ensinar, mas comentários num programa real têm o objetivo de explicar o programa para programadores e não o de ensinar a um leigo na linguagem o que está sendo feito. Por exemplo, o comentário na linha de instrução a seguir:

```
x = 2; // O conteúdo de x passa a ser 2
```

é aceitável num texto que se propõe a ensinar programação, mas não faz sentido num programa real.

A melhor forma de adquirir prática na escrita de comentários é ler programas escritos por programadores profissionais e observar os vários estilos de comentários utilizados[2].

Existem dois formatos básicos de comentários: comentário de bloco e comentário de linha, que devem ser utilizados conforme é sugerido nas seções a seguir ou de acordo com a necessidade. Em qualquer caso, você não precisa seguir obrigatoriamente o ***formato*** do modelo sugerido. Isto é, você é livre para usar sua criatividade na criação de seu próprio estilo de comentár; o que se espera é que o ***conteúdo*** seja aquele sugerido nos formatos de comentários apresentados a seguir.

6.2.1 Comentários de Bloco

Blocos de comentário são utilizados no início do programa [i.e., no início do arquivo contendo a função **main()**] com o objetivo informativo de apresentar o propósito geral do programa, data de início do projeto, nome do(s) programador(es), versão do programa, nota de direitos autorais (***copyright***) e qualquer outra informação pertinente. Por exemplo:

[2] Repetindo, textos utilizados no ensino de programação não são ideais para este propósito, pois, como foi afirmado, alguns comentários são didáticos demais para serem utilizados na prática.

```
/****
 *
 * Título do Programa: MeuPrograma
 *
 * Autor: José da Silva
 *
 * Data de Início do Projeto: 10/11/2006
 * Última modificação: 19/11/2007
 *
 * Versão: 2.01b
 *
 * Descrição Geral: Este programa faz isto e aquilo.
 *
 * Dados de Entrada: Este programa espera os seguinte dados ...
 *
 * Dados de Saída: Este programa produz como saída o seguinte
...
 *
 * Copyright © 2007 José da Silva Software Ltda.
 *
 ****/
```

Cada arquivo constituinte de um programa multiarquivo deve conter um comentário em forma de bloco como o apresentado acima, mas com um conteúdo ligeiramente diferente, como exemplificado a seguir:

```
/****
 *
 * Nome do Arquivo: MeuArquivo.c
 *
 * Programador: João da Silva
 *
 * Data de Criação: 20/11/2006
 * Última modificação: 18/11/2007
 *
 * Descrição: Este módulo implementa funções que fazem isto e
aquilo.
 *
 * Copyright © 2007 José da Silva Software Ltda.
 *
 ****/
```

Um bloco de comentário também deve ser utilizado antes da definição de cada função, exceto antes da função **main()**, que deve ser precedida pelo

comentário informativo do programa como um todo, conforme descrito no início desta seção. Esse comentário deve conter, entre outras coisas:

- O nome da função
- O propósito da função
- Como a função deve ser utilizada
- Descrição dos argumentos da função, salientando o modo de cada um deles (entrada, saída ou entrada/saída)
- Descrição do valor retornado pela função (não confunda isto com argumento de saída ou argumento de entrada/saída)
- Descrição sucinta do algoritmo utilizado (se o algoritmo for um algoritmo clássico, basta que se apresente uma referência ao mesmo)

Considere o seguinte como exemplo esquemático de comentário precedendo uma definição de função:

```
/****
 *
 * Função UmaFuncao()
 *
 * Parâmetros:
 *      x (entrada): [descrição do parâmetro x]
 *      y (entrada): [descrição do parâmetro y]
 *      z (saída): [descrição do parâmetro z]
 *      w (entrada/saída): [descrição do parâmetro w]
 *
 * Valor retornado: [os possíveis valores retornados pela função]
 *
 * Descrição: [descreva sucintamente o algoritmo usado]
 *
 ****/
```

Observe que os parâmetros da função são descritos separadamente e seus modos apresentados entre parênteses. Quando a função não possui parâmetros, escreve-se ***nenhum*** após ***Parâmetros:*** no comentário de bloco.

Após a expressão ***Valor retornado:*** deve-se descrever aquilo que a função retorna (i.e., o ***significado*** do valor retornado). Se a função não

retorna nada (i.e., se seu tipo de retorno é **void**), escreve-se *nenhum* após aquela expressão. Um engano freqüentemente cometido por iniciantes é descrever o *tipo de retorno* da função (em vez do significado do valor retornado). Esta última informação não merece nenhum comentário, já que o tipo de retorno de uma função é evidente no cabeçalho da mesma.

Outro local onde blocos de comentário se fazem necessários é em trechos de programas difíceis de serem entendidos por conterem truques, sutilezas, otimizações em código de baixo nível etc. Este tipo de comentário deve ter um formato diferente dos blocos de comentário apresentados anteriormente para distingui-lo dos demais. Por exemplo:

```
/* O trecho de programa a seguir faz isso e aquilo */
/* Descrição de como isso é implementado...        */
/* Descrição de como aquilo é implementado...      */
```

6.2.2 Comentários de Linha

Comentários de linha são utilizados para comentar uma única linha de instrução ou declaração, mas, em si, eles podem ocupar mais de uma linha, por razões de estética e legibilidade. Como o propósito aqui é explicar uma instrução ou declaração que não seja clara para um programador de C, não se deve comentar aquilo que é óbvio para um programador da linguagem. Por exemplo, o seguinte comentário não deve ser feito, pois é evidente para qualquer programador com alguma experiência em C:

```
x = ++y;  /* x recebe o valor de y incrementado de 1 */
```

Comentários como o do último exemplo não são apenas irrelevantes ou redundantes; pior, eles tornam o programa mais difícil de ler, pois desviam a atenção do leitor para informações inúteis. Entretanto, algumas instruções ou declarações mais complexas ou confusas do que a anterior devem ser comentadas não apenas para facilitar a leitura como também para que o programador verifique se a instrução faz exatamente aquilo que o comentário informa que ela faz. Por exemplo,

```
int (*(*f())[])(); /* f é declarada como uma função que retorna
um */
           /* ponteiro para um array de ponteiros para */
          /* funções que retornam um valor do tipo int */
```

Formatar comentários de modo que estes sejam legíveis e ao mesmo tempo não interrompam o fluxo de escrita do programa é muitas vezes difícil e requer alguma habilidade. Quando sobra pouco espaço à direita da instrução para a escrita de um comentário de linha, pode-se colocar o mesmo precedendo a instrução e sem espaço vertical entre ambos, como, por exemplo:

```
  /* A instrução a seguir calcula tal coisa de tal e tal modo */
minhaVariavel1 = minhaVariavel2 + 10*MinhaFuncao(minhaVariavel2);
```

Cada variável que tem papel significativo na função ou programa deve ter um comentário associado à sua declaração para clarificar o papel desempenhado por ela.

```
float   pesoDoAluno;    /* O peso do aluno em quilogramas */
```

Variáveis que não sejam significativas ***não precisam ser comentadas***. O comentário a seguir, por exemplo, é dispensável:

```
unsigned  int  i;  /* Variável utilizada como contador no laço
for */
```

Se tiver dúvida a respeito de algo ser ou não óbvio para ser comentado, comente, pois, neste caso, é melhor pecar por redundância do que por omissão. Mas lembre-se de que comentários não devem ser utilizados para compensar um programa mal-escrito.

Um breve comentário de linha é útil para indicar a quem pertence um dado fecha-chaves (“}”) em blocos longos ou aninhados. Estes comentários são dispensáveis quando um bloco é suficientemente curto. Por exemplo:

```
while (x){
     ... /* Um longo trecho de programa */
     if  (y > 0){
         ... /* Outro longo trecho de programa */
     } /* Final do if */
     ... /* Outro longo trecho de programa */
}  /* Final do while */
```

Existem outras situações nas quais comentários de linha são sugeridos:

- Para chamar atenção de que uma instrução é deliberadamente vazia.
- Idem para um corpo de função deliberadamente vazio.
- Para explicar uma instrução **switch** na qual instruções pertencentes a mais de um caso poderão ser executadas (i.e., na ausência deliberada de um **break**).
- Quando o bom senso recomendar...

Quando alterar alguma instrução com um comentário associado, lembre-se de atualizar o comentário de modo a refletir a alteração da instrução. Ter um comentário que não corresponda àquilo que está sendo descrito é pior do que não ter comentário algum.

6.3 Uso de Espaços em Branco

O uso judicioso de **espaços em branco** é essencial para uma boa legibilidade do programa. Isso inclui, além de endentação, o uso de **espaços horizontais** e **espaços verticais**.

6.3.1 Endentação

O uso *consistente* de **endentação** é essencial para uma boa legibilidade do programa. A **endentação** deve ser utilizada nas seguintes situações:

- Para indicar subordinação de uma instrução com relação a outra. Por exemplo, as chamadas de **printf()** e **scanf()** no trecho de programa a seguir estão subordinadas à instrução **if**:

```
if (x <= 0) {
   printf("Digite um valor inteiro positivo: ");
   scanf("%d", %x);
}
```

• Para ressaltar que uma instrução ou declaração faz parte de uma função. Por exemplo, a endentação da instrução **return** com relação ao cabeçalho da função `Quadrado()` a seguir indica que esta instrução pertence a esta função:

```
int Quadrado(int x)
{
   return x*x;
}
```

• Para indicar que uma linha é continuação de uma instrução ou declaração. Por exemplo:

```
printf("Esta instrucao e' longa demais para caber numa"
       "unica linha; por isso precisou ser dividida");
```

• Para destacar campos numa declaração de estrutura ou união (v. **Capítulo 9**). Por exemplo:

```
typedef struct {
   short dia, mes, ano;
} tData;
```

• Para tornar mais legível a iniciação de arrays multidimensionais (v. **Capítulo 7**). Por exemplo:

```
int  ar[5][3] = { {1, 2, 3},
                  {4},
                  {5, 6, 7} };
```

Utilize três ou quatro espaços de tabulação para endentação[3]. Endentações menores do que isto podem não ser muito legíveis, enquanto

[3] Esta medida não é aleatória, mas sim resultado de pesquisa

endentações maiores farão com que as linhas de instrução atinjam rapidamente a largura da tela.

Use sempre endentação para instruções dentro de blocos ou que fazem parte de alguma estrutura de controle. Exemplos:

```
{
   instrução1;
   ...
   instruçãoN;
}

while  (x) {
   instrução1;
   ...
   instruçãoN;
}
```

No caso de rótulos, é mais comum manter a endentação da instrução rotulada e retroceder à endentação do rótulo mantendo-o na linha anterior à instrução. Por exemplo:

```
while  (x) {
   instrução1;

rotuloDaInstrucao2: /* O rótulo é recuado em relação à instrução
*/
   instrução2; /* Mantém-se a endentação que se teria sem rótulo
*/
   ...
   instruçãoN;
}
```

Outras sugestões para endentação são apresentadas ao longo do texto. Você não precisa seguir exatamente essas sugestões, mas seja qual for sua escolha de endentação, seja consistente (i.e., use-a coerentemente em todos os seus programas).

6.3.2 Outros Espaços em Branco Horizontais

Além de servirem como endentação, espaços horizontaisdevem ainda ser utilizados em torno de identificadores e operadores para melhorar a

legibilidade de expressões. A seguir, alguns conselhos úteis sobre o uso de espaços horizontais:

• Use espaços horizontais para enfatizar precedência de operadores. Por exemplo:

```
5*3 + 4
```

é melhor do que:

```
5 * 3 + 4
```

ou:

```
5*3+4
```

• Use espaços horizontais para alinhar identificadores numa seção de declaração. Por exemplo,

```
short              var1;
register long int  var2;
```

• Use espaços horizontais para alinhar comentários. Assim, os comentários serão mais fáceis e menos cansativos de ler. Por exemplo,

```
              /* Este comentário é */
/* desagradável de ler */

        /* Mas, este é um comentário */
        /* bem mais aprazível        */
```

6.3.3 Espaços em Branco Verticais

O uso de espaços verticais em branco não apenas facilitam a leitura de um programa como também a tornam menos cansativa. A seguir são apresentados alguns conselhos úteis sobre o uso de espaços verticais.

• Use espaços verticais para separar funções e blocos com alguma afinidade lógica.

- Use espaços verticais em branco para separar seções logicamente diferentes de uma função.
- Use espaços verticais em branco entre declarações ou entre definições e instruções.

Espaços verticais em branco não devem ser utilizados para separar porções do programa que possuem afinidade. Por exemplo, não se deve utilizar espaços verticais para separar:

- Diretivas de pré-processamento de um mesmo tipo; por exemplo, não se devem separar duas diretivas **#include**.
- Alusões a variáveis ou funções entre si.
- Definições de variáveis entre si.

Em qualquer caso, use o bom senso para determinar quando o uso de espaços em branco verticais é adequado. Melhor ainda, solicite a ajuda de outro programador para verificar se sua intuição é correta.

6.4 Escolha de Identificadores

O uso consistente de **convenções** para a criação de identificadores das diversas categorias que compõem um programa facilita bastante sua leitura, pois permite determinar visualmente aquilo que um identificador representa (i.e., se ele representa uma variável, função, macro etc.).

6.4.1 Convenções

A seguir, são resumidas algumas sugestões para a escrita de identificadores que já têm sido utilizadas ao longo do texto.

- **Nomes de variáveis**. Comece com letra minúscula; se o nome da variável for composto, utilize letra maiúscula no início de cada palavra seguinte, inclusive palavras de ligação (por exemplo, preposições e conjunções); não utilize sublinha.
- **Nomes de tipos**. Siga a regra para nomes de variáveis, mas comece com a letra `t` ou termine com `_t`.

• **Nomes de funções**. Utilize a mesma regra para nomes de variáveis, mas comece com letra maiúscula.

• **Nomes de macros**. Utilize sempre letras maiúsculas; se o nome for composto, utilize sublinha para separar os componentes.

6.4.2 Representatividade de Identificadores

Identificadores que exercem papéis importantes no programa devem ter nomes que sejam significativos com relação à função exercida. Por exemplo, uma variável denominada `idadeDoAluno` é muito melhor do que uma variável denominada simplesmente `x`.

Identificadores com importância menor não precisam ter nomes significativos. Por exemplo, uma variável utilizada apenas como variável de controle num laço **for** pode ser nomeada `i` (não precisa ser denominada, por exemplo, `contador`).

Funções que não retornam nenhum valor (exceto, talvez, um código de erro), devem ser nomeadas pelo que fazem. Por exemplo, uma função que executa formatação de texto deve ser denominada `FormataTexto()`. Por outro lado, funções cujo principal objetivo é retornar um valor (por exemplo, resultante de algum cálculo) devem ser nomeadas baseadas naquilo que elas retornam. Por exemplo, uma função que calcula e retorna o valor do fatorial de um número deve ser nomeada como `Fatorial()`, e não como `CalculaFatorial()`. Funções que retornam um dentre dois valores possíveis (por exemplo, ***sim/não,*** ou ***verdadeiro/falso***) podem ser nomeadas começando com `Eh` [representando "é" – por exemplo, `EhValido()`] ou `Sao` [representando "são" – por exemplo, `SaoIguais()`].

Finalmente, não utilize identificadores nem muito longos nem muito abreviados; encontre um meio-termo que seja sensato.

6.5 Recomendações para Escrita de Instruções e Expressões

O uso de abre-chaves na mesma linha inicial de uma estrutura de controle, como é o caso da estrutura **while** acima, tem como vantagem

reduzir o risco de se encerrar acidentalmente a estrutura de controle colocando um ponto-e-vírgula nesta posição (um erro muito freqüente em programação em C). Por exemplo, é muito mais fácil cometer o erro:

```
while (x);    /* O ponto-e-vírgula encerra a instrução while */
{
     ...
```

do que o erro:

```
while (x);{   /* Aqui é mais fácil perceber o erro */
     ...
```

Escreva instruções e expressões que sejam razoavelmente curtas. Isto é, evite instruções que ocupem mais de uma linha. Quando isto for realmente inevitável, endente as linhas seguintes com relação à linha inicial da instrução. Por exemplo, escreva:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
             minhaVariavel4 + minhaVariavel5;
```

em vez de:

```
resultado = (minhaVariavel1 + minhaVariavel2)*minhaVariavel3 -
minhaVariavel4 + minhaVariavel5;
```

Tente escrever blocos que não sejam muito longos e que caibam inteiramente na tela, pois isto facilita a leitura dos mesmos (para depuração, por exemplo).

Evite o uso de expressões condicionais muito longas e complexas. Uma forma de se verificar se uma expressão condicional é muito complexa é utilizar o chamado ***teste do telefone***: leia a expressão em voz alta; se você conseguir entender a expressão à medida que a lê, a expressão passa no teste; caso contrário, se você deixar de acompanhá-la, é melhor dividir a expressão em subexpressões.

6.6 Números Mágicos

Números mágicos são valores numéricos constantes desprovidos de significado próprio e que dificultam o entendimento de um programa. Como exemplos de números mágicos considere:

```
y = 2.54*x; /* 2.54 é um número mágico */
while (i < 10)  /* 10 é um número mágico */
char c = 65; /* 65 é um número mágico */
for (i = 0; i < 20; ++i)  /* 20 é um número mágico, mas 0 não é
```

Em geral, qualquer valor numérico diferente de 0 ou 1 provavelmente poderá ser considerado um número mágico[4]. Números mágicos não possuem significado próprio e, assim, o leitor do programa precisa fazer inferências muitas vezes imprecisas para desvendar o significado deles. Em nome da boa legibilidade, é sempre recomendável que números sejam representados por constantes simbólicas ou constantes que façam parte de uma enumeração (v. **Seção 9.7**). Por exemplo, os números mágicos que aparecem no último exemplo poderiam ser representados por constantes simbólicas do seguinte modo:

```
#define CENTIMETROS_POR_POLEGADA 2.54
#define LIMITE_SUPERIOR 10
#define NUMERO_DE_ELEMENTOS 20
y = CENTIMETROS_POR_POLEGADA*x;
while (i < LIMITE_SUPERIOR)
char c = 'A';
for (i = 0; i < NUMERO_DE_ELEMENTOS; ++i)
```

Deve-se salientar que o uso de uma constante simbólica em substituição a um valor numérico é recomendado ***apenas*** quando o valor têm algum significado próprio. Por exemplo:

[4] Neste livro, aparecem muitos números mágicos em fragmentos de programas apresentados, pois, na maioria destes trechos de programas, não há contexto capaz de prover significado para essas constantes.

```
perimetro = 2*3.14*r;
```

deve ser rescrito como:

```
#define PI 3.14
...
perimetro = 2*PI*r;
```

Note que o valor `3.14`, que tem um significado próprio, foi substituído pela constante simbólica `PI` na instrução, mas o valor `2` permaneceu, visto que ele não tem nenhum significado, simplesmente faz parte de uma fórmula.

Uma situação na qual muitos iniciantes em C teimam em utilizar números mágicos é na manipulação de caracteres. Por exemplo, freqüentemente programadores inexperientes utilizam o seguinte fragmento de programa para processar os caracteres compreendidos entre A e Z:

```
char c;
...
for (c = 65; c <= 90; ++c)
   ...
```

Neste mau exemplo, os números mágicos `65` e `90` não apenas tornam o programa difícil de entender como também o tornam dependente de implementação. Ou seja, o programa não terá portabilidade, pois se está assumindo implicitamente que A é representado por `65` e Z é representado por `90`, o que não é o caso em qualquer código de caracteres[5].

Outra situação na qual números mágicos aparecem com freqüência é na definição e uso de arrays (v. **Capítulo 7**). Aqui, o problema não é apenas de legibilidade, mas também de manutenibilidade. Suponha, por exemplo, que você tenha declarado um array como:

```
float notas[10];
```

e seu programa utilize o valor constante `10` em vários outros pontos.

[5] Lembre-se de que o padrão ISO da linguagem C não especifica qual código de caracteres deve ser utilizado (v. **Seção 1.2.3**).

Se alguém desejar alterar seu programa com o objetivo de aumentar o tamanho do array para 25, terá que decidir quais dos demais valores iguais a `10` distribuídos no programa representam o tamanho do array.

6.7 Uso de Jargões

Um **jargão** numa linguagem de programação é uma construção utilizada de modo preferencial em detrimento de outras construções equivalentes. Jargões são escolhidos tendo em vista os seguintes objetivos: melhora de legibilidade, facilidade de manutenção e proteção contra erros de programação.

Um exemplo comum de jargão é a forma de alinhamento de instruções **if** aninhadas. Isto é, o uso convencional de:

```
if (x < 0) {
    ...
} else if (y > 0) {
    ...
} else if (z == 0) {
    ...
} else
    ...
```

em vez da construção equivalente:

```
if (x < 0) {
    ...
} else
    if (y > 0) {
        ...

 } else
        if (z == 0) {
            ...
        } else
            ...
```

O primeiro formato de instruções **if** aninhadas possui pelo menos duas vantagens: facilita a visualização dos testes efetuados (i.e., avaliações de condições) e em que seqüência isto ocorre. O uso do jargão também evita

que as instruções se desloquem excessivamente para a direta prejudicando, assim, a legibilidade.

Outro uso comum de jargão em C é na escrita de laços de repetição infinitos. Ambas as formas apresentadas a seguir constituem jargões em C:

```
while (1) {
   ...
}

for (;;) {
   ...
}
```

A aprendizagem de uma linguagem de programação envolve a familiarização com os jargões utilizados pela linguagem. Ao longo do texto, procura-se apresentar formatos convencionais de escrita para diversas construções da linguagem C, mas ainda existem muito mais jargões com os quais você deverá deparar-se examinando programas escritos por programadores experientes. Entretanto, mesmo programadores experientes escrevem trechos de programas em formatos não-convencionais por razões idiossincráticas. De qualquer modo, é fácil descobrir se uma dada construção trata-se de um jargão ou não analisando a sua freqüência de uso.

6.8 Interface do Usuário

O estudo daquilo em que consiste numa boa interface do usuário constitui por si só uma disciplina à parte. Este tópico está um pouco fora do escopo deste capítulo, pois, aqui, lida-se com legibilidade em termos de programa e não em termos de usuário. Entretanto, é conveniente que se apresente para o programador iniciante, que talvez desconheça a disciplina de interação humano-computador, um conjunto mínimo de recomendações básicas que devem ser seguidas enquanto ele não aprofunda seu conhecimento sobre o assunto. Estas recomendações básicas são apresentadas a seguir.

- Todo programa interativo deve iniciar sua execução apresentando ao usuário informações sobre o que o programa

faz, seu autor, versão e qualquer outra informação pertinente ***para o usuário*** do programa.

• Toda entrada de dados deve ser precedida por uma indicação (**prompt**) informando ao usuário o tipo de entrada que o programa espera que o usuário introduza e o formato dos dados.

• Se for o caso, o programa deve informar ao usuário qual ação ele deve executar para introduzir certos dados se esta ação não for óbvia. Por exemplo, você não precisa dizer ao usuário para ***pressionar uma tecla*** para introduzir um caractere, basta dizer ***digite um caractere***. No entanto, precisa informar–lhe como executar uma operação não usual, tal como ***pressione simultaneamente as teclas*** `ALT`, `SHIFT` ***e*** `A`, em vez de ***digite*** `ALT-SHIFT-A`.

• Toda saída de dados deve ser compreensível para o usuário. Lembre-se de que o usuário pode não ser versado em computação ou programação. Portanto, não utilize termos próprios da área.

• O programa deve informar ao usuário o que ele deve fazer para encerrar o programa ou uma dada iteração.

• Se um determinado processamento for demorado, informe ao usuário que o programa está executando a tarefa. Não deixe o usuário sem saber o que está acontecendo.

• Tenha sempre em mente que o programa deverá ser usado por ***usuários comuns***. Portanto, não faça suposições sobre o nível intelectual dos usuários.

Você deve criar um **manual do usuário** contendo instruções sobre como usar seu programa. Este manual não deve conter detalhes técnicos; i.e., ele deve ser dirigido para um usuário leigo em programação.

6.9 Guias de Estilo de Programação em C

As recomendações de estilo apresentadas aqui não são suficientes para constituir um manual de estilo completo. Uma fonte interessante e acessível sobre estilo de programação em C é o documento conhecido como *Indian Hill*, escrito por programadores da AT&T, encontrado facilmente na internet (v. **Bibliografia**).

Existem inúmeros outros guias de estilo de programação em C e em outras linguagens disponíveis na internet, mas é preciso ter cautela quando procurar seguir as orientações contidas em alguns desses guias, pois muitos deles refletem questões de gosto pessoal, sem nenhuma comprovação empírica de utilidade. Como recomendação prática, procure guias de estilo que demonstrem ser amplamente citados.

6.10 Teste e Depuração de Programas

Antes de começar a depurar um programa, você deve testá-lo; i.e., assegurar que cada entrada corresponde a uma resposta correta do programa, sem efeitos inesperados. Testar um programa significa verificar se o mesmo funciona conforme o esperado (i.e., conforme foi especificado no projeto do programa). Normalmente, qualquer programa não-trivial possui erros de programação. Portanto, o objetivo maior de testar um programa é o de encontrar erros que impeçam o seu funcionamento normal. Outro objetivo importante dos testes é verificar a eficiência do programa em termos de tempo de resposta e quantidade de memória utilizada pelo mesmo.

6.10.1 Testando um Programa

Em engenharia de software, existem duas fases principais de testes: **alfa** e **beta**. Na fase alfa, os testes são realizados por programadores e engenheiros que compõem a equipe de desenvolvimento do produto, enquanto na fase beta, os testadores são usuários voluntários que utilizam o programa em situações reais. Idealmente, a fase beta deve começar apenas quando a fase alfa estiver encerrada.

Quando um programa entra em fase de testes ele deve ser ***congelado***, no sentido de que as únicas modificações no programa devem ser realizadas apenas com o objetivo de corrigir erros de programação (i.e., adições de novas características não devem ser permitidas durante a fase de testes).

Técnicas utilizadas para testes ***alfa*** de programas complexos fazem parte de uma disciplina isolada em si, e uma apresentação completa destas técnicas está além do escopo deste livro. Portanto, aqui, serão descritas apenas três das técnicas mais comuns e fáceis de ser implementas.

A primeira técnica é conhecida como **inspeção de programa** e consiste em ler atenciosamente o programa e responder a uma lista de verificação contendo questões referentes a erros comuns em programação na linguagem de codificação do programa.

A segunda técnica comum de verificação de programas é conhecida como **teste exaustivo** e consiste em utilizar dados típicos de entrada e simular manualmente a execução do programa. Num teste exaustivo, é importante que sejam utilizados casos de entrada qualitativamente diferentes. Também, é igualmente importante que sejam testadas não apenas entradas válidas, mas também algumas entradas inválidas.

Um **teste de unidade** consiste em testar pequenas seções de código antes de integrá-las ao programa. Assegurar que esta pequena porção de programa funciona reduz a possibilidade de o programa inteiro não funcionar. Ao invés de incluir uma dúzia de funções no programa e testá-las de uma vez, existe maior probabilidade de que o conjunto funcionará corretamente se cada função for testada individualmente.

6.10.2 Depuração de Programas

Nem mesmo os programadores mais experientes escrevem programas livres de erros em sua primeira tentativa. Assim, uma grande parcela do tempo gasto em programação é dedicada à tarefa de encontrar e corrigir erros. **Depurar** um programa significa localizar e consertar trechos do programa que provocam seu mau funcionamento. Apesar de estarem intimamente relacionados, teste e depuração de um programa não significam a mesma coisa. Um bom teste deve ser capaz de apontar um

comportamento anormal de um programa, mas não indica precisamente aquilo que causa tal comportamento. Por outro lado, a depuração deve determinar precisamente as instruções que causam o mau funcionamento do programa e corrigi-las, reescrevendo-as ou substituindo-as.

6.10.3 Classificação de Erros de Programação

Erros de programação são usualmente classificados em três categorias: erro de compilação, erro de execução e erro de lógica.

Um **erro de compilação** (ou **erro de sintaxe**) ocorre devido a uma violação das regras de sintaxe da linguagem C. Um programa contendo erros de sintaxe não pode ser nem compilado nem executado. Causas comuns para este tipo de erro são:

- Falhas de digitação
- Omissão de ponto-e-vírgula
- Referência a variáveis que não foram declaradas
- Chamada de uma função com um número errado de argumentos ou com argumentos de tipos incompatíveis com aqueles na definição da função
- Atribuição de um valor de tipo incompatível para uma variável.

Erros de compilação são relativamente fáceis de ser corrigidos, apesar de, muitas vezes, o compilador não indicar precisamente a instrução incorreta e o tipo de erro que ocorreu. Entretanto, o programador deve observar as seguintes regras quando analisar mensagens de erros apresentadas por um compilador:

- O primeiro erro apontado pelo compilador é sempre um erro real, mas erros subseqüentes podem não corresponder a outros erros legítimos. Isto é, estes erros subseqüentes podem ser apontados simplesmente em conseqüência da ocorrência do primeiro erro detectado pelo compilador.

- Um erro de compilação pode ser causado por qualquer linha de código acima da linha indicada pelo compilador, mas não pode jamais ser causado por uma linha abaixo daquela indicada pelo compilador.

Um **erro de execução** não impede um programa de ser compilado, mas faz com que a execução do mesmo seja interrompida de maneira anormal (algumas vezes causando até mesmo a falha de todo o sistema operacional no qual o programa está sendo executado). Um exemplo comum deste tipo de erro é uma tentativa de divisão de um valor inteiro por zero.

Erro de lógica[6] é um erro que nem impede a compilação nem acarreta interrupção da execução de um programa. Entretanto, um programa contendo um erro deste tipo não funciona conforme o esperado. Por exemplo, o usuário solicita que o programa execute uma determinada tarefa e o programa não a realiza satisfatoriamente. Um erro de lógica ocorre quando o algoritmo utilizado é incorreto, mesmo que este tenha sido implementado corretamente, ou quando o algoritmo é correto mas sua implementação é equivocada. Às vezes, o algoritmo pode ser tão ruim que nenhuma ferramenta de depuração, por melhor que seja, é capaz de consertá-lo ou melhorá-lo.

6.10.4 Poupando Depuração

Depuração é uma atividade muito mais difícil do que a escrita de programas. Em resumo, depuração requer paciência, criatividade, esperteza e, principalmente, muita experiência por parte do programador. Idealmente, além de possuir profundo conhecimento sobre a linguagem utilizada, o programador que atua na depuração de programas deve possuir outros conhecimentos, como conhecimento de compiladores, assembly e arquitetura de computadores, que transcendem a tarefa básica de construção de programas.

[6] A palavra *lógica* aqui é utilizada como sinônimo de *raciocínio*. Isto é, um erro de lógica é um erro na concepção ou implementação de um algoritmo.

Levando em consideração as prováveis dificuldades que tipicamente cercam as atividades de depuração, é mais sensato tentar evitar que estas atividades se façam necessárias. Infelizmente, alguns programadores adotam, equivocadamente, uma estratégia contrária a esse argumento. O raciocínio utilizado por estes maus programadores é aproximadamente o seguinte:

1. Tão logo o programador adquire uma vaga idéia do problema em mãos, ele constrói um programa para resolvê-lo.
2. Então, ele verifica se o programa funciona com alguns poucos casos de teste.
   2.1 Se o programa funcionar, ele acha que está tudo bem.
   2.2 Se o programa não funcionar, ele passa a depurá-lo até que ele seja aprovado nos testes de verificação. (Aqui, provavelmente, o mau programador despenderá muito mais tempo do que na escrita do programa.)

A abordagem apresentada aqui é equivocada em termos de esforços porque ela transfere para a fase de depuração, que é exatamente a mais árdua, a tarefa de pôr o programa em funcionamento. No restante desta seção, serão apresentadas algumas atitudes que um bom programador deve adotar para poupar atividades de depuração.

Para poupar tempo, antes de iniciar uma sessão de depuração, certifique-se de que:

• *Você entende realmente o algoritmo seguido pelo programa*. É praticamente impossível depurar um programa cujo funcionamento não seja completamente entendido.

• *O programa foi compilado usando o nível máximo de apresentação de mensagens de advertência* (no gcc, usando `-Wall`). Além disso, ***todas as mensagens de advertência emitidas pelo compilador foram atendidas***. Quando utilizado com esta opção, o compilador é capaz de apontar muitas causas de possíveis erros (v. **Seção 6.11.1**).

• *O programa lint foi utilizado com cada arquivo-fonte que constitui o programa* (v. **Seção 6.11.1**). Atente às mensagens emitidas por este programa, pois ele é capaz de apontar causas de possíveis erros.

• *O programa-fonte foi analisado com o auxílio de uma lista de verificação de programas contendo questões relacionadas a erros comuns de programação em C*. Existem muitos erros de programação que são comuns; verificando se algum destes erros ocorre em seu programa você pode fazê-lo economizar muito tempo. No **Volume II**, encontra-se uma extensa lista de erros que freqüentemente ocorrem em programas escritos em C.

• *Outros programadores examinaram seu programa*. Não se iluda pensando que o erro não se encontra numa determinada seção do programa simplesmente porque você já a examinou várias vezes. Do mesmo modo que um texto pode conter erros evidentes que o escritor não percebe, um programa pode conter erros que são óbvios para terceiros mas o programador não consegue notá-los.

• *Verificou as expressões relacionais para ver se, involuntariamente, você utilizou o operador de atribuição em vez do operador relacional de igualdade*. Quando uma variável assume um valor inesperado, esta causa freqüente de erro é uma das principais suspeitas.

• *Todos os acessos a elementos de arrays* (v. **Capítulo 7**) *e conteúdos apontados por ponteiros foram cuidadosamente examinados*. Essas são causas muito comuns de alteração indevida de memória.

• *O compilador não introduziu erros em seu programa executável*. Compiladores são programas e, portanto, também contêm bugs. Consulte *FAQs* e listas de discussão na internet para verificar se o problema em seu programa não é causado pelo compilador utilizado.

Um **histórico de modificações** pode ajudar a detectar erros. Se você souber quando um defeito foi introduzido, pode consultar o histórico de modificações para determinar qual alteração pode ter causado o erro. Um histórico de modificações não é útil apenas para grandes projetos de programação. Existem programas, denominados **sistemas de controle de versões** (*Version Control Systems* – VCS, em inglês), capazes de produzir históricos de modificações automaticamente. Programas VCS permitem que se acompanhe todas as alterações feitas no código-fonte e podem automaticamente reproduzir o código-fonte como ele era algum tempo atrás.

6.11 Técnicas Elementares de Depuração

A etapa mais difícil de depuração de um programa consiste em localizar precisamente a instrução ou o conjunto de instruções que causa o mau funcionamento do programa. Como já foi dito anteriormente, encontrar erros sintáticos não é difícil, mesmo quando o compilador não é capaz de apontá-los com precisão e, portanto, este tópico não merece maiores considerações.

A seguir, serão apresentadas algumas técnicas comuns utilizadas para localizar causas de erros lógicos e de execução em programas. Mas, antes de utilizar alguma destas técnicas, é importante que o programador determine precisamente a natureza do erro e quando o mesmo ocorre (por exemplo, sempre que o programa recebe tal entrada, ele apresenta tal e tal comportamento). A situação ideal ocorre quando o programador é capaz de reproduzir o erro sempre que introduz dados possuindo as mesmas características (i.e., quando o erro não é aleatório, mas sistemático).

As técnicas descritas a seguir devem servir apenas como um guia introdutório. À medida que você se tornar um programador experiente, será capaz de desenvolver suas próprias técnicas e utilizar versões mais sofisticadas do que aquelas apresentadas aqui. Se você não tiver um depurador disponível, use as técnicas descritas aqui como alternativas.

6.11.1 Uso do Compilador e de Lint

O compilador não é propriamente uma ferramenta de depuração, mas pode ser seu primeiro aliado na luta contra o surgimento de erros de programação. Isto é, o uso preventivo do compilador pode ajudá-lo a evitar que alguns erros ocorram antes mesmo de o programa ser executado pela primeira vez. Para utilizar o compilador de forma preventiva, utilize-o sempre com a opção de apresentação de todas as mensagens de advertência. Examine cuidadosamente cada mensagem de advertência emitida pelo compilador e corrija todas as instruções que correspondam a uma dada advertência, mesmo que você tenha certeza que elas não causarão problemas. Agindo de modo contrário, uma mensagem de advertência importante poderá deixar de ser notada.

Como exemplo da importância de seguir esta recomendação, considere o programa apresentado a seguir, que, apesar de ser bem simples, apresenta um erro comum entre iniciantes em programação em C:

```
#include <stdio.h>

int x;

int main()
{
  printf("Digite um numero inteiro: \n");
  scanf("%d", x);

  printf("Valor introduzido = %d\n", x);

  return 0;
}
```

Quando este programa é compilado utilizando o compilador gcc sem o uso da opção `-Wall` e, em seguida, executado, ele é abortado. Mas, se você compilar este programa utilizando esta opção, como:

```
gcc -g -Wall -std=c99 Advertencia.c -o Advertencia
```

o compilador gcc emite a seguinte mensagem de advertência:

```
Advertencia.c:18: warning: format '%d' expects type 'int *', but
argument 2 has type 'int'
```

Esta mensagem informa que, na chamada da função **scanf()** que se encontra na linha 18 do arquivo `Advertencia.c`, o especificador de formato `%d` espera que o tipo da variável correspondente seja **int *** (i.e., o endereço de uma variável do tipo **int**), mas o tipo do parâmetro utilizado foi **int**. Assim, se você der a devida atenção a esta mensagem, provavelmente notará que o erro cometido foi ter-se utilizado `x` em vez de `&x`.

Uma outra maneira de prevenir erros de programação é utilizando o programa **lint**. Este programa utilitário, originalmente encontrado no sistema operacional Unix, serve para detectar possíveis fontes de erros e construções

não-portáveis em programas escritos em C[7]. Atualmente, o programa lint mais utilizado é **splint**, que existe em versões para Windows e Linux.

Assim como as mensagens de advertência apresentada pelo compilador, os alertas apresentados por um programa lint podem revelar vulnerabilidades de um programa. Assim, as mesmas recomendações apresentadas acima com relação ao uso do compilador são válidas para o uso de lint.

A utilização de lint é extremamente simples: basta utilizar, na linha de comando, o nome do programa lint seguido dos nomes do arquivos-fonte que você deseja que sejam checados. O programa lint original e outros semelhantes também aceitam a especificação de múltiplos arquivos com a utilização de caracteres curingas. Por exemplo:

```
splint *.c
```

invoca o programa splint para checar todos os programas com extensão `.c` que se encontram no diretório corrente.

Se o programa splint for utilizado para analisar o último exemplo de programa acima:

```
splint Advertencia.c
```

ele apresentará o seguinte resultado[8]:

```
Advertencia.c:20:15: Format argument 1 to scanf (%d) expects int
* gets int: x
Type of parameter is not consistent with corresponding code in
format string.

Advertencia.c:20:3: Return value (type int) ignored: scanf("%d",
x)
Result returned by function call is not used. If this is intended,
can cast result to (void) to eliminate message.
```

[7] Nos dias atuais, o nome ***lint*** tem sido empregado genericamente para identificar qualquer ferramenta utilizada na análise de programas escritos em qualquer linguagem com os mesmos objetivos do programa lint original.

[8] Parte do conteúdo considerado irrelevante para a discussão que segue foi removida e o restante foi formatada, para melhor visualização.

```
Advertencia.c:15:5: Variable exported but not used outside
Advertencia: x

A declaration is exported, but not used outside this module.
Declaration can use static qualifier.
```

A primeira mensagem apresentada pelo splint é equivalente àquela apresentada pelo gcc no exemplo anterior. A segunda mensagem chama atenção para o fato de o valor retornado pela função **scanf()** não ser utilizado. Finalmente, a terceira mensagem informa que a variável `x` tem escopo global, mas é utilizada apenas no arquivo que a contém.

Observe que, dentre as três mensagens apresentadas pelo splint, apenas a primeira é realmente importante para o programa em questão. Mesmo assim, é sempre importante dar atenção a todas as mensagens apresentadas. No mínimo, você aprenderá um pouco mais sobre como melhorar seu estilo de programação em C.

6.11.2 Uso de printf()

A função **printf()** e algumas outras funções de saída constituem uma ferramenta bastante útil em depuração. Existem dois usos principais de **printf()** em depuração:

- Para examinar o valor de uma ou mais variáveis em vários pontos do programa.
- Para verificar o fluxo de execução do programa (por exemplo, para determinar se uma determinada instrução é executada).

A técnica consiste em distribuir chamadas de **printf()** em vários pontos estratégicos do programa. Enquanto distribui chamadas de **printf()** pelo programa, certifique-se de que você será capaz de distinguir cada uma destas chamadas quando a mesma for executada. Como exemplos de uso de **printf()** na depuração de um programa têm-se:

```
printf("Valor de x antes da instrucao tal e': %f.\n", x);
printf("Valor de x apos o segundo while e': %f.\n", x);
printf("Instrucoes seguindo o else do 3o. if executadas.\n");
```

No caso de um programa que é abortado devido a um erro de execução, as chamadas de **printf()** que foram executadas, evidentemente, estão antes do erro que causou o aborto.

6.11.3 Uso de Comentários

Comentários são utilizados para excluir da compilação um trecho de programa que esteja sob suspeita de estar causando o mau funcionamento do programa. Esta técnica de depuração funciona da seguinte maneira:

(1) O programa apresenta um comportamento inesperado e você suspeita que um determinado trecho do programa esteja provocando este comportamento indesejável.

(2) Comente este trecho de programa para excluí-lo do código executável e veja como o programa resultante se comporta. Talvez algumas outras adaptações no programa, como, por exemplo, retirada de comentários preeexistentes, sejam necessárias antes de recompilá-lo.

(3) Se o programa continuar apresentando o mesmo erro, é provável que este erro não seja provocado pelo trecho de programa comentado. Assim, você deve procurar o erro em outro local do programa.

(4) Se o programa não apresentar o mesmo erro, é provável que sua conjetura sobre a causa do erro tenha sido adequada e que o trecho de programa comentado seja realmente o causador do erro. Se este trecho for grande ao ponto de não permitir identificar exatamente qual a instrução causadora do erro, repita o procedimento a partir do passo (2), mas agora comente um trecho de programa menor dentro da porção de programa anteriormente comentada.

O procedimento aqui descrito constitui um caso de abordagem mais geral de depuração (ou, mais precisamente, de busca de erros) denominada **busca binária**. Esta abordagem pode ser utilizada em conjunto com todas as técnicas de depuração descritas aqui.

6.11.4 Uso de Asserções e Compilação Condicional

A compilação condicional é útil em depuração, uma vez que ela permite incluir ou excluir trechos de programa de acordo com o valor de uma macro que indica se o programa está em processo de depuração.

Quando o programa está em fase de depuração, esta macro é definida com 1 e é utilizada para incluir trechos de programa utilizados exclusivamente para depuração. Quando a fase de depuração é encerrada, redefine-se esta macro com 0 e aqueles trechos utilizados apenas na depuração do programa não serão incluídos no código final[9]. Conseqüentemente, o código gerado pelo compilador quando a fase de depuração estiver encerrada será menor do que durante aquela fase.

O uso da macro `ASSERT`, apresentada na **Seção 5.3**, para verificar se certas condições são satisfeitas em vários pontos de um programa, também é uma ferramenta extremamente útil em depuração. Aqui, esta macro será definida de modo mais sofisticado em conjunção com compilação condicional, como:

```
#define  DEBUG  1

#if  DEBUG
#define ASSERT(x)  if (!(x) ) {\
             printf("A condicao %s na linha %d do arquivo %s
falhou.", \
                      #x, __LINE__, __FILE__); \
           exit(1); \
         }
#else
#define ASSERT(x) 0
#endif
```

A diferença entre esta definição e a definição da macro `ASSERT` apresentada na **Seção 5.3** é que, nessa nova versão, quando a macro `DEBUG` expande-se em 0, indicando final da fase de depuração, a macro `ASSERT` expande-se simplesmente em 0.

---

[9] Alternativamente, a macro pode ser simplesmente definida como vazia durante a fase de depuração. Encerrada esta fase, pode-se remover a definição com **#undef** ou comentando-a. Na prática, é mais fácil implementar o que é apresentado aqui.

Uma asserção expressa uma condição que deve ser verdadeira num dado ponto de um programa. Suponha, por exemplo, que seu programa contenha uma divisão `x/y` que deve sempre ser executada. Então, para que não ocorra erro de execução, a variável `y` deve ser diferente de zero no instante da divisão. Você poderia assegurar isto incluindo uma asserção imediatamente antes desta divisão, como no exemplo a seguir:

```
ASSERT (y != 0);
z = x/y;
```

Com a inclusão desta asserção, quando `y` assumisse o valor 0 indevido, o programa seria abortado e uma mensagem de erro seria emitida indicando a linha e o arquivo nos quais a expressão `y != 0` deixou de ser satisfeita. É importante notar a diferença entre o uso de uma asserção como a do exemplo acima e o simples uso de um **if** aparentemente eqüivalente. Por exemplo, se a chamada de `ASSERT` fosse substituída pela instrução **if**:

```
if (y != 0)
    z = x/y;
```

talvez não fosse possível determinar quando a instrução `z = x/y` é executada ou não.

Inserindo asserções em diversos pontos do programa, você pode detectar bugs muito mais rápido e facilmente do seria o caso utilizando um depurador. A regra básica para decidir sobre que condição usar com uma asserção precedendo uma dada instrução consiste em responder à pergunta: Que propriedades devem ser verdadeiras para tal instrução funcionar adequadamente?

A compilação condicional também é útil quando utilizada em conjunto com **printf()**. Isto é, em vez de usar simplesmente **printf()**, é aconselhável utilizar uma macro, como:

```
#ifdef DEBUG
#define IMPRIME(msg, exp) printf("%s %s = %d", msg, #exp, exp)
#else
#define IMPRIME(msg, exp) 0
```

Supondo que x é uma variável do tipo **int**, esta macro poderia ser utilizada do seguinte modo:

```
IMPRIME("Valor no interior do laco while de ", x);
```

A vantagem de utilizar macros como a apresentada acima em vez de chamadas **printf()** é que você não tem que removê-las manualmente quando encerrar as fases de teste e depuração do programa.

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 nível**. 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 as 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*** no 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

A 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. No entanto, usuários de Windows também podem usar essas ferramentas de desenvolvimento em conjunto com o programa Cygwin.

6.13.1 Preparação

Na sessão de depuração a 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 algum 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, 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[10]:

[10] 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 este 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. a seguir).**

Para que um programa possa ser depurado usando um depurador, ele deve ter incluídas 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, tornando-o 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 é mensagem 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[11], 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

[11] 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`.

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 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.

6.13.2 Executando gdb

Para executar gdb com a finalidade de depurar um programa, navegue até o diretório[12] 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[13]:

```
(gdb) h
```

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

[13] 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.

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 destes 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
```

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 capturada pelo programa[14].

A segunda linha apresentada pelo gdb parece ainda mais enigmática:

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

[14] O tratamento de sinais é abordado no **Volume II**, mas este conhecimento não é essencial no presente contexto.

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 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, por meio do uso do 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()** em vez 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 estar 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 foi novamente 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, 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 por que a correção adotada resolve o problema. Certifique-se ainda de 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[15]. 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 que um arquivo core realmente foi criado no diretório onde o programa está sendo executado[16]. 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:

---

[15] 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, e 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.

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

```
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[17]. Entretanto, 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:

```
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) )
```

[17] 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.

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 ele ainda não foi encontrado, de modo que se possa explorar mais algumas facilidades oferecidas pelo gdb.

6.13.9 Instalando e Removendo Breakpoints

Suponha que, neste instante, tudo o que se saiba 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 isso, 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[18]:

```
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***.

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,

[18] 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.

não é recomendável ter muitos ***breakpoints*** ativos numa mesma sessão de depuração, o comando `delete` é mais utilizado. Este comando tem a seguinte sintaxe[19]:

```
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[20]:

```
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 que o 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 moda Unix. Se você não acredita que o ***breakpoint*** foi realmente removido, digite novamente o comando:

```
i b
```

---

[19] 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.

[20] 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.

e o gdb responderá:

```
No breakpoints or watchpoints.
```

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, 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[21]:

```
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 o seguinte como resposta:

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

[21] 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.

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
```

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 seqüencialmente 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 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 28.**

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 28: Principais comandos de execução do depurador gdb

Conforme mostra a **Tabela 28**, 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 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 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 você realmente 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 de 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 esteja informando que o programa ainda está em execução. A verdade é que a informação anterior do gdb indica que o programa ***será*** abortado e não que já ***foi*** 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, em vez 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 moda Unix[22].

[22] 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, precisa informar ao 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[23]:

```
$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) )
```

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

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 indica o gdb:

```
18              divisor--;
```

Agora, suponha que você deseje 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`).

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, 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, pode-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 utilizando 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, 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 não apenas estará adquirindo experiência no uso de depuradores de alto nível como também entenderá melhor como o compilador traduz programas e como estes são executados.

6.14 Depuração Usando ddd

O programa ddd, cujo nome é derivado de *Data Display Debugger*, foi desenvolvido para o sistema operacional Linux[24] e consiste em uma interface gráfica (*front-end*) para o depurador gdb bem como outros depuradores. O objetivo do ddd é tornar mais confortável a depuração de programas. Por exemplo, o ddd permite que se instale um ***breakpoint*** simplesmente localizando a instrução desejada e clicando sobre a mesma com o mouse. Entretanto, este programa não acrescenta nenhuma funcionalidade adicional a nenhum depurador.

A **Figura 22** mostra o ddd sendo utilizado na depuração do programa apresentado na **Seção 6.13**.

[24] Nenhuma das principais distribuições de Linux instala automaticamente o ddd, de modo que a instalação deste programa deve ser efetuada pelo administrador do sistema.

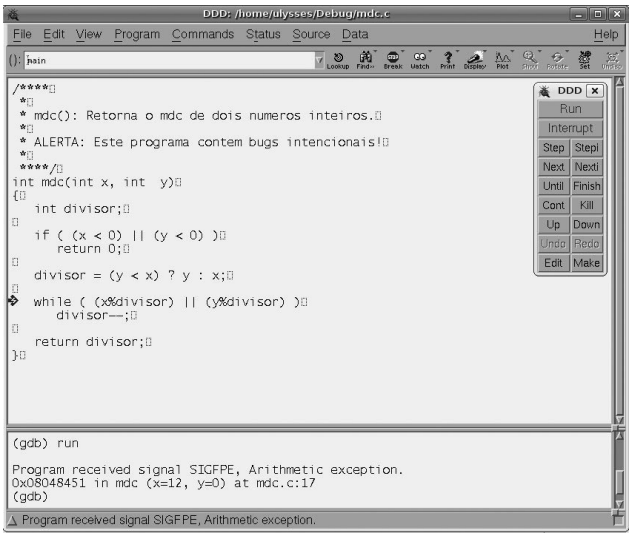

Figura 22: Interface ddd em uso

Observe, à direita da **Figura 23,** que o ddd possui uma janela flutuante contendo alguns dos principais comandos do gdb, conforme mostra a imagem ampliada na **Figura 23**.

Figura 23: Janela de comandos do ddd

Se você aprender a utilizar o gdb, não terá dificuldades em aprender também o ddd. Caso contrário, o ddd será totalmente inútil.

Uma recomendação de cautela, que deve ser levada em consideração se você decidir utilizar a interface ddd, é que, como ocorre com muitos programas do mundo Linux, algumas versões deste programa não são estáveis e podem falhar com relativa facilidade. Outro ponto negativo do ddd, que também aflige outros depuradores que utilizam interfaces gráficas, é que ele consome muito mais recursos computacionais do que depuradores de linha de comando.

6.15 Otimização de Código

Um programa com código otimizado é, em princípio, mais eficiente em termos de rapidez e uso de memória do que um programa não-otimizado. Entretanto, otimizações fazem muito pouca diferença quando o próprio programa original (i.e., sem otimização) já utiliza poucos recurso de memória e CPU. A otimização de um programa pode ser feita manualmente pelo programador ou pelo compilador.

A maioria dos compiladores modernos oferece diversas opções de otimização de código. No compilador gcc, as opções de otimização começam com `-O` (letra "o") e terminam com um número que varia de 1 até um valor que depende do número de otimizações oferecidas pela corrente versão do compilador. Quanto maior for este número, mais sofisticada é a otimização (e maior a probabilidade de conter bugs). Por exemplo, o comando:

```
gcc -O2 main.c Arq1.c Arq2.c -o MeuProg
```

compila um programa utilizando o segundo nível de otimização do gcc.

Às vezes, algumas implementações de algoritmos de otimização usados por compiladores apresentam bugs que fazem com que programas otimizados por elas também apresentem bugs. Portanto, é importante que, após otimizar seu programa, você execute os mesmos testes realizados com ele antes da otimização. Se o programa não-otimizado

funcionava satisfatoriamente bem e depois da otimização passou a não mais funcionar a contento, desconfie da otimização. Assim, otimizar um programa utilizando um compilador nem sempre é uma idéia tão boa quanto parece. Um programa otimizado é susceptível a erros introduzidos pelo compilador e estes erros muitas vezes só podem ser corrigidos utilizando depuração de baixo nível.

Um programa pode ser otimizado manualmente por meio da escolha de algoritmos ou construções da linguagem que ofereçam melhor desempenho. Em qualquer caso, o programador deve começar o processo de otimização apenas quando tiver certeza que o programa atual funciona satisfatoriamente. Também é importante anotar todas as alterações efetuadas no programa desde a última versão em perfeito estado de funcionamento. Um programa de controle de versões (VCS) pode auxiliar muito nesta última tarefa.

Um modo de tornar um programa mais eficiente é determinar quais funções consomem mais tempo de processamento e, então, otimizá-las. Uma ferramenta de desenvolvimento capaz de realizar a primeira destas tarefas é um **profiler**, que é um programa utilizado para analisar o desempenho de outros programas. Em particular, um ***profiler*** é capaz de determinar o tempo de CPU gasto e a freqüência de execução de cada função de um programa. Um ***profiler*** registra estes dados num arquivo que pode ser posteriormente analisado.

Antes de pensar em otimizar um programa, certifique-se de que realmente o programa contém gargalos que comprometem seu bom funcionamento. Para tal, utilize um programa ***profiler*** para determinar o impacto de cada função no desempenho geral do programa e otimize apenas, se for realmente necessário, as funções que são críticas para um melhor desempenho do programa.

6.16 Exercícios de Revisão

1. (a) Por que algumas instruções são endentadas num programa em C? (b) Endentação é absolutamente necessária num programa em C?

2. (a) O que são comentários de bloco e onde eles devem ser utilizados? (b) O que são comentários de linha e onde eles devem ser utilizados?

3. Qual é a melhor ocasião para escrever comentários num programa e por quê?

4. Por que o uso de comentários irrelevantes pode prejudicar a legibilidade de um programa?

5. Comentários que não correspondem ao código que eles tentam explicar são altamente prejudiciais à legibilidade de um programa. Apresente situações nas quais o programador pode, por falta de atenção ou conhecimento, introduzir comentários desta natureza num programa.

6. Critique a seguinte reflexão: *Usar um comentário ruim é pior do que não usar nenhum comentário*.

7. No contexto de legibilidade de programas, o que é um número mágico?

8. (a) O que é um jargão de uma linguagem de programação? (b) Que vantagens são obtidas por meio do uso de jargões?

9. Reescreva as seguintes construções de C utilizando formas mais convencionais (i.e., jargões):

(a)

```
int i = 0;
while (i <= 10) {
   printf("i = %d", i);
   ++i;
}
```

(b)

```
int i = 0;
while (i <= 100)
   printf("i = %d", i++);
```

(c)

```
for (int i = 0; i++ < 15; )
   printf("i = %d", i - 1);
```

(d)

```
while (-5) {
   ...
}
```

(e)

```
if (x < 0) {
    return 0;
} else
    if (y > 0 || x == 25) {
        return -1;
    } else
        if (z >= 0) {
            return -2;
        } else
            return -3;
```

(f)

```
for (int i = 0; i++ < 10; x += delta) {
   printf("\nDigite o valor de delta: ");
   scanf("%f", &delta);
}
```

(g)

```
if (valor != OK)
    return valor;
return OK;
```

10. Que números mágicos aparecem nas instruções do exercício anterior?

11. Escreva o trecho de programa a seguir de uma forma mais convencional:

```
while(valor != 0)
{
   valor = -1;

   printf("Introduza o proximo numero: ");
   scanf("%d", &valor);
   LimpaBuffer();

   if(valor == 0)
   {
      continue;
   }

   if(valor < 0)
   {
      printf("%d nao e' um valor valido.\n",valor);
      continue;
   }

   else
   {
      menor == 0 ? menor = valor : 0;
      n += 1;
      media += valor;
      maior < valor ? maior = valor : 0;
      valor < menor ? menor = valor : 0;
   }
}
```

12. Como são classificados os tipos de erros de programação?

13. Qual é a diferença entre teste e depuração de programas?

14. Como funciona o método de busca binária em depuração?

15. Para que serve um programa lint?

16. O que é um programa *profiler*?

17. O que é e para que serve um histórico de modificações de um programa?

18. Para que serve um programa de controle de versões (CVS)?

19. (a) Diferencie depuradores de alto nível de depuradores de baixo nível.

(b) Por que depuradores de baixo nível são mais difíceis de utilizar do que depuradores de alto nível?

20. (a) Como funciona a técnica de depuração que utiliza **printf()**? (b) Compare esta técnica de depuração com a técnica de depuração que faz uso de comentários.

21. Explique como funciona em depuração o uso da macro `ASSERT` .

22. Que vantagens são obtidas com o uso de compilação condicional em depuração de programas?

23. O que significa um ponto de parada (***breakpoint***) em depuração de alto nível?

24. Qual é a diferença entre os comandos ***step over*** e ***step into*** num depurador de alto nível?

25. O programa original utilizado como exemplo na **Seção 6.13.1** é abortado quando a instrução:

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

é executada. Mas, se as variáveis `a` e `b`, que, no programa em questão, têm escopo de arquivo, forem declaradas dentro da função **main()**, passando a ter escopo de bloco, o aborto do programa nem sempre irá acontecer.

(a) Explique por que isso acontece.

(b) Supondo que as variáveis `a` e `b` tivessem sido declaradas com escopo de bloco, no corpo da função **main()**, como você utilizaria o gdb para determinar o erro apresentado pelo programa?

(**NB**: Nas duas situações o programa continua errado. Mas, na situação original, ocorre um erro de tempo de execução, enquanto na segunda situação provavelmente ocorrerá um erro lógico, já que, apesar de o programa não ser abortado, os resultados apresentados não serão corretos.)

6.17 Exercícios de Programação

EP6.1) Execute um programa lint com os arquivos originais `main.c` e `mdc.c` utilizados na **Seção 6.13** e interprete as mensagens de advertência emitidas pelo programa. Em seguida responda à seguinte pergunta: Algum erro apresentado pelo programa poderia ser evitado levando em consideração as mensagens emitidas pelo programa lint?

EP6.2) Utilize o programa apresentado na **Seção 6.13** para explorar outros comandos do depurador gdb. Em particular, experimente e explique o uso dos seguintes comandos:

(a) `disable`

(b) `enable`

(c) `cond`

(d) `watch`

(e) `break +n`

(f) `break -n`

(g) `info locals`

(h) `shell`

(i) `until`

EP6.3) Utilize o gdb e a versão corrigida do programa apresentado na **Seção 6.13** para executar o seguinte experimento:

1. Instale um ***breakpoint*** na instrução que contém a chamada da função `mdc()`.

2. Inicie a execução do programa no gdb.

3. Antes de o programa chamar a função `mdc()`, utilize o comando `print` para imprimir os valores das variáveis `a` e `b` e dos endereços destas variáveis.

4. Utilize o comando `step` para fazer com a execução do programa pare na primeira da função `mdc()`.

5. Utilize o comando `print` para imprimir os valores dos parâmetros `x` e `y` e dos endereços destes parâmetros.

6. Compare os valores e os endereços dos respectivos parâmetros reais (`a` e `b`) e formais (`x` e `y`) na chamada da função `mdc()`.

Que conclusão você é capaz de tirar deste experimento?