MAKRON
Books

Parte 4

Desenvolvimento de Software Usando C

Esta parte examina diversos aspectos do processo de desenvolvimento de software no que se refere ao ambiente de programação de C. O Capítulo 25 cobre a utilização de sub-rotinas em linguagem assembly e otimizações. O Capítulo 26 fornece uma visão geral do processo de desenvolvimento utilizando C. Por último, o Capítulo 27 fornece uma noção de portabilidade, eficiência e depuração.

MAKRON Books

25

Interfaceamento com Rotinas em Linguagem Assembly

Mesmo com toda a capacidade de C, existem momentos em que você precisa escrever uma rotina utilizando assembler. A maneira de fazer isso varia de compilador para compilador, mas o processo geral, descrito neste capítulo, aplica-se à maioria dos compiladores.

A interface entre C e a linguagem assembly é afetada fundamentalmente por dois fatores: o tipo de CPU e as convencões de chamada do compilador. Cada CPU define a sua própria linguagem assembly. Cada compilador é livre para definir as suas próprias convenções de chamada, o que determina como a informação é passada de e para cada função. Este capítulo usa a linguagem assembly da família 8086. Os exemplos de interface com linguagem assembly neste capítulo usam Microsoft C/C++ (mais um exemplo usando Borland C/C++), mas você pode aplicar a informação a outros compiladores C. Mesmo que você tenha um compilador ou computador diferente, a discussão seguinte pode servir como um guia. Lembre-se, no entanto, de que o interfaceamento com linguagem assembly é um tópico avançado.

Interface com a Linguagem Assembly

Existem três razões para querer utilizar uma rotina escrita em assembler:

- Para aumentar a velocidade e eficiência.
- Para executar uma função específica de máquina não disponível em C.
- Para utilizar uma rotina pronta de caráter geral da linguagem assembly.

Vamos dar uma olhada mais detalhada nestes pontos agora.

Embora os compiladores C tendam a produzir códigos-objetos extremamente rápidos e compactos, nenhum compilador cria consistentemente um código tão rápido ou compacto como aquele escrito por um excelente programador usando assembler. Na maioria das vezes, a pequena diferença não tem muita importância nem justifica o tempo extra necessário para escrever em assembler No entanto, em casos especiais, uma função específica precisa ser codificada em assembler de forma que seja executada mais rapidamente. Por exemplo, um pacote matemático em ponto flutuante poderia ser codificado em assembly, porque ele é usado freqüentemente e afeta significativamente a velocidade de execução de um programa. Além disso, alguns dispositivos de hardware especiais necessitam de uma temporização exata, o que significa que você precisa codificar em assembler para satisfazer as exigências da temporização.

Muitos computadores, incluindo máquinas baseadas em 8086, têm certas instruções que a maioria dos compiladores C não pode executar. Por exemplo, você não pode alterar os segmentos de dados com nenhuma instrução do código ANSI para C. Além disso, você não pode implementar uma interrupção por software ou controlar o conteúdo de registradores específicos utilizando um comando padrão de C.

É muito comum, em ambientes profissionais de programação, adquirir bibliotecas de sub-rotinas para manipulação de gráficos e matemática de ponto flutuante. Algumas vezes, é necessário utilizar essas rotinas no formato de código-objeto, porque o fornecedor não vende o código-fonte. Ocasionalmente, você pode simplesmente ligar essas rotinas com o código compilado. Outras vezes, torna-se necessário escrever um módulo de interface para corrigir quaisquer diferenças entre a interface utilizada pelo seu compilador e as rotinas adquiridas.

Existem, basicamente, duas maneiras de integrar módulos em código assembly em seus programas em C. A rotina pode ser codificada separadamente e ligada ao resto do seu programa. Alternativamente, podem ser utilizadas as capacidades de código assembly em linha (*in-line*) de muitos compiladores C. Este capítulo explora ambos os métodos.

Uma palavra de aviso: Este capítulo *não* ensina a codificar em assembler, mas assume que você já sabe. Se você não sabe, não tente os exemplos. É extremamente fácil fazer algo errado e provocar um desastre. Você pode apagar seu disco rígido, por exemplo. Antes de tentar interfacear com um módulo em linguagem assembly que tenha criado, você deve consultar o manual do usuário do seu compilador para detalhes relativos à sua implementação específica.

As Convenções de Chamada de um Compilador C

Uma *convenção de chamada* é o método que um compilador particular utiliza para passar informações às funções e devolver valores. Quase todos os compiladores C utilizam a pilha para passar argumentos às funções. Se o argumento é um dos tipos de dados intrínsecos ou uma estrutura, união ou enumeração, o valor real é colocado na pilha. Se o argumento é uma matriz, seu endereço é colocado na pilha. Quando uma função C termina, ela passa um valor de retorno à rotina chamadora. Tipicamente, esse valor de retorno é colocado em um registrador, embora teoricamente possa ser passado na pilha.

A convenção de chamada também determina exatamente quais registradores devem ser preservados e quais podem ser usados livremente. Geralmente, o compilador produz um código-objeto que necessita apenas de uma parte dos registradores disponíveis. Você deve conservar o conteúdo dos registradores utilizados por seu compilador, colocando-os na pilha antes de usá-los. Quaisquer outros registradores estão livres para ser utilizados.

Quando você escreve um módulo em linguagem assembly, que precisa interfacear com o código compilado por seu compilador C, é necessário seguir as convenções definidas e utilizadas por seu compilador. Apenas dessa forma você tem rotinas em linguagem assembly interfaceando corretamente com seu código em C. A próxima seção examina, em detalhes, as convenções de chamada do Microsoft C/C++.

As Convenções de Chamada do Microsoft C/C++

Como a maioria dos compiladores, o Microsoft C passa argumentos para funções na pilha. Os argumentos são colocados na pilha da direita para a esquerda. Ou seja, na chamada

```
func(a, b, c);
```

c é colocado primeiro, seguido por **b** e **a**. A Tabela 25.1 mostra o número de bytes que cada tipo ocupa na pilha.

Na entrada de uma função em código assembly, o conteúdo do registrador **BP** deve ser gravado na pilha e o valor atual do ponteiro de pilha (**SP**) é colocado em **BP**. Os únicos registradores que precisam ser preservados são **SI**, **DI**, **SS** e **DS** (se a sua rotina os usa).

Tabela 25.1 O número de bytes na pilha necessário para cada tipo de dado quando passado para uma função para o Microsoft C\C++.

Tipo	Número de bytes
char	2
short	2
signed char	2
signed short	2
unsigned char	2
unsigned short	2
int	2
signed int	2
unsigned int	2
long	4
unsigned long	4
float	4
double	8
long double	10
ponteiro (near)	2 (offset apenas)
ponteiro(far)	4 (segmento e offset)

Antes de retornar à sua função em linguagem assembly, você deve restaurar o valor de **BP, SI, DI, SS** e **DS** e repor o valor do apontador de pilha.

Se sua função devolve um valor de 8 ou 16 bits, ele é colocado no registrador **AX**. Caso contrário, é devolvido de acordo com a Tabela 25.2.

Tabela 25.2 Uso dos registradores para retornar valores com o Microsoft C/C++.

Tipo	Registrador(es) e significados
char	AL
unsigned char	AL
short	AX
unsigned short	AX
int	AX
unsigned int	AX
long	Palavra de baixa ordem em AX Palavra de alta ordem em DX
unsigned long	Palavra de baixa ordem em AX Palavra de alta ordem em DX
float & double	Endereço do valor retornado AX contém deslocamento, DX contém segmento
struct & union	Endereço do valor retornado AX contém deslocamento, DX contém segmento.
Ponteiro (near)	AX
Ponteiro (far)	Deslocamento em AX, segmento em DX

Tabela 25.2 Uso dos registradores para retornar valores com o Microsoft C/C++.

O último ponto: Um programa C aloca espaço para variáveis locais na pilha. Quando você escreve suas próprias funções em (*Assembly*), você deve seguir o mesmo procedimento para suas variáveis locais.

Criando uma Função em Código Assembly

Sem dúvida, observar como seu compilador gera o código é a maneira mais fácil de aprender a criar funções, em linguagem assembly, que sejam compatíveis com a convenção de chamada do mesmo. Virtualmente todo compilador C tem uma opção de tempo de compilação que faz com que o compilador gere uma listagem em linguagem assembly do código que ele produz. Examinando esse arquivo, você pode aprender bastante, não apenas sobre como interfacear com o compilador, mas também sobre como o compilador realmente funciona.

A opção **-Fa** do compilador Microsoft C/C++ cria um arquivo em linguagem assembly. **-Fc** cria um arquivo que contém as instruções em linguagem assembly, os códigos hexadecimais para as instruções e as linhas de código em C que geram essas instruções. Esses dois arquivos têm as extensões .ASM e .COD, respectivamente. Este capítulo utiliza essas características para mostrar como o Microsoft C/C++ gera código para dois pequenos programas.

Uma Função Simples em Código Assembly

O primeiro programa, mostrado aqui, ilustra como o código é gerado para uma função com argumentos:

```
int sum;
int add(int a, int b);

void main(void)
{
  sum = add(10, 12);
}
```

```
add(int a, int b)
{
  int t;

  t = a+b;
  return t;
}
```

A variável **sum** é intencionalmente declarada como global para que você possa ver exemplos de dados tanto globais e locais. Se este programa for chamado **test**, a linha de comando a seguir criará o arquivo **test.asm**:

cl -Fa test.c

Isto faz com que o programa seja compilado usando o modelo de memória pequeno. (Modelos de memória são discutidos no Capítulo 16.) O conteúdo de **test.msc.asm** é mostrado aqui:

```
; File test.c
; Line 4
_main:
        push    bp
        mov     bp, sp
        mov     ax, OFFSET L00114
        call     __aNchkstk
        push    si
        push    di
; Line 5
        mov     ax, OFFSET 12
        push    ax
        mov     ax,OFFSET 10
        push    ax
        call    _add
        add     sp,OFFSET 4
        mov     WORD PTR _sum,ax
; Line 6
; Line 6
L00107:
                pop di
                pop si
                mov sp, bp
                pop bp
                ret OFFSET 0
; Line 9
; a = 0004
```



```
; b = 0006
_add:
        push    bp
        mov     bp, sp
        mov     ax, OFFSET L00116
        call    __aNchkstk
        push    si
        push    di
; t = fffc
; Line 10
; Line 12
        mov     ax, WORD PTR 4 [bp]
        add     ax, WORD PTR 6[bp]
        mov     WORD PTR -4[bp], ax
; Line 13
       mov      ax, WORD PTR -4[bp]
       jmp      L00112
; Line 14
; Line 14
L00112:
        pop     di
        pop     si
        mov     sp, bp
        pop     bp
        ret     OFFSET 0
```

O compilador adiciona o caractere sublinhado na frente de **sum**, **main** e **add** para evitar confusão com quaisquer nomes internos do compilador. De fato, o sublinhado é adicionado na frente de todos os nomes de variáveis e funções. (Isto é uma prática comum, adotada por quase todos os compiladores.)

A primeira coisa que **_main** faz é empilhar **BP** e mover **SP** para **BP**. A seguir, a rotina _ _aNchkstk fornecida pelo compilador é chamada (isto gerencia a pilha). Depois, **SI** e **DI** são salvos. Este passo é tecnicamente desnecessário porque eles não são usados neste programa. A seguir, os dois argumentos de **_add** são empilhados e **_add** é chamada. Quando **_add** retorna, seu valor de retorno é copiado para **_sum**, e **_main** retorna.

A função **_add** começa salvando **BP**, movendo o valor de **SP** para **BP**, definindo a pilha, e novamente salvando **SI** e **DI**. (De novo, salvar **SI** e **DI** é desnecessário neste caso.) As próximas três linhas de código somam os números e colocam o resultado na posição de **t** na pilha. (Lembre-se: dados locais são armazenados na pilha em programas C.) Note como os parâmetros de **_add** são acessados usando **BP**. Depois de a soma ter sido efetuada, o valor de retorno (neste caso **t**) é carregado em **AX** e a função retorna.

Para poder usar este arquivo em linguagem assembly, você precisará adicionar a ele as especificações de segmento, as declarações de dados e outras inicializações exigidas por seu compilador. (Consulte o manual do usuário do seu compilador.) No entanto, depois que você tiver feito isto, poderá montar este arquivo, ligá-lo com as funções necessárias da biblioteca e executá-lo. Mais ainda, você pode modificar o arquivo para torná-lo mais rápido, deixando o código em C inalterado. Por exemplo, você poderia remover as instruções que carregam **AX** com o valor de **t** antes do retorno de **add()** — já que **AX** já contém o resultado. Isto é chamado de *otimização manual*. Você também poderia remover as instruções que salvam e restauram **SI** e **DI**, já que eles não são usados neste programa.

Lembre-se de que compiladores diferentes geram código ligeiramente diferente. Para ver um exemplo, examine o seguinte programa em linguagem assembly. Este programa foi gerado compilando o programa C anterior usando Borland C/C++, usando a opção **-S**. Identifique as semelhanças (e diferenças) com o código gerado pelo compilador da Microsoft. (O arquivo produzido pelo compilador Borland também inclui o código de inicialização de segmentos que não foi incluído na versão Microsoft.) Como uma regra geral, as convenções de chamada do Borland C/C++ são as mesmas do que as do Microsoft C/C++.

```
          ifndef  ??version
?debug    macro
          endm
publicdll macro   name
          public  name
          endm
$comm     macro   name,dist,size,count
          comm    dist name:BYTE:count*size
          endm
          else
$comm     macro   name,dist,size,count
          comm    dist name[size]:BYTE:count
          endm
          endif
          ?debug  V 300h
          ?debug  S "test.c"
          ?debug  C E9878D421F06746573742E63
_TEXT     segment byte public 'CODE'
_TEXT     ends
DGROUP    group   _DATA,_BSS
          assume cs:_TEXT,ds:DGROUP
_DATA     segment word public 'DATA'
d@        label   byte
```



```
d@w       label   word
_DATA     ends
_BSS      segment word public 'BSS'
b@        label   byte
b@w       label   word
_BSS      ends
_TEXT segment byte public 'CODE'
  ;
  ; void main(void)
  ;
          assume cs:_TEXT
_main     proc    near
          push    bp
          mov     bp,sp
  ;
  ;       {
  ;       sum=add(10,12);
  ;
          mov     ax,12
          push    ax
          mov     ax,10
          push    ax
          call    near ptr _add
          pop     cx
          pop     cx
          mov     word ptr DGROUP:_sum,ax
  ;
  ;       }
  ;
          pop bp
          ret
_main     endp
  ;
  ;       add(int a, int b)
  ;
          assume cs:_TEXT
_add      proc     near
          push     bp
          mov      bp,sp
          sub      sp,2
  ;
  ;       {
  ;       int t;
  ;
  ;       t=a+b;
```

```
  ;
            mov     ax,word ptr [bp+4]
            add     ax,word ptr [bp+6]
            mov     word ptr [bp-2],ax
  ;
  ;          return t;
  ;
            mov     ax,word ptr [bp-2]
            jmp     short @2@58
@2@58:
  ;
  ;         }
  ;
            mov     sp,bp
            pop     bp
            ret
_add        endp
_TEXT       ends
_BSS        segment word public 'BSS'
_sum        label   word
            db      2 dup (?)
            ?debug  C E9
            ?debug  C FA00000000
_BSS        ends
_DATA       segment word public 'DATA'
s@          label   byte
_DATA       ends
_TEXT       segment byte public 'CODE'
_TEXT       ends
            public  _main
            public  _add
            public  _sum
_s@         equ      s@
            end
```

Um Exemplo de Chamada por Referência

No programa seguinte, a função **get_val()** é chamada utilizando-se o endereço de **a**, para ilustrar o código produzido quando ponteiros são usados.

```
#include <stdio.h>

void get_val(int *x);
```



```
void main(void)
{
  int a;

  get_val(&a)
  printf("%d", a);
}

void get_val(int *x)

{
  *x = 100;
}
```

O arquivo em linguagem assembly produzido pelo Microsoft C/C++ é o seguinte:

```
; File get_val.c
; Line 6
_main:
          push  bp
          mov   bp,sp
          mov   ax,OFFSET L00184
          call  __aNchkstk
          push  si
          push  di
; a = fffc
; Line 7
; Line 9
          lea   ax,WORD PTR -4[bp]
          push  ax
          call  _get_val
          add   sp,OFFSET 2
; Line 10
          push  WORD PTR -4[bp]
          mov   ax, OFFSET L00180
          push  ax
          call  __printf
          add   sp,OFFSET 4
; Line 11
; Line 11
L00178:
          pop   di
          pop   si
          mov   sp, bp
          pop   bp
```

```
        ret    OFFSET 0
; Line 14
; x = 0004
_get_val:
        push   bp
        mov    bp, sp
        mov    ax,OFFSET L00186
        call   __aNchkstk
        push   si
        push   di
; Line 15
        mov    bx,WORD PTR 4[bp]
        mov    WORD PTR [bx],OFFSET 100
; Line 16
; Line 16
L00183:
        pop    did
        pop    si
        mov    sp,bp
        pop    bp
        ret    OFFSET 0
```

Como você pode observar, **_get_val** é chamada usando o endereço de **a**. O endereço de **a** é encontrado usando a instrução LEA (carrega endereço efetivo) da linguagem assembly. Dentro de **_get_val**, o valor **100** é colocado em **a**, que está nesse endereço usando o modo de endereçamento indireto do 8086.

Utilizando o Modelo de Memória Grande para Dados e Código

Como exemplo final da forma como um compilador gera o código, compilemos o mesmo programa-teste usado nas seções anteriores utilizando o modelo de memória *huge* (enorme). Isso faz com que as referências às funções da biblioteca e aos dados globais sejam **FAR**. Isso é obtido com o compilador do Microsoft especificando a opção do compilador **-AH**. O seguinte módulo em linguagem assembly é produzido:

```
; File get_val.c
; Line 6
_main:
        push   bp
```



```
        mov    bp,sp
        mov    ax,OFFSET L00184
        call   FAR PTR __aFchkstk
        push   si
        push   di
; a = fffc
; Line 7
; Line 9
        lea    ax,WORD PTR -4[bp]
        mov    dx, ss
        push   dx
        push   ax
        call   FAR PTR _get_val
        add    sp,OFFSET 4
; Line 10
        push   WORD PTR -4[bp]
        mov    ax, OFFSET L00180
        mov    dx, ds
        push   dx
        push   ax
        call   FAR PTR __printf
        add    sp,OFFSET 6
; Line 11
; Line 11
L00178:
        pop    di
        pop    si
        mov    sp, bp
        pop    bp
        ret    OFFSET 0
; Line 14
; x = 0006
_get_val:
        push   bp
        mov    bp, sp
        mov    ax,OFFSET L00186
        call   FAR PTR __aFchkstk
        push   si
        push   di
; Line 15
        les    bx,WORD PTR 6[pb]
        mov    WORD PTRes [bx],OFFSET 100
; Line 16
; Line 16
```

```
L00183:
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret     OFFSET 0
```

Note três importantes diferenças entre esta versão e a anterior. Primeiro, a pilha agora é definida com uma chamada a **__aFchkstk** no lugar de **__aNchkstk**; **__aFchkstk** é usada quando um programa é compilado para o modelo de memória grande. Segundo, o endereço de **a** agora exige que 4 (e não 2) bytes sejam empilhados antes da chamada de **get_val()**. Isto permite que tanto o segmento quanto o deslocamento de **a** sejam passados. Dentro de **_get_val()**, **a** é acessada usando seu endereço completo de 32 bits. Terceiro, as chamadas a **_get_val** e **printf()** agora são **FAR**. Se você deseja ligar seu próprio código em linguagem assembly com código C compilado para um modelo com código e dados grandes, você deve gerar código de retorno compatível ao retornar de uma chamada **FAR**. Se você confundir os modelos, corromperá a pilha e travará o programa. Além disso, você deve usar **FAR** ao referenciar dados globais.

Criando um Esqueleto de Código Assembly

Agora que já foi visto como os compiladores C chamam as funções, o próximo passo é escrever suas próprias funções em linguagem assembly. O método mais fácil de fazer isso é deixar o compilador gerar um esqueleto em linguagem assembly para você. Uma vez com o esqueleto, tudo o que resta a fazer é preencher os detalhes. Por exemplo, suponha que você precise criar uma rotina em linguagem assembly que multiplique dois inteiros. Para que o compilador gere o esqueleto para essa função, é necessário, primeiro, criar um arquivo que contenha apenas essa função.

```
mul(int a, int b)
{
}
```

Em seguida, o arquivo deve ser compilado com a opção apropriada para que seja produzido um arquivo em linguagem assembly. Se for utilizado o compilador do Microsoft, este arquivo será produzido:


```
; File mul.c
; Line 2
; a = 0004
; b = 0006
_mul:
        push    bp
        mov     bp,sp
        mov     ax,OFFSET L00106
        call    aNchkstk
        push    si
        push    di
; Line 3
; Line 3
L00105:
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret     OFFSET 0
```

Nesse esqueleto, tudo que você precisa fazer é preencher os detalhes. A função **mul()** completa é mostrada aqui:

```
; File mul.c
; Line 2
; a = 0004
; b = 0006
_mul:
        push    bp
        mov     bp,sp
        mov     ax,OFFSET L00106
        call    __aNchkstk
        push    si
        push    di
; Aqui é onde a multiplicação realmente acontece.
        mov     ax,word ptr [bp+4]
        imul    word ptr [bp+6]
; AX agora contém o resultado, portanto retorne.
L00105:
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret     OFFSET 0
```

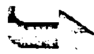
Se sua função em linguagem assembly use variáveis locais, então você precisará alocar espaço para elas na pilha. Para tanto, subtraia a quantidade necessária de bytes de **SP** depois que tiver sido salvo em **BP**. Depois, para acessar uma variável local, indexe a pilha apropriadamente usando deslocamentos negativos em relação a **BP**.

A melhor maneira de aprender mais sobre como interfacear código, em linguagem assembly, com seus programas em C é escrever funções pequenas em C que façam aproximadamente o que você deseja em assembly. E então, usando a opção de gerar linguagem assembly do compilador, criar um arquivo nessa linguagem. Na maioria das vezes será necessário apenas otimizar manualmente esse código em lugar de criar desde o princípio uma rotina em linguagem assembly.

Usando asm

Embora não suportado por alguns compiladores C (incluindo o do Microsoft), muitos outros compiladores, como o Turbo C, acrescentaram uma extensão à linguagem C, que permite que haja código assembly em linha (*in-line*), como parte de um programa em C, sem a utilização de um módulo completamente separado. Existem duas vantagens. Primeiro, não é necessário escrever todo o código da interface para cada função; segundo, todo o código está em um só lugar, tornando mais fácil o suporte.

A extensão acrescentada pelo Turbo C é chamada de **asm**. (No entanto, Microsoft C/C++ chama a palavra-chave estendida de **__asm**.) Para inserir código assembly em um programa, deve-se preceder a instrução assembly com **asm**. Isto é, cada linha que contém código assembly deve começar com **asm**. O compilador Turbo C simplesmente passa a instrução, sem alteração, para a fase assembler do compilador.



NOTA: Embora o padrão C ANSI não defina a palavra-chave **asm**, *C++ o faz.*

Por exemplo, a função seguinte, chamada **init_port1()**, move o valor 88 para **AX** e o envia para as portas 20 e 21:

```
void init_port(void)
{
  printf("Inicializando a porta\n");
asm     mov AX, 88
asm     out 20, AX
asm     out 21, AX
}
```


Aqui, o Turbo C produz automaticamente o código de interface para salvar os registradores e retornar da função. Você apenas precisa fornecer o código que está dentro da função.

Esse método poderia ser utilizado para criar uma função que multiplique dois números, chamada **mul()**, sem criar realmente um arquivo separado em linguagem assembly. Usando essa abordagem, o código para **mul()** é mostrado aqui:

```
mul(int a, int b)
{
asm     mov ax, word ptr [bp + 4]
asm     imul word ptr [bp + 6]
}
```

O compilador C fornece todo o suporte usual para montar e retornar de uma chamada de função. Você deve simplesmente fornecer o corpo da função e seguir as convenções de chamada para acessar os argumentos.

Qualquer que seja o método utilizado, lembre-se de que você está criando situações dependentes da máquina, que tornarão seu programa difícil de ser transportado para outra máquina. No entanto, para as situações que exigem o uso de código assembly, normalmente vale a pena o esforço.

Quando Codificar em Assembler

Por tratar-se de uma codificação difícil, a maioria dos programadores só codifica em assembler quando é absolutamente necessário. Como regra geral: não use assembler, cria problemas demais! No entanto, há duas situações em que faz sentido codificar em assembler. A primeira é quando não há absolutamente outra maneira de alcançar o resultado desejado. Por exemplo, quando você precisa interfacear diretamente com um dispositivo de hardware que não pode ser operado usando C. A segunda situação ocorre quando você precisa reduzir o tempo de execução de um programa em C.

Quando você precisa aumentar a velocidade de um programa, deve escolher cuidadosamente que funções codificar em assembler. Se codificar as funções erradas, verá um aumento muito pequeno na velocidade. Se escolher as funções corretas, seu programa voará! Você pode determinar facilmente que funções recodificar revendo o fluxo operacional do seu programa. Geralmente, as funções usadas dentro dos laços são as que devem ser programadas em assembler, pois são executadas repetidamente. Codificar em assembler uma função utilizada apenas uma ou duas vezes não aumentará a velocidade do seu programa, mas recodificar uma função usada várias vezes aumentará. Por exemplo, considere a seguinte função **main()**:

```
#include <stdio.h>

void main(void)

{
  register int t;
  init();

  for(t=0; t<1000; ++t) {
    phase1();
    phase2();
    if(t==10) phase3();
  }
  byebye()
}
```

Evidentemente, recodificar **init()** e **byebye()** não deve afetar de forma mensurável a velocidade do programa porque elas são executadas apenas uma vez. No entanto, **phase1()** e **phase2()** são executadas mil vezes, e codificá-las em assembler definitivamente diminuirá o tempo de processamento do programa. A função **phase3()** é executada apenas uma vez, mesmo estando dentro do laço; assim, recodificar essa função em assembler provavelmente não valerá a pena.

Refletindo com cuidado você pode aumentar a velocidade dos seus programas, recodificando apenas algumas funções em assembler.

26

Engenharia de Software Usando C

Toda a disciplina da ciência da computação emergiu com uma velocidade espantosa. Antes de 1970, pouca distinção era feita entre os engenheiros que projetavam o computador e aqueles que o programavam. Se você entendia de computadores, pressuponha-se que você poderia desenvolver tanto o hardware como o software. Essa situação se modificou radicalmente durante os anos 70. Agora, a maioria das faculdades oferece currículos diferentes para engenheiros de computadores e engenheiros de software.

A arte e a ciência da engenharia de software encerram uma ampla gama de tópicos. Criar um grande programa de computador é semelhante a projetar um grande prédio. Há tantas partes envolvidas que parece quase impossível fazer tudo funcionar junto. Evidentemente, o que faz a criação de um grande programa possível é a aplicação dos métodos de engenharia adequados. Neste capítulo, serão examinadas diversas técnicas e dois utilitários que dizem respeito especificamente ao ambiente de programação em C e que tornam a criação e o suporte de um programa muito mais fáceis.

Projeto em Top-Down

Sem dúvida a coisa mais importante que pode ser feita para simplificar a criação de um grande programa é aplicar uma abordagem sólida. Existem três métodos gerais utilizados para se escrever um programa: *top-down* (de cima para baixo), *bottom-up* (de baixo para cima) e *ad hoc*. Na *abordagem top-down*, você inicia pela rotina de nível mais alto e desce às rotinas de baixo nível. A *abordagem bottom-up*

opera na direção oposta: você inicia pelas rotinas específicas e as constrói progressivamente dentro de estruturas mais complexas, finalizando na rotina de alto nível. A *abordagem ad hoc* não tem método predeterminado.

Como uma linguagem estruturada, C presta-se à abordagem top-down. O método top-down pode produzir um código limpo e legível de fácil manutenção. Essa abordagem ajuda, também, a esclarecer a estrutura completa do programa antes de codificar as funções de baixo nível, reduzindo o tempo desperdiçado por falsos começos.

Delineando Seu Programa

Como um esboço, o método top-down começa com uma descrição geral e caminha na direção da particularização. Uma boa maneira de projetar um programa é definir exatamente o que o programa fará no nível mais alto. Por exemplo, suponha que você tenha de escrever um programa de lista postal. Primeiro, você deve fazer uma lista do que o programa fará. Cada entrada na lista deve conter apenas uma unidade funcional. (Uma unidade funcional pode ser imaginada como uma caixa preta que executa uma única tarefa.) Por exemplo, a sua lista se pareceria com isto:

- Inserir um novo nome
- Deletar um nome
- Imprimir a lista
- Procurar um nome
- Gravar a lista em um arquivo em disco
- Carregar a lista
- Terminar o programa

Essas unidades podem formar a base das funções do programa.

Após ter definido o funcionamento geral do programa, pode-se esboçar os detalhes de cada unidade funcional, começando com o laço principal. O laço principal deste programa é

```
laço principal
{
  do {
    exibe o menu
    obtém a seleção do usuário
    processa a seleção
  } while  a seleção não for igual a quit (terminar)
}
```


Esse tipo de notação algorítmica (também chamada de *pseudocódigo*) pode ajudar a esclarecer a estrutura geral do seu programa antes que você sente ao computador. Foi usada a sintaxe de C porque é familiar, mas qualquer tipo de sintaxe pode ser usada.

Você pode dar uma definição similar a cada área funcional. Por exemplo, você pode definir a função que escreve a lista postal em um arquivo em disco desta forma:

```
salva no disco {
  abre o arquivo em disco
  while  houver dados a escrever {
    escreva o dado no disco
  }
  fecha o arquivo em disco
}
```

Nesse ponto, a função gravar no disco criou novas unidades funcionais específicas. Cada uma delas deve ser definida. Se, no curso da definição, novas unidades funcionais forem criadas, elas também serão definidas e assim por diante. Esse processo termina quando não é criada mais nenhuma unidade funcional e tudo o que resta é codificar a rotina. Por exemplo, a unidade que fecha o arquivo em disco provavelmente consistirá apenas em uma chamada de **fclose()**.

Note que a definição não menciona estrutura de dados ou variáveis. Isso é intencional. Até agora, o que se deseja definir é o que o seu programa fará, não como ele de fato o fará. Esse processo de definição ajuda-o a decidir quanto à real estrutura de dados utilizada. (Evidentemente, é preciso determinar a estrutura de dados antes que se possa codificar qualquer unidade funcional.)

Escolhendo uma Estrutura de Dados

Após ter determinado o perfil geral do seu programa, você precisa decidir como os dados serão estruturados. A escolha da estrutura de dados e sua implementação são críticas porque ajudam a determinar os limites do seu programa.

Uma lista postal trabalha com coleções de informações: nomes e endereços. Utilizando a abordagem top-down, isso imediatamente sugere o uso de uma estrutura para guardar a informação. Porém, como essas estruturas serão armazenadas e manipuladas? Para um programa de lista postal, poderia ser utilizada uma matriz de estruturas de tamanho pré-fixado. Mas uma matriz de tamanho fixo tem duas sérias desvantagens. Primeiro, o tamanho da matriz limita arbitrariamente a extensão da lista postal. Segundo, uma matriz de tamanho fixo

não tirará vantagem de qualquer memória adicional que você possa colocar em seu computador. Por outro lado, se a memória for subtraída (por exemplo, se uma placa de memória falhar), o programa não funcionará, porque a matriz de tamanho fixo pode não caber mais na memória. Portanto, um programa de lista postal deve utilizar alocação dinâmica de memória de forma que a lista seja do tamanho que a memória permitir.

Embora a alocação dinâmica tenha sido escolhida em lugar de uma matriz de tamanho fixo, a forma exata dos dados ainda não foi definida. Há diversas possibilidades: você pode utilizar uma lista encadeada, uma lista duplamente encadeada, uma árvore binária ou mesmo um método de fragmentação. Cada método tem seus méritos e suas desvantagens. Se você decidir utilizar uma árvore binária, devido ao seu rápido tempo de busca, poderá então, definir a estrutura que contém cada nome e endereço da lista, como mostrado aqui:

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  char zip[11];
  struct addr *left; /* ponteiro para a subárvore esquerda */
  struct addr *right; /* ponteiro para a subárvore direita */
};
```

Uma vez que a estrutura de dados tenha sido definida, você está pronto para codificar seu programa. Para tanto, simplesmente preencha os detalhes descritos no pseudocódigo que você criou anteriormente.

Se você seguir a abordagem top-down, seus programas não apenas serão mais fáceis de ler como levarão menos tempo para ser desenvolvidos e exigirão menor esforço para ser mantidos.

Funções à Prova de Bala

Em grandes programas, especialmente aqueles que controlam potencialmente eventos que ameacem a vida humana, a possibilidade de erros deve ser ínfima. Embora pequenos programas possam ser verificados em todas as condições, isso não acontece com os grandes. (Um programa examinado é dito livre de erros e jamais deixará de funcionar — pelo menos em teoria.) Por exemplo, considere um programa que controle os flaps das asas de um jato 767. Você não pode

testar todas as possíveis interações das numerosas forças que serão exercidas sobre o avião. Isso significa que o programa não pode ser testado exaustivamente. No máximo, tudo o que se pode dizer é que ele foi executado corretamente em algumas situações específicas. Em um programa desse tipo, a última coisa que você (como passageiro ou programador) quer é uma quebra (do programa ou do avião).

Após ter programado por alguns anos, aprende-se que a maioria das quebras de programa fora do estágio de desenvolvimento inicial ocorre porque uma função inadvertidamente interfere no código ou nos dados de uma outra função. Por exemplo, muitos erros de programa catastróficos são causados por um destes erros relativamente comuns:

- Alguma condição faz com que seja iniciada um laço infinito não prevista.
- Os limites de uma matriz foram violados, causando danos a código ou dados adjacentes.
- Um tipo de dados ultrapassa sua capacidade de forma imprevista.

Na teoria, estes tipos de erros podem ser evitados por meio de práticas de projeto e programação cuidadosas e bem pensadas. (De fato, programas escritos profissionalmente devem estar razoavelmente livres deste tipo de erros.)

No entanto, existe outro tipo de erro que freqüentemente aparece depois do estágio de desenvolvimento inicial do programa, ocorrendo ou durante a fase final de "ajuste fino" ou durante a fase de manutenção do programa. Este erro é causado por uma função interferindo de maneira inadvertida no código ou nos dados de outra função. Este tipo de erro é especialmente difícil de encontrar porque o código em ambas as funções parece correto. Em vez disso, é a interação entre as duas funções que gera o erro. Portanto, para reduzir as chances de uma falha catastrófica, você irá querer que suas funções e seus dados sejam tão robustos quanto possível. A melhor maneira de conseguir isso é conservar o código e os dados relacionados a cada função ocultos do restante do programa. Isto é chamado algumas vezes de *encapsular código e dados*.

Ocultar código e dados é similar a contar um segredo apenas às pessoas que precisam conhecê-lo. Se uma função não precisa saber nada a respeito de uma outra função ou variável, não deixe a função ter acesso a ela. Para conseguir isso, estas quatro regras devem ser seguidas.

1. Cada unidade funcional deve ter um ponto de entrada e um ponto de saída.

2. Sempre que possível, passe as informações para as funções em lugar de usar variáveis globais.
3. Onde variáveis globais são necessárias para umas poucas funções relacionadas, você deve colocar as variáveis e as funções em um arquivo separado. Além disso, as variáveis globais devem ser declaradas como **static**.
4. Cada função deve ser capaz de informar o sucesso ou falha da operação para a qual foi chamada. Isto é, o código que chama a função deve ser capaz de saber se a função teve sucesso ou falhou.

A regra 1 determina que cada área funcional tem um ponto de entrada e um ponto de saída. Isto significa que, embora uma unidade funcional possa conter diversas funções, o resto do programa se comunica apenas por meio de uma delas. Considere o programa de lista postal discutido anteriormente. Há sete áreas funcionais. Você poderia colocar todas as funções necessárias a cada área funcional em seu próprio arquivo e compilá-las separadamente. Se isso for feito corretamente, a única maneira de entrar ou sair de cada unidade funcional será por meio da sua função de nível mais alto. E, no programa de lista postal, essas funções de nível mais alto são chamadas apenas por **main()**, desse modo evitando que uma unidade funcional danifique acidentalmente outra. Essa situação é representada na Figura 26.1.

Embora diminua a performance do programa, a melhor maneira de reduzir a possibilidade de efeitos colaterais é sempre passar toda informação necessária para a função e nunca usar dados globais. Essa é a regra 2 e, se você alguma vez já programou em BASIC padrão — no qual todas as variáveis são globais — você entende a sua importância.

A regra 3 determina que, quando dados globais devem ser utilizados, eles e as funções que precisam acessá-los devem ser colocados em um arquivo e compilados separadamente. Os dados globais devem ser declarados como **static**, escondendo-os, dessa forma, dos outros arquivos. Também, as funções que acessam os dados **static** podem, elas mesmas, ser declaradas como **static**, previnindo-as de serem chamadas por outras funções não declaradas dentro do mesmo arquivo.

Dito de maneira simples, a regra 4 garante que os programas tenham uma segunda chance ao permitir que o chamador de uma função possa responder de maneira razoável ante uma condição de erro. Por exemplo, se a função que controla os flaps de um avião resulta em uma condição errada, você não quer que o programa inteiro falhe (e o avião caia). Em vez disso, você deseja que o programa saiba que ocorreu um erro dentro da função. Como a condição de erro pode ser uma situação temporária para um programa que opera com dados obtidos em tempo real, o programa pode responder a um erro dessa natureza simplesmente aguardando por alguns instantes e tentando novamente.

Tenha em mente que a submissão rigorosa a estas regras não se aplica em todas as situações. No entanto, você deve seguir as regras quando for necessário o mais alto grau de tolerâncias a falhas. Objetivo deste enfoque é criar um programa que tenha a mais alta probabilidade de se recuperar sem prejuízo de uma condição de erro.

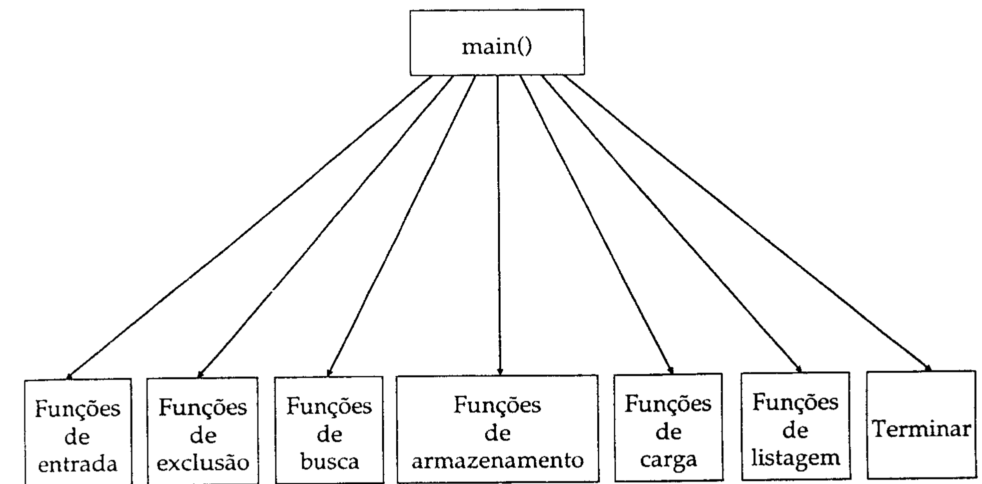


Figura 26.1 Cada unidade funcional tem apenas um ponto de entrada.

NOTA: Se você estiver especialmente interessado nos conceitos que suportam funções à prova de bala, deverá explorar C++. C++ fornece um mecanismo de proteção mais forte chamado encapsulamento, que diminui ainda mais a chance de uma função danificar outra.

Usando MAKE

Outro tipo de erro que tende a afetar a criação de programas grandes ocorre com maior freqüência durante o estágio de desenvolvimento e pode levar um projeto virtualmente a parar. Este erro ocorre quando um ou mais arquivos-fontes estão desatualizados em relação a seus respectivos códigos-objeto quando o programa é compilado. Quando isto ocorrer, o programa compilado não se comportará de acordo com o estado atual do código-fonte. Qualquer um que tenha estado envolvido na criação ou manutenção de um projeto grande de software provavel-

mente já experimentou este problema. Para ajudar a eliminar este tipo frustrante de erro, a maioria dos compiladores C inclui um utilitário chamado MAKE que ajuda a sincronizar arquivos-fonte e objeto. (O nome exato do utilitário MAKE para seu compilador pode não ser MAKE, portanto certifique-se verificando o nome correto no manual do usuário do compilador.)

MAKE é um programa utilitário que automatiza o processo de recompilação para grandes programas compostos de diversos arquivos. Como você provavelmente sabe, no decurso do desenvolvimento de um programa, muitas alterações pequenas são feitas em alguns arquivos. Então, o programa é recompilado e testado. Infelizmente, é fácil esquecer quais arquivos precisam ser recompilados. Nessa situação, você pode recompilar todos os arquivos — um desperdício de tempo — ou acidentalmente não recompilar um arquivo que foi modificado, acrescentando potencialmente diversas horas de depuração frustrante. O programa MAKE resolve esse problema recompilando automaticamente apenas os arquivos que foram alterados.

Os exemplos apresentados aqui usam o programa MAKE fornecido com o Borland C/C++ e o Microsoft C/C++. Borland chama seu MAKE de MAKE. No entanto, as versões contemporâneas do Microsoft C/C++ o chamam de NMAKE. Os exemplos também devem funcionar com outros utilitários MAKE do mercado, e os conceitos genéricos apresentados são aplicáveis a todos os programas MAKE.

NOTA: Nos últimos anos, os programas MAKE tornaram-se muito sofisticados. Os exemplos aqui apresentados ilustram a essência do MAKE. Você irá querer explorar o utilitário MAKE fornecido com seu compilador. Ele pode conter recursos especialmente úteis para seu ambiente de desenvolvimento.

MAKE é guiado por um *arquivo make*, que contém uma lista de arquivos-destinos, arquivos dependentes e comandos. Um *arquivo-destino* requer seus *arquivos dependentes* para produzi-lo. Por exemplo, T.C seria o arquivo dependente de T.OBJ, pois T.C é necessário para produzir T.OBJ. MAKE opera comparando as datas entre um arquivo dependente e seu arquivo-destino (como usado aqui, o termo *data* inclui a data de calendário e a hora). Se o arquivo-destino tem uma data anterior à do seu arquivo dependente ou não existe, a seqüência de comandos especificada é executada. Se a seqüência de comandos contém arquivos-destino definidos por outras dependências, então essas dependências também são atualizadas, conforme seja necessário. Quando o processo MAKE terminar, todos os arquivos-destino terão sido atualizados. Portanto, em um arquivo make construído corretamente, todos os arquivos-fonte que exigem compilação são automaticamente compilados e ligados, formando o novo executável. Desta maneira, os arquivos-fonte são mantidos em sincronismo com os arquivos-objeto.

A forma geral do arquivo make é

arquivo_destino1 : lista de arquivos_dependentes
seqüência_de_comandos

arquivo_destino2 : lista de arquivos_dependentes
seqüência_de_comandos

arquivo_destino3 : lista de arquivos_dependentes
seqüência_de_comandos

.
.
.

arquivo_destinoN : lista de arquivos_dependentes
seqüência_de_comandos

O nome do arquivo-destino deve começar na coluna mais à esquerda e ser seguido por dois-pontos e por sua lista de arquivos dependentes. A seqüência de comandos associada a cada destino deve ser precedida pelo menos por um espaço ou uma tabulação. Os comentários são precedidos por um # e podem vir após a lista de arquivos dependentes e a seqüência de comandos. Se eles aparecerem em uma linha própria, devem começar na coluna mais à esquerda. Cada especificação de arquivo-destino deve ser separada da próxima por, no mínimo, uma linha em branco.

A coisa mais importante que você precisa entender a respeito de um arquivo make é isto: a execução de um arquivo make pára tão logo a primeira dependência seja bem-sucedida. Isto significa que você deve projetar seus arquivos make de tal forma que as dependências sejam hierárquicas. Lembre-se de que nenhuma dependência pode ser bem-sucedida enquanto todas as dependências subordinadas não tiverem sido resolvidas.

Para ver como MAKE funciona, considere um programa muito simples. O programa é dividido em quatro arquivos chamados TEST.H, TEST.C, TEST2.C e TEST3.C. (Para seguir adiante, insira cada parte do programa nos arquivos indicados.) Essa situação é ilustrada na Figura 26.2. (Para acompanhar, digite cada parte do programa nos arquivos indicados.)

Se você estiver usando Borland C/C++, o próximo arquivo de make deverá recompilar o programa quando você fizer alterações. (Se você está usando um compilador C/C++, troque **bcc** por **cl**.)

```
test.exe:   test.h test.obj test2.obj test3.obj
        bcc test.obj test2.obj test3.obj

test.obj:   test.c test.h
        bcc -c test.c

test2.obj:  test2.c test.h
        bcc -c test2.c

test3.obj:  test3.c test.h
        bcc -c test3.c
```

TEST.H

```
extern int count;
```

TEST.C

```
#include <stdio.h>
void test2(void), test3(void);

int count = 0;

void main(void)
{
  printf("count=%d\n", count);
  test2();
  printf("count=%d\n", count);
  test3();
  printf("count=%d\n", count);
}
```

TEST2.C

```
#include <stdio.h>
#include "test.h"

void test2(void)
{
  count = 30;
}
```

TEST3.C

```
#include <stdio.h>
#include "test.h"

void test3(void)
{
  count = -100;
}
```

Figura 26.2 Um programa simples com quatro arquivos.

Normalmente um programa MAKE usará as diretivas de um arquivo chamado MAKEFILE. No entanto, você vai provavelmente querer usar um outro nome para seu arquivo make. Quando você usa outro nome para o arquivo make, você deve usar a opção **-f** na linha de comando. Por exemplo, se o nome do arquivo anterior for TEST, você deverá digitar

make -f test

no aviso da linha de comando para compilar os módulos necessários e criar um programa executável. (Isto se aplica tanto à versão de MAKE da Borland quanto ao NMAKE da Microsoft. Uma opção diferente pode ser necessária se você usa um utilitário MAKE diferente.)

A ordem é muito importante no arquivo make, porque muitas versões de MAKE movem-se na lista apenas para frente. Por exemplo, se o arquivo make TEST fosse alterado desta forma:

```
# Este é um arquivo de make incorreto.
test.obj:  test.c test.h
        bcc -c test.c

test2.obj:  test2.c test.h
        bcc -c test2.c

test3.obj:  test3.c test.h
        bcc -c test3.c

test.exe:  test.h test.obj test2.obj test3.obj
        bcc test.obj test2.obj test3.obj
```

não mais funcionaria corretamente quando o arquivo TEST.H (ou qualquer outro arquivo-fonte) fosse alterado. A razão disso é que a diretiva final (que cria um novo TEST.EXE) não será mais executada.

Usando Macros com MAKE

MAKE permite que sejam definidas macros no arquivo make. Esses nomes de macro são simplesmente substituídos pela informação que realmente seria determinada, por uma especificação na linha de comandos ou pela definição da macro no arquivo make. As macros são definidas de acordo com esta forma geral:

nome_da_macro = definição

Se deve haver algum espaço em branco na definição da macro, você deve colocar a definição entre aspas.

Uma vez a macro definida, ela é usada no arquivo desta forma:

$(*nome_da_macro*)

Cada vez que essa sentença é encontrada, a definição ligada à macro é substituída. Por exemplo, este arquivo make utiliza a macro **LIBFIL** para determinar qual biblioteca será utilizada pelo ligador:

```
LIBFIL = graphics.lib

prog.exe: prog.obj prog2.obj prog3.obj
        bcc prog.obj prog2.obj prog3.obj $(LIBFIL)
```

Muitos programas MAKE têm recursos adicionais, de forma que é muito importante que você consulte seu manual do usuário.

Usando um Ambiente Integrado de Desenvolvimento

A maioria dos compiladores modernos é fornecida de duas formas diferentes. A primeira forma é o compilador isolado, acessado a partir da linha de comando. Usando esta forma, você usa um editor separado para criar seu programa, depois você compila seu programa e, finalmente, você executa seu programa. Todos estes eventos ocorrem como comandos separados que você emite na linha de comando. Qualquer depuração ou controle de arquivos-fonte (tais como MAKE) também ocorre de maneira separada. O compilador de linha de comando é a maneira pela qual os compiladores eram implementados tradicionalmente.

A segunda maneira de um compilador é encontrada em um ambiente integrado de desenvolvimento (IDE, de Integrated Development Environment). Nesta forma, o compilador é integrado com um editor, um depurador, um gerenciador de projetos (que toma o lugar de um utilitário MAKE separado) e um sistema de suporte da execução. Usando um IDE, você edita, compila e executa seus programas sem jamais sair do IDE. Quando os IDEs foram inventados, eles eram meio difíceis de usar e tediosos de trabalhar. No entanto, hoje os IDEs fornecidos pelos principais fabricantes de compiladores têm muito a oferecer aos programadores. Se você gastar tempo para definir as opções do IDE de forma a otimizá-lo de acordo com as suas necessidades, você achará que o uso do IDE acelera seu processo de desenvolvimento.

É claro, usar um IDE ou o enfoque tradicional da linha de comando também é uma questão de gosto. Se você gosta de usar a linha de comando, então use-a. Além disso, um ponto que ainda favorece o enfoque tradicional é que você pode escolher pessoalmente cada uma das ferramentas que usa, em vez de ter de contentar-se com aquelas que são fornecidas pelo IDE.

MAKRON Books

27

Eficiência, Portabilidade e Depuração

A habilidade de escrever programas que façam uso eficiente dos recursos do sistema, sejam livres de erros e possam ser transportados para um novo computador é a marca característica de um programador profissional. É também nessas áreas que a ciência da computação torna-se a "arte da ciência da computação" devido a poucas técnicas formais que garantem êxito. Este capítulo apresenta alguns dos métodos para se obter eficiência, aperfeiçoar a depuração de programas e aumentar a portabilidade.

Eficiência

Em programação, o termo *eficiência* pode referir-se à velocidade de execução, utilização dos recursos do sistema, ou ambos. Os recursos do sistema incluem coisas como RAM, espaço em disco, papel de impressora e coisas do gênero — basicamente qualquer coisa que você possa alocar e utilizar. Se um programa é eficiente ou não é, algumas vezes, um julgamento subjetivo que pode variar de situação para situação. Por exemplo, considere um programa de ordenação que utilize 128 K de RAM, 2 MB de espaço em disco e cujo tempo médio de processamento seja sete horas. Se esse programa está ordenando 100 endereços de um banco de dados de uma lista postal, ele não é muito eficiente. Porém, se o programa está ordenando a lista telefônica de New York, então é provavelmente muito eficiente.

Além disso, a otimização de um aspecto de um programa geralmente degrada um outro. Por exemplo, fazer um programa ser executado mais rapidamente geralmente também significa torná-lo maior quando é utilizado código

em linha (in-line) para eliminar o tempo extra despendido por uma chamada a uma função. Da mesma forma, tornar um programa menor, substituindo código em linha por chamadas a funções, faz o programa rodar mais lentamente. Assim também, fazer um uso mais eficiente do espaço em disco significa compactar os dados, o que pode tornar os acessos ao disco mais lentos. Esses e outros tipos de trocas na eficiência podem ser muito frustrantes — especialmente para não-programadores e usuários finais, que não conseguem perceber por que uma coisa deve afetar outra.

Felizmente, há algumas práticas de programação que sempre são eficientes — ou, pelo menos, mais eficientes que outras. Além disso, existem algumas técnicas que tornam os programas mais rápidos e menores. Este capítulo examina essas técnicas.

Os Operadores de Incremento e Decremento

As discussões sobre o uso eficiente de C quase sempre começam com os operadores de incremento e decremento. Caso você tenha esquecido, os operadores de incremento, **++**, incrementam seu operando em um e o operador de decremento, **--**, decrementa-o em um. Por exemplo, estes dois comandos são iguais no efeito final.

```
x = x + 1;

x++;
```

Ambos incrementam o valor de **x** de um. No entanto, o comando de incremento é executado mais rápido e requer menos RAM do que seu comando de atribuição correspondente. Isso acontece devido à forma como o código-objeto é gerado pelo compilador. Geralmente, como no caso da maioria dos microcomputadores, é possível incrementar ou decrementar uma palavra da memória sem utilizar explicitamente instruções de carga e armazenamento. Por exemplo, utilizando uma linguagem assembly imaginária que se aproxima de forma imprecisa daquela encontrada em muitos microprocessadores, o comando

```
x = x + 1;
```

gera um código-objeto semelhante a isto:

```
mova A,x   ;    carrega o valor de x da memória no
           ;    acumulador
some A1    ;    soma 1 ao acumulador
armazene x;     armazena o novo valor de volta em x
```

Se, porém, fosse utilizado o operador de incremento, o seguinte código seria produzido:

```
incr x    ;      incrementa x de 1
```

Como você pode observar, as instruções de carga e armazenamento foram eliminadas, o que significa que o código roda mais rápido e é menor. Embora muitos dos melhores compiladores otimizem automaticamente um comando como x = x + 1 para **x++**, isso não pode ser tomado como certo.

Utilizando Variáveis em Registradores

Sem dúvida, você deve utilizar sempre que possível variáveis em registradores para controle de laços. Qualquer variável especificada como **register** é armazenada de uma maneira que produz o tempo de acesso mais curto. Para tipos inteiros, isso normalmente significa um registrador da CPU. Isso é importante porque a velocidade em que os laços críticos de um programa são executados determina a velocidade geral do programa.

Para mostrar como o código difere entre uma variável **register** e uma variável normal armazenada na memória este programa foi compilado de forma a gerar um arquivo em linguagem assembly. Como você deve saber, muitos compiladores C fornecem uma opção que faz com que o compilador crie um arquivo em código assembly antes de um arquivo código objeto. Usando esta opção verifique o código produzido pelo compilador, observando que cada tipo de variável é manipulada. (Declarando que J como uma variável global garanta que sem a otimização do compilador automaticamente o converterá para dentro da variável **register**.)

```
int j;

void main(void)
{
  register int i;

  for(i=0; i<100; i++) ;

  for(j=0; j<100; j++) ;
}
```

Este arquivo foi produzido pelo Borland C/C++. (O arquivo em código assembly produzido é mostrado a seguir. Os comentários que iniciam com asteriscos foram acrescentados pelo autor.) Note as diferenças nas instruções usadas na repetição controlada por registrador e a repetição controlada sem registrador. Embora este código tenha sido produzido pelo Borland C/C++, código similar será produzido por qualquer compilador C.


```
        ifndef  ??version
?debug  macro
        endm
publicdll macro  name
        public name
        endm
$comm   macro      name,dist,size,count
        comm       dist name:BYTE:count*size
        endm
        else
$comm   macro       name,dist,size,count
        comm        dist name[size]:BYTE:count
        endm
        endif
        ?debug V 300h
        ?debug S "regvars.c"
        ?debug C E90364441F09726567766172732E63
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label byte
b@w     label word
_BSS    ends
_TEXT   segment byte public 'CODE'
   ;
   ;    void main (void)
   ;
        assume   cs:_TEXT
_main   proc     near
        push     bp
        mov      bp,sp
   ;
   ;    {
   ;      register int i;
   ;
   ;      for(i=0; i<100; i++ ;
   ;
;**************************************************
; O código seguinte inicializa a repetição controlada
```

```
; por variável registrador. Note como a variável de
; controle é inicializada com uma instrução XOR.
;************************************************
      xor    ax,ax
      jmp    short @1@86
@1@58:
;******* Este é um incremento de registrador.
      inc    ax
@1@86:
;******* Esta é uma comparação de registrador.
      cmp    ax,100
      jl     short @1@58
;
;
;     for(j=0; j>100; j++) ;
;************************************************
; O código seguinte inicializa a repetição controlada
; por variável de memória. Note que é necessário um
; acesso a memória para inicializar a variável.
;************************************************
         mov     word ptr DGROUP:_j,0
         jmp     short @1@170
@1@142:
; ******* Aqui, um acesso à memória é usado para incrementar j.
          inc    word ptr DGROUP:_j
@1@170:
; ******* Aqui, um acesso à memória é usado para comparar j.
          cmp    word ptr DGROUP:_j,100
          jl     short @1@142
   ;
   ;      }
   ;
          pop    bp
          ret
_main     endp
_TEXT     ends
_BSS      segment word public 'BSS'
_j        label   word
          db      2 dup (?)
          ?debug  C E9
          ?debug  C FA00000000
_BSS      ends
_DATA     segment word public 'DATA'
s@        label byte
```



```
_DATA     ends
_TEXT     segment byte public 'CODE'
_TEXT     ends
          public _main
          public _j
_s@       equ    s@
          end
```

Como você pode observar no arquivo em linguagem assembly, a repetição controlada por variável registrador não requer nenhum acesso à memória. No entanto, a repetição controlada pela variável de memória requer numerosos acessos à memória. Como os acessos à memória são mais custosos em termos de tempo do que os acessos a registradores, é óbvio qual repetição executará mais rapidamente.

Embora seja possível declarar quantas variáveis se queira como **register**, na realidade, a maioria dos compiladores pode otimizar o tempo de acesso de apenas umas poucas. Por exemplo, geralmente apenas duas variáveis do tipo inteiro podem ser guardadas em registradores da CPU. O padrão ANSI determina que o compilador pode desprezar o especificador **register** e manipular a variável normalmente. A razão para essa provisão é que, em muitos ambientes, há apenas um número limitado de posições de armazenamento de acesso rápido. Assim, é conveniente escolher cuidadosamente aquelas variáveis que você deseja que sejam acessadas mais rapidamente.

Ponteiros *Versus* Indexação de Matrizes

A indexação de matrizes pode, também, ser substituída por aritmética de ponteiros para produzir um código menor e mais rápido. Ponteiros geralmente tornam seu código mais rápido e tomam menos espaço. Por exemplo, os dois fragmentos de código seguintes executam a mesma tarefa:

Indexação de matrizes

```
for(;;) {
  a = array[t++];
  .
  .
  .
}
```

Aritmética de ponteiros

```
p = array;
for(;;) {
  a = *(p++)
  .
  .
  .
  }
```

A vantagem do método ponteiro é que, uma vez que **p** tenha sido carregado com o endereço de **array** (talvez em um registrador de índice, como SI no processador 8086), apenas um incremento precisa ser executado cada vez que o laço se repete. Já, a versão com indexação de matriz precisa calcular o índice da matriz baseado no valor de **t** — uma tarefa mais complexa. A disparidade nas velocidades de execução da indexação de matrizes e aritmética de ponteiro aumenta à medida que são utilizados índices múltiplos. Cada índice requer sua própria seqüência de instruções, enquanto a aritmética de ponteiro equivalente pode usar simples adição.

Mas, cuidado! Você deve utilizar indexação de matrizes quando o índice for derivado de uma fórmula muito complexa e a aritmética de ponteiro obscureceria o significado do programa. Normalmente, é melhor perder um pouco em desempenho que sacrificar a clareza.

Uso de Funções

Lembre-se, a todo momento, de que o uso de funções isoladas com variáveis locais ajuda a formar a base da programação estruturada. Funções são os blocos construcionais de programas em C e um dos seus mais fortes recursos. Não deixe que nada que seja discutido neste capítulo faça-o pensar diferente. Tendo sido avisado, há umas poucas coisas que você deve saber sobre funções em C e suas contribuições no tamanho e na velocidade do seu código.

Primeiramente, C é uma linguagem orientada pela pilha. Isso significa que todas as variáveis locais e os parâmetros para as funções utilizam a pilha para armazenamento temporário. Quando uma função é chamada, o endereço de retorno da rotina chamadora é colocado na pilha também. Isso permite que a sub-rotina retorne à posição em que foi chamada. Quando uma função retorna, esse endereço e todas as variáveis locais e parâmetros têm de ser removidos da pilha. O processo de colocação dessa informação é geralmente chamado de *seqüência de chamada* e o processo de retirada é denominado *seqüência de retorno.* Essas seqüências tomam tempo — algumas vezes bastante tempo.

Para entender como uma chamada à função pode retardar seu programa, veja estes dois fragmentos de código:

Versão 1

```
for(x=1; x<100; ++x) {
   t = compute(x);
}
```

Versão 2

```
for(x=1; x<100; ++x){
   t = abs(sin(x)/100/3.1416);
}
```



```
float compute(int q)
{
   return abs(sin(q)/100/3.1416);
}
```

Embora cada laço execute a mesma função, a versão 2 é muito mais rápida, porque a sobrecarga das seqüências de chamada e retorno foi eliminada pelo uso de código em linha.

Veja outro exemplo, dessa vez com o código assembly gerado pelo compilador Borland C/C++. Este programa

```
max(int a, int  b);

void main(void)
{
  int x;

  x = max(10, 20);
}

max(int a, int b)
{
  return a>b ? a : b;
}
```

produz o seguinte código assembly. As seqüências de chamada e retorno são indicadas por comentários, começando com asteriscos acrescentados pelo autor. Como você pode observar, elas formam uma parte considerável do código do programa.

```
        .286p
        ifndef    ??version
?debug  macro
        endm
publicdll macro name
        public  name
        endm
$comm   macro   name,dist,size,count
        comm    dist name: BYTE:count *size
        endm
        else
$comm   macro   name,dist,size,count
        comm    dist name[size]:BYTE:count
        endm
        endif
```

```
        ?debug V 300h
        ?debug S "funções.C"
        ?debug C E90765441F0966756E636F65732E63
_TEXT   segment byte public 'CODE'
_TEXT   ends
DGROUP  group   _DATA,_BSS
        assume  cs:_TEXT,ds:DGROUP
_DATA   segment word public 'DATA'
d@      label   byte
d@w     label   word
_DATA   ends
_BSS    segment word public 'BSS'
b@      label   byte
b@w     label   word
_BSS    ends
_TEXT   segment byte public 'CODE'
   ;
   ;    void main(void)
   ;
        assume cs:_TEXT
_main   proc   near
        enter  2,0
   ;
   ;    {
   ;      int x;
   ;
   ;      x = max(10,20);
   ;
; ***********************************************
; Este é o começo da seqüência de chamada.
; ***********************************************
         push    20
         push    10
         call    near ptr _max
; ***********************************************
;
;
; ***********************************************
; A próxima linha faz parte da seqüência de retorno.
; ***********************************************
         add     sp,4
         mov     word ptr [bp-2],ax
         ;
         ; }
         ;
         leave
         ret
_main    endp
   ;
```



```
   ;      max(int a, int b)

          assume  cs:_TEXT
_max      proc    near
; ***********************************************
; Mais da seqüência de chamada.
; ***********************************************
        push    bp
        mov     bp,sp
        mov     dx,word ptr [bp+4]
        mov     bx,word ptr [bp+6]
; ***********************************************
   ;
   ;    {
   ;      return a>b ? a : b;
   ;
        cmp     dx,bx
        jle     short @2@86
        mov     ax,dx
        jmp     short @2@114
@2@86:
; ***********************************************
; Aqui está a primeira parte da seqüência de retorno.
; ***********************************************
        mov     ax,bx
@2@114:
        jmp     short @2@142
@2@142:
   ;
   ;    }
   ;
         pop    bp
         ret
_max     endp
         ?debug C E9
         ?debug C FA00000000
_TEXT    ends
_DATA    segment word public 'DATA'
s@       label   byte
_DATA    ends
_TEXT    segment byte public 'CODE'
_TEXT    ends
         public _main
         public _max
_s@      equ     s@
         end
```

O código real produzido depende de como o compilador é implementado e de que processador está sendo utilizado, mas geralmente segue o mesmo padrão desse exemplo.

Neste momento, você poderia pensar que deve escrever programas com apenas poucas funções maiores e, assim, eles rodariam mais rápido. Isso provavelmente não é uma boa idéia. Primeiro, na grande maioria dos casos a pequena diferença de tempo obtida por meio do uso de grandes funções não é significativa e a perda de estrutura é considerável. Mas há um outro problema. A substituição de funções, que são usadas por diversas rotinas, por código em linha pode fazer com que seu programa se torne muito grande, pois o mesmo código é duplicado diversas vezes. Tenha em mente que as sub-rotinas foram inventadas principalmente como forma de fazer um uso mais eficiente da memória. Isso explica por que, como regra geral, tornar seu programa mais rápido significa fazê-lo maior, enquanto torná-lo menor significa fazê-lo mais lento.

Como análise final, só faz sentido utilizar código em linha em lugar de uma chamada a uma função quando a velocidade é de absoluta prioridade. Caso contrário, o uso deliberado de funções é definitivamente recomendado.

Programas Portáveis

É comum que um programa escrito para uma máquina seja transportado para outra com um processador ou sistema operacional diferente, ou ambos. Esse processo é chamado de *portabilidade* e pode ser muito fácil ou extremamente difícil de ser obtido, dependendo de como o programa foi originalmente escrito. Um programa que pode ser facilmente transportado é chamado de *portável*. Um programa não é muito portável quando contém numerosas *dependências da máquina* — fragmentos de código que funcionam somente com um sistema operacional e processador específico. C lhe permite criar códigos portáveis, mas, para isso, é necessário cuidado e atenção aos detalhes. Esta seção examina algumas áreas com problemas específicos e oferece algumas soluções.

Usando #define

Talvez a maneira mais fácil de tornar programas portáveis seja fazer uma diretiva de substituição de macro **#define** para todos os "números mágicos" dependentes do processador ou do sistema. Esses "números mágicos" incluem tamanho dos buffers para acesso a disco, comandos especiais para tela e teclado, informações sobre alocação de memória e qualquer outra coisa que tenha a menor possibilidade de ser trocada quando o programa for transportado. Esses **#defines** não só

tornam claros os números mágicos para a pessoa que está fazendo o transporte como também simplificam o trabalho de edição, porque seus valores têm de ser trocados apenas uma vez e não ao longo de todo o programa.

Por exemplo, aqui está um comando **fread()** que é inerentemente não-portável:

```
fread(buf, 128, 1, fp);
```

O problema é que o tamanho do buffer, 128, está fixado em **fread()**. Isso pode funcionar para um sistema operacional, mas não ser a melhor opção para outro. A melhor maneira para codificar essa função é vista aqui:

```
#define BUF_SIZE 128

fread(buf, BUF_SIZE, 1, fp);
```

Nesse caso, quando for necessário mover-se para um sistema diferente, apenas o **#define** tem de mudar e todas as referências a **BUF_SIZE** são automaticamente corrigidas. Isso não só torna mais fácil alterar como também evita muitos erros de edição. Lembre-se de que, provavelmente, haverá muitas referências a **BUF_SIZE** em um programa real, assim, o ganho em portabilidade é significativo.

Dependências do Sistema Operacional

Virtualmente todos os programas comerciais contêm código específico para o sistema operacional sob o qual são executados. Por exemplo, uma planilha para MS-DOS pode chamar diretamente rotinas da BIOS para conseguir mudar o modo de vídeo mais rapidamente, ou um aplicativo de contabilidade para Windows pode usar fontes especiais disponíveis somente nesse ambiente. O ponto é que algumas dependências em relação ao sistema operacional são necessárias para programas que sejam bons, rápidos e comercialmente viáveis. Mas as dependências para com os sistemas operacionais também dificultam a portabilidade dos seus programas.

Embora não exista nenhuma regra simples que você possa seguir para minimizar as suas dependências em relação ao sistema operacional, existe um conselho que pode ser oferecido: separe as partes de seu programa que lidam diretamente com a sua aplicação das partes que formam a interface com o sistema operacional. Desta forma, se você portar seu programa para um novo ambiente, precisará modificar apenas os módulos de interface.

Diferenças no Tamanho dos Dados

Se você quiser escrever código portátil, você nunca deve fazer suposições sobre o tamanho dos tipos de dados. Como você provavelmente sabe, o tamanho de uma palavra em um processador de 16 bits é de 16 bits; para um processador de 32 bits será de 32 bits. Como o tamanho de uma palavra tende a ser o mesmo tamanho de um inteiro, código que suponha que um inteiro tem 16 bits, por exemplo, não funcionará quando portado para um ambiente de 32 bits. Para evitar dependências do tamanho, use **sizeof** sempre que seu programa precisa saber quantos bytes são ocupados por alguma coisa. Por exemplo, este comando escreve um inteiro para um arquivo em disco e funciona em qualquer ambiente:

```
fwrite(&i, sizeof(int), 1, stream);
```

Algumas vezes, porém, não é possível criar um código portátil, mesmo com **sizeof**. Por exemplo, esta função (que troca os bytes de um inteiro) funciona com inteiros de 16 bits, mas falha com inteiros de 32 bits.

```
void swap_bytes(int *x)
{
  union sb {
    int t;
    unsigned char c[2];
  } swap;

  unsigned char temp;

  swap.t = *x;
  temp = swap.c[1];
  swap.c[1] = swap.c[0];
  swap.c[0] = temp;
  *x = swap.t;
}
```

Se você sabe de antemão que precisará de uma versão de 32 bits desta função, precisar criar uma segunda versão e então usar uma diretiva de compilação condicional (tal como **#ifdef**) para compilar a versão correta para cada ambiente.

Depuração

Plageando Thomas Edison, programação é 10% de inspiração e 90% de depuração. Todos os programadores realmente bons são bons depuradores. Os tipos de erros (bugs) que podem facilmente ocorrer quando se está usando C são o tópico desta seção.

Erros de Ordem de Processamento

Os operadores de incremento e decremento são utilizados na maioria dos programas escritos em C, e a ordem em que as operações ocorrem é afetada caso esses operadores precedam ou sigam a variável. Considere o seguinte:

```
y = 10;          y = 10;
x = y++;         x = ++y;
```

Esses dois comandos não são iguais. O primeiro atribui o valor **10** a **x** e, então, incrementa **y**. O segundo incrementa **y** para **11** e atribui o valor **11** a **x**. Portanto, no primeiro caso, **x** contém **10**; no segundo, **x** contém **11**. A regra é que as operações de incremento e decremento ocorrem antes das outras operações, se elas precedem o operando; caso contrário, ocorrem depois.

Um erro de ordem de processamento normalmente ocorre quando são feitas alterações em um comando já existente. Por exemplo, o comando

```
x = *p++;
```

atribui o valor apontado por **p** a **x** e incrementa o ponteiro **p**. Imagine, porém, que, mais tarde, você decida que **x** precisa realmente do valor apontado por **p** vezes o valor apontado por **p**. Para tanto, você tenta

```
x = *p++ * (*p);
```

Porém, isso não funciona, porque **p** já foi incrementado. A solução apropriada é escrever

```
x = *p * (*p++);
```

Erros como esse podem ser muito difíceis de encontrar. Pode haver indícios como em laços que não rodam corretamente ou rotinas que erram por um. Se tem qualquer dúvida sobre um comando, recodifique-o de maneira a sentir-se seguro.

Problemas com Ponteiros

Um erro muito comum em programas em C é o mau uso de ponteiros. Problemas com ponteiros recaem em duas categorias gerais. A má compreensão da indireção e dos operadores de ponteiros e o uso acidental de ponteiros inválidos. A solução para o primeiro problema é entender a linguagem C; a solução para o segundo é sempre verificar a validade de um ponteiro antes de usá-lo.

O que segue é um típico erro de programação em C:

```
/* Este programa tem um erro. */
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
  char *p;

  *p = malloc(100); /* esta linha está errada */
  gets(p);
  printf(p);
}
```

Esse programa muito provavelmente quebrará (crash) — possivelmente levando consigo o sistema operacional. A razão é que o endereço devolvido por **malloc()** não foi atribuído a **p**, mas à posição de memória apontada por **p**, que, nesse caso, é completamente desconhecida. Para corrigir esse programa, você deve substituir

```
p = malloc(100); /* isto está correto */
```

na linha incorreta.

O programa também contém um segundo e mais insidioso erro. Não há verificação, em tempo de execução, do endereço devolvido por **malloc()**. Lembre-se: se a memória está cheia, **malloc()** devolve **NULL**, que nunca é um ponteiro válido em C. O problema ocasionado por esse tipo de erro é muito complicado, porque raramente ocorre, só quando um pedido de alocação falha. A melhor maneira de trabalhar com esse tipo de problema é evitá-lo. O que segue é uma versão corrigida do programa, que inclui uma verificação da validade do ponteiro:


```
/* Este programa está correto. */

#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  char *p;

  p = malloc(100); /* esta linha está correta */

  if(!p) {
    printf("Sem memória.\n");
    exit(1);
  }

  gets(p);
  printf(p);
}
```

A coisa terrível com os ponteiros "selvagens" é que eles são muito difíceis de rastrear. Se você estiver fazendo atribuições a uma variável ponteiro que não contém um endereço válido, seu programa parecerá funcionar corretamente algumas vezes e quebrará em outras. Quanto menor seu programa, maior a probabilidade de ele rodar corretamente, mesmo com um ponteiro extraviado. Isso ocorre porque muito pouca memória está sendo utilizada e as chances de essa memória ser utilizada por alguma outra coisa são estatisticamente pequenas. Conforme seu programa cresce, as falhas tornam-se mais comuns, mas, ao tentar depurar, você pensará nos acréscimos recentes ou alterações no seu programa e não em erros de ponteiros. Assim, você tenderá a procurar o erro no lugar errado.

Uma maneira de reconhecer um problema com ponteiros é lembrar que os erros tendem a ser erráticos. Seu programa poderá funcionar corretamente uma vez e erradamente em outra. Algumas vezes outras variáveis conterão "lixo", sem razão aparente. Se esses problemas ocorrerem, verifique seus ponteiros. Como regra de procedimento, você sempre deve verificar todos os ponteiros quando os erros começarem a aparecer.

Embora os ponteiros possam ser problemáticos, eles são também um dos aspectos mais poderosos da linguagem C e valem a pena, apesar de todos os problemas que podem causar. Faça um esforço no início para aprender a usá-los corretamente.

Um último ponto a lembrar sobre ponteiros é que você deve inicializá-los antes de serem utilizados. Considere o fragmento de código:

```
int *x;

*x = 100;
```

Isso será um desastre, porque você não sabe para onde **x** está apontando. Atribuir um valor a uma posição desconhecida provavelmente destruirá alguma coisa de valor, como outro código ou dado do programa.

Erros Bizarros de Sintaxe

Ocasionalmente, você verá um erro de sintaxe que não conseguirá entender ou mesmo saber por que é um erro. Algumas vezes, o próprio compilador C pode ter um erro que o leva a apresentar falsos erros. A única maneira de contornar isso é refazer seu código. Outros erros não usuais requerem simplesmente uma busca mais detalhada para serem encontrados.

Um erro particularmente desconcertante ocorre quando você tenta compilar o seguinte código.

```
char *myfunc(void);

void main(void)
{
    .
    .
    .
}

myfunc(void) /* erro informado aqui */
{
    .
    .
    .
}
```

Seu compilador emitirá uma mensagem de erro semelhante a **Type mismatch in redeclaration of f()** em referência à linha indicada na listagem. Como pode ser isso? Não há duas **myfunc()**s. A resposta é que **myfunc()** foi declarada como devolvendo um ponteiro a caractere no início do programa. Essa declaração provoca uma entrada na tabela de símbolos com essa informação. Quando o compilador encontra **myfunc()**, mais tarde, dentro do programa, não há nenhuma indicação de que **myfunc()** devolverá outra coisa que não seja um inteiro, o tipo padrão. Logo, você estará "redeclarando" ou "redefinindo" a função.

Outro erro de sintaxe que é difícil de entender aparece no seguinte código:

```
/* Este programa tem um erro de sintaxe. */
#include <stdio.h>

void func1(void);

void main(void)
{
  func1();
}

void func1(void);
{
  printf("isto é func1 \n");
}
```

O erro aqui é o ponto-e-vírgula após a declaração de **func1()**. O compilador verá isso como um comando fora de qualquer função, o que caracteriza um erro. Porém, a maneira como o erro será apresentado varia entre compiladores. Muitos compiladores mostrarão uma mensagem de erro, como **bad declaration syntax** (declaração com sintaxe errada), apontando para a primeira chave após **func1()**. Como você está acostumado a ver pontos-e-vírgulas depois de declarações, poderá ser difícil ver a fonte desse erro.

Erros por Um

Como você deve saber, em C, todas as indexações de matrizes começam em 0. Um erro comum aparece, como o uso do laço **for** para acessar os elementos de uma matriz. Considere o programa seguinte, que inicializa uma matriz de 100 inteiros:

```
/* Este programa não funcionará. */

void main(void)
{
  int x, num[100];

  for(x=1; x<=100; ++x) num[x]=x;
}
```

O laço **for**, nesse programa, está errado por duas razões. Primeiro, ele não inicializa **num[0]**, o primeiro elemento da matriz **num**. Segundo, ele passa do final da matriz, porque **num[99]** é o último elemento. A maneira correta de escrever este programa é

```
/* Isto está certo. */

void main(void)
{
  int x, num[100];

  for(x=0; x<100; ++x) num[x]=x;
}
```

Lembre-se: uma matriz de 100 elementos vai de 0 a 99.

Erros de Limites

O ambiente de tempo de execução de C e muitas funções da biblioteca padrão fazem pouca ou nenhuma verificação de limites. Por exemplo, é possível escrever após o final de matrizes. Considere o programa seguinte, que lê uma string do teclado e exibe-a na tela.

```
#include <stdio.h>

void main(void)
{
  int var1;
  char s[10];
  int var2;

  var1 = 10;  var2 = 10;
  gets(s);
  printf("%s %d %s", s, var1, var2);
}
```

Nesse caso, não há nenhum erro de código. Indiretamente, porém, um erro pode ser provocado ao chamar **gets()** com **s**. No programa, **s** é declarada com 10 caracteres de comprimento, mas, e se o usuário digitar mais de 10 caracteres? Isso fará com que o limite de **s** seja ultrapassado. O problema real é que **s** pode exibir todos os caracteres, mas nem **var1** nem **var2** possuirão valores corretos. Virtualmente, todos os compiladores C usam a pilha para armazenar variáveis locais. As variáveis **var1**, **var2** e **s** estarão localizadas na memória, como mostra a Figura 27.1.

Seu compilador C pode inverter a ordem de **var1** e **var2**, mas elas ainda estarão ao redor de **s**. Quando se escreve além do final de **s**, a informação adicional é colocada na área que é reservada a **var2**, destruindo qualquer conteúdo anterior. Assim, em lugar de imprimir o número 10 para ambas as variáveis

inteiras, o programa exibirá algo a mais para aquela variável destruída pela escrita em s. Além disso, nesse exemplo específico, o endereço de retorno da função também pode ser perdido, provocando uma quebra.

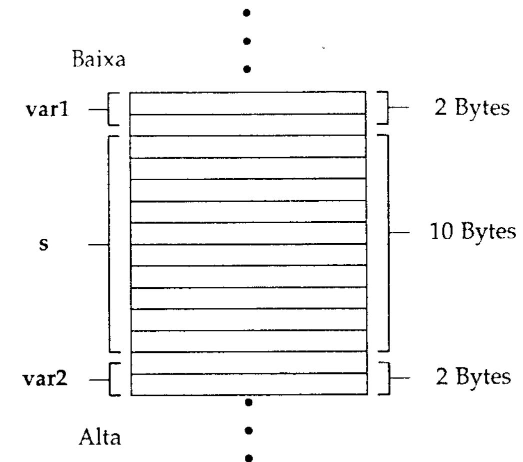


Figura 27.1 As variáveis **var1**, **var2** e **s** na memória.

Omissão de Protótipo de Função

Sempre que uma função devolve um tipo de valor diferente de inteiro, ela deve ser declarada como tal usando seu protótipo. Se você não incluir um protótipo de função, o compilador assumirá que ela devolve um inteiro. Considere o programa seguinte, que multiplica dois números em ponto flutuante:

```
/* Este programa está errado. */
#include <stdio.h>

void main(void)
{
  float x, y;
  scanf(%f%f", &x, &y);
  printf("%f", mul(x, y));
}

float mul(float a, float b)
{
```

```
    return a*b;
}
```

Aqui, **main()** espera um valor inteiro de **mul()**, mas **mul()** devolve um número em ponto flutuante. Você obterá respostas sem sentido porque **main()** copiará apenas 2 bytes dos 8 necessários para um **float**. Embora C perceba esse erro, caso as funções estejam no mesmo arquivo, isso não é possível se elas estiverem em módulos compilados separadamente e nenhum protótipo para **mul()** tenha sido incluído. O programa anterior é uma excelente ilustração de por que é preciso colocar protótipos para todas as funções em um programa.

A maneira de corrigir esse programa é pôr um protótipo para **mul()**. A versão corrigida é mostrada aqui:

```
/* Este programa está correto. */
#include <stdio.h>

float mul(float a, float b);

void main(void)
{
  float x, y;

  scanf(%f%f", &x, &y);
  printf("%f", mul(x, y));
}

float mul(float a, float b)
{
  return a*b;
}
```

Nesse caso, o protótipo diz a **main()** que **mul()** devolve um valor em ponto flutuante.

Erros de Argumentos

Você deve certificar-se de casar o tipo de argumento que uma função espera com o tipo que a ela é passado. Um exemplo importante é **scanf()**. Lembre-se de que **scanf()** espera receber os *endereços* de seus argumentos, não seus valores. Por exemplo,

```
int x;
char string[10];
```



```
scanf("%d%s", x, string);
```

está errado, enquanto

```
scanf("%d%s", &x, string);
```

está correto. Recorde que strings já passam seus endereços para funções, assim, você não deve utilizar o operador & nesse caso.

Colisões entre a Pilha e o Heap

Embora alguns compiladores não permitam que a pilha colida com o heap, muitos não fazem nenhuma verificação. Quando isso acontece, o programa "morre" completamente ou continua a rodar de modo estranho. Esse segundo sintoma ocorre quando um dado é usado acidentalmente como endereço de retorno.

O que há de pior nas colisões entre a pilha e o heap é que, geralmente, ocorrem sem nenhum aviso e "matam" o programa completamente, de forma que o código de depuração não pode ser executado. Outro problema é que as colisões entre a pilha e o heap aparecem freqüentemente como ponteiros "selvagens", o que o desencaminha. A única coisa que se pode dizer é que a maioria das colisões entre a pilha e o heap é provocada por funções recursivas desgovernadas. Se seu programa usa recursão e ocorrem falhas inexplicáveis, verifique as condições para terminação de suas funções recursivas.

Teoria Geral de Depuração

Cada um tem uma abordagem diferente para programar e depurar. Porém, certas técnicas têm provado ser melhores do que outras. No caso da depuração, testes incrementais são considerados os de menor custo e os mais eficientes, muito embora aparentem retardar o processo de desenvolvimento.

Teste incremental é o processo de sempre ter um programa funcionando. Isto é, no início do processo de desenvolvimento é estabelecida uma unidade operacional. *Uma unidade operacional* é simplesmente uma porção de código que trabalha. Quando um novo código é acrescentado a essa unidade, ele é testado e depurado. Dessa forma, o programador pode encontrar erros facilmente, porque eles provavelmente ocorrerão no código acrescentado ou na forma em que ele interage com a unidade operacional.

O tempo de depuração é proporcional ao número total de linhas nas quais o *bug* (erro) poderia estar com os testes incrementais, é possível restringir o número de linhas de código a apenas aquelas que foram recentemente adicionadas — isto é, fora da unidade operacional. Essa situação é mostrada na Figura 27.2. No processo de depuração, você, como programador, deve trabalhar com a menor área possível. Por meio dos testes incrementais, é possível subtrair a área já testada da área total, reduzindo, dessa forma, a região em que um erro pode ser encontrado.

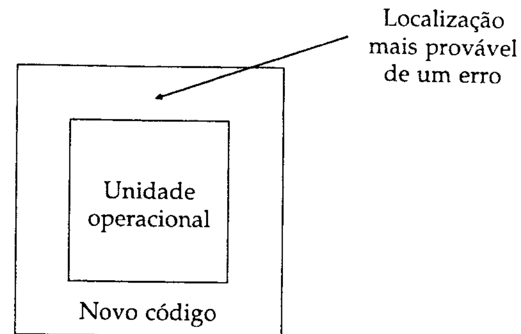


Figura 27.2 A posição mais provável para um erro quando são usados testes incrementais.

Em grandes projetos, há freqüentemente vários módulos que têm pouca interação. Nestes casos, você pode estabelecer várias unidades operacionais, para permitir desenvolvimento simultâneo.

O teste incremental é simplesmente o processo de sempre ter um código funcionando. Logo que for possível executar uma parte do seu programa, você deverá fazê-lo, testando essa parte completamente. Conforme for aumentando o programa, continue testando as novas partes bem como o modo como elas se relacionam com o código operacional. Dessa forma, você concentrará possíveis erros em uma área de código pequena. Naturalmente, você deverá estar sempre alerta para a possibilidade de deixar passar um erro na unidade operacional. Mas você reduziu a probabilidade de isso ocorrer.

A Arte da Manutenção de Programas

Uma vez que um programa tenha sido escrito, testado, depurado e — finalmente — julgado pronto para ser usado, a fase de desenvolvimento do programa acabou e começa sua fase de manutenção. A maioria dos programadores gosta do encanto de desenvolver um programa novo, mas evita ser aquele a fazer a sua manutenção. Isso ocorre, em parte, porque a fase de manutenção nunca termina. Quando um programa está sendo desenvolvido, mesmo um muito grande, há sempre uma luz no fim do túnel. Algum dia o programa estará pronto. No entanto, a fase de manutenção é uma luta diária de evasivas, irregularidades, erros e "bugs". O programador de suporte pode nunca sentir a emoção da realização e a melhor parte do dia talvez seja a hora de sair. Por mais desolador que possa parecer, a manutenção de programas poderá ser uma tarefa desafiante e gratificante se for abordada corretamente.

O programador de suporte tem duas responsabilidades:

- Corrigir erros
- Fornecer proteção ao código-fonte

Consertando Erros

Todo programa não-trivial tem erros. Essa é uma verdade que não se pode provar, porém inseparável da ciência da computação. O que torna difícil a manutenção de um programa é que todos os erros simples são encontrados durante o estágio de desenvolvimento. Os erros que o programador de suporte tem de encontrar e consertar são, geralmente, obscuros e só surgem sob circunstâncias diabolicamente complexas e difíceis de recriar. Se você gosta de verdadeiros desafios, talvez a manutenção de programas seja o que você estava esperando.

Existem basicamente três tipos de erros: aqueles que você tem de consertar (categoria 1), aqueles que você gostaria de consertar (categoria 2) e aqueles que você não precisa necessariamente preocupar-se (categoria 3). Os erros da categoria 1 quebram o sistema, escrevem lixo nos discos ou destroem dados. Por exemplo, o erro que faz com que um programa ocasionalmente destrua o arquivo em disco do banco de dados deve ser consertado, porque ele simplesmente torna o programa inaproveitável. Os erros da categoria 2 são consertados apenas quando não há erros da categoria 1 para consertar. Um erro da categoria 2 é aquele que faz com que um processador de texto, em raras ocasiões, reformate incorretamente um parágrafo. Nada é perdido, o programa não morre e o usuário faz um reparo manual. Os erros da categoria 2 são, na sua maioria, um incômodo para o usuário, porém, podem ser contornados. Esses erros devem ser consertados, mas

não são a prioridade. Finalmente, os erros da categoria 3 trazem apenas incômodos, como um processador de texto que sempre ejeta uma folha extra de papel no final de uma impressão. Certamente, papel custa dinheiro, mas não muito. Um outro tipo de erro da categoria 3 não é realmente um erro, mas uma diferença entre a forma como a documentação diz que o programa funcionará e como ele realmente funciona. Os erros da categoria 3 raramente são consertados; não porque não o devam ser, mas porque sempre há muito mais erros nas categorias 1 e 2.

Se você puder organizar os erros que encontrar nessas três categorias, poderá dividir seu tempo em conformidade.

Proteção do Código-Fonte

O programador de suporte está freqüentemente incumbido do código-fonte do programa. Embora a maioria das companhias ponha uma cópia do código-fonte da versão atual do programa em um cofre de banco, esta normalmente é desatualizada se alguma vez for necessária. Geralmente, o código no banco é visto como último recurso. O programador de suporte é encarregado de proteger o código-fonte da companhia. O que realmente significa não perdê-lo!

A maneira mais comum de um código-fonte perder-se é durante o processo de correção dos erros. Funciona desta forma: o programador A "conserta" um erro. No processo, sem que A perceba, um erro de edição apaga cinco linhas do código em algum outro lugar do arquivo. O programador A compila o programa e verifica se o erro foi consertado. O erro parece ter sido consertado, assim — e aqui está a parte importante —, o programador A copia o código-fonte "consertado" do diretório de trabalho para o diretório de armazenamento. Agora, cinco linhas de código estão faltando e o programa decididamente tem um novo erro. Mas antes que isso seja descoberto, o programador B, cujo trabalho é fazer o backup do disco rígido, copia a versão mutilada no disco de armazenamento externo. Agora, a versão antiga do código-fonte está realmente perdida.

Só há uma maneira de evitar o incidente anterior: nunca destruir versões antigas do programa. O erro de fato cometido, além da edição descuidada, não foi o programador A ter copiado o código-fonte alterado de volta ao diretório de armazenanento. O erro ocorreu quando o programador B escreveu sobre a versão anterior na mídia de armazenamento externo.

Eis como você deve trabalhar com o código-fonte de um programa em desenvolvimento para evitar sua perda. Primeiro, devem-se criar três diretórios. O primeiro contém a versão do programa atualmente em uso. Esse diretório só é atualizado quando uma nova versão é liberada. O segundo diretório contém uma versão estável, porém não liberada do programa. O terceiro contém o código em desenvolvimento. Em seguida, deve-se realizar o armazenamento externo de backup em um período regular — semanalmente, por exemplo — sempre usando um disco (ou fita) novo. Mantenha guardados todos os backups anteriores. Dessa forma, o armazenamento externo nunca está mais que cinco dias atrasado, caso seja necessária uma recuperação.