Técnicas de Programação

TP1301

Prof. Giovane Barcelos
giovane_barcelos@uniritter.edu.br

Plano de Ensino
Conteúdo programático

1. Introdução à programação em C 2. Desenvolvimento estruturado de programas em C 3. Controle de programa 4. Funções 5. Arrays 6. Ponteiros	N1
7. Caracteres e strings 8. Entrada/Saída formatada 9. Estruturas, uniões, manipulações de bits e enumerações 10. Processamento de arquivos 11. Estruturas de dados 12. O pré-processador 13. Outros tópicos sobre C	N2

Objetivos

- A redirecionar a entrada do teclado para que venha de um arquivo.
- A redirecionar a saída da tela para que seja gravada em um arquivo.
- A escrever funções que usem listas de argumentos de tamanho variável.
- A processar argumentos da linha de comandos.
- A atribuir tipos específicos a constantes numéricas.
- A usar arquivos temporários.
- A processar eventos assíncronos externos em um programa.
- A alocar memória para arrays dinamicamente.
- A mudar o tamanho da memória que já tenha sido alocada dinamicamente.

Introdução

- **Muitos dos recursos aqui discutidos são aplicáveis em sistemas operacionais específicos, especialmente Linux/UNIX e Windows.**

Redirecionamento de entrada/saída

- Normalmente, a entrada de dados em um programa é feita por meio do teclado (entrada-padrão), e a saída de dados de um programa é exibida na tela (saída-padrão).
- Na maioria dos sistemas operacionais de computadores — em particular nos sistemas Linux/UNIX e Windows —, é possível redirecionar a entrada de dados para que sejam lidos de arquivos, e não do teclado, e redirecionar as saídas para que sejam armazenadas em arquivos, em vez de serem enviadas para a tela.
- Ambas as formas de redirecionamento podem ser completadas sem o uso dos recursos de processamento de arquivos da biblioteca-padrão.
- Existem várias maneiras de redirecionar entrada e saída a partir da linha de comando

Redirecionamento de entrada/saída

- **Considere o arquivo executável sum (nos sistemas Linux/UNIX), que lê números inteiros, um por vez, e acumula o total dos valores lidos até que um indicador de final de arquivo seja encontrado e, então, imprima o resultado.**
- **Normalmente, o usuário digita números inteiros no teclado, bem como o indicador de final de arquivo, sendo este uma combinação de teclas que indica que não há mais valores a serem lidos.**
- **Com o redirecionamento de entrada, esses dados podem ser armazenados em um arquivo.**

Redirecionamento de entrada/saída

- Por exemplo, se os dados são armazenados no arquivo input, a linha de comando
  - $ sum < input

  faz com que o programa sum; o símbolo de redirecionamento de entrada (<) indica que os dados do arquivo input devem ser usados como os dados de entrada pelo programa.
- O redirecionamento da entrada em um sistema Windows é executado de forma idêntica.
- O caractere $ é o prompt da linha de comando do Linux/UNIX (alguns sistemas usam % como prompt, ou outro símbolo).
- O segundo método de redirecionamento de entrada é a canalização (ou piping).
- Um pipe (|) faz com que a saída de um programa seja redirecionada como entrada de outro programa.

Redirecionamento de entrada/saída

- Suponha que o programa random tenha como saída uma série de números inteiros aleatórios; a saída do programa random pode ser 'canalizada' diretamente para o programa sum usando a linha de comando
  - $ random | sum
- Isso faz com que a soma dos números inteiros produzidos por random seja calculada.
- O uso de pipes pode ser feito no Linux/
- UNIX e no Windows.
- A saída de um programa pode ser redirecionada para um arquivo com o uso do símbolo de redirecionamento de saída (>).
- Por exemplo, para redirecionar a saída do programa random para o arquivo out, use
  - $ random > out

Redirecionamento de entrada/saída

- Por fim, a saída de um programa pode ser acrescentada ao final de um arquivo que já exista, utilizando-se o símbolo de acréscimo (>>).
- Por exemplo, para acrescentar a saída do programa random ao final do arquivo out criado na linha de comando acima, use a linha de comando
  - $ random >> out

Listas de argumentos de tamanhos variáveis de entrada/saída

- É possível criar funções que recebam uma quantidade não especificada de argumentos.
- A maior parte dos programas no texto usa a função printf da biblioteca-padrão que, como você sabe, utiliza uma quantidade variável de argumentos.
- No mínimo, printf precisa receber uma string como seu primeiro argumento, mas printf pode receber qualquer número de argumentos adicionais.
- O protótipo de função para printf é
  - `int printf( const char *format, ... );`
- Em um protótipo de função, as reticências (...) indicam que a função recebe um número variável de argumentos de qualquer tipo.

Listas de argumentos de tamanhos variáveis de entrada/saída

- Note que as reticências sempre devem ser colocadas ao final da lista de argumentos.
- Macros e definições de cabeçalhos de argumentos variáveis <stdarg.h> (Figura 14.1) oferecem os recursos necessários para a construção de funções com listas de argumentos de tamanho variável.
- A Figura 14.2 demonstra a função average (linhas 26-41), que recebe um número variável de argumentos.
- O primeiro argumento de average é sempre o número de valores que terão suas médias calculadas.

Listas de argumentos de tamanhos variáveis de entrada/saída

Identificador	Explicação
`va_list`	Tipo adequado para a armazenagem das informações necessárias para as macros `va_start`, `va_arg` e `va_end`. Para acessar os argumentos em uma lista de argumentos de tamanhos variáveis, deve-se declarar um objeto do tipo `va_list`.
`va_start`	Uma macro que é chamada antes que se possa acessar os argumentos de uma lista de argumentos de tamanhos variáveis. A macro inicializa o objeto declarado com `va_list` para ser utilizado pelas macros `va_arg` e `va_end`.
`va_arg`	Uma macro que é expandida em uma expressão com os mesmos valor e tipo do próximo argumento da lista de argumentos de tamanhos variáveis. Cada chamada de `va_arg` modifica o objeto declarado com `va_list`, de modo que ele passe a apontar para o próximo argumento na lista.
`va_end`	Uma macro que facilita o retorno normal de uma função cuja lista de argumentos de tamanhos variáveis foi referenciada pela macro `va_start`.

Figura 14.1 ■ O tipo e as macros da lista de argumentos de tamanhos variáveis definidos em `stdarg.h`.

Listas de argumentos de tamanhos variáveis de entrada/saída

```
1  /* Fig. 14.2: fig14_02.c
2     Usando listas de argumentos de tamanhos variáveis */
3  #include <stdio.h>
4  #include <stdarg.h>
5
6  double average( int i, ... ); /* protótipo */
7
8  int main( void )
9  {
10    double w = 37.5;
11    double x = 22.5;
12    double y = 1.7;
13    double z = 10.2;
14
```

Figura 14.2 ■ Uso das listas de argumentos de tamanhos variáveis. (Parte 1 de 2.)

Listas de argumentos de tamanhos variáveis de entrada/saída

```
15      printf( "%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
16         "w = ", w, "x = ", x, "y = ", y, "z = ", z );
17      printf( "%s%.3f\n%s%.3f\n%s%.3f\n",
18         "A média de w e x is ", average( 2, w, x ),
19         "A média de w, x e y é ", average( 3, w, x, y ),
20         "A média de w, x, y e z é ",
21         average( 4, w, x, y, z ) );
22      return 0; /* indica conclusão bem-sucedida */
23   } /* fim do main */
24
25   /* calcula média */
26   double average( int i, ... )
27   {
28      double total = 0; /* inicializa total */
29      int j; /* contador para selecionar argumentos */
30      va_list ap; /* armazena informações necessárias para va_start e va_end */
31
32      va_start( ap, i ); /* inicializa o objeto va_list */
33
34      /* processa lista de argumentos de tamanho variável */
35      for ( j = 1; j <= i; j++ ) {
36         total += va_arg( ap, double );
37      } /* fim do for */
38
39      va_end( ap ); /* limpa lista de argumentos de tamanho variável */
40      return total / i; /* calcula média */
41   } /* fim da função average */
```

Listas de argumentos de tamanhos variáveis de entrada/saída

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

A média de w e x é 30.000
A média de w, x e y é 20.567
A média de w, x, y e z é 17.975
```

Figura 14.2 ■ Uso das listas de argumentos de tamanhos variáveis. (Parte 2 de 2.)

Listas de argumentos de tamanhos variáveis de entrada/saída

- A função average (linhas 26-41) usa todas as definições e macros do cabeçalho <stdarg.h>.
- O objeto ap, do tipo va_list (linha 30), é usado pelas macros va_start, va_arg e va_end para processar a lista de argumentos de tamanhos variáveis da função average.
- A função começa chamando va_start (linha 32) para inicializar o objeto ap que será utilizado em va_arg e em va_end.
- A macro recebe dois argumentos — o objeto ap e o identificador do argumento mais à direita da lista de argumentos antes das reticências — i nesse caso (va_start usa i aqui para determinar o início da lista de argumentos).

Listas de argumentos de tamanhos variáveis de entrada/saída

- Em seguida, a função average repetidamente, adiciona argumentos da lista de argumentos de tamanhos variáveis à variável total (linhas 37–39).
- O valor a ser adicionado em total é recuperado por meio da lista de argumentos, chamando-se a macro va_arg.
- A macro va_arg recebe dois argumentos — o objeto ap e o tipo de valor esperado na lista de argumentos (double, nesse caso).
- A macro retorna o valor do argumento.
- A função average chama a macro va_end (linha 39) com o objeto ap como argumento para facilitar o retorno normal de average para main.
- Finalmente, a média é calculada e retornada para main.

Listas de argumentos de tamanhos variáveis de entrada/saída

Erro comum de programação 14.1

Colocar as reticências no meio da lista de parâmetros da função consiste em um erro de sintaxe. As reticências só podem ser colocadas ao final da lista de parâmetros.

Listas de argumentos de tamanhos variáveis de entrada/saída

- O leitor poderá questionar como as funções printf e scanf sabem que tipo usar em cada macro va_arg.
- A resposta é que printf e scanf varrem os especificadores de conversão de formato na string de controle de formato para determinar o tipo do próximo argumento a ser processado.

Uso de argumentos na linha de comando

- Em muitos sistemas, é possível passar argumentos para main a partir de uma linha de comandos ao incluir parâmetros int argc e char *argv[] na lista de parâmetros de main.
- O parâmetro argc recebe o número de argumentos da linha de comando.
- O parâmetro argv é um array de strings onde foram armazenados os argumentos efetivos da linha de comando.
- O uso comum de argumentos da linha de comando inclui a passagem de opções para o programa e a passagem de nomes de arquivos para o programa.
- A Figura 14.3 copia um arquivo em outro, um caractere por vez.

Uso de argumentos na linha de comando

- O nome do arquivo que contém o programa executável chama-se mycopy.
- Uma linha de comando típica para o programa mycopy no sistema Linux/UNIX é
  - $ mycopy input output
- Essa linha de comando indica que o arquivo input é copiado para o arquivo output.
- Quando o programa é executado, se argc não é 3 (mycopy é considerado um dos argumentos), o programa imprime uma mensagem de erro e termina.
- Caso contrário, o array argv contém as strings "mycopy", "input" e "output".

Uso de argumentos na linha de comando

- O segundo e o terceiro argumentos da linha de comando são utilizados como nomes de arquivos pelo programa.
- Os arquivos são abertos por meio da função fopen.
- Se os dois arquivos são abertos com sucesso, os caracteres são lidos do arquivo input e gravados no arquivo output até que o indicador de final do arquivo input seja encontrado.
- Então, o programa termina.
- O resultado é a cópia exata do arquivo input.
- Consulte os manuais de seu sistema para obter mais informações sobre argumentos da linha de comandos.

Uso de argumentos na linha de comando

- **No Visual C++, você pode especificar os argumentos da linha de comandos indo para Project Properties > Configuration Properties > Debugging e entrando com os argumentos na caixa de texto à direita de Command Arguments.**

Uso de argumentos na linha de comando

```
1  /* Fig. 14.3: fig14_03.c
2     Usando argumentos na linha de comandos */
3  #include <stdio.h>
4
5  int main( int argc, char *argv[] )
6  {
7     FILE *inFilePtr;  /* ponteiro do arquivo de entrada */
8     FILE *outFilePtr; /* ponteiro do arquivo de saída */
9     int c;   /* c mantém caracteres digitados pelo usuário */
10
11    /* verifica número de argumentos da linha de comandos */
12    if ( argc != 3 ) {
13       printf( "Uso: mycopy arquivo-entrada arquivo-saída\n" );
14    } /* fim do if */
15    else {
16       /* se arquivo de entrada puder ser aberto */
17       if ( ( inFilePtr = fopen( argv[ 1 ], "r" ) ) != NULL ) {
18          /* se arquivo de saída puder ser aberto */
19          if ( ( outFilePtr = fopen( argv[ 2 ], "w" ) ) != NULL ) {
20             /* lê e envia caracteres */
21             while ( ( c = fgetc( inFilePtr ) ) != EOF ) {
22                fputc( c, outFilePtr );
23             } /* fim do while */
24          } /* fim do if */
25          else { /* arquivo de saída não pode ser aberto */
26             printf( "Arquivo \"%s\" não pode ser aberto\n", argv[ 2 ] );
27          } /* fim do else */
28       } /* fim do if */
```

Figura 14.3 ■ Uso de argumentos na linha de comando. (Parte 1 de 2.)

Uso de argumentos na linha de comando

```
29          else { /* arquivo de entrada não pode ser aberto */
30             printf( "Arquivo \"%s\" não pode ser aberto\n", argv[ 1 ] );
31          } /* fim do else */
32       } /* fim do else */
33
34       return 0; /* indica conclusão bem-sucedida */
35    } /* fim do main */
```

Figura 14.3 ■ Uso de argumentos na linha de comando. (Parte 2 de 2.)

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- É possível criar programas que consistam em múltiplos arquivos-fonte.
- Existem várias considerações a serem feitas quando se cria programas a partir de múltiplos arquivos.
- Por exemplo, a definição de uma função deve estar contida em um único arquivo — não pode estar espalhada em dois ou mais arquivos.
- No Capítulo 5, introduzimos os conceitos de classes de memória e escopo.
- Aprendemos que variáveis declaradas fora das definições de função são da classe de memória **static** , por default, e são conhecidas como variáveis globais.
- Variáveis globais são acessíveis a qualquer função definida no mesmo arquivo após a declaração da variável.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- Variáveis globais também são acessíveis a funções que estejam em outros arquivos, porém, devem ser declaradas em todos os arquivos em que são utilizadas.
- Por exemplo, se definirmos uma variável global inteira flag em um arquivo e nos referirmos a ela em um segundo arquivo, o segundo arquivo deverá conter a declaração
  - `extern int flag;`
- antes de as variáveis serem usadas nesse arquivo.
- Essa declaração utiliza o especificador de classe de memória `extern` para indicar ao compilador que a variável flag será definida adiante no mesmo arquivo ou em outro arquivo.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- O compilador informa ao editor de ligação (*linker*) que uma referência não resolvida para a variável flag aparece no arquivo (o compilador não sabe onde flag foi definida, então deixa o linker tentar encontrar flag).
- Se o linker não localizar a definição de flag, um erro de edição de ligação é reportado e o arquivo executável não é gerado.
- Se o linker encontrar uma definição global apropriada, ele resolverá a referência indicando onde flag foi localizado.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

Observação sobre engenharia de software 14.1
Variáveis globais devem ser evitadas, a menos que o desempenho da aplicação seja importante, porque elas violam o princípio do menor privilégio e tornam difícil a manutenção do software.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- Da mesma forma que as declarações extern podem ser usadas para declarar variáveis globais em outros arquivos de programa, protótipos de função podem estender o escopo de uma função além do arquivo em que ela foi definida (o especificador extern não é requerido no protótipo de função).
- Simplesmente inclua o protótipo da função em cada arquivo em que a função for chamada, e compile os arquivos todos juntos (ver Seção 13.2).
- Protótipos de função indicam ao compilador que a função especificada foi definida mais à frente, no mesmo arquivo, ou em outro arquivo

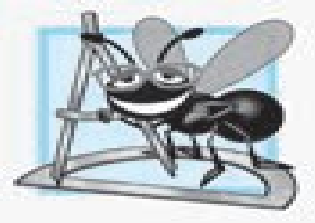
Notas sobre a compilação de programas de múltiplos arquivos-fonte

- Novamente, o compilador não tenta resolver referências a tal função — esta é uma tarefa para o linker.
- Se o linker não localizar uma definição apropriada de função, uma mensagem de erro é gerada.
- Como exemplo do uso de protótipos de função para estender o escopo de uma função, considere qualquer programa que contenha a diretiva do pré-processador #include <stdio.h>, que inclui em um arquivo protótipos de funções, tais como printf e scanf.
- Outras funções no arquivo podem usar printf e scanf para completar as suas tarefas.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- As funções printf e scanf são definidas em outros arquivos.
- Não precisamos saber onde elas foram definidas.
- Simplesmente reutilizamos o código em nossos programas.
- O *linker* resolve automaticamente nossas referências para essas funções.
- Esse processo permite usar as funções da biblioteca-padrão

Notas sobre a compilação de programas de múltiplos arquivos-fonte



Observação sobre engenharia de software 14.2
Criar programas em múltiplos arquivos-fonte facilita a reutilização e a boa engenharia de software. Funções podem ser comuns a muitas aplicações. Em tais casos, essas funções devem ser armazenadas em seu arquivo-fonte, e cada arquivo-fonte deve ter um arquivo de cabeçalho correspondente, contendo os protótipos das funções. Isso permite que os programadores de diferentes aplicações reutilizem o mesmo código, incluindo o arquivo de cabeçalho apropriado e compilando sua aplicação com o arquivo-fonte correspondente.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- É possível restringir o escopo de uma variável global ou função ao arquivo em que elas são definidas.
- O especificador de classe de armazenamento static, quando aplicado a uma variável global ou a uma função, evita que ela seja usada por qualquer função que não tenha sido definida no mesmo arquivo.
- Isso é chamado de ligação interna.
- Variáveis globais e funções que não sejam precedidas por static em suas definições têm ligação externa — elas podem ser acessadas em outros arquivos se eles contiverem declarações e/ou protótipos de funções apropriados.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- A declaração da variável global
  - `static const double PI = 3.14159;`

  cria a variável PI do tipo double, inicializa-a em 3.14159 e indica que PI somente é conhecida por funções no arquivo no qual é definida.
- O uso do especificador static é comum no caso de funções utilitárias que são chamadas apenas por funções em um arquivo particular.
- Se uma função não é requerida fora de um arquivo particular, o princípio do menor privilégio deve ser reforçado com o uso de static.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- Se uma função é definida antes de ser usada em um arquivo, static deve ser aplicado à definição da função.
- Caso contrário, static deve ser aplicado ao protótipo da função.
- Quando construímos programas grandes, distribuídos em múltiplos arquivos-fonte, compilar o programa se tornará tedioso se pequenas alterações tiverem de ser feitas em um dos arquivos e for preciso recompilar todo o programa.
- Muitos sistemas oferecem utilitários que recompilam apenas o arquivo do programa modificado.

Notas sobre a compilação de programas de múltiplos arquivos-fonte

- Nos sistemas Linux/UNIX, o utilitário é chamado **make**.
- O utilitário make lê um arquivo chamado **makefile** , que contém instruções para compilar e ligar o programa. Produtos como Eclipse™ e Microsoft® Visual C++® oferecem utilitários semelhantes.
- Para obter mais informações sobre os utilitários make, veja o manual de sua ferramenta de desenvolvimento.

Término de programas com exit e atexit

- A biblioteca de utilitários gerais (<stdlib.h>) oferece métodos para finalizar a execução do programa por outros meios além de um retorno convencional da função main.
- A função exit força um programa a terminar como se fosse executado normalmente.
- A função é frequentemente usada para finalizar um programa quando é detectado um erro de entrada, ou se um arquivo a ser processado pelo programa não pode ser aberto.
- A função atexit registra uma função que deve ser chamada no término bem-sucedido do programa, ou seja, quando o programa termina alcançando o final main ou quando a função exit é chamada.

Término de programas com exit e atexit

- A função atexit usa como argumento um ponteiro para uma função (ou seja, o nome da função).
- As funções chamadas no término do programa não podem ter argumentos e não podem retornar um valor.
- Até 32 funções podem ser registradas para execução no término do programa.
- A função exit usa um argumento que, normalmente, é a constante simbólica **EXIT_SUCCESS** ou a constante simbólica **EXIT_FAILURE**.
- Se exit é chamado com EXIT_SUCCESS, o valor definido na implementação para término bem-sucedido é retornado ao ambiente para o qual o programa foi chamado.

Término de programas com exit e atexit

```c
/* Fig. 14.4: fig14_04.c
   Usando as funções exit e atexit */
#include <stdio.h>
#include <stdlib.h>

void print( void ); /* protótipo */

int main( void )
{
   int resposta; /* escolha de menu do usuário */

   atexit( print ); /* registra função print */
   printf( "Digite 1 para terminar programa com função exit"
      "\nDigite 2 para terminar programa normalmente\n" );
   scanf( "%d", &resposta );

   /* chama exit se resposta é 1 */
   if ( answer == 1 ) {
      printf( "\nTerminando programa com função exit\n" );
      exit( EXIT_SUCCESS );
   } /* fim do if */

   printf( "\nTerminando programa alcançando o fim do main\n" );
   return 0; /* indica conclusão bem-sucedida */
} /* fim do main */

/* exibe mensagem antes do término */
void print( void )
{
   printf( "Executando a função print no término "
      "do programa \nPrograma encerrado\n" );
} /* fim da função print */
```

Término de programas com exit e atexit

```
Digite 1 para terminar o programa com a função exit
Digite 2 para terminar o programa normalmente
1

Terminando programa com função exit
Executando a função print no término do programa
Programa encerrado
```

```
Digite 1 para terminar o programa com a função exit
Digite 2 para terminar o programa normalmente
2

Terminando programa alcançando o final de main
Executando a função print no término do programa
Programa encerrado
```

Figura 14.4 ■ Funções exit e atexit.

Término de programas com exit e atexit

- Se exit é chamado com EXIT_FAILURE, o valor definido na implementação para término malsucedido é retornado.
- Quando a função exit é chamada, todas as funções previamente registradas por atexit são chamadas na ordem inversa à de seu registro, todos os streams associados ao programa são liberados e fechados, e o controle retorna ao ambiente hospedeiro.
- A Figura 14.4 testa as funções exit e atexit.
- O programa sugere que o usuário determine de que forma deve ser terminado, com exit ou alcançando o fim de main.
- Note que a função print é executada ao término do programa em cada caso.

O qualificador de tipo volatile

- A linguagem em C também oferece o qualificador de tipo volatile para suprimir diversos tipos de otimizações.
- O padrão C indica que, quando volatile é usado para qualificar um tipo, a natureza do acesso a um objeto desse tipo depende da implementação.
- Isso normalmente significa que a variável pode ser mudada por outro programa ou pelo hardware do computador.

Sufixos para constantes inteiras e de ponto flutuante

- C oferece sufixos de inteiros e sufixos de ponto flutuante para especificar os tipos de constantes inteiras e de ponto flutuante.
- Os sufixos de inteiros são: u ou U para um inteiro unsigned integer, l ou L para um inteiro long integer, e ul, lu, UL ou LU para um inteiro unsigned long.
- As seguintes constantes são do tipo unsigned, long e unsigned long, respectivamente:
  - 174u
    8358L
    28373ul

Sufixos para constantes inteiras e de ponto flutuante

- Se uma constante inteira não tiver sufixo, seu tipo será determinado pelo primeiro tipo capaz de armazenar um valor daquele tamanho (primeiro int, depois long int e, então, unsigned long int).
- Os sufixos de ponto flutuante são: f ou F para um float, e l ou L para um long double.
- As constantes a seguir são do tipo float e long double, respectivamente:
  - 1.28f
    3.14159L
- Uma constante de ponto flutuante sem sufixo é automaticamente do tipo double.

Mais sobre arquivos

Modo	Descrição
rb	Abre um arquivo binário existente para leitura.
wb	Cria um arquivo binário para gravação. Se o arquivo já existir, descarta o conteúdo atual.
ab	Acréscimo; abre ou cria um arquivo binário para gravação no final do arquivo.
rb+	Abre um arquivo binário existente para atualização (leitura e gravação).
wb+	Cria um arquivo binário para atualização. Se o arquivo já existir, descarta o conteúdo atual.
ab+	Acréscimo; abre ou cria um arquivo binário para atualização; toda gravação é feita no final do arquivo.

Figura 14.5 ■ Modos de abertura de arquivo binário.

Mais sobre arquivos

- Foram apresentadas capacidades para o processamento de arquivos de texto com acesso sequencial e acesso aleatório.
- C também oferece capacidade para o processamento de arquivos binários, mas alguns sistemas de computador não aceitam esses arquivos.
- Se os arquivos binários não forem aceitos, e um arquivo for aberto em um modo de arquivo binário (Figura 14.5), o arquivo será processado como um arquivo de texto.
- Os arquivos binários devem ser usados no lugar dos arquivos de texto somente em situações em que as condições rígidas de velocidade, o armazenamento e/ou a compatibilidade exigem arquivos binários.
- Caso contrário, os arquivos de texto têm sempre preferência por sua inerente portabilidade e pela capacidade de usar outras ferramentas-padrão para examinar e manipular os dados do arquivo.

Mais sobre arquivos

Dica de desempenho 14.1
Use arquivos binários no lugar de arquivos de texto em aplicações que exijam alto desempenho.

Dica de portabilidade 14.1
Use arquivos de texto ao escrever programas portáveis.

Mais sobre arquivos

- A biblioteca-padrão também oferece a função tmpfile, que abre um arquivo temporário no modo"wb+".
- Embora esse seja um modo de arquivo binário, alguns sistemas processam arquivos temporários como arquivos de texto.
- Um arquivo temporário existe até que seja fechado com fclose, ou até que o programa termine.
- A Microsoft eliminou essa função por 'motivos de segurança'.
- A Figura 14.6 troca as tabulações de um arquivo para espaços. O programa pede ao usuário que informe o nome do arquivo a ser modificado.

Mais sobre arquivos

- Se o arquivo informado pelo usuário e o arquivo temporário forem abertos com sucesso, o programa lerá os caracteres do arquivo a ser modificado e os gravará no arquivo temporário.
- Se o caractere lido for uma tabulação ('\t'), ele será substituído por um espaço e gravado no arquivo temporário.
- Quando o final do arquivo que está sendo modificado for alcançado, os ponteiros de arquivo para cada tipo serão reposicionados no início de cada arquivo com rewind.
- Em seguida, o arquivo temporário será copiado para o arquivo original, um caractere de cada vez.

Mais sobre arquivos

- **O programa imprimirá o arquivo original enquanto copia os caracteres para o arquivo temporário, e imprime o novo arquivo enquanto copia caracteres do arquivo temporário para o arquivo original, para confirmar os caracteres que estão sendo gravados.**

Mais sobre arquivos

```
1  /* Fig. 14.6: fig14_06.c
2     Usando arquivos temporários */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     FILE *filePtr; /* ponteiro para arquivo sendo modificado */
8     FILE *tempFilePtr; /* ponteiro de arquivo temporário */
9     int c; /* c mantém caracteres lidos de um arquivo */
10    char fileName[ 30 ]; /* cria array char */
11
12    printf( "Esse programa transforma tabulações em espaços.\n"
13       "Informe o arquivo a ser modificado: " );
14    scanf( "%29s", fileName );
15
16    /* fopen abre o arquivo */
17    if ( ( filePtr = fopen( fileName, "r+" ) ) != NULL ) {
18       /* cria arquivo temporário */
19       if ( ( tempFilePtr = tmpfile() ) != NULL ) {
20          printf( "\nO arquivo antes da modificação é:\n" );
21
22          /* lê caracteres do arquivo e coloca no arquivo temporário */
23          while ( ( c = getc( filePtr ) ) != EOF ) {
24             putchar( c );
25             putc( c == '\t' ? ' ' : c, tempFilePtr );
26          } /* fim do while */
27
28          rewind( tempFilePtr );
29          rewind( filePtr );
30          printf( "\n\nO arquivo após a modificação é:\n" );
31
```

Mais sobre arquivos

```
32          /* lê arquivo temporário e grava no arquivo original */
33          while ( ( c = getc( tempFilePtr ) ) != EOF ) {
34             putchar( c );
35             putc( c, filePtr );
36          } /* fim do while */
37       } /* fim do if */
38       else { /* se o arquivo temporário não pode ser aberto */
39          printf( "Impossível abrir arquivo temporário\n" );
40       } /* fim do else */
41    } /* fim do if */
42    else { /* se o arquivo não pode ser aberto */
43       printf( "Impossível abrir %s\n", fileName );
```

Figura 14.6 ■ Arquivos temporários. (Parte 1 de 2.)

Mais sobre arquivos

```
44      } /* fim do else */
45
46      return 0; /* indica conclusão bem-sucedida */
47   } /* fim do main */
```

```
Esse programa muda tabulações para espaços.
Informe o arquivo a ser modificado: data.txt

O arquivo antes da modificação é:
0    1    2    3    4
     5    6    7    8    9

O arquivo após a modificação é:
0 1 2 3 4
 5 6 7 8 9
```

Figura 14.6 ■ Arquivos temporários. (Parte 2 de 2.)

Tratamento de sinais

- Um evento assíncrono externo, ou sinal, pode terminar um programa prematuramente.
- Alguns eventos incluem interrupções (pressionar <Ctrl> c em um sistema Linux/UNIX ou Windows), instruções ilegais, violações de segmentação, ordens de término vindas do sistema operacional e exceções de ponto flutuante (divisão por zero ou multiplicação de ponto flutuante com valores muito grandes).

Tratamento de sinais

Sinal	Explicação
SIGABRT	Término anormal de um programa (como uma chamada a `abort`).
SIGFPE	Uma operação aritmética errada, como divisão por zero ou uma operação resultando em overflow.
SIGILL	Detecção de uma instrução ilegal.
SIGINT	Recebimento de um sinal de atenção interativo.
SIGSEGV	Acesso inválido à memória.
SIGTERM	Requisição de término enviada a um programa.

Figura 14.7 ■ Sinais-padrão de `signal.h`.

Tratamento de sinais

- A biblioteca de tratamento de sinais (<signal.h>) oferece a capacidade de interceptar eventos inesperados com a função signal.
- A função signal recebe dois argumentos — um número de sinal inteiro e um ponteiro de acesso à função de tratamento de sinal.
- Sinais podem ser gerados pela função raise, que recebe um número de sinal inteiro como argumento.
- A Figura 14.7 resume os sinais-padrão definidos no arquivo de cabeçalho <signal.h>.

Tratamento de sinais

- A Figura 14.8 usa a função signal para interceptar um sinal interativo (SIGINT).
- A linha 15 chama signal com SIGINT, e um ponteiro para a função signalHandler (lembre-se de que o nome de uma função é um ponteiro para o início da função).
- Quando um sinal do tipo SIGINT aparece, o controle passa para a função signalHandler, que imprime uma mensagem e dá ao usuário a opção de continuar a executar o programa normalmente.
- Se o usuário quiser continuar a execução, o signal handler é reinicializado chamando signal novamente, e o controle retorna ao ponto do programa em que o sinal foi detectado.

Tratamento de sinais

- **Nesse programa, a função raise (linha 24) é usada para simular um sinal interativo.**
- **Um número aleatório entre 1 e 50 é escolhido.**
- **Se o número escolhido for 25, raise será chamado para gerar o sinal.**
- **Normalmente, os sinais interativos são iniciados fora do programa.**
- **Por exemplo, digitar <*Ctrl*> *c* durante a execução do programa em um sistema Linux/UNIX ou Windows gera um sinal interativo que termina a execução do programa.**
- **O tratamento de sinal pode ser usado para interceptar o sinal interativo e impedir que o programa seja terminado.**

Tratamento de sinais

```
1   /* Fig. 14.8: fig14_08.c
2      Usando o tratamento de sinal */
```

Figura 14.8 ■ Tratamento de sinais. (Parte I de 3.)

Tratamento de sinais

```
 3  #include <stdio.h>
 4  #include <signal.h>
 5  #include <stdlib.h>
 6  #include <time.h>
 7
 8  void signalHandler( int signalValue ); /* protótipo */
 9
10  int main( void )
11  {
12     int i; /* contador usado para percorrer loop 100 vezes */
13     int x; /* variável para manter valores aleatórios entre 1-50 */
14
15     signal( SIGINT, signalHandler ); /* tratador de sinal do registrador */
16     srand( time( NULL ) );
17
18     /* envia números de 1 a 100 */
19     for ( i = 1; i <= 100; i++ ) {
20        x = 1 + rand() % 50; /* gera número aleatório para elevar SIGINT */
21
22        /* eleva SIGINT quando x é 25 */
23        if ( x == 25 ) {
24           raise( SIGINT );
25        } /* fim do if */
26
27        printf( "%4d", i );
28
29        /* envia \n quando i é um múltiplo de  10 */
30        if ( i % 10 == 0 ) {
31           printf( "\n" );
32        } /* fim do if */
33     } /* fim do for */
34
```

Tratamento de sinais

```
35       return 0; /* indica conclusão bem-sucedida */
36    } /* fim do main */
37
38    /* trata o sinal */
39    void signalHandler( int signalValue )
40    {
41       int response; /* resposta do usuário ao sinal (1 ou 2) */
42
43       printf( "%s%d%s\n%s",
44          "\nSinal de interrupção ( ", signalValue, " ) recebido.",
45          "Deseja continuar ( 1 = sim ou 2 = não )? " );
46
47       scanf( "%d", &response );
48
49       /* verifica respostas inválidas */
50       while ( response != 1 && response != 2 ) {
51          printf( "( 1 = yes or 2 = no )? " );
52          scanf( "%d", &response );
53       } /* fim do while */
54
55       /* determina se é hora de sair */
56       if ( response == 1 ) {
57          /* registra novamente tratador de sinal para próximo SIGINT */
58          signal( SIGINT, signalHandler );
59       } /* fim do if */
60       else {
61          exit( EXIT_SUCCESS );
```

Figura 14.8 ■ Tratamento de sinais. (Parte 2 de 3.)

Tratamento de sinais

```
62      } /* fim do else */
63   } /* fim da função signalHandler */
```

```
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50
    51    52    53    54    55    56    57    58    59    60
    61    62    63    64    65    66    67    68    69    70
    71    72    73    74    75    76    77    78    79    80
    81    82    83    84    85    86    87    88    89    90
    91    92    93
Sinal de interrupção ( 2 ) recebido.
Deseja continuar ( 1 = sim ou 2 = não )? 1
    94    95    96
Sinal de interrupção ( 2 ) recebido.
Deseja continuar ( 1 = sim ou 2 = não )?  2
```

Figura 14.8 ■ Tratamento de sinais. (Parte 3 de 3.)

Alocação dinâmica de memória: funções calloc e realloc

- Discutimos o estilo de alocação dinâmica de memória usando a função malloc.
- Como dissemos os arrays são melhores que as listas vinculadas de classificação rápida, pesquisa e acesso a dados.
- Porém, os arrays normalmente são estruturas estáticas de dados.
- A biblioteca de utilitários genéricos (stdlib.h) oferece duas outras funções para alocação dinâmica de memória — calloc e realloc.
- Essas funções podem ser utilizadas para criar e modificar arrays dinâmicos.

Alocação dinâmica de memória: funções calloc e realloc

- O ponteiro para um array pode ser subscritado como um array.
- Assim, um ponteiro para uma porção contígua de memória, criada por calloc , pode ser manipulado como um array.
- A função calloc aloca memória dinamicamente para um array.
- O protótipo de calloc é
  - `void *calloc( size_t nmemb, size_t size );`
- Seus dois argumentos representam o número de elementos (nmemb) e o tamanho de cada elemento (size).
- A função calloc também inicializa os elementos do array em zero.

Alocação dinâmica de memória: funções calloc e realloc

- A função retorna um ponteiro para a memória alocada, ou um ponteiro NULL se a memória não tiver sido alocada.
- A principal diferença entre malloc e calloc é qie calloc apaga a memória que aloca, e malloc não.
- A função realloc altera o tamanho de um objeto alocado anteriormente por uma chamada a malloc, calloc ou realloc.
- O conteúdo original do objeto não é modificado, desde que a memória alocada seja maior que a quantidade alocada anteriormente.

Alocação dinâmica de memória: funções calloc e realloc

- Caso contrário, o conteúdo se mantém inalterado até alcançar o tamanho do novo objeto.
- O protótipo para realloc é
  - `void *realloc( void *ptr, size_t size );`
- A função realloc recebe dois argumentos — um ponteiro para o objeto original (ptr) e o novo tamanho do objeto (size).
- Se ptr for NULL, realloc funcionará da mesma forma que malloc.
- Se size for 0 e ptr for diferente de NULL, a memória para o objeto será liberada.

Alocação dinâmica de memória: funções calloc e realloc

- **Por outro lado, se ptr for diferente de NULL e size for maior que zero, realloc tentará alocar um novo bloco de memória para o objeto.**
- **Se o novo espaço não puder ser alocado, o objeto apontado por ptr não será alterado.**
- **A função realloc retornará ou um ponteiro para a memória realocada ou um ponteiro NULL para indicar que a memória não foi realocada.**

Desvio incondicional com goto

- Ao longo deste livro, temos salientado a importância do uso das técnicas de programação estruturada para construir um software consistente de fácil depuração, manutenção e modificação.
- Em alguns casos, o desempenho é mais importante que uma restrita observância às técnicas de programação estruturada.
- Nesses casos, podem ser utilizadas algumas técnicas de programação não estruturada.
- Por exemplo, podemos usar break para terminar a execução de uma estrutura de repetição antes que a condição do loop de continuação se torne falsa.
- Isso evita repetições desnecessárias do loop se a tarefa for completada antes do término do loopn.

Desvio incondicional com goto

```
1  /* Fig. 14.9: fig14_09.c
2     Usando goto */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int contador = 1; /* inicializa contador */
8
9     start: /* rótulo */
10
11       if ( contador > 10 ) {
12          goto end;
13       } /* fim do if */
14
15       printf( "%d  ", contador );
16       count++;
17
18       goto start; /* vai para início da linha 9 */
19
20    end: /* rótulo */
21       putchar( '\n' );
22
23    return 0; /* indica conclusão bem-sucedida */
24 } /* fim do main */
```

```
   1  2  3  4  5  6  7  8  9  10
```

Figura 14.9 ■ Comando goto.

Desvio incondicional com goto

- Outro exemplo da programação não estruturada é o comando goto — um desvio incondicional.
- O resultado do comando goto é o desvio do fluxo de controle do programa para o primeiro comando após o rótulo especificado no comando goto.
- Um rótulo é um identificador seguido por dois-pontos (:).
- Um rótulo deve aparecer na mesma função que o comando goto referente a esse rótulo.
- A Figura 14.9 usa comandos goto para repetir dez vezes um loop, e imprime o valor do contador a cada iteração.

Desvio incondicional com goto

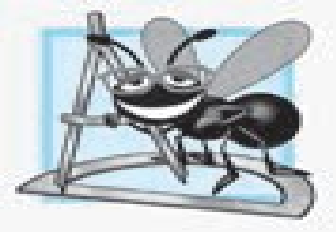
- Depois de inicializar count em 1, a linha 11 testa se count é maior que 10 (o rótulo start é saltado porque rótulos não executam nenhuma ação).
- Se for, o controle será transferido por goto para o primeiro comando depois do rótulo endereço (que aparece na linha 20).
- Caso contrário count será impresso e incrementado nas linhas 15-16, e o controle será transferido de goto (linha 18) para o primeiro comando depois do rótulo start (que aparece na linha 9).

Desvio incondicional com goto

- **Estabelecemos que eram necessárias apenas três estruturas de controle para escrever qualquer programa — sequência, seleção e repetição.**
- **Quando as regras da programação estruturada são seguidas, é possível criar estruturas de controle profundamente aninhadas, das quais é difícil sair de uma maneira eficiente.**
- **Alguns programadores usam o comando goto em tais situações, como uma saída rápida de uma estrutura profundamente aninhada.**
- **Isso elimina a necessidade de testar diversas condições para sair de uma estrutura de controle..**

Desvio incondicional com goto

Dica de desempenho 14.2
O comando goto pode ser usado para sair de uma estrutura de controle profundamente aninhada de maneira eficiente.



Observação sobre engenharia de software 14.3
O comando goto deve ser utilizado apenas em aplicações orientadas a desempenho. O comando goto é não estruturado e pode levar a programas mais difíceis de serem depurados, mantidos e modificados.

Lembre-se

“ Zero linhas de código tem zero defeitos. ”