Técnicas de Programação

TP1201

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 criar, ler, gravar e atualizar arquivos.
- Processar arquivos por acesso sequencial.
- Processar arquivos por acesso aleatório.

Objetivos

- A usar #include para desenvolver programas de grande porte.
- A usar #define para criar macros, e macros com argumentos.
- A usar a compilação condicional para especificar partes de um programa que nem sempre deverão ser compiladas (assim como o código que auxilia na depuração).
- A exibir mensagens de erro durante a compilação condicional.
- A usar asserções para testar se os valores das expressões estão corretos.

Introdução

- O pré-processador em C é executado antes de um programa ser compilado.
- Algumas das ações possíveis são a inclusão de outros arquivos no arquivo que está sendo compilado, a definição de constantes simbólicas e de macros, a compilação condicional do código do programa e a execução condicional das diretivas do pré-processador.
- Todas as diretivas do pré-processador começam com # e somente caracteres de espaço em branco e comentários podem aparecer antes de uma diretiva de pré-processador em uma linha.

A diretiva #include do pré-processador

- A diretiva #include do pré-processador tem sido usada ao longo de todo o livro.
- Ela faz com que uma cópia de um arquivo juntar a linha abaixo especificado seja incluída no lugar da diretiva. As duas formas da diretiva #include são:
  - #include <nome-arquivo>
    #include "nome-arquivo"
- A diferença entre as duas formas é o local em que o pré-processador procura o arquivo a ser incluído.
- Se o nome do arquivo está entre aspas, o pré-processador pesquisa, em primeiro lugar, o diretório em que está o arquivo que está sendo compilado (ele também pode procurar em outros locais).

A diretiva #include do pré-processador

- **Normalmente, esse método é usado para incluir arquivos de cabeçalho definidos pelo programador.**
- **Se o nome do arquivo está entre os sinais de menor e maior (< e >) — usados para arquivos de cabeçalho da biblioteca-padrão —, o pré-processador procura o arquivo especificado de uma maneira dependente da implementação, normalmente em diretórios pré-designados pelo compilador e pelo sistema.**

A diretiva #include do pré-processador

- A diretiva #include é usada para incluir arquivos de cabeçalho da biblioteca-padrão como stdio.h e stdlib.h (ver Figura 5.6), e com programas que consistem em diversos arquivos-fonte que devem ser compilados juntos.
- Um cabeçalho que contenha declarações comuns aos arquivos de programas separados frequentemente é criado e incluído no arquivo.
- Exemplos de tais declarações são declarações de estrutura e união, de enumerações e de protótipos de funções.

A diretiva #define do pré-processador: constantes simbólicas

- A diretiva #define cria constantes simbólicas — constantes representadas por símbolos — e macros — operações definidas por símbolos.
- O formato da diretiva #define é
  - #define *identificador texto-substituto*
- Quando essa linha aparece em um arquivo, todas as ocorrências subsequentes de identificador (exceto aquelas dentro de uma string literal) serão substituídas automaticamente pelo texto substituto antes de o programa ser compilado.

A diretiva #define do pré-processador: constantes simbólicas

- **Por exemplo,**
  - **#define PI 3.14159**

  substitui todas as ocorrências subsequentes da constante simbólica PI pela constante numérica 3.14159.
- **Constantes simbólicas permitem ao programador criar um nome para uma constante e usar esse nome ao longo de todo o programa.**
- **Se a constante precisar ser modificada em todo o programa, ela pode ser modificada uma vez na diretiva #define.**
- **Quando o programa for recompilado, todas as ocorrências da constante no programa serão modificadas.**

A diretiva #define do pré-processador: constantes simbólicas

- Tudo o que está à direita do nome da constante simbólica substitui a constante simbólica. Por exemplo, #define PI = 3.14159 faz com que o compilador substitua cada ocorrência de PI por = 3.14159.
- Isso é a causa de muitos erros sutis de lógica e de sintaxe.
- Redefinir uma constante simbólica com um novo valor também é um erro.

A diretiva #define do pré-processador: constantes simbólicas

Boa prática de programação 13.1
Usar nomes significativos para constantes simbólicas ajuda a tornar os programas mais autodocumentados.

Boa prática de programação 13.2
Por convenção, as constantes simbólicas são definidas por apenas letras maiúsculas e caracteres de sublinhado.

A diretiva #define do pré-processador: macros

- Uma macro é um identificador definido em uma diretiva #define para o pré-processador.
- Da mesma maneira que ocorre com as constantes simbólicas, o identificador de macro é trocado pelo texto substituto antes de o programa ser compilado.
- Macros podem ser definidas com ou sem argumentos.
- A macro sem argumentos é processada como uma constante simbólica.
- Em uma macro com argumentos, os argumentos são trocados no texto substituto e, então, a macro é expandida — ou seja, o texto substituto entra no lugar do identificador da macro e da lista de argumentos no programa.

A diretiva #define do pré-processador: macros

- [Nota: uma constante simbólica é um tipo de macro.]
- Considere a seguinte definição de uma macro com um argumento para o cálculo da área de um círculo:
  - `#define AREA_CIRCULO( x ) ( ( PI ) * ( x ) * ( x ) )`
- Onde quer que AREA_CIRCULO(y) apareça no arquivo, o valor de y é trocado por x no texto de substituição, a constante simbólica PI é substituída por seu valor (definido anteriormente) e a macro é expandida no programa.

A diretiva #define do pré-processador: macros

- Por exemplo, a instrução
  - area = AREA_CIRCULO(4);

  é expandida para
  - area = ((3.14159) * (4) * (4));

  e o valor da expressão é calculado e atribuído à variável area.
- Os parênteses em torno de cada x no texto substituto forçam a ordem correta de cálculo quando o argumento da macro é uma expressão.

A diretiva #define do pré-processador: macros

- Por exemplo, a instrução
  - area = AREA_CIRCULO(c + 2);

  é expandida para
  - area = ((3.14159) * (c + 2) * (c + 2));

  a qual é avaliada corretamente, porque os parênteses forçam a ordem certa de cálculo.

A diretiva #define do pré-processador: macros

- Se os parênteses são omitidos, a expansão da macro é
  - area = 3.14159 * c + 2 * c + 2;

que é calculada, incorretamente, como

  - area = (3.14159 * c) + (2 * c) + 2;

por causa das regras de precedência de operadores.

A diretiva #define do pré-processador: macros

Erro comum de programação 13.1

Esquecer de colocar os argumentos de uma macro entre parênteses no texto substituto pode provocar erros lógicos.

A diretiva #define do pré-processador: macros

- A macro AREA_CIRCULO poderia ser definida como uma função.
- A função areaCirculo
  - ```
    double areaCirculo( double x )
    {
      return 3.14159 * x * x;
    }
    ```

  executa o mesmo cálculo que AREA_CIRCULO, mas existe o overhead ou sobrecarga de uma chamada à função areaCirculo.
    ```

A diretiva #define do pré-processador: macros

- As vantagens do uso de AREA_CIRCULO estão no fato de que macros inserem o código diretamente no programa — evitando o overhead das chamadas de funções —, e o programa permanece legível porque AREA_CIRCULO é definida separadamente e recebe um nome significativo.
- A desvantagem é que o argumento é avaliado duas vezes.

A diretiva #define do pré-processador: macros

Dica de desempenho 13.1

Às vezes, as macros podem ser usadas para substituir uma chamada de função pelo código inline para eliminar o
overhead de uma chamada de função. Os compiladores otimizadores de hoje colocam funções inline para você com
frequência, de modo que muitos programadores não precisam mais usar macros para essa finalidade. A C99 também oferece a palavra-chave inline (ver Apêndice G).

A diretiva #define do pré-processador: macros

- A definição seguinte é uma definição de macro com dois argumentos para o cálculo da área de um retângulo:
  - `#define AREA_RETANGULO( x, y )  ( ( x ) * ( y ) )`
- Onde quer que AREA_RETANGULO(x, y) apareça no programa, os valores de x e y são substituídos no texto substituto da macro, e a macro é expandida no lugar do nome da macro.
- Por exemplo, o comando
  - `areaRetang = AREA_RETANGULO( a + 4, b + 7 );`
- é expandido para
  - `areaRetang = ( ( a + 4 ) * ( b + 7 ) );`

A diretiva #define do pré-processador: macros

- O valor da expressão é avaliado e atribuído à variável areaRetang.
- O texto de substituição de uma macro ou de uma constante simbólica normalmente é qualquer texto que apareça na linha após o identificador na diretiva #define.
- Se o texto substituto para uma macro ou para uma constante simbólica é mais longo que o restante
- da linha, uma barra invertida (\) deve ser colocada no fim da linha para indicar que o texto de substituição continua na próxima linha.

A diretiva #define do pré-processador: macros

- Constantes simbólicas e macros podem ser descartadas com o uso da diretiva para o pré-processador #undef.
- A diretiva #undef 'anula a definição' do nome de uma constante simbólica ou de uma macro.
- O escopo de uma constante simbólica ou de uma macro vai de sua definição até o ponto em que sua definição é anulada com #undef, ou até o final do arquivo.
- Uma vez anulado, um nome pode ser redefinido com #define.
- Algumas vezes, as funções da biblioteca-padrão são definidas como macros baseadas em outras funções de biblioteca.

A diretiva #define do pré-processador: macros

- Uma macro comumente definida no arquivo de cabeçalho stdio.h é
  - #define getchar() getc(stdin)
- A definição de macro de getchar usa a função getc para obter um caractere do stream padrão de entrada.
- A função putchar do arquivo de cabeçalho stdio.h e as funções de manipulação de caracteres do arquivo de cabeçalho ctype.h também são, frequentemente, implementadas como macros.
- Note que expressões com efeitos colaterais (ou seja, valores de variáveis podem mudar) não devem ser passadas para uma macro, porque os argumentos de uma macro podem ser avaliados mais de uma vez.

Compilação condicional

- A **compilação condicional** permite que você controle a execução das diretivas do pré-processador e a compilação do código do programa.
- Cada uma das diretivas condicionais do pré-processador avalia uma expressão constante inteira.
- Expressões de coerção, expressões com sizeof e constantes de enumerações não podem ser avaliadas em diretivas do pré-processador.
- A instrução condicional do pré-processador é bastante semelhante à estrutura de seleção if.

Compilação condicional

- Considere o seguinte código de pré-processador::

```
#if !defined(MINHA_CONSTANTE)
  #define MINHA_CONSTANTE 0
#endif
```

- Essas diretivas determinam se MINHA_CONSTANTE está definida.
- A expressão defined(MINHA_CONSTANTE) é calculada e produz o valor 1 se MY_CONSTANT estiver definida; caso contrário, o valor produzido será 0.
- Se o resultado for 0, !defined(MINHA_CONSTANTE) produz 1 e MINHA_CONSTANTE é definida.
- Caso contrário, a diretiva #define é omitida.

Compilação condicional

- Cada construção #if termina com #endif.
- As diretivas #ifdef e #ifndef são abreviações de #if defined(*nome)* e *#if ! defined(nome).*
- Uma construção de pré-processador com múltiplas partes pode ser testada com o uso das diretivas #elif (o equivalente de else if em uma estrutura if) e #else (o equivalente de else em uma estrutura if).
- Essas diretivas são usadas constantemente para impedir que arquivos de cabeçalho sejam incluídos várias vezes no mesmo arquivo-fonte.
- Usaremos bastante essa técnica na parte C++ deste livro.

Compilação condicional

- **Muitas vezes durante o desenvolvimento do programa será útil eliminar partes significativas do código, transformando-o em comentários, evitando, assim, que ele seja compilado.**
- **Se o código contém comentários, /* e */ não podem ser usados para realizar essa tarefa.**
- **Em vez disso, você pode usar a seguinte construção de pré-processador:**
  - ```
    #if 0
       código que não será compilado
    #endif
    ```
- **Para habilitar o código de compilação, simplesmente substitua o valor 0 na instrução precedente pelo valor 1.**
    ```

Compilação condicional

- Frequentemente, a compilação condicional é usada como um auxílio para a depuração do programa.
- Muitas implementações em C possuem depuradores, que oferecem recursos muito mais poderosos que a compilação condicional.
- Se um depurador não estiver disponível, instruções, printf frequentemente serão usadas para imprimir valores e confirmar o fluxo de controle.
- Essas instruções printf podem ser delimitadas em diretivas condicionais do pré-processador, de maneira que os comandos serão compilados somente até que termine o processo de depuração.

Compilação condicional

- **Por exemplo,**
  - ```
    #ifdef DEBUG
       printf( "Variável x = %d\n", x );
    #endif
    ```

faz com que o comando printf seja compilado no programa se a constante simbólica DEBUG tiver sido definida (#define DEBUG) antes da diretiva #ifdef DEBUG.
    ```

Compilação condicional

- **Quando a depuração estiver completa, a diretiva #define será removida do arquivo-fonte (ou se torna um comentário) e os comandos printf que foram inseridos para fins de depuração serão ignorados durante a compilação.**
- **Em programas maiores, pode ser desejável definir várias constantes simbólicas diferentes que controlarão a compilação condicional em seções separadas do arquivo-fonte.**

Compilação condicional

Erro comum de programação 13.2

Inserir comandos printf *compilados condicionalmente para fins de depuração em lugares em que C espera por um comando simples. Nesse caso, o comando compilado condicionalmente deverá ser incluído em um comando composto. Assim, quando o programa for compilado com comandos de depuração, o fluxo de controle do programa não será alterado.*

As diretivas #error e #pragma do pré-processador

- **A diretiva #error**
  - **#error *tokens***

  imprime uma mensagem dependente de implementação que inclui os *tokens* especificados na diretiva.
- **Os tokens (unidades léxicas) são sequências de caracteres separadas por espaços.**
- **Por exemplo,**
  - **#error 1 - Out of range error**

  contém 6 tokens.
- **Quando uma diretiva #error é processada em alguns sistemas, os tokens na diretiva são exibidos como uma mensagem de erro, o pré-processamento é interrompido e o programa não é compilado.**

As diretivas #error e #pragma do pré-processador

- **A diretiva #pragma**
  - **#pragma *tokens***

  provoca uma ação definida pela implementação.
- **Um pragma não reconhecido pela implementação é ignorado.**

Operadores # e

- Os operadores # e ## do pré-processador estão disponíveis em C padrão.
- O operador # faz com que o token de um texto substituto seja convertido em uma string entre aspas.
- Considere a seguinte definição de macro:
  - `#define HELLO(x) printf( «Olá, " #x "\n" );`
- Quando HELLO(John) aparece em um arquivo de programa, isso é expandido para
  - `printf( "Olá, " "John" "\n" );`
- A string "John" substitui #x no texto substituto.

Operadores # e

- Strings separadas por espaços em branco são concatenadas durante o pré-processamento, de forma que a instrução acima é equivalente a
  - printf("Olá, John\n");
- O operador # deve ser usado em uma macro com argumentos, porque o operando de # se refere a um argumento da macro.
- O operador ## concatena dois tokens.

Operadores # e

- Considere a seguinte definição de macro:
  - `#define TOKENCONCAT(x, y)  x ## y`
- Quando TOKENCONCAT aparece em um programa, seus argumentos são concatenados e usados para substituir a macro.
- Por exemplo, TOKENCONCAT(O, K) é substituído por OK no programa.
- O operador ## deve ter dois operandos.

Números de linhas

- A diretiva do pré-processador #line faz com que as linhas subsequentes do código-fonte sejam renumeradas e comecem com o valor inteiro constante especificado.
- A diretiva
  - #line 100

  começa a numerar as linhas a partir de 100 iniciando com a próxima linha do código-fonte.
- Um nome de arquivo pode ser incluído na diretiva #line.

Números de linhas

- A diretiva
  - #line 100 “arquivo1.c"

  indica que as linhas são numeradas a partir de 100 , começando com a próxima linha de código-fonte, e que o nome do arquivo com a finalidade de receber qualquer mensagem do compilador é "file1.c".
- A diretiva normalmente é usada para ajudar a tornar mais significativas as mensagens produzidas por erros de sintaxe e advertências do compilador.
- Os números de linhas não aparecem no arquivo-fonte.

Constantes simbólicas predefinidas

Constante simbólica	Explicação
__LINE__	O número de linha da linha atual no código-fonte (uma constante inteira).
__FILE__	O nome presumido do arquivo de código-fonte (uma string).
__DATE__	A data em que o arquivo-fonte foi compilado (uma string na forma "Mmm dd yyyy", como "Jan 19 2002").
__TIME__	A hora em que o arquivo-fonte foi compilado (uma string literal na forma "hh:mm:ss").
__STDC__	O valor 1 se o compilador aceita a C padrão.

Figura 13.1 ■ Algumas constantes simbólicas predefinidas.

Constantes simbólicas predefinidas

- A C padrão oferece constantes simbólicas predefinidas, várias delas mostradas na Figura 13.1.
- Os identificadores para cada uma das constantes simbólicas predefinidas começam e terminam com dois caracteres sublinhados.
- Esses identificadores e o identificador defined identifier (usado na Seção 13.5) não podem ser utilizados em diretivas #define ou #undef.

Asserções

- A macro assert — definida no arquivo de cabeçalho <assert.h> — testa o valor de uma expressão.
- Se o valor da expressão é falso (0), então assert imprime uma mensagem de erro e chama a função abort (da biblioteca de utilitários gerais, <stdlib.h>) para terminar a execução do programa.
- Esta é uma ferramenta útil de depuração para testar se uma variável tem um valor correto.
- Por exemplo, suponha que a variável x nunca deva ser maior que 10 em um programa.

Asserções

- Pode-se usar uma asserção para testar o valor de x e imprimir uma mensagem de erro se o valor de x estiver incorreto.
- O comando seria
  - assert(x <= 10);
- Se x for maior que 10 quando esse comando for encontrado em um programa, uma mensagem de erro contendo o número da linha e o nome do arquivo é impressa, e o programa termina.

Asserções

- Você poderá, então, concentrar-se nessa área do código para encontrar o erro.
- Se a constante simbólica NDEBUG estiver definida, as asserções subsequentes serão ignoradas.
- Assim, quando as asserções não são mais necessárias, a linha
  - `#define NDEBUG`

  é inserida no arquivo de programa em vez de excluir cada asserção manualmente.

Asserções

Observação sobre engenharia de software 13.1

Asserções não têm por finalidade substituir o tratamento de erros durante as condições normais em tempo de execução. Seu uso deve ser limitado a localizar erros lógicos.

Lembre-se

“ Experiência é o nome que todos dão aos seus erros. ” Oscar Wilde