CAPÍTULO 5

O PRÉ-PROCESSADOR DE C

5.1 Introdução

O **pré-processador de C** é um programa *conceitualmente* independente do compilador que prepara arquivos-fonte para compilação. Apesar de ser executado automaticamente pelo compilador, o pré-processador é considerado um programa distinto que funciona separadamente do compilador. Em particular, o pré-processador de C utiliza uma linguagem própria, cujas instruções, denominadas **diretivas**, começam sempre com o símbolo #.

Uma diretiva pode aparecer em qualquer ponto de um programa-fonte. Dois tipos de diretivas, **#define** e **#include**, já foram introduzidos em seções anteriores.

Diferentemente de instruções da linguagem C, uma diretiva termina quando se introduz uma nova linha e não com ponto-e-vírgula. Quando deseja-se que uma diretiva ocupe mais de uma linha, utiliza-se uma barra invertida para indicar este fato. Por exemplo:

```
#define UMA_CONSTANTE_SIMBOLICA_LONGA "Este e' um exemplo \
de diretiva que ocupa duas linhas"
```

Compiladores antigos requerem que qualquer diretiva comece sempre com o símbolo # na primeira coluna e que não exista espaço entre este símbolo e a palavra que identifica a instrução.Apesar de estas

restrições não mais existirem em compiladores mais modernos, esta forma de escrever diretivas ainda é comum.

O pré-processador de C provê as facilidades enumeradas a seguir, que serão discutidas neste capítulo:

- Processamento de macros
- Inclusão de arquivos
- Compilação condicional
- Processamento de diretivas **#error**
- Processamento de diretivas **#pragma**
- Processamento de diretivas **#line**
- Resolução de expressões contendo os operadores #, ## e **_Pragma**

5.2 Unidades de Tradução

Levando em consideração a tarefa executada pelo pré-processador, o processo completo de compilação de um único arquivo-fonte escrito em C pode ser apresentado esquematicamente como na **Figura 19**. A origem da referência não foi encontrada. e o processo completo de construção de um programa multiarquivo pode ser visualizado conforme mostra a **Figura 20**. Nessas figuras, cada retângulo representa uma etapa do respectivo processo. Com exceção dos nomes contendo ***código-fonte*** ou ***códigos-fonte***, os demais nomes que não se encontram dentro de retângulos correspondem a arquivos gerados como resultado da execução de uma etapa do processo.

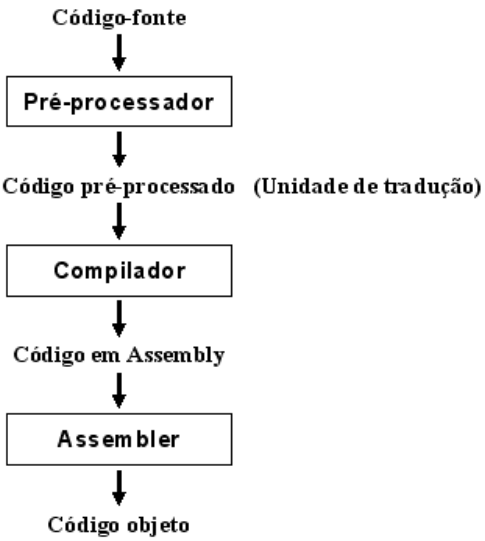

Figura 19: Processo completo de compilação de um arquivo-fonte

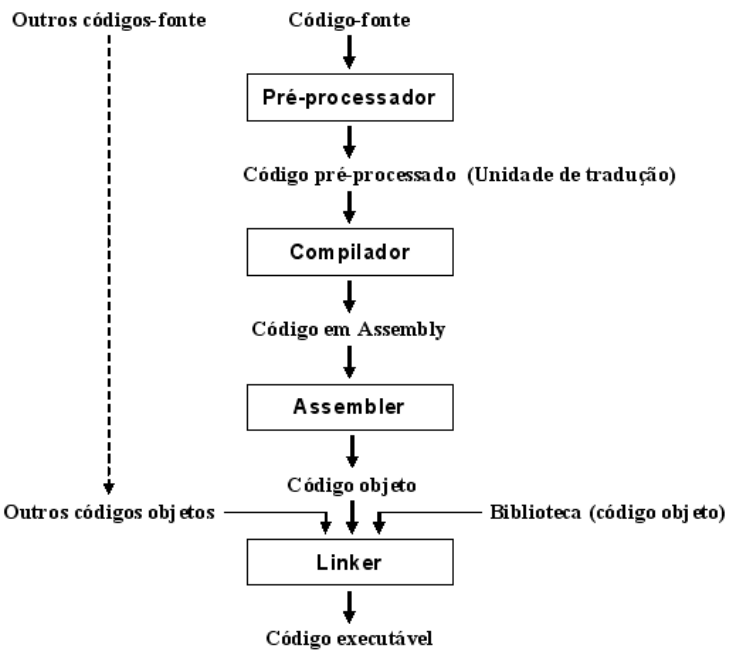

Figura 20: Processo completo de construção de um arquivo executável

Os arquivos intitulados *Código pré-processado* e *Código em Assembly* nas figuras são normalmente arquivos temporários apagados ao final de cada processo. No entanto, alguns compiladores permitem que se obtenha um arquivo contendo apenas o resultado da etapa de pré-processamento. Por exemplo, quando o compilador gcc é utilizado com a opção `-E`, como em:

```
gcc MeuArquivo.c -E -o MeuArquivoPP.c
```

ele cria um arquivo denominado `MeuArquivoPP.c`, que difere do arquivo original `MeuArquivo.c` apenas pelo fato de não possuir mais diretivas[1].

O arquivo pré-processado é formalmente conhecido como **unidade de tradução** pelo padrão ISO de C e consiste em um arquivo de programa mais os arquivos que ele inclui menos as linhas de código desprezadas devido a compilação condicional (v. **Seção 5.4**). Enquanto compila uma unidade de tradução, o compilador de C não leva em consideração as demais unidades de tradução que fazem parte de um programa multiarquivo.

5.3 Macros

Uma macro é um identificador que possui usualmente uma seqüência de símbolos associada denominada **corpo da macro**. Basicamente, existem dois tipos de macros: sem argumentos e com argumentos.

Convencionalmente, nomes de macros são constituídos apenas de letras maiúsculas e sublinha, pois esta convenção facilita a identificação visual de macros espalhadas pelo programa.

Macros são usualmente definidas no início de um arquivo de programa ou em qualquer local de um arquivo de cabeçalho e cada macro tem validade de seu ponto de definição até o final do arquivo. Em outras palavras, uma macro não tem validade em arquivos diferentes daquele no qual a macro foi definida. Se você deseja que uma macro seja utilizada em vários arquivos, coloque-a num arquivo de cabeçalho e inclua-o nos arquivos em que desejar ter a macro disponível.

---

[1] Não espere, entretanto, que este arquivo seja escrito em puro C, pois o pré-processador pode acrescentar informações que serão utilizadas pelo compilador no processo subseqüente de compilação.

5.3.1 Macros sem Argumentos

Uma definição de macro sem argumentos tem a seguinte sintaxe:

#define *nome-da-macro seqüência-de-símbolos*

Como exemplo de macro sem argumentos, considere:

```
#define  NUMERO_DE_MESES_POR_ANO  12
```

Quando o nome de uma macro aparece num local diferente daquele de sua definição, o nome é substituído ***literalmente*** pelo corpo da macro. Este ato de substituição é denominado **expansão de macro**. Por exemplo, considerando a definição de macro do exemplo anterior, na expressão a seguir, a macro `NUMERO_DE_MESES_POR_ANO` seria expandida em `12`:

```
numeroDeAnos = totalDeMeses / NUMERO_DE_MESES_POR_ANO;
```

Macros sem argumentos são também conhecidas como constantes simbólicas (v. **Seção 1.2.6**).

5.3.2 Macros com Argumentos

A utilização de argumentos torna as macros semelhantes às funções. Este tipo de macro é definida usando-se a seguinte sintaxe:

#define *nome-da-macro*(*argumentos-da-macro*) *corpo-da-macro*

Os argumentos de uma macro devem ser separados por vírgulas e usualmente são representados por letras minúsculas. Deve-se notar ainda que argumentos de macros não são variáveis; i.e., eles não têm nem tipo nem memória alocada para si. Como exemplo de macro com argumento, considere a seguinte definição de macro:

```
#define  QUADRADO(a) ((a)*(a))
```

A diretiva apresentada acima define a macro `QUADRADO` com um argumento e tem por objetivo definir o quadrado de um número.

Uma macro com argumentos é utilizada da mesma forma que uma chamada de função. Por exemplo, na instrução:

```
x = QUADRADO(5);
```

a macro QUADRADO seria expandida pelo pré-processador, resultando em:

```
x = ((5)*(5));
```

Neste ponto, você deve estar se perguntando por que são utilizados tantos parênteses, que aparentemente são redundantes, na definição da macro QUADRADO. A resposta a esta questão é que, devido ao modo como macros são expandidas, a ausência desses parênteses pode fazer com que a macro não funcione em certas circunstâncias. Por exemplo, suponha que os parênteses em torno do corpo da macro do último exemplo sejam deixados de fora, resultando em:

```
#define  QUADRADO_ERRADO1(a) (a)*(a)
```

então, se esta macro fosse utilizada na instrução:

```
y = 100/QUADRADO_ERRADO1(5);
```

a expansão resultaria em:

```
y = 100/(5)*(5);
```

que é equivalente a:

```
y = (100/5)*5;
```

Em resumo, no último exemplo, a intenção do programador era certamente dividir 100 pelo quadrado de 5, o que deveria resultar em 4. Entretanto, devido à ausência de parênteses em torno do corpo da macro, a expressão foi expandida para 100/5*5, que resulta em 100. Esse exemplo explica, portanto, a razão da existência dos parênteses externos no corpo de uma macro.

Agora, suponha que se decida deixar de fora os parênteses em torno de cada argumento da macro, resultando na seguinte macro:

```
#define  QUADRADO_ERRADO2(a) (a*a)
```

O problema aqui surgiria quando a macro acima fosse utilizada numa expressão como a seguinte:

```
z = QUADRADO_ERRADO2(3 + 1);
```

A macro deste último exemplo seria expandida em:

```
z = (3 + 1*3 + 1);
```

que resultaria, finalmente, em `7`, enquanto de acordo com a mais provável intenção do programador a expressão deveria resultar em `16`. Assim, esse exemplo explica a razão da existência dos parênteses em torno de cada argumento no corpo de uma macro.

Além desses problemas decorrentes da ausência de parênteses em definições de macros, existem outros problemas potenciais que poderão se manifestar com o uso indevido de macros. Algumas recomendações para evitar estes problemas serão apresentadas na **Seção 5.3.11**.

De acordo com o padrão C99, macros com argumentos podem ser invocadas com argumentos vazios. Por exemplo, considerando a definição de macro:

```
#define SOMA(x, y) (x + y)
```

a seguinte invocação da macro seria considerada legal:

```
SOMA(, 5)
```

e a expansão da macro resultante desta invocação seria:

```
( + 5)
```

Note que, quando se utilizam argumentos vazios, as vírgulas que separam estes argumentos devem aparecer como quando os argumentos não são vazios.

Nem sempre o uso de uma macro com argumentos vazios é expandida numa expressão legal. Por exemplo, a invocação da macro SOMA:

```
SOMA(5, )
```

seria expandida em:

```
(5 + )
```

que é uma expressão ilegal em C.

O uso de parênteses em torno de argumentos da macro pode impedir que ela seja chamada com qualquer argumento vazio. Por exemplo, se a macro SOMA tivesse sido definida como:

```
#define SOMA2(x, y) ((x) + (y))
```

qualquer das chamadas a seguir seria expandida numa expressão ilegal:

```
SOMA2(, 5); /* Expansão ILEGAL: (() + (5)) */
SOMA2(5, ); /* Expansão ILEGAL: ((5) + ()) */
```

O corpo de uma macro pode conter uma ou mais instruções ou declarações de variáveis. Quando o corpo de uma macro possui mais de uma instrução ou declaração de variáveis, é comum utilizar-se o seguinte artifício:

```
do {instruções-e-declarações} while (0)
```

Esse jargão (v. **Seção 6.7**) permite que as instruções e declarações que constituem o corpo da macro sejam encapsuladas numa única instrução. Assim, a macro pode ser invocada terminando com ponto-e-vírgula, sem que este seja interpretado como instrução vazia. Por exemplo, considere a definição de macro a seguir:

```
#define ATRIBUI(a, b) {a = 5; b = -5;}
```

Quando esta macro for invocada como:

```
ATRIBUI(x, y);
```

o ponto-e-vírgula utilizado nesta chamada representa a instrução vazia, pois, quando a macro for expandida, ele sucederá o fecha-chaves. Por outro lado, se a macro for definida como:

```
#define ATRIBUI(a, b) do {a = 5; b = -5;} while (0)
```

a expansão da mesma chamada resulta em:

```
do {x = 5; y = -5;} while (0);
```

e o ponto-e-vírgula será considerado normalmente como um terminal de instrução.

5.3.3 Comparações entre Macros com Argumentos e Funções

Normalmente, macros são executadas mais rapidamente do que funções equivalentes, mas nem sempre o uso de macros é mais indicado do que o uso de funções. De fato, algumas vezes, é difícil decidir se uma dada operação deve ser implementada como função ou como macro. A seguir serão enumeradas algumas vantagens e desvantagens decorrentes do uso de macros em comparação com funções que não são definidas como ***inline*** (C99). Você deve fazer um balanço entre vantagens e desvantagens antes de decidir utilizar uma macro em substituição a uma função numa situação particular.

► Vantagens de Macros

Macros são mais rápidas, pois não demandam o ônus associado com chamadas de funções, como empilhamento/desempilhamento de variáveis locais e parâmetros.

Uma macro pode servir para vários tipos de dados, pois não existe restrição quanto aos tipos de dados dos argumentos de uma macro. Em contraste, é muito difícil escrever uma função igualmente adequada para quaisquer tipos de dados[2].

[2] A linguagem C++ permite a escrita de funções genéricas utilizando o conceito de molde.

▶ Desvantagens de Macros

Um argumento é avaliado a cada vez que é utilizado no corpo da macro, o que pode levar a resultados inesperados quando a macro é invocada com argumentos contendo operadores com efeitos colaterais (v. **Seção 5.3.11**).

O corpo de uma função é compilado apenas uma vez, enquanto o corpo de uma macro é compilado sempre que ela é invocada. Assim, um programa contendo muitas macros pode ocupar muito mais espaço em memória do que um programa que utiliza funções equivalentes.

Macros passam por um nível de tradução a mais do que funções. Isto é, macros são processadas pelo pré-processador e pelo compilador, enquanto funções são processadas apenas por este último. Este fato torna mais difícil depurar programas contendo macros.

Macros não checam os tipos de seus argumentos, o que torna o uso de macros mais sensível a erros do que o uso de funções.

Macros não podem chamar recursivamente a si mesmas. Conseqüentemente, elas não podem implementar algoritmos recursivos.

Macros não possuem endereços e, portanto, não podem ser representadas por ponteiros. Funções possuem esta propriedade, que pode ser muito útil em algumas aplicações, conforme será visto no **Capítulo 10**.

Pode-se concluir das observações acima que existe uma compensação entre macros e funções com relação à rapidez de execução e ao tamanho do programa objeto resultante. De um lado, macros são executadas mais rapidamente, mas o programa executável poderá ocupar muito espaço. Por outro lado, funções são relativamente mais lentas, mas o programa resultante ocupará menos espaço. O impacto decorrente da substituição de funções por macros em termos de rapidez de execução só será percebido se a função substituída for chamada com muita freqüência.

Certamente, o uso de macros é mais vantajoso em substituição a funções que aparecem em poucas instruções do programas, mas que são chamadas um número repetido de vezes. Por exemplo, se uma operação é executada em 100 locais diferentes de um programa, é melhor utilizar

uma função para implementar tal operação. Mas, se uma operação aparece numa única instrução que é executada 100 vezes (por exemplo, uma operação dentro de um laço de repetição), é melhor usar uma macro. Também, funções pequenas (i.e., cujos corpos contêm poucas instruções) são melhores candidatas à substituição por macros do que funções contendo muitas instruções.

Finalmente, devido às suas diferentes características, uma macro e uma função aparentemente equivalentes podem produzir resultados diferentes. Por exemplo, considere a seguinte definição de função:

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

À primeira vista, esta função parece ser equivalente à macro `QUADRADO` definida anteriormente. Realmente, ambas produzirão os mesmos resultados se os argumentos utilizados forem do tipo **int**. Agora, observe o que acontece quando um argumento do tipo **float**, digamos `2.5`, é passado para a função e para a macro. A macro produzirá o resultado correto (`6.25`), enquanto a função retornará `4` (por quê?), que provavelmente não seria o resultado esperado.

Muitos ambientes de desenvolvimento de C oferecem certas operações, contidas em arquivos de biblioteca, em dois formatos: função e macro. Em tal situação, o programador tem a liberdade de escolher o formato que melhor convém a um determinado programa (v. **Volume II**).

5.3.4 Comparações entre Macros com Argumentos e Funções Inline (C99)

Virtualmente, a única vantagem do uso de macros com relação ao uso de funções *inline* é que macros podem servir para vários tipos de dados. Na prática, porém, conforme foi visto na **Seção 3.6**, um dado compilador pode decidir não expandir uma função definida como *inline*.

5.3.5 Macros Produtoras de Strings: O Operador

Quando colocado antes de um argumento no corpo de uma macro, o símbolo #, que representa um operador de pré-processamento, faz com que o pré-processador envolva o argumento com aspas, que resultam numa cadeia de caracteres (*string*) constante em C.

O uso do operador # para produção de *strings* é indicado quando se deseja que um dado argumento de uma macro seja tratado tanto como uma expressão quanto como uma cadeia de caracteres. Considere, por exemplo, a seguinte macro:

```
#define IMPRIME_VALOR(x) printf("Valor de %s = %Lf\n", #x, x)
```

Quando esta macro é invocada, como no trecho de programa a seguir:

```
double x = 1.0, y = 2.0, z = 3.0;

...

IMPRIME_VALOR(x);
IMPRIME_VALOR(x*y);
IMPRIME_VALOR(x + y + z);
```

o pré-processador expande as chamadas da macro para:

```
printf("Valor de %s = %lf\n", "x", x);
printf("Valor de %s = %lf\n", "x*y", x*y);
printf("Valor de %s = %lf\n", "x + y + z", x + y + z);
```

É permitido concatenar *strings* constantes com strings produzidos por meio do operador # simplesmente colocando-os justapostos. Utilizando este conhecimento, a macro apresentada acima poderia ter sido definida como:

```
#define IMPRIME_VALOR2(x) printf("Valor de " #x " = %Lf\n", x)
```

Quando esta última macro é invocada, como em:

```
IMPRIME_VALOR(x + y + z);
```

o pré-processador expande esta macro em:

```
printf("Valor de " "x + y + z" " = %Lf\n", x + y + z);
```

O compilador (e não o pré-processador) é responsável pela concatenação dos strings que resulta em:

```
printf("Valor de x + y + z = %Lf\n", x + y + z);
```

De acordo com o padrão C99, o operador # pode ser aplicado sobre um argumento vazio, resultando num ***string*** vazio. Por exemplo:

```
#define TRANSFORMA_EM_STRING(x) #x
...
printf("%s", TRANSFORMA_EM_STRING());
```

Após a expansão da macro `TRANSFORMA_EM_STRING`, esta última instrução é transformada em:

```
printf("%s", "");
```

5.3.6 Concatenação de Tokens: O Operador

O operador de pré-processamento **##** faz com que dois ***tokens***[3] numa definição de macro sejam concatenados para formar um único item. Por exemplo:

```
#define CRIA_VAR(n)  variavel##n
```

Esta macro concatena o ***token*** `variavel` com o argumento da macro. Assim, se esta macro fosse invocada como:

```
double CRIA_VAR(2);
```

ela seria expandida em:

```
double variavel2;
```

[3] Um ***token*** é uma seqüência de caracteres que têm algum significado próprio num determinado contexto.

Observe que concatenação de ***tokens*** não é o mesmo que concatenação de ***strings*** e que o resultado obtido com o operador ## não seria possível utilizando-se outros tipos de macros vistos anteriormente.

O operador ## pode ser usado com argumentos vazios, de acordo com o padrão C99. Quando um dos operandos do operador ## é um argumento vazio, o resultado é o outro operando, se este não for um argumento vazio. Se ambos os operandos forem argumentos vazios, nada é produzido como resultado. Considere, como exemplo, a seguinte definição de macro:

```
#define CONCATENA(x, y) x ## y
```

As invocações:

```
CONCATENA(, zzz)
```

e

```
CONCATENA(zzz, )
```

seriam ambas expandidas em:

```
zzz
```

enquanto a chamada:

```
CONCATENA(, )
```

não produziria nada.

5.3.7 Removendo a Definição de uma Macro

Uma vez definida, uma macro retém sua definição até o final do arquivo-fonte que a contém, mas pode-se remover a definição de uma macro por meio da diretiva **#undef**. Esta diretiva é necessária quando se deseja redefinir uma macro com um valor diferente do valor atual, pois, caso contrário, isto não seria permitido. Isto é, podem existir várias definições de uma mesma macro num arquivo, mas estas definições devem ser iguais. Se você precisar redefinir uma macro com um valor diferente, precisará antes remover sua antiga definição por meio de **#undef**.

5.3.8 Macros Predefinidas

Existem algumas macros que são predefinidas (ou embutidas) no pré-processador de C e não é permitido remover a definição ou redefinir estas macros. Os nomes destas macros sempre começam e terminam com um duplo sublinha ("__"). Estas macros, juntamente com suas expansões, são apresentadas na **Tabela 26**.

MACRO	EXPANSÃO
__LINE__	O número da linha no arquivo-fonte na qual a macro é invocada.
__FILE__	O nome do arquivo-fonte no qual a macro é invocada.
__TIME__	A hora (***string***) na qual o arquivo atual foi compilado.
__DATE__	A data (***string***) na qual o arquivo atual foi compilado.
__STDC__	1, se o compilador é 100% ANSI/ISO; 0 ou indefinida se o compilador não for ANSI/ISO.
__STDC_VERSION__	`199901L` se o compilador segue a versão C99 do padrão ISO; `199409L` se o compilador segue a versão C95 deste padrão; indefinida em outros casos.

Tabela 26: Macros predefinidas

As macros **__LINE__** e **__FILE__** podem ser muito úteis durante a fase de depuração de um programa. Um exemplo prático de uso das macros **__LINE__** e **__FILE__** é a definição da macro `ASSERT` a seguir:

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

Como ilustração de uso dessa última macro, suponha que na linha número 100 do arquivo `main.c` de um programa a macro `ASSERT` seja invocada como:

```
ASSERT( i > 0 )
```

A macro ASSERT seria então expandida em:

```
if (!( i > 0) ) {
    printf("A condicao %s na linha %d do arquivo %s falhou.",
           "i > 0", 100, "main.c");
    exit(1);
}
```

Suponha ainda que, durante a execução do programa, neste ponto do mesmo, i seja menor do que 0. Então, a condição da instrução **if** anterior resultante da expansão da macro seria satisfeita e o programa imprimiria o seguinte no meio de saída padrão:

```
A condição i > 0 na linha 100 do arquivo main.c falhou.
```

O programa seria, então, abortado devido à chamada da função **exit()**[4]. A macro ASSERT é uma macro extremamente útil em depuração de programas, conforme será visto no **Capítulo 6**. Uma macro similar a esta apresentada aqui, denominada **assert** (em letras minúsculas), pode ser encontrada no arquivo de cabeçalho <assert.h> da biblioteca padrão de C.

As macros **__TIME__** e **__DATE__**, que são expandidas em ***strings***, servem para registrar a hora e a data da última vez que um arquivo foi compilado. Como exemplo de uso das macros **__TIME__** e **__DATE__**, considere a seguinte função que, quando invocada, imprime a data e a hora de compilação do arquivo que a contém:

```
void  ImprimeDataEHoraDeCompilacao( void )
{
   printf("O arquivo %s foi compilado as %s do dia %s\n",
              __FILE__, __TIME__,__DATE__);
}
```

A macro **__STDC__** serve para indicar se um compilador obedece à especificação ISO ou não e é útil quando utilizada em conjunto com compilação condicional (v. mais adiante).

[4] A função **exit()** (#include <stdlib.h>) provoca o encerramento imediato do programa. Utiliza-se o valor 0 como parâmetro desta função para indicar encerramento normal do programa e um valor diferente de zero em caso contrário. Maiores detalhes sobre esta função podem ser encontrados no **Volume II**.

Além das macros predefinidas apresentadas na **Tabela 26**, o padrão C99 introduz novas macros, que são apresentadas na **Tabela 27**.

MACRO	EXPANSÃO
__STDC_IEC_559__	1 se a implementação estiver em conformidade com o padrão IEC 559[5] para aritmética de ponto-flutuante; 0 ou indefinida em caso contrário.
__STDC_IEC_559_COMPLEX__	1 se a implementação estiver em consonância com o padrão IEC 559 para aritmética de números complexos; 0 ou indefinida em caso contrário.
__STDC_ISO_10646__	Um valor da forma `aaaammL` (por exemplo, `199712L`) se o tipo `wchar_t` da implementação representar caracteres de acordo com o padrão ISO/IEC 10646; neste caso, `aaaa` e `mm` representam, respectivamente, o ano e o mês de publicação das emendas e correções deste padrão incorporadas pela implementação.
__STDC_HOSTED__	1 se a implementação for baseada em hospedeiro (i.e., sistema operacional); 0 ou indefinida em caso contrário.

Tabela 27: Novas macros predefinidas introduzidas pelo Padrão C99

5.3.9 Macros Vazias

O corpo de uma macro pode ser vazio, dando origem a uma **macro vazia**. Uma macro vazia não é expandida em nada e é útil apenas quando utilizada em conjunto com compilação condicional (v. mais adiante). Pode-se, por exemplo, utilizar uma macro vazia para indicar que um programa está em fase de depuração. Por exemplo:

```
#define EM_DEPURACAO
```

[5] Este padrão também é conhecido como IEC 60559 e IEEE 754.

Neste exemplo, o que interessa é apenas o fato de a macro ser definida; o valor da macro em si não importa.

5.3.10 Macros com Listas de Argumentos Variáveis (C99)

O padrão C99 introduz o conceito de macros com listas de argumentos variáveis, que são semelhantes às funções com listas de argumentos variáveis vistas na **Seção 3.4**. Para definir-se uma macro desse tipo utiliza-se o identificador **__VA_ARGS__**. O uso deste identificador na definição de uma macro com uma lista variável de argumentos é exemplificado a seguir:

```
#define IMPRIME_ERRO(...) fprintf(stderr, __VA_ARGS__)
```

A macro do exemplo acima não possui nenhum argumento fixo, como seria exigido de uma função com lista de argumentos variáveis equivalente. Entretanto, requer-se que a lista de argumentos variáveis apareça depois do último argumento fixo da macro (como também ocorre com funções com listas de argumentos variáveis).

Quando uma macro com lista de argumentos variáveis é invocada, os argumentos reais que aparecem após o último argumento real fixo casa com o identificador **__VA_ARGS__**. Portanto, se a macro do exemplo anterior fosse invocada na linha 25 do arquivo `modulo3.c` como:

```
IMPRIME_ERRO("Erro na linha %d do arquivo %s", __LINE__,
__FILE__);
```

a expansão desta macro seria:

```
fprintf(stderr, "Erro na linha %d do arquivo %s", 25,
modulo3.c);
```

É importante notar que, na invocação de uma macro com lista de argumentos variáveis, se houver macros nos argumentos variáveis, estas serão expandidas antes de ocorrer o casamento da lista de argumentos variáveis com o identificador **__VA_ARGS__**. No último exemplo, é

exatamente isto que ocorre com as macros **__LINE__** e **__FILE__**. O mesmo raciocínio aplica-se a macros construídas com os operadores # e ##.

Note que macros que não possuem argumentos fixos (como a do exemplo anterior) podem ser invocadas sem nenhum argumento. Por exemplo:

```
IMPRIME_ERRO();
```

seria expandida em:

```
fprintf(stderr);
```

o que resultaria, neste caso, em erro na chamada da função **fprintf()**, que requer, no mínimo, dois argumentos. Mas este erro seria detectado pelo compilador ou pelo editor de ligações (***linker***) e não pelo pré-processador.

Por outro lado, se uma macro com lista de argumentos variáveis for definida com argumentos fixos, além dos argumentos variáveis, ela deve ser invocada com no mínimo o mesmo número de argumentos fixos com o qual a macro foi definida. Por exemplo, se a macro `IMPRIME_ERRO2`, definida como:

```
#define IMPRIME_ERRO2(arquivo) fprintf(arquivo, __VA_ARGS_
_)
```

fosse invocada como:

```
IMPRIME_ERRO2();
```

ocorreria um erro, pois a macro `IMPRIME_ERRO2` requer, no mínimo, um argumento.

O padrão C99 sugere que o identificador **__VA_ARGS__** seja utilizado apenas na definição de macros com listas de argumentos variáveis. Qualquer outro uso deste identificador produzirá erro de compilação ou um comportamento indefinido no programa.

5.3.11 Cuidados com o Uso de Macros

Macros devem ser utilizadas com cautela, pois algum descuido pode acarretar erros difíceis de depurar. Conforme poder-se-á constatar nesta seção, macros são fontes potenciais de problemas. Portanto, seu uso deve ser restringido ao mínimo. Se seu compilador dá suporte a funções ***inline*** (v. **Seção 3.6**), o uso de macros pode ser preterido completamente em favor desse tipo de função. Mas, se realmente você acha que a melhor opção numa dada situação é o uso de macro, a seguir estão enumerados alguns cuidados especiais que você deve seguir ao definir e invocar macros.

► **Nunca termine uma macro com ponto-e-vírgula**

Este erro é comum e muitas vezes é inócuo, mas algumas vezes pode acarretar em erros de programação que são difíceis de corrigir. Por exemplo, considere a seguinte definição (errônea) de macro:

```
#define TAMANHO_MAXIMO   100;
```

Se esta macro for utilizada numa instrução tal como:

```
x = TAMANHO_MAXIMO;
```

após a expansão, a instrução aparecerá como:

```
x = 100;;
```

e o ponto-e-vírgula excedente será interpretado pelo compilador como uma instrução vazia. Nesta situação, portanto, não haveria nenhum problema. Mas, suponha que esta macro fosse utilizada na instrução a seguir:

```
y = TAMANHO_MAXIMO / 10;
```

Esta última instrução apareceria após a expansão como:

```
y = 100; / 10;
```

Quando o compilador encontrar esta instrução, obviamente ele irá indicar a existência de um erro sintático, pois esta instrução não é legal em C. A dificuldade que o programador inexperiente encontra para corrigir este

erro é que o compilador irá indicá-lo exatamente nesta linha de instrução e não na diretiva **#define**, onde o erro foi realmente introduzido. Afinal, tudo o que o programador vê como indicação de erro é a instrução:

```
y = TAMANHO_MAXIMO / 10;
```

que aparentemente é legal[6].

Este erro, causado pelo uso indevido de ponto-e-vírgula na macro, às vezes é difícil de ser consertado, mas isto ainda não é o pior, pois, como ocorre com todo erro sintático, o programa será executado apenas quando o erro for corrigido. O erro muito mais difícil de corrigir é aquele que o compilador não indica. Suponha, por exemplo, a seguinte definição (novamente errônea) de macro:

```
#define  CONDICAO_LEGAL  (indicador == 1);
```

Se esta macro for utilizada na instrução:

```
while  CONDICAO_LEGAL {
    ...
}
```

ela irá ser expandida em:

```
while  (indicador == 1); {
    ...
}
```

O problema aqui é similar àquele no qual coloca-se indevidamente ponto-e-vírgula antes do corpo de uma instrução **while** (v. **Seção 1.7.3**). Entretanto, aqui o problema é mais difícil de detectar, pois a instrução original (i.e., a primeira instrução **while** acima) parece ser perfeitamente legal.

Para concluir, se você se deparar com erros de programação não identificados num programa, você deve examinar todas as macros para

[6] Lembre-se de que o compilador começa a atuar após a expansão de todas as macros, o que é realizado pelo pré-processador.

verificar se alguma delas possui ponto-e-vírgula no final. A recomendação de usar apenas letras maiúsculas em identificadores de macros facilita a identificação visual de macros e, por conseguinte, a depuração de programas que as contêm.

► **Quando o corpo da macro contiver uma expressão, envolva cada argumento, bem como toda a expressão, com parênteses**

A não-observância desta regra pode levar a resultados inesperados, conforme foi discutido anteriormente. Portanto, mesmo que você tenha certeza que a macro não será utilizada em situações que acarretariam em erro, habitue-se a sempre seguir esta regra.

► **Nunca invoque uma macro utilizando expressões contendo operadores com efeitos colaterais**

Este problema refere-se à utilização de macros e não a definições em si. Considere, por exemplo, a seguinte definição de macro:

```
#define  MAXIMO(a, b) ((a) > (b) ? (a) : (b))
```

Esta última macro, se utilizada corretamente, resulta no maior entre dois números. Agora, suponha que esta macro seja utilizada como a seguir:

```
z = MAXIMO(x++, y);
```

Esta última instrução será expandida como:

```
z = ((x++) > (y) ? (x++) : (y));
```

O que acontece aqui é que, quando `x++` for maior do que `y`, a variável `x` será incrementada duas vezes, o que provavelmente não é aquilo que se deseja. Portanto, por segurança, não chame macros utilizando argumentos contendo operadores com efeitos colaterais.

5.4 Compilação Condicional

A compilação condicional permite que certos trechos de um

programa sejam compilados ou não de acordo com o valor de uma ou mais condições. Isto pode ser obtido por meio de dois conjuntos de diretivas similares à instrução condicional **if-else** de C. Esses conjuntos de diretivas serão explorados a seguir.

5.4.1 Testando o Valor de Macros: Diretivas #if, #else, #elif e #endif

As diretivas **#if**, **#else**, **#elif** e **#endif** possuem a seguinte sintaxe:

```
#if condição1
trecho-de-programa-a-ser-compilado-se-condição1-for-satisfeita
#elif condição2
trecho-de-programa-a-ser-compilado-se-condição2-for-satisfeita
 ...
#elif condiçãoN
trecho-de-programa-a-ser-compilado-se-condiçãoN-for-satisfeita
#else
trecho-de-programa-a-ser-compilado-se-nenhuma-condição-for-satisfeita
#endif
```

A expressão condicional que segue um **#if** ou **#elif** deve ser uma constante (usualmente representada por uma macro sem argumentos) e sua interpretação é semelhante àquela vista anteriormente para expressões condicionais em C. Existem, entretanto, algumas diferenças sintáticas adicionais com relação ao **if-else** de C que não são aparentes no esquema sintático apresentado acima:

- A expressão condicional não precisa estar entre parênteses (mas eles podem ser acrescentados se você assim desejar).
- **#elif** é análogo a uma construção **else if** de C.
- Os trechos de programa não são delimitados por { e }, como blocos dentro de uma instrução **if-else**.
- Cada bloco de diretivas **#if** deve ser terminado com **#endif**.

Outra diferença importante entre blocos de diretivas **#if** e instruções **if-else** é que essas diretivas não determinam se uma instrução será executada ou não; ***elas apenas especificam aquilo que será efetivamente compilado posteriormente pelo compilador***.

O pré-processador expande macros antes da avaliação de qualquer bloco de diretivas **#if**. Além disso, se uma expressão condicional contiver um nome que não tenha ainda sido definido, ele será expandido em zero. Por exemplo:

```
#undef  x
#if  x
```

é expandida em:

```
#if  0
```

Um uso comum de compilação condicional é na escolha entre os dois formatos de alusões de funções de acordo com o compilador utilizado, como ilustrado a seguir:

```
#if  (__STDC__ == 1)
    extern  int  MinhaFuncao(char  a, long  b);
#else
    extern  int  MinhaFuncao();
#endif
```

De acordo com as diretivas acima, se o programa estiver sendo compilado por um compilador ANSI/ISO, **__STDC__** é definido como sendo 1 e o formato moderno de alusão será utilizado; caso contrário, o estilo antigo será compilado. De qualquer modo, apenas uma das duas alusões será incluída no programa resultante.

A compilação condicional também é particularmente útil durante o estágio de depuração de um programa, conforme será visto no **Capítulo 6**.

5.4.2 Testando a Existência de Macros: Diretivas #ifdef e #ifndef

Pode-se especificar compilação condicional com base na existência ou não de uma macro (independentemente do valor da macro) utilizando,

respectivamente, as diretivas **#ifdef** ou **#ifndef**[7]. Como exemplo de compilação condicional com o uso da diretiva **#ifdef**, considere o seguinte trecho de programa:

```
#ifdef  TESTE
    printf("Isto é um teste.\n");
#else
    printf("Isto não é um teste.\n");
#endif
```

No último exemplo, se a macro `TESTE` for definida (com qualquer valor ou mesmo se for uma macro vazia), a primeira instrução **printf()** será compilada; caso contrário, a segunda chamada de **printf()** será compilada. Um exemplo mais prático de uso dessas macros é apresentado na **Seção 5.5**.

5.5 Inclusão de Arquivos

A diretiva **#include** para inclusão de arquivos já foi suficientemente discutida anteriormente (v. **Seção 2.2**). Aqui será apresentado um esquema que previne a inclusão múltipla de um arquivo de cabeçalho e que constitui um exemplo prático de uso de diretivas de compilação condicional e de macros vazias.

Prevenir a inclusão múltipla de um arquivo de cabeçalho é útil, pois economiza tempo de processamento do arquivo e, às vezes, é realmente necessário. Por exemplo, se o arquivo multiplamente incluído contiver uma declaração de tipo, o compilador indicará erro quando encontrar mais de uma declaração do mesmo tipo.

Suponha que um arquivo denominado `c1.h` inclua um arquivo denominado `c2.h` e que um arquivo denominado `arq.c` inclua ambos `c1.h` e `c2.h`. Como `c1.h` já é incluído em `c2.h`, `c1.h` seria incluído duas vezes em `arq.c` na ausência de uma estratégia que previna inclusão múltipla de arquivos. A situação é representada esquematicamente na **Figura 21** a seguir.

[7] Estas diretivas podem ser escritas como **#if defined** e **#if !defined**, respectivamente.

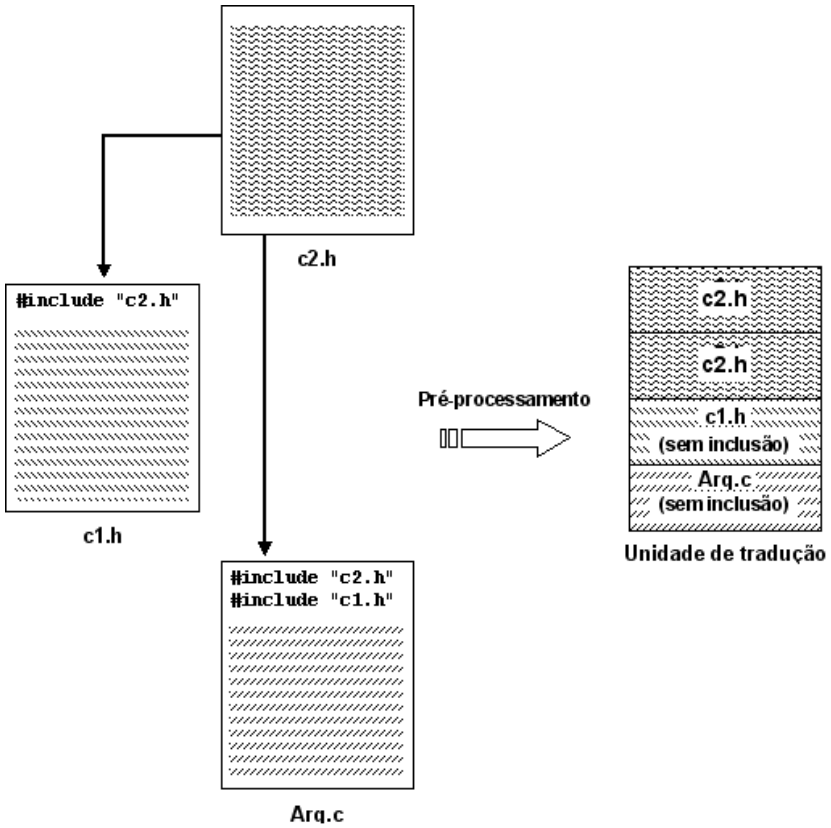

Figura 21: Inclusão múltipla de um arquivo de cabeçalho

Supondo que o arquivo de cabeçalho seja denominado `Arq1.h`, a estratégia utilizada para prevenir inclusão múltipla de arquivos de cabeçalho é esquematizada a seguir[8]:

```
#ifndef _Arq1_H_ /* Arq1.h é o nome do arquivo de cabeçalho */
#define _Arq1_H_

/* O conteúdo do arquivo vem aqui  */

#endif
```

[8] Note que os modelos de arquivos de cabeçalho para programas multiarquivos apresentados na **Seção 4.11** utilizam esta estratégia.

Utilizando-se essa estratégia, se um arquivo tenta incluir o cabeçalho `Arq1.h` sem tê-lo incluído antes, isto indica que a macro `_Arq1_H_` ainda não foi definida. Logo, esta macro é definida na segunda linha do arquivo `Arq1.h` como uma macro vazia e a inclusão do arquivo ocorre normalmente. Agora, suponha que um arquivo tente incluir o arquivo `Arq1.h` mais de uma vez. Neste caso, como a macro `_Arq1_H_` foi definida na primeira inclusão do arquivo `Arq1.h`, a diretiva de compilação condicional na primeira linha deste arquivo faz com que o restante do conteúdo do arquivo seja saltado, evitando assim a inclusão múltipla. Note que o sucesso dessa estratégia depende de uma boa escolha para o nome da macro de controle (aqui denominada `_Arq1_H_`), de modo que ele não coincida com o nome de nenhuma outra macro do programa. Usualmente, esta macro é escolhida como uma combinação do nome do arquivo com caracteres de sublinha.

5.6 A Diretiva #error

A diretiva **#error** permite que um erro seja indicado e a compilação seja interrompida durante o estágio de pré-processamento do programa. Esta diretiva segue esta sintaxe:

#error *caracteres*

Os caracteres que seguem a diretiva usualmente formam um texto explicativo e são impressos no meio de saída padrão logo antes de a compilação ser encerrada. Tipicamente, esta diretiva é utilizada para verificar se alguma condição ilegal de compilação está sendo utilizada. Por exemplo:

```
#ifdef  __WINDOWS__
#  error  Este programa não deve ser compilado para Windows
#endif
```

Portanto, se alguém tentar compilar o programa contendo as diretivas acima quando a macro __WINDOWS__ estiver definida, o pré-processador emitirá a mensagem de erro:

```
#error Este programa não deve ser compilado para Windows
```

e a compilação não prosseguirá.

5.7 A Diretiva #pragma

Uma diretiva **#pragma** é uma instrução para o compilador compilar o programa de um modo específico. Diretivas **#pragma** representam tarefas dependentes da implementação utilizada; isto é, cada compilador tem a liberdade de incluir identificadores que têm significados especiais quando precedidos pela diretiva **#pragma**. Por exemplo, o compilador Borland C++ contém a diretiva **#pragma warn,** que, quando utilizada com um nome que representa uma mensagem de advertência, faz com que o compilador suprima (nome da mensagem precedido por "-") ou ative (nome da mensagem precedido por "+") a apresentação da respectiva mensagem de advertência. A seção a seguir apresenta mais alguns exemplos de diretivas **#pragma**.

Consulte o manual do compilador de C que você está utilizando para verificar quais são as diretivas **#pragma** que ele suporta.

5.8 O Operador _Pragma (C99)

O padrão C99 introduziu o operador de pré-processamento **_Pragma**, que permite que diretivas **#pragma** possam ser produzidas como resultado de uma expansão de macro. A sintaxe de uso do operador **_Pragma** é:

_Pragma (*string-constante*)

onde ***string-constante*** é um ***string*** constante. Por exemplo:

```
_Pragma("comment(user, \"Meu Nome\")")
```

Note que a barra invertida é utilizada para permitir a escrita de aspas no interior do ***string*** que constitui o operando do operador **_Pragma**.

Quando encontra uma expressão composta pelo operador **_Pragma**, o pré-processador procede da seguinte maneira:

- As aspas externas do ***string*** são removidas.
- Os caracteres de escape (\) são removidos.
- Se o ***string*** for extenso, a letra L, que deve precedê-lo, é removida.
- Antepõe-se **#pragma** ao resultado da execução dos passos acima, criando, assim, uma diretiva pragma.

Seguindo estes passos, o pré-processador resolveria a expressão:

```
_Pragma("comment(user, \"Meu Nome\")")
```

como[9]:

```
#pragma comment(user, "Meu Nome")
```

Portanto, conforme demonstra o exemplo, a expressão:

```
_Pragma ("string-constante")
```

funciona de modo equivalente a:

```
#pragma string-constante
```

Em termos práticos, entretanto, uma expressão composta pelo operador **_Pragma** leva vantagem sobre uma diretiva **#pragma** equivalente porque tal expressão pode ser utilizada no corpo de uma macro. Como exemplo, considere o seguinte fragmento de programa:

[9] Esta diretiva pragma é encontrada no compilador Borland C++ versão 5 e permite que o programador ***assine*** o código-objeto produzido pelo compilador.

```
#define ASSINATURA(tipo, ass) PRAGMA(comment(tipo, ass))
#define PRAGMA(x) _Pragma(#x)
...
ASSINATURA(user, "Um Nome")
```

O uso da macro ASSINATURA definida no último exemplo facilita o uso da diretiva **#pragma** correspondente e melhora a legibilidade do programa.

O padrão C99 é ambíguo com relação a outras situações, além daquelas apresentadas aqui, nas quais uma expressão **_Pragma** pode ser utilizada. Portanto, recomenda-se que o uso de **_Pragma** seja restrito a estas situações.

5.9 A Diretiva #line

A diretiva **#line** pode ser utilizada para alterar a interpretação do compilador com relação à numeração de linhas de um arquivo-fonte e pode assumir três formatos. No formato mais simples, esta diretiva atribui um número de linha à próxima linha do programa que sucede a diretiva e, assim, as linhas seguintes terão numerações obtidas por meio de sucessivos incrementos a partir da primeira linha que sofreu alteração. Neste caso, a diretiva **#line** assume o seguinte formato:

#line *valor-inteiro-constante*

Como exemplo de uso da diretiva **#line** nesse formato, considere:

```
#line 150
x++;
printf("Valor de x = %d\n", x);
...
```

No fragmento de código acima, a linha de código contendo a expressão x++ será interpretada pelo compilador como tendo numeração 150, a linha contendo a chamada de **printf()** será interpretada como tendo numeração 151 e assim por diante.

A diretiva **#line** também pode ser utilizada para fazer com que o

compilador considere que o arquivo sendo compilado tem um nome diferente do original. Neste caso, a diretiva **#line** tem o seguinte formato:

#line *valor-inteiro-constante "nome-de-arquivo"*

Como exemplo de uso da diretiva **#line** neste último formato, considere:

```
#line 150 "teste.c"
x++;
printf("Valor de x = %d\n", x);
...
```

Este último exemplo é semelhante ao anterior, mas agora o compilador irá considerar que o nome do arquivo é `teste.c`, quer este seja ou não o verdadeiro nome do arquivo.

É importante ressaltar que nem o número de linhas nem o nome original do arquivo são alterados com o uso desta diretiva. Ou seja, esta diretiva apenas altera o modo como o compilador apresenta mensagens de erro e advertência. Portanto, esta diretiva é útil apenas em depuração de programas.

O último formato permitido para a diretiva **#line** tem a seguinte sintaxe:

#line *macro*

Neste formato, a macro contida na diretiva **#line** deve ser expandida e resultar num dos dois outros formatos anteriores e, então, ser processada conforme descrito anteriormente.

De acordo com o padrão C99, a diretiva **#line** deve permitir a especificação de um número maior do que zero até 2^{31}-1 linhas. Em versões anteriores do padrão de C, este limite superior era 2^{15}-1.

5.10 Diretiva Nula

Uma diretiva nula consiste simplesmente em # seguido de quebra de linha. A função desta diretiva é separar outras diretivas com o objetivo de

melhorar a legibilidade de um trecho de programa contendo várias outras diretivas.

5.11 Exercícios de Revisão

1. (a) O que é uma macro? (b) Descreva o uso de argumentos em macros. (c) Quais são as principais diferenças entre macros e funções em C?

2. Quais são os cuidados que devem ser tomados quando se utiliza macros?

3. Descreva (a) vantagens e (b) desvantagens do uso de macros em relação ao uso de funções.

4. Descreva (a) vantagens e (b) desvantagens do uso de macros em relação ao uso de funções definidas com **inline**.

5. (a) Por que é recomendável envolver parâmetros de macros entre parênteses? (b) Por que é recomendável envolver expressões contendo parâmetros de macros entre parênteses?

6. (a) O que é uma macro com lista de parâmetros variáveis? (b) Além das diferenças comuns entre macros e funções, o que distingue uma macro com lista de parâmetros variáveis de uma função com lista de parâmetros variáveis?

7. (a) Uma função pode ser chamada com algum parâmetro ausente? (b) E uma macro, pode?

8. O que é uma macro predefinida?

9. Para que serve uma macro vazia?

10. Descreva os seguintes operadores de pré-processamento:

(a) #
(b) ##
(c) **_Pragma**

11. (a) O que é compilação condicional? (b) Quais são as diretivas de pré-processador utilizadas em compilação condicional?

12. Qual é a diferença entre as diretivas **#if** e #**ifdef**?

13. Qual é a diferença entre **compilação condicional** e **execução condicional**?

14. O que é uma diretiva **#pragma**?

15. Qual é a utilidade prática do operador **_Pragma**?

16. Por que deve-se evitar inclusão múltipla de arquivos de cabeçalho?

17. Descreva a estratégia utilizada para prevenir a inclusão múltipla de arquivos de cabeçalho.

18. Explique o propósito de cada um dos seguintes grupos de diretivas:

(a)
```
#if !defined (SINALIZADOR)
   #define SINALIZADOR
#endif
```

(b)
```
#ifdef PASCAL
   #define  BEGIN  {
   #define  END    }
#endif
```

(c)
```
#ifdef  CELSIUS
   #define TEMPERATURA(t)   0.55*(t - 32)
#else
   #define TEMPERATURA(t)   1.8*t + 32
#endif
```

(d)
```
#ifndef  DEBUG
#define IMPRIME(x) printf("%s = %f\n", #x, double)x)
 #else
   #define  IMPRIME(x)  ;
 #endif
```

(e)
```
#ifdef DEBUG
    #undef DEBUG
#endif
```

(f)
```
#ifdef  DEBUG
    #define  MENSAGEM(msg)   printf("%s", msg)
#else
    #define  MENSAGEM(msg)   0
#endif
```

19. Considere a seguinte definição de macro:

```
#define IMPRIME(a, b) printf(#a " = %d " #b " = %d  \n", a,
b)
```

Como a invocação da macro `IMPRIME` no trecho de programa a seguir seria expandida pelo pré-processador?

```
int     a = 10;
int     b = 11;
...
IMPRIME(a, b);
```

20. Em Matemática, define-se o valor absoluto de um número real como:

$$|x| = \sqrt{x^2}$$

Assim, poder-se-ia, naturalmente, concluir que a macro:

```
ABS(x)  (sqrt((x)*(x)))
```

onde **sqrt()** é a função que calcula a raiz quadrada encontrada no módulo math da biblioteca padrão de C (`#include <math.h>`) seria adequada para calcular o valor absoluto de um número de ponto flutuante. Entretanto,

esta macro apresenta um sério problema. Qual é este problema? (**Sugestão**: Qual é o valor resultante da expansão desta macro quando x é igual a um enorme valor do tipo **double**?)

21. Defina uma macro, denominada `ABS(x)`, que deve ser expandida para o valor absoluto do argumento `x`.

22. Por que o programa a seguir não consegue ser compilado?

```
#include <stdio.h>

#define INVERSO (X) (1.0/(X))

int main()
{
   float umFloat;

   scanf("%f", &umFloat);

   printf("O inverso de %f e' %f\n",
           umFloat, INVERSO(umFloat));

   return 0;
}
```

23. O programa a seguir imprime o valor esperado? Por quê?

```
#include <stdio.h>

#define UM_VALOR     10;
#define OUTRO_VALOR  UM_VALOR -2;

int main()
{
   int x;

   x = OUTRO_VALOR;

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

   return 0;
}
```

24. O programa a seguir imprime o resultado esperado? Por quê?

```
#include <stdio.h>

#define PARCELA1        7
#define PARCELA2        5
#define DUAS_PARCELAS   PARCELA1 + PARCELA2

int main()
{
    printf("O quadrado das duas parcelas e' %d\n",
          DUAS_PARCELAS * DUAS_PARCELAS);

    return (0);
}
```

5.12 Exercícios de Programação

EP5.1) Desenvolva um projeto semelhante àquele do exercício **EP4.2**, mas, desta vez, substitua o módulo `Geometria` por um arquivo de cabeçalho (denominado `GeometriaM.h`) que substitui as funções providas por aquele módulo por macros correspondentes. Por exemplo, a função `AreaDeCirculo()` seria substituída pela macro:

```
#define AREA_DE_CIRCULO(raio) (PI*(raio)*(raio))
```

onde `PI` é uma constante simbólica definida no início do próprio arquivo `GeometriaM.h`.

SUGESTÕES

1. Utilize a notação sugerida para macros na **Seção 6.4**.

2. O módulo Interface do exercício **EP4.2** não precisa ser modificado.

3. O arquivo que contém a função **main()**, que você poderá denominar `main2.c`, precisa ser modificado apenas substituindo-se as chamadas de funções por chamadas das macros correspondentes.

4. Seu projeto terá agora apenas dois arquivos de programa: o arquivo `Interface.c` e o arquivo `main2.c`.