APÊNDICE **D**

Manual de Sintaxe da Linguagem C

SUMÁRIO

D.1 Elementos básicos de um programa
D.2 Estrutura de um programa C
D.3 O primeiro programa ANSI C
D.4 Palavras reservadas ANSI C
D.5 Diretivas do pré-processador
D.6 Arquivos de cabeçalho
D.7 Definição de macros
D.8 Comentários
D.9 Tipos de dados
D.10 Variáveis
D.11 Expressões e operadores
D.12 Funções de entrada e saída
D.13 Sentenças de controle
D.14 Funções
D.15 Estruturas de dados
D.16 Cadeias
D.17 Estruturas
D.18 Uniões
D.19 Campos de bits
D.20 Ponteiros
D.21 Pré-processador de C

D.1 ELEMENTOS BÁSICOS DE UM PROGRAMA

A linguagem C foi desenvolvida no Bell Laboratories para seu uso em pesquisa e é caracterizada por um grande número de propriedades que a tornam ideal para uso científico e de administração.

Uma das grandes vantagens da linguagem C é ser *estruturada.* Podem ser escritos laços que tenham condições de entrada e saída claras e se pode escrever funções cujos argumentos sejam sempre verificados para sua completa exatidão.

Sua excelente biblioteca-padrão de funções converte C em uma das melhores linguagens de programação que os profissionais da informática podem utilizar.

D.2 ESTRUTURA DE UM PROGRAMA C

Um programa típico em C se organiza em um ou mais *arquivos-fontes* ou *módulos.* Cada arquivo tem uma estrutura semelhante com comentários, diretivas de pré-processador, declarações de variáveis e funções e suas definições. Normalmente, cada grupo de funções e variáveis relacionadas está situado em um único arquivo-fonte. Dentro de cada arquivo-fonte, os componentes de um programa costumam se colocar em uma determinada forma padrão. A Figura D.1 mostra a organização típica de um arquivo-fonte em C.

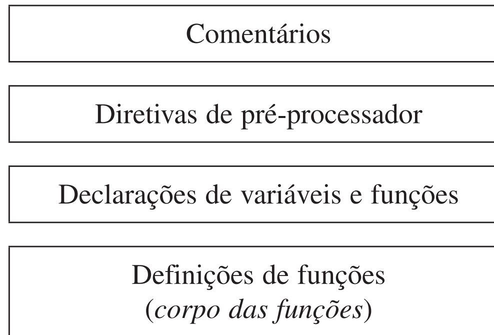


Figura D.1 Organização de um programa C.

Os componentes típicos de um arquivo-fonte do programa são:

1. O arquivo começa com alguns comentários que descrevem o propósito do módulo e informação adicional, como nome do autor e data e nome do arquivo. Os comentários começam com `/*` e terminam com `*/`.
2. Ordens ao pré-processador, conhecidas como *diretivas do pré-processador*. Normalmente, incluem arquivos de cabeçalho e definição de constantes.
3. Declarações de variáveis e funções visíveis em todo o arquivo. Em outras palavras, os nomes dessas variáveis e funções podem ser utilizados em qualquer função desse arquivo. Caso se deseje limitar a visibilidade das variáveis e funções *somente* a esse módulo, deve ser colocado diante de seus nomes o prefixo `static`; caso contrário, a palavra reservada `extern` indica que os elementos são declarados e definidos em outro arquivo.
4. O resto do arquivo inclui definições das funções (seu corpo). Dentro de um corpo de uma função, podem ser definidas variáveis que são locais à função e que somente existem no código da função que se está executando.

D.3 O PRIMEIRO PROGRAMA ANSI C

```
#include <stdio.h>
int main ()
{
  printf (“;Olá mundo!”);
  return 0;
}
```

D.4 PALAVRAS RESERVADAS ANSI C

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

D.5 DIRETIVAS DO PRÉ-PROCESSADOR

O *pré-processador* é a parte do compilador que realiza a primeira etapa da tradução ou compilação de um arquivo ANSI C em instruções de máquina. O pré-processador processa o arquivo-fonte e atua sobre as ordens, denominadas *diretivas de pré-processador,* incluídas no programa. Essas diretivas começam com o sinal de jogo da velha, #. Em geral, o compilador invoca automaticamente o pré-processador antes de começar a compilação. Podemos utilizar o pré-processador de três formas diferentes para tornar seus programas mais modulares, mais legíveis e mais fáceis de personalizar:

1. Pode ser utilizada a diretiva `#include` para inserir o conteúdo de um arquivo em seu programa.
2. Mediante a diretiva `#define`, podem ser definidas macros que permitem substituir uma cadeia por outra. Pode ser utilizada a diretiva `#define` para dar nomes significativos a constantes numéricas, melhorando a legibilidade de seus arquivos-fonte.
3. Diretivas tais como `#if`, `#ifdef`, `#else` e `#endif` podem compilar somente partes de seu programa. Essa característica pode ser utilizada para escrever arquivos-fonte com código para dois ou mais sistemas, mas compilar somente aquelas partes que sejam aplicadas ao sistema informático em que se compila o programa.

D.6 ARQUIVOS DE CABEÇALHO

Diretivas como `#include <stdio.h>` indicam ao compilador que leia o arquivo stdio.h de modo que suas linhas sejam situadas na posição da diretiva. ANSI C suporta dois formatos para a diretiva `#include`:

1. `#include <stdio.h>`
2. `#include "demo.h"`

O primeiro formato de `#include` lê o conteúdo de um arquivo — o arquivo-padrão de C, *stdio.h.* O segundo formato visualiza o nome do arquivo entre as aspas que está no diretório atual.

D.7 DEFINIÇÃO DE MACROS

Uma macro define um símbolo equivalente a uma parte de código C e é utilizada para isso a diretiva `#define`. Podem ser representadas constantes como `PI`, `IVA` e `BUFFER`.

```
#define PI 3.14159
#define IVA 16
#define BUFFER 1024
```

que assumem os valores `3.14159`, `16` e `1.024`, respectivamente. Uma macro também pode aceitar um parâmetro e substituir cada ocorrência desse parâmetro com o valor proporcionado quando a macro é utilizada em um programa. Conseqüentemente, o código que resulta da expansão de uma macro podem mudar dependendo do parâmetro que é utilizado quando executamos a macro. Por exemplo, a macro seguinte aceita um parâmetro e expande a uma expressão projetada para calcular o quadrado do parâmetro.

```
#define quadrado(x) ((x)*(x))
```

D.8 COMENTÁRIOS

O compilador ignora os comentários contidos entre os símbolos `/*` e `*/`.

```
/* Meu primeiro programa */
```

Podemos escrever comentários multilinha.

```
/* Meu segundo programa C
escrito no dia 15 de agosto de 1985
em Carchelejo – Jaén – Espanha */
```

Os comentários não podem ser aninhados. A linha seguinte não é legal:

```
/* Comentário /* comentário interno */ externo */
```

D.9 TIPOS DE DADOS

Os tipos de dados básicos incorporados a C são *inteiros, reais* e *caractere.*

Tabela D.1 Tipos de dados inteiros

Tipo de dado	Tamanho em bytes	Tamanho em bits	Valor mínimo	Valor máximo
`signed char`	1	8	–128	127
`unsigned char`	1	8	0	255
`signed short`	2	16	–32.768	32.767
`unsigned short`	2	16	0	65.535
`signed int`	2	16	–32.768	32.767
`unsigned int`	2	16	0	65.535
`signed long`	4	32	–2.147.483.648	2.147.483.647
`unsigned long`	4	32	0	4.294.967.295

O tipo `char` é utilizado para representar caracteres ou valores integrais. As constantes de tipo `char` podem ser caracteres entre aspas (`'A'`, `'b'`, `'p'`). Caracteres não impressos (tabulação, avanço de página etc.) podem ser representados com seqüências de escape (`'\t'`, `'\f'`).

Tabela D.2 Seqüências de escape

Caractere	Significado	Código ASCII
`\a`	Caractere de alerta (som)	7
`\b`	Retrocesso de espaço	8
`\f`	Avanço de página	12
`\h`	Nova linha	10
`\r`	Retorno de carro	13
`\t`	Tabulação (horizontal)	9
`\v`	Tabulação (vertical)	11
`\\`	Barra inclinada	92
`\?`	Sinal de interrogação	63
`\'`	Apóstrofo	39
`\"`	Aspas	34
`\nnn`	Número octal	–
`\xnn`	Número hexadecimal	–
`'\0'`	Caractere nulo (término da cadeia)	–

Tabela D.3 Tipos de dados de ponto flutuante

Tipo de dado	Tamanho em bytes	Tamanho em bits	Valor mínimo	Valor máximo
`float`	4	32	3.4E-38	3.4E+38
`double`	8	64	1.7E-308	1.7E+308
`long double`	10	80	3.4E-4932	3.4E+4932

Todos os números sem um ponto decimal em programas C são tratados como inteiros e todos os números com um ponto decimal são considerados reais de ponto flutuante de dupla precisão. Desejando representar números em base 16 (hexadecimal) ou em base 8 (octal), precede-se o número com o caractere `'OX'` para hexadecimal e `'0'` para octal. Desejando especificar que um valor inteiro se armazene como um inteiro longo, devemos seguir com um `"L"`:

```
025     /* octal 25 ou decimal 21 */
0x25   /* hexadecimal 25 ou decimal 37 */
250L   /* inteiro longo 250 */
```

D.10 VARIÁVEIS

Todas as variáveis em C são declaradas ou definidas antes de serem utilizadas. Uma *declaração* indica o tipo de uma variável. Se a declaração produz também armazenamento (se inicia), então é uma *definição*.

D.10.1 Nomes de variáveis em C

Os nomes de variáveis em C constam de letras, números e caractere sublinhado. Podem ser maiúsculas ou minúsculas ou uma mistura de tamanhos. O tamanho da letra é significativo. As variáveis seguintes são *todas diferentes*.

```
Temperatura         TEMPERATURA         temperatura
```

Às vezes utilizamos caracteres sublinhados e mistura de maiúsculas e minúsculas para aumentar a legibilidade.

```
Dia_da_Semana              DiaDaSemana        Nome_cidade        PagamentoMen

int x;                    /*     declara x variável inteira */
char nome, conforme;      /*     declara nome, conforme de tipo char */

int x = 0, não = 0;       /*     definem as variáveis x e y não */
float total = 42,125;     /*     define a variável total */
```

Podem ser declaradas variáveis múltiplas do mesmo tipo de duas formas. Assim, uma declaração:

```
int v1; int v2; int v3; inte v4;
```

ou

```
int v1;
int v2;
int v3;
int v4;
```

podendo-se declarar também da forma seguinte:

```
int v1, v2, v3, v4;
```

C não dá suporte a tipos de dados lógicos, mas mediante inteiros, podem ser representados: 0, significa *falso*; diferente de zero, significa *verdadeiro* (certo).

A palavra reservada `const` permite definir determinadas variáveis com valores constantes, que não podem ser modificadas. Assim, declaramos:

```
const int z = 4350;
```

e se tratamos de modificar seu valor

```
z = 3475;
```

o compilador emite uma mensagem de erro semelhante a "Cannot modify a `const` object in function main" ("Não podemos modificar um objeto `const` na função main"). As variáveis declaradas como `const` podem receber valores iniciais, mas não se pode modificar seu valor com outras sentenças, nem se utilizarem onde C espera uma expressão constante.

D.10.2 Variáveis tipo `char`

As variáveis de tipo `char` (caractere) podem armazenar caracteres individuais. Por exemplo, a definição:

```
char car = 'M';
```

declara uma variável `car` e lhe atribui o valor ASCII do caractere `M`. O compilador converte a constante caractere `'M'` em um valor inteiro (`int`), igual ao código ASCII de `'M'` que é armazenado a seguir em byte reservado para `car`.

Dado que os caracteres literais são armazenados internamente como valores `int`, pode ser mudada a linha

```
char car;
```

por

```
int car;
```

e o programa funcionará corretamente.

D.10.3 Constantes de cadeia

As cadeias de caracteres constam de zero ou mais caracteres separados por aspas. A cadeia é armazenada na memória como uma série de valores ASCII de tipo `char` de somente um byte e termina com um byte zero, que é chamado caractere nulo.

```
"Serra Magna em Jaén"
```

Além dos caracteres que são impressos, podemos guardar em constantes cadeia, códigos de escape, símbolos especiais que representam códigos de controle e outros valores ASCII não impressos. Os códigos de escape são representados na Tabela D.2 como um caractere único, armazenado internamente como um valor inteiro e composto de uma barra inclinada seguida por uma letra, sinal de pontuação ou dígitos octais ou hexadecimais. Por exemplo, a declaração

```
char c = '\n';
```

atribui o símbolo nova linha à variável C. Nos PCs, quando enviamos um caractere `'\n'` para um dispositivo de saída, ou quando escrevemos `'\n'` em um arquivo de texto, o símbolo nova linha se converte em um retorno de carro e um avanço de linha.

D.10.4 Tipos enumerados

O tipo enum é uma "*lista ordenada*" de elementos como constantes inteiras. A menos que seja indicado o contrário, o primeiro membro de um conjunto enumerado de valores recebe o valor 0, mas podem ser especificados valores. A declaração:

```
enum nome {enum_1, enum_2,...}lista_variáveis;
enum diasSemana {Segunda, Terça, Quarta, Quinta, Sexta, Sábado, Domingo};
```

significa que `Segunda = 0`, `Terça = 1` etc. Se fazemos `Quinta = 10`, então `Segunda` segue sendo 0, `Terça` é igual a 2 etc., mas agora `Sexta = 11`, `Sábado = 12` etc.

Um tipo enumerado pode ser utilizado para declarar uma variável.

```
enum diasSemana Trabalhados;
```

e a seguir utilizá-la com

```
Trabalhado = Quinta;
```

ou

```
Trabalhado = Sábado
if (Trabalhado >= Sexta)
  printf ("Hoje não é dia de trabalho \n";
```

D.10.5 `typedef`

A sentença `typedef` é utilizada para atribuir um nome novo a um tipo de dado derivado ou básico. `typedef` não define um novo tipo, mas simplesmente um nome novo para um tipo existente:

```
typedef struct
{
  float x;
  float y;
} PONTO;
```

A declaração de uma variável de tipo `PONTO`:

```
Ponto de origem = {0.0, 0.0}
```

D.10.6 Qualificadores de tipos `const` e `volatile`

A palavra reservada `const` pode ser situada antes de uma declaração de tipo para indicar ao compilador que não se pode modificar o valor

```
const int x5 = 100;
```

O modificador `volatile` indica explicitamente ao compilador que o valor muda (normalmente, de maneira dinâmica).

O qualificador `volatile` é utilizado na seguinte declaração de `porto17` para assegurar que o compilador avalie sempre qualquer acesso indireto por meio do ponteiro:

```
#define TTYPORT 0x17755U;
volatile char *porto17 =  (char *) TTYPORT;
*porto17 = '0';
*porto17 = 'N';
```

D.11 EXPRESSÕES E OPERADORES

As expressões são operações que o programa realiza.

```
a+b+c;
```

Tabela D.4 Operadores aritméticos

Operador	Descrição	Exemplo
*	Multiplicação	(a * b)
/	Divisão	(a / b)
+	Adição	(a + b)
-	Subtração	(a – b)
%	Módulo	(a % b)

Tabela D.5 Operadores relacionais

Operador	Descrição	Exemplo
<	Menor que	(a < b)
<=	Menor que ou igual a	(a <= b)
>	Maior que	(a > b)
>=	Maior que ou igual a	(a >= b)
==	Igual	(a == b)
!=	Diferente	(a != b)

Tabela D. 6 Operadores de incremento e decremento

Operador	Descrição	Exemplo
++	Incremento em i	++i, i++
--	Decremento em i	--j, j--

Exemplos

```
++i;        /*   Somar um a i */
i++;        /*   Igual ao anterior */
--i:        /*   Subtrai um a i */
i--;        /*   Igual ao anterior */
```

Tabela D. 7 Operadores de manipulação de bits (*bitwise*)

<table>
<tr><th>Operador</th><th>Descrição</th><th>Exemplo</th></tr>
<tr><td>&</td><td>AND bit a bit</td><td>C = A&B;</td></tr>
<tr><td>|</td><td>OR inclusivo bit a bit</td><td>C = A|B;</td></tr>
<tr><td>^</td><td>OR exclusivo bit a bit</td><td>C = A^B;</td></tr>
<tr><td><<</td><td>Deslocar bits para a esquerda</td><td>C = A<<B;</td></tr>
<tr><td>>></td><td>Deslocar bits para a direita</td><td>C = A>>B;</td></tr>
<tr><td>~</td><td>Complemento a um</td><td>C = ~B</td></tr>
</table>

D.11.1 Operadores de atribuição

Os operadores de atribuição são binários e combinações de operadores e do sinal = são utilizadas para abreviar expressões:

```
A = B                /* atribui o valor de B a A */
C = (A=B)            /* C e A são iguais a B */
C = A = B            /* atribui B a A e a C */
```

A = A + 45;	*equivale a*	A + = 45;

O compilador pode gerar um código mais eficiente, recorrendo-se a operações de atribuição compostas do tipo `*=`, `+=` etc., cada operador composto (*op*) reduz a expressão em pseudocódigo:

```
a = a op b
```

a forma abreviada

```
a op = b;
```

Tabela D.8 Operadores de atribuição

Operador	Descrição		Exemplo
=	Operação de atribuição simples		a = b;
* =	z *= 10;	*equivale a*	z = z * 10;
/ =	z /= 5;	*equivale a*	z = z / 5;
% =	z %= 2;	*equivale a*	z = z % 2;
+ =	z += 4;	*equivale a*	z = z + 4;
− =	z −= 5;	*equivale a*	z = z - 5;
<< =	z <<= 3;	*equivale a*	z = z << 3;
>> =	z >>= 4;	*equivale a*	z = z >> 4;
& =	z &= j;	*equivale a*	z = z & j;
^ =	z ^= j;	*equivale a*	z = z ^ j;
| =	z |= j;	*equivale a*	z = z | j;

D.11.2 Operador série

Na expressão `a, b`, é avaliada primeiro a expressão `a` e a seguir avalia-se a expressão `b`. O tipo e o valor da expressão são os de `b`.

O operador em série, a vírgula, indica uma série de sentenças executadas da esquerda para a direita. Utilizamos normalmente em laços `for`. Por exemplo:

```
for (conta=1; conta<100; ++conta, ++linhasporpágina);
```

produz o incremento da variável `conta` e da variável `linhasporpágina` cada vez que executamos o laço (realizamos uma iteração).

D.11.3 Operador condicional

Dadas as expressões `a`, `b` e `c`, a expressão

```
a ? b : c
```

recebe como valor `b` se `a` é diferente de zero, e `c` caso contrário (as expressões `b` e `c` devem ser do mesmo tipo de dados).

D.11.4 Operador de conversão de tipos

`(tipo)` `a` converte a um tipo especificado

expressão

nome de um tipo de dado básico, um tipo de dado enumerado, um tipo definido `typedef` ou um tipo de dado derivado.

Exemplo

`(int)29.55 + (int)21.99` *se avaliam em C como* `29 + 21`
`(float)6 / (float)4` *produz o resultado* `1.5 (6/4)`

D.11.5 Operador `sizeof`

`sizeof(tipo)` Recebe como valor o número de bytes necessários para conter um valor de um tipo especificado; `sizeof(int)` *proporciona 2 bytes.*

`sizeof a` Recebe como valor o número de bytes requeridos para conter o resultado da avaliação de `a`.

D.11.6 Prioridade (precedência) de operadores

As expressões C constam de diversos operandos e operadores. Em expressões complexas, as subexpressões com operadores de prioridade (precedência) mais alta são avaliadas antes das subexpressões com operadores de menor prioridade.

Tabela D.9 Ordem de avaliação e prioridade de operadores (associatividade)

<table>
<tr><th>Nível</th><th>Operadores</th><th>Ordem de avaliação</th></tr>
<tr><td>1</td><td>( ) . [ ] ></td><td>esquerda-direita</td></tr>
<tr><td>2</td><td>* & ! ~ ++ -- + -
(conversão de tipo) sizeof</td><td>direita-esquerda</td></tr>
<tr><td>3</td><td>* / %</td><td>esquerda-direita</td></tr>
<tr><td>4</td><td>+ -</td><td>esquerda-direita</td></tr>
<tr><td>5</td><td><< >></td><td>esquerda-direita</td></tr>
<tr><td>6</td><td>< <= > >=</td><td>esquerda-direita</td></tr>
<tr><td>7</td><td>= = !=</td><td>esquerda-direita</td></tr>
<tr><td>8</td><td>&</td><td>esquerda-direita</td></tr>
<tr><td>9</td><td>^</td><td>esquerda-direita</td></tr>
<tr><td>10</td><td>|</td><td>esquerda-direita</td></tr>
<tr><td>11</td><td>&&</td><td>esquerda-direita</td></tr>
<tr><td>12</td><td>||</td><td>esquerda-direita</td></tr>
<tr><td>13</td><td>?:</td><td>direita-esquerda</td></tr>
<tr><td>14</td><td>= *= /= += -= %= <<= >>= &= ^= |=</td><td>direita-esquerda</td></tr>
<tr><td>15</td><td>,</td><td>esquerda-direita</td></tr>
</table>

D.12 FUNÇÕES DE ENTRADA E SÁIDA

As funções `printf()` e `scanf()` permitem comunicar-se com um programa. Denominamos funções de E/S. `printf()` é uma função de saída e `scanf()` é uma função de entrada e ambas utilizam uma cadeia de controle e uma lista de argumentos.

D.12.1 `printf`

A função `printf` escreve no dispositivo de saída os argumentos da lista de argumentos. Requer o arquivo de cabeçalho *stdio.h.* A saída de `printf` é realizada com formato e seu formato consta de uma cadeia de controle e uma lista de dados.

```
printf (cadeia de controle [, item1, item2,...item]);
```

O primeiro argumento é a *cadeia de controle* (ou formato); determina o formato da escrita dos dados. Os argumentos restantes são os dados ou variáveis de dados a ser escritos.

```
printf (“Isto é um teste %d\n”, teste);
```

A cadeia de controle tem três componentes: texto, identificadores e seqüências de escape. Podemos utilizar qualquer texto e qualquer número de seqüências de escape. O número de identificadores tem de corresponder ao número de variáveis ou valores a serem escritos.

Os identificadores da cadeia de formato determinam como é escrito cada um dos argumentos.

```
printf (‘Meu povoado favorito é Cazorla%s”, msg);
```

cada identificador começa com um sinal de porcentagem `(%)` e um código que indica o formato de saída da variável.

Tabela D.10 Códigos de identificadores

Identificador	Formato
`%d`	Inteiro decimal
`%c`	Caractere simples
`%s`	Cadeia de caracteres
`%f`	Ponto flutuante (decimal)
`%e`	Ponto flutuante (notação exponencial)
`%g`	Usa o `%f` ou o `%e` mais curto
`%lf`	Tipo `double`
`%u`	Inteiro decimal sem sinal
`%o`	Inteiro octal sem sinal
`%x`	Inteiro hexadecimal sem sinal

As seqüências de escape são as indicadas na tabela:

```
print (“Minha flor favorita é a %s\n”, msg);
printf (“A temperatura é %f graus centígrados \n”,
                   centígrados);
```

Exemplo D.1

```
#include <stdio.h>
#define PI 3.141593
#define SERRA “Serra Magna”
int main (void)
{
```

```
  printf ("O valor de PI é %f.\n", PI);
  printf ("/%s/ \n", SERRA);
  print ("/%-20s/ \n", SERRA);
  return 0;
}
```

A saída produzida ao executar o programa é:

```
O valor de PI é 3.141593.
/Serra Magna/
/        Serra Magna/
```

D.12.2 scanf

A função `scanf()` é a função de entrada com formato. Pode ser utilizada para introduzir números com formato de máquina, caracteres ou cadeias de caracteres, em um programa.

```
scanf ("%f", &fahrenheit);
```

O formato geral da função `scanf()` é uma cadeia de formato e uma ou mais variáveis de entrada. A cadeia de controle consta somente de identificadores.

Tabela D.11 Identificadores de formato de `scanf`

Identificador	Formato
%d	Inteiro decimal
%c	Caractere simples
%s	Cadeia de caracteres
%f	Ponto flutuante
%lf	Ponto flutuante em tipo duplo
%e	Ponto flutuante
%ld	Inteiro longo
%u	Inteiro decimal sem sinal
%o	Inteiro octal sem sinal
%x	Inteiro hexadecimal sem sinal
%h	Inteiro curto

Um exemplo típico de uso de `scanf` é

```
print ("Introduza cidade e província : ");
scanf ("%s %s", cidade, província);
```

Outros exemplos de uso de `scanf`

```
scanf ("%d", &conta);
scanf ("%s", endereço);
scanf ("%d%d", &r, &c);
scanf ("%d*c%d", &x, &y);
```

D.13 SENTENÇAS DE CONTROLE

Uma sentença consta de palavras reservadas, expressões e outras sentenças. Cada sentença termina com um ponto-e-vírgula (;).

Um tipo especial de sentença, a *sentença composta* ou *bloco*, é um grupo de sentenças entre chaves ({...}). O corpo de uma função é uma sentença composta. Uma sentença composta pode ter variáveis locais.

D.13.1 Sentença `if`

1. `if (expressão)` / `sentença;` *ou* `if (expressão) sentença;`

```
if (expressão)
   sentença;
```

ou

```
if (expressão) sentença;
```

2. Se a sentença é *composta*

```
if (expressão) {

     sentença1;
     sentença2;
     ...
}
```

ou

```
if (expressão)
{
     sentença1;
     sentença2;
     ...
}
```

3.

```
if (expressão == valor)
     sentença;
```

4.

```
if (expressão != 0)
sentença;
```

equivale a

```
if (expressão)
```

Nota

(expressão != 0) e (expressão) são equivalentes, já que qualquer valor diferente de zero representa certo.

D.13.2 Sentença `if-else`

1.

```
if (expressão)
     sentença;
else
     sentença2;
```

2.

```
if (expressão) {

     sentença1;
     sentença2;
     ...
} else{

     sentença3;
     sentença4;
     ...
}
```

ou

```
if (expressão)
{
     sentença1;
     sentença2;
}
else
{
     sentença3;
     sentença4;

}
```

D.13.2.1 Sentenças `if` aninhadas

1.

```
if (expressão1)
   sentença1;
else if (expressão2)
   sentença2;
else
   sentença3;
```

2.
```
if (expressão1)
   sentença1;
else if (expressão2)
   sentença2;
else if (expressão3)
   sentença3;
else if (expressãoN)
   sentençaN;
else
   SentençaPorOmissão;        /* opcional */
```

D.13.3 Expressão condicional `(?:)`

Expressão condicional é uma simplificação de uma sentença `if-else`. Sua sintaxe é:

```
expressão1 ? expressão2 : expressão3;
```

que equivale a

```
if (expressão1)
  expressão2;
else
  expressão3;
```

Exemplo D.2

```
if (opção == 'S')           equivale a        prêmio = (opção == 'S')? 1000 : 0;
  prêmio = 1000
else
  prêmio = 0;
```

D.13.4 Sentença `switch`

A sentença `switch` realiza uma bifurcação múltipla, dependendo do valor de uma expressão:

```
switch (expressão) {
case valor1:
  sentença1;                /* é executada se expressão é igual a valor1 */
  break;                    /* saída de sentença switch           */
case valor2:
  sentença2;
  break;
case valor3;
  sentença3;
  break;
  ...
default;
sentença por omissão;       /* é executada se nenhum valor coincide com
                               expressão */

}
```

Exemplo D.3

```
switch (op)
{
case 'a':
  func1();
  break;
case 'b':
  func2();
  break;
case 'H':
  printf ("Olá \n");
default:
  print ("Saída \n");
};
```

D.13.5 Sentença `while`

```
while (expressão)
      sentença;
```

ou

```
while (expressão) {
   sentença1;
   sentença2;
      ...
}
```

D.13.6 Sentença `do-while`

```
do {
   sentença;
   ...
} while (expressão);
```

ou

```
do {
   sentença1;
   sentença2;
      ...
} while (expressão)
```

As sentenças `do-while` monossentenças podem ser escritas também assim:

```
do sentença; while (expressão);
```

D.13.7 Sentença `for`

```
1. for (expressão1; expressão2; expressão3) {
     sentença;
   }
2. for (expressão1; expressão2; expresão3) sentença;
```

A sentença `for` equivale a:

```
expressão1;
while (expressão2)      {
sentença;
expressão3;
}
```

Exemplo D.4

```
a. for (i = 1; i <= 100; i++)
   printf ("i = = %d\n" , i);
b. for (i = 0, soma = 0; i <= 100; soma + = i, i++);
```

D.13.8 Sentença nula

A *sentença nula* representada por um ponto-e-vírgula não faz nada. São utilizadas sentenças nulas em laços, quando todo o processo é feito nas expressões do laço em lugar do corpo. Por exemplo, localizar o byte zero que marca o final de uma cadeia:

```
char cad [80] = "Teste";
int i;
for (i = 0; cad [i] != '\0'; i++);
  /* sentença nula*/
```

D.13.9 Sentença `break`

A sentença `break` é utilizada para sair incondicionalmente de um laço `for`, `while`, `do-while` ou de uma sessão `case` de uma sentença `switch`.

```
for (;;) {
...
if (expressão)
   break;
}
```

D.13.10 Sentença `continue`

A sentença `continue` salta no interior do laço até o princípio do laço para prosseguir com a *execução*, deixando sem executar as linhas restantes depois da sentença `continue` até o final do laço. `continue` é semelhante à sentença `break`.

```
while (expressão1) {
    ...
if (expressão2)
   continue;
    ...
}
```

D.13.11 Sentença `goto`

Uma sentença `goto` dirige o programa a uma sentença específica que tem etiqueta utilizada nessa sentença `goto`. As etiquetas devem terminar com um símbolo dois pontos.

```
salto:
...
if (expressão) goto salto;
```

D.14 FUNÇÕES

As funções são os blocos de construção de programas C. Uma *função* é uma coleção de declarações e sentenças. Cada programa C tem pelo menos uma função: a função `main`. Esta é a função onde começa a execução de um programa C. A biblioteca ANSI C contém grande quantidade de funções-padrão.

Formato

```
tipo_retorno nome (tipo1 param1, tipo2 param2)
{
   declarações_variáveis_locais
   sentença
   sentença
   ...
   return expressão;
};
```

Ativação de funções

```
nome (arg1, arg2...)
```

Declaração de protótipos

```
tipo_retorno nome (tipo1 param1,tipo2 param2...);
```

Exemplo

```
float valor_absoluto (float x)
{
     if (x < 0)
       x = - x;
     return (x);
}
...
resultado = valor_absoluto (-15.5);
```

D.14.1 Sentença `return`

A sentença `return` detém a execução da função atual e devolve ao controle a função ativadora. A sintaxe é:

```
return expressão
```

onde o valor de *expressão* é devolvido como valor da função.

```
#include <stdio.h>
int main()
{
   printf ("Serra de Cazorla e \n");
   printf ("Serra Magna \n");
   printf ("são duas bonitas serras andaluzas \n");
   return 0;
}
```

D.14.2 Protótipos de funções

Em `ANSI C` deve ser declarada uma função antes de ser utilizada. A declaração da função indica ao compilador o tipo de valor que a função devolve e o número e os tipos de argumentos que são aceitos.

O *protótipo de uma função* é o nome da função, a lista de seus argumentos e o tipo de dado que devolve.

```
int SelecionarMenu (void);
double Área (int x, int y);
void sair (int estado);
```

D.14.3 O tipo `void`

Se uma função não devolve nem aceita nenhum parâmetro, `ANSI C` incorporou o tipo de dado `void`, que é útil para declarar funções que não devolvam nada e para descrever ponteiros que possam apontar qualquer tipo de dado.

```
void exit (int estado);
void ContaAcima (void);
void ContaAbaixo (void);
```

Erros típicos de funções

1. *Nenhum retorno de valor*. A função carece de uma sentença `return`. Se uma função termina sem executar `return`, devolve um valor imprescindível, que pode produzir erros graves.
2. *Retornos omitidos*. Se há funções que têm sentenças `if`, é preciso assegurar-se de que existe uma sentença `return` por caminho de saída possível.
3. *Ausência de protótipos*. As funções que carecem de protótipos devolvem `int`, inclusive ainda que estejam definidas para devolver valores de outro tipo. Como regra geral, declarar protótipos para todas as funções.
4. *Efeitos colaterais.* Esse problema é produzido normalmente por uma função que muda o valor de uma ou mais variáveis globais.

D.14.4 Deter um programa com `exit`

Um meio para terminar um programa sem mensagens de erro no compilador é utilizar a sentença

```
return valor;
```

onde *`valor`* é qualquer expressão inteira.
Outro meio para deter um programa é ativar `exit`. A execução da sentença

```
exit (0);
```

termina o programa imediatamente e fecha todos os arquivos abertos.

D.15 ESTRUTURAS DE DADOS

Os tipos complexos ou estruturas de dados em C são: *arrays, estruturas, uniões, cadeias* e *campos de bits.*

D.15.1 Arrays

Todos os arrays em C começam com o índice `[0]`.

```
int notas[25];          /* declara array de 25 elementos*/
float lista[10][25];    /* declara array de 10 por 25 elementos */
```

O número de elementos de um array pode ser determinado dividindo-se o tamanho do array completo pelo tamanho de um dos elementos.

```
nãoelementos = sizeof(lista)  /sizeof (lista[0]);
```

Um *array estático* pode ser iniciado quando é declarado com:

```
static char nome [] = "Serra Magna";
```

Um *array auto* pode ser iniciado somente com uma expressão constante em ANSI C.

```
int listamenor [2][2] = {{25, 4}, {100, 75}};
int dígitos[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Um *array de ponteiros* a caracteres pode ser iniciado com:

```
char*cores[] = {"verde", "vermelho", "amarelo", "rosa", "azul"};
```

Arrays unidimensionais

Os arrays podem ser definidos para conter qualquer tipo de dado básico ou qualquer tipo derivado. A declaração de um array tem o seguinte formato básico:

```
tipo nome [n] = {valor_inicial, valor_inicial ...}
```

expressão constante (valor_inicial)

número de elementos do array ([n])

Exemplos

```
char hoje [] = "Domingo";   se inicializa a:  'D', 'o', 'm', 'i', 'n', 'g', 'o', '\0'
char hoje [7] = "Domingo";  se inicializa a:  'D', 'o', 'm', 'i', 'n', 'g', 'o'.
```

Arrays multidimensionais

O formato geral para declarar um array multidimensional é:

```
tipo nome [d1] [d2]...[dn] = lista_inicialização
```

expressão constante

Exemplo

```
int três_d [5][2][20];       define um array de 3 dimensões três_d que contém 200 inteiros
três_d [4][0][15] = 100;     são armazenados 100 no elemento citado de três_d.
```

Declaração de um array de duas dimensões de quatro filas e três colunas

```
int matriz [4][3] ={
  {1,2, 3},         primeira fila: 1, 2, e 3
  { 4, 5, 6},       segunda fila: 4, 5 e 6
  { 7, 8, 9}        terceira fila: 7, 8, e 9
  {0,0,0} };        quarta fila: 0, 0, 0
```

A declaração anterior é equivalente a:

```
int matriz [4] [3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Por último, a declaração

```
int matriz [4][3] ={                {1}     primeiro elemento da primeira fila a 1
                                    {4}     segundo elemento da primeira fila a 4
                                    {7}};   terceiro elemento da terceira fila a 7
                                            resto dos elementos a zero
```

D.16 CADEIAS

As *cadeias* são simplesmente arrays de caracteres. As cadeias sempre terminam com um valor nulo `('\0')`.

```
char cadeia[80];        /* declara uma cadeia de 80 caracteres */
char mensagem[] = "Carchelejo está em Serra Magna";
char frutas[][10] = {"laranja", "banana", "maçã" };
```

Concatenação de cadeias

O pré-processador concatena automaticamente constantes de cadeias de caracteres adjacentes. As cadeias devem estar separadas por zero ou mais caracteres ou espaços em branco.

```
"um" "caractere" "cadeia"
```

equivale a

```
"umcaracterecadeia"
```

D.17 ESTRUTURAS

As **estruturas** são coleções de dados, normalmente de tipos diferentes, que atuam como um todo. Podem conter tipos de dados simples (caractere, float, array e enumerado) ou compostos (estruturas, array ou uniões).

Formato geral

```
struct nome
{
  declaração_membro
} lista_variáveis
```

declaração_membro	especificação de tipo seguida por uma lista de um ou mais nomes de membros.

Exemplos

1.
```
struct data
{
  int mês;
  int dia;
  int ano;
};
struct data hoje;
struct dada_data_compra;
struct data hoje, data_compra;

hoje.dia = 21;
hoje.ano = 2000;

if (hoje.mês == 12)
  mês_seguinte = 1;
```

2.
```
struct coordenada {
  int x;
  int y;
};

struct disciplina {
  char título [40];
  char autor [30];
  int páginas;
  int anopublic;
};
```

Uma variável de tipo estrutura pode ser declarada assim:

```
struct coordenada ponto;
struct disciplina livrosinfantis;
```

Para atribuirmos valores aos membros de uma estrutura, utilizamos a notação ponto (`.`).

```
ponto.x = 12;
ponto.y = 15;
```

Existe outra forma de declarar estruturas e variáveis estrutura.

```
struct coordenada {
  int x;
  int y;
} ponto;
```

Para evitar ter de escrever `struct` cada vez que declaramos a estrutura, podemos usar `typedef` na declaração:

```
typedef struct coordenada {
int x;
int y;
} Coordenadas;
```

e a seguir podemos declarar a variável `ponto`:

```
Coordenadas ponto;
```

e utilizar a notação ponto (.)

```
ponto.x = 12
ponto.y = 15;
```

D.18 UNIÕES

As uniões são quase idênticas às estruturas em sua sintaxe. As uniões proporcionam um meio de armazenar mais de um tipo em uma posição de memória. Definimos um tipo união assim:

```
união tipodemo {
  short x;
  long 1;
  float f:
};
```

e se declara uma variável com

```
união tipodemo demo;
```

e pode ser utilizada

```
demo.x = 345;
```

ou

```
demo.y = 324567;
```

A união anterior pode ser iniciada assim:

```
união tipodemo { short x; long 1; float f; } = {75};
```

D.19 CAMPOS DE BITS

Os *campos de bits* são utilizados com freqüência para colocar inteiros em espaços menores dos que o compilador possa normalmente utilizar e são, conseqüentemente, dependentes da implementação. Uma estrutura de campos de bits especifica o número de bits que cada membro ocupa. Uma declaração de uma estrutura `pessoa`:

```
struct pessoa {
  unsigned idade;            /* 0 .. 99 */
  unsigned sexo;             /* 0 = homem, 1 = mulher */
  unsigned filhos;           /* 0 .. 15 */
};
```

e de uma estrutura de campos de bits

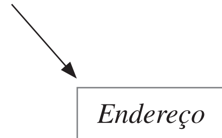
```
struct pessoa {
  unsigned idade: 7;    /* 0..1127 */
  unsigned sexo: 1;           /0 = homem, 1 = mulher */
  unsigned filhos: 4;   /* 0 ..15 */
}
```

D.20 PONTEIROS (Apontadores)

O **ponteiro** é um tipo de dado especial que contém o endereço de outra variável. Um ponteiro se declara utilizando o asterisco (*) diante de um nome de variável.

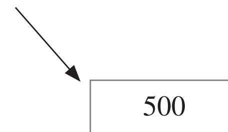
```
float *comprimentoOnda;          /* ponteiro a dados float */
char *índice;                    /* ponteiro a dados char */
int *p;                          /* ponteiro a dados int */
```

posição do ponteiro



Variável ponteiro

posição da variável



Dado direcionado por ponteiro

Figura D.2 Um ponteiro é uma variável que contém um endereço.

D.20.1 Declaração e indicação de ponteiros

```
int m;        /* uma variável inteira um */
int *p        /* um ponteiro p aponta um valor inteiro */
p = &m;       /* se atribui a p o endereço da variável m */
```

O operador de indicação (*) é utilizado para acessar o valor do endereço contido em um ponteiro.

```
float *comprimentoOnda;
comprimentoOnda = *cabo1;
*comprimentoOnda = 40.5;
```

D.20.2 Ponteiros `nulos` e `void`

Um ponteiro nulo aponta nenhuma parte específica; ou seja, não direciona a nenhum dado válido em memória:

```
fp = NULL;             /*atribui Nulo a fp */
fp = 0;                /* atribui Nulo a fp */

if (fp != NULL)        /* o programa verifica se o ponteiro é válido*/
```

Como uma função `void` que não devolve nenhum valor, um ponteiro `void` aponta um tipo de dado não especificado.

```
void *nãoapontafixo;
```

D.20.3 Ponteiros a valores

Para utilizar um ponteiro que aponte um valor, devemos utilizar o operador de indicação (*) diante da variável ponteiro.

```
double *total = &x;
*total = 34750.75;
```

O operador de endereço (&) atribui o ponteiro ao endereço da variável

```
double *total;
double ctotal;
total = &total;
```

D.20.4 Ponteiros e arrays

Se um programa declara

```
float *ponteirolista;
float listafloat [50];
```

então a `ponteirolista` pode ser atribuído o endereço do princípio de `listafloat` (o primeiro elemento).

```
ponteirolista = listafloat;
```

ou

```
ponteirolista = &listafloat [0];
listafloat [4]        é o mesmo que        *(listafloat + 4)
listafloat [2]        é o mesmo que        *(listafloat + 2)
```

D.20.5 Ponteiros a ponteiros

Os ponteiros podem apontar outros ponteiros (dupla indicação):

```
char **lista;          /* se declara um ponteiro a outro ponteiro char */
```

equivale a

```
char *lista[];
```

e

```
char ***ptr /* um ponteiro a um ponteiro a outro ponteiro char */;
```

D.20.6 Ponteiros a funções

```
float *ptr;                   /* ptr aponta a um valor float */
float *minhafun (void);       /* a função minhafun devolve um ponteiro a
                                 um valor float */
```

A declaração

```
int (*ptrafunção)(void);
```

cria um ponteiro a uma função que devolve um inteiro e a declaração

```
float (*ptrafunção1)(int x, int y);
```

declara `ptrafunção1` como um ponteiro a uma função que devolve um valor `float` e requer dois argumentos inteiros.

D.21 PRÉ-PROCESSADOR DE C

O pré-processador de C é um conjunto de sentenças específicas, denominadas diretivas, que são executadas quando começa o processo de compilação.

Uma diretiva começa com o símbolo # como primeiro caractere, que indica ao pré-processador que deve ser executada uma ação específica.

D.21.1 Diretiva `#define`

Utiliza-se para associar identificadores com uma seqüência de caracteres, e estes podem ser uma palavra reservada, uma constante, uma sentença ou uma expressão. No caso de o identificador representar sentenças ou expressões, denomina-se *macro*.

```
Sintaxe     #define     <identificador>[(parâmetro)] <texto>
```

Exemplos

1. `#define PI 3.141592`
2. `#define quadrado(x) x*x`
3. `#define área_círculo(r) PI*quadradro(r)`

1. `Se declara` a constante `PI`. Qualquer aparição de `PI` no programa será substituída pelo valor associado: `3,141592`.
2. `Quando` `quadrado(x)` aparece em uma linha do programa, o pré-processador a substitui por `x*x`.
3. A terceira diretiva declara uma operação matemática, uma fórmula, que inclui as macros `quadrado(x)` e `PI`. Calcula a área de um círculo.

Outros exemplos

```
#define LONGMÁX 81
#define SOMA(x,y) (x) + (y)
```

O pré-processador irá substituir no arquivo-fonte que contém estas duas macros pela constante `81` e a expressão `(x)+(y)`, respectivamente. Assim:

`float vetor[LONGMÁX];`	define um vetor de 81 elementos.
`printf("%d", SOMA(5,7));`	escreve a soma de 5 + 7.

Nota

Sendo uma diretiva do pré-processador, não se pode colocar (ponto-e-vírgula); ao final.

D.21.2 Diretiva `#erro`

Essa diretiva está associada à compilação condicional. No caso de que se cumpra uma condição, escreve-se uma mensagem.

```
Sintaxe     #erro     texto
```

O `texto` é escrito como uma mensagem de erro pelo pré-processador e termina a compilação.

D.21.2 Compilação condicional

A compilação condicional permite que certas sessões de código sejam compiladas dependendo das condições assinaladas no código. Por exemplo, pode desejar-se que o código não seja compilado se é utilizado um modelo de memória específico ou se um valor não está definido. As diretivas para a compilação condicional têm a forma de sentenças `if`; e são:

```
#if, #elif, #ifndef, #ifdef, #endif.
```

D.21.2.1 Diretiva `#if`, `#elif`, `#endif`

Permite selecionar código-fonte para ser compilado. Podemos fazer seleção simples ou seleção múltipla.

Formato 1

```
#if expressão_constante
...
#endif
```

Avalia-se o valor da `expressão_constante`. Se o resultado é diferente de zero, processam-se todas as linhas de programa até a diretiva `#endif`: caso contrário, são saltadas automaticamente e não são processadas pelo compilador.

Formato 2

```
#if expressão_constante1
...
#elif expressão_constante2
...
#elif expressão_constanteN
...
#else
...
#endif
```

Avalia-se o valor da `expressão_constante1`. Se o resultado é diferente de zero, todas as linhas são processadas até a diretiva `#elif`. Caso contrário, avalia-se a `expressão_constante2`, se é diferente de zero, são processadas todas as linhas até a seguinte `#elif` ou `#endif`. Caso contrário, segue-se avaliando a expressão seguinte, até que uma seja diferente de zero, ou processar-se as linhas incluídas depois de `#else`.

Exemplo

```
#if VGA
puts ("Está utilizando cartões VGA,");
#else
puts ("hardware de gráficos desconhecido.");
#endif
```

Escrevemos `"Está utilizando cartão VGA."` se VGA é diferente de zero, caso contrário escrevemos a segunda frase.

D.21.2.2 Diretiva `#ifdef`

Permite selecionar o código-fonte como está definido um macro (`#define`).

Formato

```
#ifdef identificador
...
#endif
```

Se `identificador` está definido previamente com `#define` identificador, então se processam todas as linhas de programas até `#endif`; caso contrário, não são compiladas essas linhas. Como a diretiva `#if`, as diretivas `#elif` e `#else` podem ser utilizadas conjuntamente com `#ifdef`.

D.21.2.3 Diretiva `#ifndef`

Agora a seleção do código-fonte é produzida se não está definida uma macro.

Formato 1

```
#ifndef identificador
...
#endif
```

Se `identificador` não está definido previamente com `#define` identificador, então todas as linhas do programa são processadas até `#endif`; em caso contrário, não se compilam essas linhas. Como a diretiva `#if`, as diretivas `#elif` e `#else` não podem ser utilizadas conjuntamente com `#ifndef`.

Exemplo

```
#ifndef _PAREJA
#define _PAREJA
struc pareja {
...
}
...
#endif
```

No exemplo, se não está definido a macro `_DUPLA`, é definido e processado o código-fonte que está escrito até `#endif`. Evita-se que o código-fonte seja processado mais de uma vez.

D.21.3 Diretiva `#include`

Permite acrescentar o código-fonte escrito em um arquivo ao arquivo que atualmente estamos escrevendo.

Formato 1 `#include` "*nome_arquivo*"

O arquivo *nome_arquivo* é incluído no módulo-fonte. O pré-processador busca no diretório ou diretórios especificados no *path* do arquivo. Normalmente buscamos no mesmo diretório que contém o arquivo-fonte. Uma vez encontrado, incluímos o conteúdo do arquivo no programa, no ponto preciso onde aparece a diretiva `#include`.

Formato 2 `#include` <*nome_arquivo*>

O pré-processador busca o arquivo especificado somente no diretório estabelecido para conter os arquivos de inclusão.

D.21.4 Diretiva `#line`

Formato `#line` constante <*nome_arquivo*>

A diretiva produz que o compilador trate as linhas posteriores do programa como se o nome do arquivo-fonte fosse <"*nome_arquivo*" e como se o número da linha de todas as linhas posteriores começasse por *constante. Se* <"*nome_arquivo*"> não for especificado, utiliza-se o nome do arquivo especificado pela última diretiva `#line`, ou o nome do arquivo-fonte (se nenhum arquivo foi especificado previamente).

A diretiva `#line` é utilizada principalmente para controlar o nome do arquivo e o número de linha que será visualizado sempre que seja emitida uma mensagem de erro pelo compilador.

D.21.5 Diretiva `#pragma`

Formato `#pragma` *nome_diretiva*

Com essa diretiva, são declaradas as diretivas que o compilador de C usa. Se, ao compilarmos o programa com outro compilador de C, este não reconhecer a diretiva, ela será ignorada.

D.21.6 Diretiva `#undef`

Formato `#undef` *identificador*

O *identificador* especificado é o de uma macro definida com `#define`; com a diretiva `#undef`, o *identificador* é convertido em não definido pelo pré-processador. As diretivas posteriores `#ifdef`, `#ifndef` irão comportar-se como se o identificador nunca tivesse sido definido.

D.21.7 Diretiva

Formato `#`

É a diretiva nula; o pré-processador ignora a linha que contém a diretiva `#`.

D.21.8 Identificadores predefinidos

Os identificadores seguintes estão definidos para o pré-processador.

Identificador	**Significado**
`__LINE__`	Número de linha atual que é compilada.
`__FILE__`	Nome do arquivo atual que é compilado.
`__DATE__`	Data de início da compilação do arquivo atual.
`__TIME__`	Hora de início da compilação do arquivo, no formato `"hh:mm:ss"`.
`__STDC__`	Constante definida como 1 se o compilador segue o padrão ANSI C.