UNIVERSIDADE DE SÃO PAULO - ICMC
DEPARTAMENTO DE CIÊNCIAS DE COMPUTAÇÃO E ESTATÍSTICA
SCE-181 INTRODUÇÃO À CIÊNCIA DE COMPUTAÇÃO II - 2º Sem /2004
Prof. Rudinei Goularte (rudinei@icmc.usp.br)
Sala 4-233.

1.

Elementos Léxicos, Operadores e o Sistema C

Material extraído de:
A Book on C. Al Kelley e Ira Pohl. Capítulo 2. Addison-Wesley, 2001 (4ª Edição).

C é uma linguagem. Como outras linguagens, tem um alfabeto e regras para colocar juntas palavras e pontuação de modo a programas corretos, ou legais. Essas regras são a *sintaxe* da linguagem. O programa que verifica a legalidade de um código em C é chamado de *compilador*. Se existe um erro, o compilador irá imprimir uma mensagem de erro e parar. Se não existem erros no código fonte, então o código é legal, e o compilador o traduz em um código objeto, o qual por sua vez é usado pelo carregador (montador) para produzir um arquivo executável.

Quando o compilador é chamado, o pré-processador faz seu trabalho primeiro. Por essa razão, nós podemos pensar no pré-processador como sendo construído dentro do compilador. Em alguns sistemas, esse é de fato o caso, enquanto em outros sistemas o pré-processador é separado. Temos de estar cientes de que podemos obter mensagens de erro tanto do pré-processador quanto do compilador. Nesse capítulo usaremos o termo *compilador* no sentido de que, conceitualmente, o pré-processador está construído dentro do compilador.

Um programa C é uma seqüência de caracteres que irão ser convertidos por um compilador C em código objeto, o qual, por sua vez, para uma linguagem-alvo em uma máquina em particular. Em muitos sistemas, a linguagem-alvo irá ter a forma de linguagem de máquina que pode ser executada ou interpretada. Para isso acontecer, o programa deve estar **sintaticamente correto**. O compilador primeiro classifica os caracteres do programa em *tokens*, os quais podem ser pensados como sendo o vocabulário básico da linguagem.

Em ANSI C (ou seja, C padrão ANSI, ou ainda, C padrão), existem seis tipos de *tokens*: palavras-chave (também conhecidas como palavras reservadas), identificadores, constantes, constantes string, operadores e pontuadores. O compilador verifica se cada *token* pode ser convertido em strings legais no contexto da sintaxe da linguagem. Vários compiladores são muito precisos nos seus requisitos. Diferente de um leitor humano de Português, o qual é capaz de entender o significado de uma sentença com um sinal de pontuação extra ou com uma palavra escrita incorretamente, um compilador C irá falhar em fornecer uma tradução de um programa sintaticamente incorreto, não importando o quão trivial seja o erro. Então, o programador deve aprender a ser preciso quando estiver escrevendo código. O programador deve ser bem-sucedido em escrever código legível ("entendível"). A chave para isso é produzir código bem documentado e com nomes significativos para identificadores.

1.1. Caracteres e Elementos Léxicos

Um programa C é construído pelo programador como uma seqüência de caracteres. Entre os caracteres que podem ser usados em um programa estão os seguintes:

Letras minúsculas	a b c d ... z
Letras maiúsculas	A B C D ... Z
Dígitos	0 1 2 3 ... 9
Outros caracteres	+ - * / = ( ) { } [ ] < > ‘ “ ! # % & _ | ^~\ . ; : ?
Caracteres espaço em branco	branco, nova linha, tabulação (tab), etc.

Esses caracteres são classificados pelo compilador em unidades sintáticas chamadas *tokens*. Vamos olhar um programa simples e informalmente analisar alguns de seus *tokens* antes de prosseguir em uma definição estrita da sintaxe C.

Arquivo sum.c

```
/*Lê dois inteiros e imprime sua soma */
#include <stdio.h>

int main (void)
{
        int a, b, sum;

        printf (“Entre dois números inteiros: “);
        scanf(“%d%d”, &a, &b);
        sum = a + b;
        printf(“%d + %d = %d\n”, a, b, sum);
        return 0;
}
```

- /*Lê dois inteiros e imprime sua soma */

Comentários são delimitados por /* e */. O compilador primeiro troca cada comentário por um espaço em branco. Então, o compilador ou ignora espaço em branco ou os usa para separar *tokens*.

- #include <stdio.h>

É uma diretiva de pré-processamento que faz com que o arquivo de cabeçalho stdio.h seja incluído. Esse cabeçalho foi incluído no programa por conter os protótipos das funções printf ( ) e scanf ( ). O compilador necessita de protótipos de funções para fazer seu trabalho.

- int main (void)
  {
      int a, b, sum;

O compilador agrupa esses caracteres em quatro tipos de *tokens*. A função main é um identificador, e os parênteses ( ) imediatamente seguindo main correspondem a um operador. À primeira vista essa idéia é confusa, porque o que se vê em seguida a main é (void). Contudo, apenas os parênteses ( ) em si constituem o operador. Esse operador indica ao compilador que main é uma função. Os caracteres "{", ",", e ";" são pontuadores; int é uma palavra-chave; a, b e sum são identificadores.

- int a, b, sum;

O compilador usa o espaço em branco entre int e a para distinguir esses dois *tokens*. Não se pode escrever:

inta, b, sum;

Por outro lado, o espaço em branco após a vírgula é supérfluo. Pode-se escrever int a,b,sum; mas não se pode escrever int absum;. O compilador consideraria absum como sendo um identificador.

- printf ("Entre dois números inteiros: ");
  scanf("%d%d", &a, &b);

Os nomes printf e scanf são identificadores e os parênteses que os seguem informam ao compilador que eles são funções.

- "Entre dois números inteiros: "

A série de caracteres entre aspas duplas é uma constante string. O compilador a trata como um único *token*.

- &a, &b

O caracter & é o operador de endereço. O compilador o trata como um *token*. Mesmo que os caracteres & e a estejam adjacentes, o compilador trata cada um como um *token* separado. Poderíamos ter escrito

```
& a , & b ou &a,&b
```

mas não

```
&a &b    /*falta o pontuador vírgula*/
a&, &b  /*& requer que seu operando esteja à direita*/
```

- sum = a + b;

Os caracteres = e + são operadores. Os espaços em branco serão ignorados, então, poderíamos ter escrito

```
sum=a+b; ou sum = a + b ;
```

mas não

```
s u m = a + b ;
```

Se tivéssemos feito do último modo, então cada letra naquela linha seria tratada pelo compilador como um identificador separado.

O compilador ou ignora espaços em branco ou os usa para separar elementos da linguagem. O programador usa espaços em branco para fornecer maior legibilidade ao código. Para o compilador, o texto do programa é implicitamente uma seqüência de caracteres, mas para o leitor humano, é um palco bidimensional.

1.2. Regras de Sintaxe

A sintaxe de C será descrita usando a Forma de Backus-Naur (Backus-Naur Form – BNF). Embora não seja adequada para descrever strings legais de C, ela é um meio padrão para descrever linguagens de alto nível modernas.

Uma categoria sintática irá ser escrita em itálico e definida por pontuações, também chamdas de regras de redefinição (ou de reescrita), tais como

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Isso deve ser lido como

> A categoria sintática *digit* é redefinida (ou reescrita) pelo símbolo 0, ou pelo símbolo 1, ..., ou pelo símbolo 9.

A barra vertical separa escolhas alternadas. Símbolos não em itálico são símbolos terminais da linguagem para os quis não se aplicam redefinições.

Itálico	Indica uma categoria sintática
::=	Símbolo “é reescrito como”
|	Barra vertical para separar escolhas
$\{\}_{1}$	Escolha um dos itens
$\{\}_{0+}$	Repete os itens 0 ou mais vezes
$\{\}_{1+}$	Repete os itens 1 um mais vezes
$\{\}_{opt}$	Itens opcionais

Outros símbolos são símbolos terminais da linguagem.

Vamos definir a categoria *letter_or_digit* significando qualquer letra maiúscula ou minúscula do alfabeto e qualquer digito decimal. Aqui está um modo de fazer isso:

letter_or_digit ::= *letter* | *digit*
letter ::= *lowercase_letter* | *uppercase_letter*
lowercase_letter ::= a | b | c | ... | z
uppercaseletter ::= A | B | C | ... | Z
digit ::= 0 | 1 | 2 | ... | 9

Agora vamos criar a categoria *alphanumeric_string* significando uma seqüência arbitrária de letras ou dígitos.

alphanumeric_string ::= $\{$*letter_or_digit*$\}_{0+}$

Usando essas produções vemos que strings de um único caracter como “3”, e strings de vários caractres como “ab6687c” assim como a string vazia “” são todas strings alfanuméricas. As aspas duplas em si não são parte da string.

Se nós desejamos garantir que uma string tenha pelo menos um caracter, nós devemos definir uma nova categoria sintática, tal como

alpha_string_1 ::= $\{letter\_or\_digit\}_{1+}$

1.3. Comentários

Comentários são strings arbitrárias de símbolos colocados entre os delimitadores /* e */. Comentários não são *tokens*. O compilador troca cada comentário por um espaço em branco. Então, comentários não são parte do programa executável. Exemplos:

```
/*comentário*/          /**** outro comentário ****/          /*****/
/*
*       Um comentário pode ser escrito desse modo para
*       destacá-lo do código ao redor
*/
```

1.4. Palavras-chave

Palavras-chave são explicitamente palavras reservadas que tem um significado estrito como *tokens* individuais em C. Elas não podem ser redefinidas e usadas em outros contextos.

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

Algumas implementações (compiladores) de C podem ter palavras-chave adicionais. Isso irá variar de uma implementação, ou sistema, para outro. Por exemplo, aqui estão algumas das palavras-chave adicionais do Turbo C: asm, cdecl, far, huge, interrupt, near, pascal.

1.5. Identificadores

Um identificador é um *token* que é uma seqüência de letras, dígitos e o caracter especial _, o qual é chamado de *underscore*. Uma letra ou um *underscore* deve ser o primeiro caracter de um identificador. Em muitas implementações de C, letras maiúsculas e minúsculas são tratadas de modo distinto. É uma boa prática de programação escolher identificadores que tenham significado mnemônico para que eles contribuam para a legibilidade e documentação do programa.

identifier ::= {*letter* | _ }$_1$ {*letter* | _ | *digit*}$_{0+}$

Alguns exemplos de identificadores são

K
_id
isiisimsm2
so_am_i

Os seguintes não são identificadores

```
not#me        /*o caracter especial # não é permitido*/
101_south     /* não deve iniciar com dígito*/
-plus         /*não confundir _ com -*/
```

Identificadores são criados para dar nomes a certos elementos de um programa. Palavras-chave podem ser pensadas como identificadores que são reservados por terem um significado especial. Identificadores como printf e scanf já são conhecidos pelo sistema C como funções de entrada e saída da biblioteca padrão. Esses nomes normalmente não deveriam ser redefinidos (criar identificadores com esses nomes para variáveis, por exemplo). O identificador main, em especial, devido aos programas C começarem sua execução na função chamada main.

Em ANSI C, os 31 primeiros caracteres de um identificador são discriminados (são usados de fato). Bons estilos de programação requerem que o programador escolha nomes que sejam significativos. Se você está escrevendo um programa para calcular taxas, você deveria ter identificadores como taxa, preco, taxa_basica. Assim, a sentença

```
taxa = preco * taxa_basica;
```

teria um significado óbvio. O *underscore* é usado para criar um único identificador para o que normalmente seria uma string de palavras separadas.

Cuidado: identificadores que começam com *underscore* podem entrar em conflito com nomes do sistema. Somente programadores do sistema deveriam usar tais identificadores. Considere o identificador _iob, o qual é freqüentemente definido como o nome de um array de estruturas em *stdio.h*. Se um programador tenta usar _iob para algum outro propósito, o compilador pode queixar-se ou, caso não acuse erro, o programa pode "se comportar mal" (ter um comportamento não esperado).

1.6. Constantes

Constantes caracter são escritas entre aspas simples, por exemplo, 'a' e '8'. Constantes em ponto flutuante apresentam o ponto decimal, por exemplo, 1.234 e 5.09852233. Inteiros decimais (base 10 – nosso sistema de numeração) são strings finitas de dígitos decimais. Devido a C fornecer inteiros octais (base 8) e hexadecimais (base 16), além de inteiros decimais, devemos ser cuidadosos para distinguir entre esses inteiros. Por exemplo, 17 é uma constante inteira decimal, 017 é uma constante inteira octal e 0x17 é uma constante inteira hexadecimal.

decimal_integer ::= 0 | *positive_decimal_integer*
positive_decimal_integer ::= *positive_digit* {*digit*}$_{0+}$
positive_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Alguns exemplos de constantes inteiras decimais:
0
77
123456
Mas não
0123 /* um inteiro octal*/
-49 /*uma constante expressão*/
123.0 /*uma constante em ponto flutuante*/

1.7. Constantes String

Uma seqüência de caracteres entre aspas duplas, como “abc”, é uma constante string, ou uma string literal. Elas são classificadas pelo compilador como um único *token*. Constantes string são sempre tratadas diferentemente de constante caracter. Por exemplo, ‘a’ e “a” não são a mesma coisa.

Note que aspas duplas “ corresponde a um único caracter e não a dois. Se o caracter “ em si deve ocorrer dentro de uma string, ela deve ser precedida pelo caracter barra invertida (backslash) \. Se o caracter \ deve ocorrer dentro de uma string, ele deve ser precedido por \. Alguns exemplos de constantes string são:

“uma string de texto”
“” /*a string nula*/
“ “ /*uma string com caracteres brancos*/
“a = b + c;” /*nada é executado*/
“/*isso não é um comentário*/”
“uma string com aspa dupla \” dentro”
“uma string com um único backslash \\ dentro”

Contra-exemplos:
/* “isso não é uma constante string*/
“e
nem isso”

Duas constante strings que são separadas apenas por espaço em branco são concatenadas pelo compilador em uma única string. Então,

“abc” “def” é equivalente a “abcdef”

Isso é uma característica nova da linguagem, disponível em ANSI C, mas não no C tradicional.

1.8. Operadores e Pontuadores

Em C existem muitos caracteres com significado particular. Exemplos incluem os operadores aritméticos

+ - * / %

os quais correspondem às operação usuais de adição, subtração, multiplicação, divisão e módulo, respectivamente. Relembre que em matemática o valor de *a* módulo *b* corresponde ao resto da divisão de *a* por *b*. Assim, 5 % 3 é 2 e 7 % 2 é 1.

Em um programa, operadores podem ser usados para separar identificadores. Apesar de tipicamente nós colocarmos espaço em branco ao redor de operadores binários para aumentar a legibilidade, em C isso não requerido.

```
a+b     /*isso é a expressão a mais b*/
a_b     /*isso é um identificador de 3 caracteres*/
```

Alguns símbolos têm significado dependente do contexto. Como um exemplo disso, considere o símbolo % nas duas sentenças

```
printf("%d", a);        e        a = b % 7;
```

O primeiro símbolo % é o início de uma conversão de especificação, ou formato, e o segundo % é o operador módulo.

Exemplos de pontuação incluem parênteses, colchetes, vírgula e ponto-e-vírgula. Considere o seguinte código

```
int main (void)
{
        int a, b = 2, c = 3;

        a = 17 * (b + c);
        ...
```

Os parênteses imediatamente após main são tratados como um operador. Eles dizem ao compilador que main é o nome de uma função. Após isso, os símbolos "{", ",", ";", "(" e ")" são pontuadores.

Ambos operadores e pontuadores são classificados pelo compilador como *tokens*, juntamente com espaço em branco, servem para separar elementos da linguagem.

Alguns caracteres especiais são usados em diferentes contextos, e o contexto em si pode determinar qual o uso apropriado. Por exemplo, parênteses são usados às vezes para indicar um nome de função; e outras vezes são usados como pontuadores. Outro exemplo é dado pelas expressões

```
a + b           ++a           a += b
```

Todos eles usam + como um caracter, mas ++ é um único operador, assim como +=. Ter o significado de um símbolo dependente do contexto faz com que o conjunto de símbolos se torne pequeno a linguagem concisa.

1.9. Precedência e Associatividade de Operadores

Operadores têm regras de precedência e de associatividade que são usadas para determinar como expressões são avaliadas. Essas regras não determinam completamente a avaliação porque o compilador tem liberdade para fazer mudanças para seus próprios propósitos. Uma vez que Expressões dentro de parênteses são avaliadas primeiro, parênteses podem ser usados para esclarecer ou mudar a ordem na qual operações são realizadas. Considere a expressão

1 + 2 * 3

Em C, o operador * tem prioridade mais alta que o operador +, fazendo com que a multiplicação seja realizada primeiro, e depois a adição. Então o valor da expressão é 7. Uma expressão equivalente é

1 + (2 * 3)

Por outro lado, devido a expressões entre parênteses serem avaliadas primeiro, a expressão

(1 + 2) * 3

é diferente, seu valor é 9. Agora considere a expressão

1 + 2 – 3 + 4 – 5

Devido aos operadores binários + e – terem a mesma precedência, a regra de associatividade "esquerda para direita" é usada para determinar como a expressão é avaliada. A regra "esquerda para direita" significa que as operações são realizadas da esquerda para a direita. Então,

(((1 + 2) – 3) + 4) – 5

é uma expressão equivalente. A tabela a seguir mostra as regras de precedência a associatividade para alguns dos operadores mais comuns se C.

Precedência de Operadores e Associatividade	
Operador	**Associatividade**
( ) ++ (pós-fixo) --(pós-fixo)	Esquerda para direita
+(unário) -(unário) ++(pré-fixo) --(pré-fixo)	Direita para esquerda
* / %	Esquerda para direita
+ -	Esquerda para direita
= += -= *= /= etc.	Direita para esquerda

Todos os operadores em uma mesma linha, tais como *, / e % têm precedência igual com respeito umas às outras, mas têm precedência maior que os operadores que ocorrem na linha abaixo deles. A regra de associatividade para todos os operadores em uma dada linha aparecem no lado direito da tabela. Quando nós introduzimos novos operadores, iremos fornecer suas regras de precedência e encapsular a informação aumentando a tabela. **Essas regras são informação essencial para qualquer programador C**.

Em adição ao operador binário mais que representa adição, existe o operador mais unário, e ambos os operadores são representados pelo sinal mais (+). O sinal menos também têm significado unário e binário. Note que o mais unário foi introduzido pelo ANSI C. Não exsite o mais unário no C tradicional, apenas o menos unário.

Pela tabela de precedência podemos ver que os operadores unários têm precedência maior que o mais e o menos binário. Na expressão

- a * b – c

o primeiro sinal de menos é unário. Usando as regras de precedência, nós podemos chegar à seguinte expressão equivalente

((-a) * b) – c

1.10. Operadores de Incremento e Decremento

O operador de incremento ++ e o operador de decremento -- são operadores unários. Eles são especiais porque eles podem ser usados tanto como operadores pré-fixos quanto como operadores pós-fixos. Seja val uma variável do tipo int. Então, ambos ++val e val++ são expressões válidas, com ++val ilustrando o caso de pré-fixação do operador ++ e, val++ ilustrando o caso de pós-fixação do operador ++.

No C tradicional, esses operadores têm a mesma precedência que os operadores unários. Em ANSI C, por razões técnicas, eles têm o nível mais alto de precedência e associatividade "esquerda para direita" para os operadores pós-fixos, e eles têm a mesma precedência que os outros operadores unários e associatividade "direita para esquerda" para os operadores pré-fixos.

Ambos, ++ e --, podem ser aplicados a variáveis, mas não a constantes ou expressões ordinárias. Mais ainda, diferentes efeitos podem ocorrer dependendo de como os operadores ocorrem em posições pré-fixas ou pós-fixas. Alguns exemplos são

```
++i
cnt—
```

mas não

```
777++           /*constantes não podem ser incrementadas*/
++(a * b -1)    /*expressões ordinárias não podem ser incrementadas*/
```

Cada uma das expressões, ++i e i++, tem um valor; mais ainda, cada uma faz com que o valor i seja armazenado na memória e incrementado de 1. A expressão ++i faz com que o valor armazenado de i seja primeiro incrementado, com a expressão tendo o seu valor como sendo o novo valor armazenado de i. Em contraste, a expressão i++ tem como seu valor o valor corrente de i; então, a expressão faz com que o valor armazenado de i seja incrementado de 1 (i passa a ser i + 1 após a expressão receber o valor de i). O código a seguir ilustra essa situação

```
int a, b, c = 0;

a = ++c;
b = c++;
printf("d %d %d\n", a, b, c);  /*1 1 3 é impresso*/
```

As expressões --i e i-- funcionam de modo semelhante.

Note que ++ e -- fazem o valor da variável na memória ser alterado. Outros operadores não fazem isso. Por exemplo, uma expressão como a + b deixa os valores das variáveis a e b intocados. Essas idéias são expressas quando dizemos que os operadores ++ e -- têm um efeito colateral; eles não apenas mantém um valor, eles mudam o valor armazenado de uma variável na memória.

Em alguns casos, o uso de ++ e -- em posições pré-fixas ou pós-fixas irão produzir resultados equivalentes. Por exemplo, cada uma das duas sentenças

```
++i;    e       i++;
```

é equivalente a

```
i = i + 1;
```

Em situações simples, pode-se considerar ++ e -- como operadores que fornecem notação concisa para incrementar e decrementar uma variável. Em outras situações, deve-se ser cuidadoso para qual posição, pré-fixa ou pós-fixa, é a desejada.

Declarações e Inicializações		
int a = 1, b = 2, c = 3, d = 4;		
Expressão	**Expressão equivalente**	**Valor**
a * b / c	(a * b) / c	0
a * b % c + 1	((a*b) % c) + 1	3
++ a * b – c --	((++a) * b) – (c--)	1
7 - - b * ++ d	7 – ((-b) * (++ d))	17

1.11. Operadores de Atribuição

Para mudar o valor de uma variável, nós temos usado sentenças de atribuição tais como

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

Diferente de outras linguagens, C trata = como um operador. É menor do que todos os operadores que temos visto até o momento, e sua associatividade é "direita para esquerda". Nessa seção explicamos em detalhes o significado disso.

Para entender = como um operador, vamos primeiro considerar +, para efeito de comparação. O operador binário + usa dois operandos, como na expressão a + b. O valor da expressão é a soma dos valores de a e b. Por comparação, uma simples sentença de atribuição é da forma

```
variável = lado_direito
```

onde lado_direito é em si uma expressão. Note que um ponto-e-vírgula colocado no fim teria transformado essa atribuição em uma sentença. O operador de atribuição = tem dois operandos: variável e lado_direito. O valor de lado_direito é atribuído à variável e esse valor se torna o valor da expressão de atribuição como um todo. Para ilustrar isso, considere as sentenças

```
b = 2;
c = 3;
a = b + c;
```

onde as variáveis são todas do tipo int. Fazendo uso de expressões de atribuição, nós podemos considerar isto

```
a = ( b = 2) + (c = 3);
```

A Expressão de atribuição b = 2 atribui o valor 2 para a variável b, e a expressão de atribuição em si toma esse valor. Similarmente, a expressão de atribuição c = 3 atribui o valo 3 à variável c, e a expressão de atribuição em si toma esse valor. Finalmente, os valores das duas sentenças de atribuição são somados, e o resultado é atribuído à variável a.

Apesar desse exemplo ser artificial, existem muitas situações onde atribuições ocorrem naturalmente como parte de uma expressão. Uma situação que freqüentemente ocorre é a atribuição múltipla. Considere a sentença

```
a = b = c = 0;
```

Devido ao operador = fazer associações da direita para a esquerda, uma sentença equivalente é

```
a = (b = (c = 0));
```

Primeiro, a c é atribuído o valor 0, e a expressão (c = 0) tem valor 0. Então, a b é atribuído o valor 0, e a expressão b = (c = 0) tem valor 0. Finalmente, à a é atribuído o valor 0 e a expressão a = (b = (c = 0)) tem valor 0. Muitas linguagens não usam atribuição de modo tão elaborado. C é diferente.

Em adição a =, existem outros operadores de atribuição, tais como += e -=. Um expressão como

```
k = k + 2
```

irá adicionar 2 ao antigo valor de k, e a expressão como um todo irá ter esse valor. A expressão

```
k += 2;
```

tem a mesma tarefa. A tabela a seguir possui todos os operadores de atribuição:

Operadores de Atribuição
= += -= *= /= %= >>= <<= &= ^= |=

Todos esses operadores têm a mesma precedência, e todos eles têm associatividade "direita para esquerda". A semântica é especificada por

variável op= expressão

a qual equivale a

variável = variável op (expressão)

com a exceção de que se variável em si é uma expressão, ela é avaliada somente uma vez. Quando trabalhando com *arrays*, este é um importante ponto técnico. Note que uma expressão de atribuição, tal como

```
j *= k + 3 é equivalente a j = j * (k + 3)
```

em vez de

```
j = j * k + 3
```

A tabela a seguir ilustra como expressões de atribuições são avaliadas:

Declarações e Inicializações			
int i = 1, j = 2, k = 3, m = 4;			
Expressão	**Expressão equivalente**	**Expressão equivalente**	**Valor**
i += j + k	i += (j + k)	i = (i + (j + k))	6
j *= k = m + 5	j *= (k = (m + 5))	j = (j * (k = (m + 5)))	18

1.12. O Sistema C

O sistema C consiste da Linguagem C, do pré-processador, do compilador, da biblioteca e de outras ferramentas úteis ao programador, tais como editores e depuradores. Nesta seção discutiremos o pré-processador e a biblioteca.

O Pré-processador

Linhas que começam com um # são chamadas diretivas de pré-processamento (ou de diretivas ao pré-processador). Essas linhas se comunicam com o pré-processador. Em C tradicional, as diretivas de pré-processamento deviam começa na coluna 1. Em ANSI C, essa restrição foi removida. Apesar de # poder ser precedido de uma linha em branco, começar diretivas de pré-processamento na coluna 1 continua sendo um estilo de programação comum.

Nós temos usado diretivas de pré-processamento tais como

```
#include <stdio.h>      e       #define PI 3.14159
```

Outra forma de usar #include é dada por

```
#include "nome_do_arquivo"
```

Isso diz ao pré-processador para trocar a linha com a cópia do conteúdo do arquivo nome_do_arquivo. Primeiramente, uma busca pelo arquivo especificado é feito no diretório corrente e então em outros locais dependentes do sistema. Com a diretiva de pré-processamento na forma

```
#include <nome_do_arquivo>
```

o pré-processador busca pelo arquivo apenas em "outros locais dependentes do sistema", e não no diretório corrente.

Devido à diretivas #include ocorrerem no início do programa, os arquivos incluídos são denominados arquivos de cabeçalho (*header files*), e um .h é usado no fim do nome do arquivo. Isto é uma convenção; o pré-processador não requer isso. Não existe restrição sobre o que um arquivo de cabeçalho pode conter. Em particular, pode conter outras diretivas de pré-processamento que serão expandidas, por sua vez, pelo pré-processador. Apesar de arquivos de qualquer tipo poderem ser incluídos, é considerado um estilo de programação **pobre** incluir arquivos que contenham o código para a definição de funções.

Em sistemas UNIX, arquivos padrões de cabeçalho, tais como *stdio.h* são tipicamente encontrados no diretório /usr/include. Em sistemas (tipicamente Windows) usando o compilador Borland C, esses arquivos devem ser encontrados no diretório c:\bc\include ou em algum outro diretório. Em geral, a localização dos arquivos #inlcude padrões é dependente do sistema. Todos esses arquivos são legíveis, e programadores, por uma série de razões, necessitam lê-los em certas ocasiões.

Um dos usos primários de arquivos de cabeçalho é fornecer protótipos de funções. Por exemplo, o arquivo *stdio.h* contém as seguintes linhas:

```
int printf (const char *format, ...);
int scanf (const char *format, ...);
```

Estes são protótipos para as funções printf ( ) e scanf ( ) na biblioteca padrão. Genericamente falando, um protótipo de função diz ao compilador quais os tipos dos argumentos passados para a função e o tipo do valor retornado pela função. Antes de nós podermos entender os protótipos de função para pirntf ( ) e scanf ( ), nós precisamos aprender sobre o mecanismo de definição de função, ponteiros e qualificadores de tipo. Esses conceitos serão apresentados no futuro. O objetivo por ora é entender que quando um programador usa uma função de uma biblioteca padrão, então, o correspondente arquivo de cabeçalho deve ser incluído. O arquivo de cabeçalho irá fornecer os protótipos de função e outros elementos necessários. O compilador necessita dos protótipos de função para realizar seu trabalho corretamente.

A Biblioteca Padrão

A biblioteca padrão contém diversas funções úteis que adicionam considerável poder e flexibilidade a sistemas C. Muitas dessas funções são usadas extensivamente por todos os programadores C, enquanto outras são usadas mais seletivamente.

Programadores não estão preocupados com a localização no sistema da biblioteca padrão porque ela contém código compilado que é ilegível para humanos. A biblioteca padrão pode compreender mais de um arquivo. A biblioteca matemática, por exemplo, é conceitualmente parte da biblioteca padrão, mas ela freqüentemente se encontra em um arquivo separado. Seja qual for o caso, o sistema sabe onde encontrar o código que corresponde à função que o programador usou – printf ( ) e scanf ( ), por exemplo - a partir da biblioteca padrão. Note, contudo, que apesar de o sistema fornecer o código, **é responsabilidade do programador fornecer o protótipo das funções**. Isto é usualmente feito incluindo os arquivos de cabeçalho apropriados.

Cuidado. Não confundir arquivos de cabeçalho com as bibliotecas em si. A biblioteca padrão contém códigos-objeto das funções que já foram compiladas. Os arquivos de cabeçalho padrão não contém código compilado.