2. CONCEITOS BÁSICOS

2.1. Estrutura de um Programa

Um programa em C é constituído de:

- Um cabeçalho contendo as diretivas de compilador onde se definem o valor de constantes simbólicas, declaração de variáveis globais, inclusão de bibliotecas, declaração de rotinas, etc.;
- Um bloco de instruções principal e outros blocos de rotinas;
- Documentação do programa: comentários.

Em C, existem comandos que são processados durante a compilação do programa. Estes comandos são genericamente chamados de diretivas de compilação. Estes comandos, que fazem parte do cabeçalho, informam ao compilador do C basicamente quais são as constantes simbólicas usadas no programa e quais bibliotecas devem ser anexadas ao programa executável. A diretiva #include diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente estes arquivos contêm bibliotecas de funções ou rotinas do usuário. A diretiva #define diz ao compilador quais são as constantes simbólicas usadas no programa [ADA 97]. Detalhes sobre estas diretivas serão vistos posteriormente.

Um bloco de instruções é delimitado por { e }, equivalente ao *begin* e *end* de Pascal. Após cada comando dentro de um bloco de instruções deve-se c colocar um ; (ponto-e-vírgula). Além disso, um programa em C deve ser identado para que possa ser lido com mais facilidade. Blocos de rotinas são as funções que compõem um programa, que, como mencionado anteriormente, começa a ser executado pela função *main*.

Em C, os comentários podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo. Para que o comentário seja identificado como tal, ele deve ter um "/*" antes e um "*/" depois. Por exemplo:

```
/* Este programa não faz nada */
main() { }
```

Compiladores C/C++ também permitem que comentários sejam escritos de outra forma: colocando um // em uma linha, o compilador entenderá que tudo que estiver a direita do símbolo é um comentário. Por exemplo:

```
// Este programa não faz nada
main() { }
```

A seguir pode-se observar um exemplo completo de um programa em C [COP 96]:

```
/* Este programa realiza um somatório e
   exibe o resultado na tela do computador */

# include <stdio.h>      // Diretiva de compilação
# include <conio.h>      // Diretiva de compilação

void Total (int);        // Protótipo da função "Total"
void Exibe(void);        // Protótipo da função "Exibe"

int soma;                // Variável global
```

```
void Total (int x)          // Função "Total" tem um parâmetro
{
        soma = x + soma;
}

void Exibe()                // Função "Exibe" não tem parâmetros
{
        int cont;           // Declaração de variável local

        for (cont=0; cont<10; cont++)                   // Laço de 1 até 10
                printf(".");                            // Exibe um ponto na tela
        printf("O somatorio corrente eh: %d\n", soma);  // Exibe uma mensagem na tela
}

void main()                         // Função principal
{
        int cont;                   // Declaração de variável local

        clrscr();                   // Limpa a tela
        soma = 0;                   // Inicializa a variável com 0
        for (cont=0; cont<10; cont++)   // Laço de 1 até 10
        {
                Total(cont);        // Chama a função "Total"
                Exibe();            // Chama a função "Exibe"
        }
        getch();                    // Espera que o usuário pressione uma tecla
}
```

2.2. Identificadores

Em C, os nomes de variáveis, funções, rótulos e vários outros objetos definidos pelo usuário são chamados de **identificadores**. Esses identificadores podem variar de um a diversos caracteres, entretanto para a escolha dos nomes é necessário seguir algumas regras:

- Um identificador deve iniciar por uma letra ou por um "_" (*underscore*);
- A partir do segundo caracter pode conter letras, números e *underscore*;
- Deve-se usar nomes significativos dentro do contexto do programa;
- Lembrar que C é uma linguagem *case-sensitive* (nomes com letras maiúsculas ≠ nomes com letras);
- Costuma-se usar maiúsculas e minúsculas para separar palavras (por exemplo, "PesoDoCarro");
- Deve ser diferente dos comandos da linguagem;
- O padrão C ANSI determina que identificadores podem ter qualquer tamanho, mas se o identificador for um nome externo (nomes de funções e variáveis globais compartilhadas entre arquivos), pelo menos os primeiros 6 caracteres devem ser significativos, e se o identificador for um nome interno os primeiros 31 caracteres serão significativos;
- Pode conter números a partir do segundo caracter [PIN 99, SCH 96].

Exemplos de identificadores:

Correto	Incorreto
cont	1cont
Teste23	hi!there
usuario_1	usuario...1
RaioDoCirculo	Raio Do Circulo

Torna-se importante salientar que existem certos nomes que não podem ser usados como identificadores. São as chamadas "palavras reservadas", que são de uso restrito da linguagem C (comandos, estruturas, declarações, etc.). O conjunto de palavras reservadas do C padrão é o seguinte:

auto	*double*	*if*	*static*
break	*else*	*int*	*struct*
case	*entry*	*long*	*switch*
char	*extern*	*register*	*typedef*
continue	*float*	*return*	*union*
default	*for*	*sizeof*	*unsigned*
do	*goto*	*short*	*while*

2.3. Tipos de Dados Básicos

Todas as variáveis em C possuem um tipo, que define os valores que podem ser armazenados. Há cinco tipos básicos de dados em C: caractere, inteiro, ponto flutuante, ponto flutuante de precisão dupla e sem valor (*char*, *int*, *float*, *double* e *void*, respectivamente). Os outros tipos de dados são baseados em um desses tipos. O tamanho e a faixa desses tipos de dados variam de acordo com o processador e com a implementação do compilador C. Um caractere ocupa geralmente um *byte* e um inteiro tem normalmente 2 *bytes*, mas não é possível fazer esta suposição se for desejável que o programa seja portável. O padrão ANSI estipula apenas a faixa mínima de cada tipo de dado, não o seu tamanho em *bytes*.

Exceto o *void*, os tipos de dados básicos podem ter vários modificadores precedendo-os. Um modificador é usado para alterar o significado de um tipo básico para adaptá-lo mais precisamente às necessidades de diversas situações. A lista de modificadores é a seguinte: *signed*, *unsigned*, *long* e *short*. Todos estes podem ser aplicados aos tipos básicos *char* e *int*, e o *long* também pode ser aplicado a *double*.

O padrão ANSI elimina o *long float* porque é igual a *double*. O uso de *signed* com *int* é permitido mas redundante, pois a declaração de um *int* assume um número com sinal. O uso mais importante de *signed* é para modificar o *char*. A próxima tabela apresenta todos os tipos de dados definidos no padrão ANSI [SCH 96].

Tipo	Extensão (em *bits*)	Faixa Mínima		
void	0	Sem valor		
char	8	-127	a	127
unsigned char	8	0	a	255
signed char	8	-127	a	127
int	16	-32.767	a	32.767
unsigned int	16	0	a	65.535
signed int	16	-32.767	a	32.767
short int	16	-32.767	a	32.767
unsigned short int	16	0	a	65.535
signed short int	16	-32.767	a	32.767
long	32	-2.147.483.647	a	2.147.483.647
long int	32	-2.147.483.647	a	2.147.483.647
unsigned long int	32	0	a	4.294.967.295
signed long int	32	-2.147.483.647	a	2.147.483.647
float	32	-3.4×10^{38}	a	$3.4 \times 10^{+38}$
double	64	-1.7×10^{308}	a	$1.7 \times 10^{+308}$
long double	80	-3.4×10^{4932}	a	$1.1 \times 10^{+4932}$

2.4. Declaração de Variáveis e Constantes

Uma variável é uma posição de memória que pode ser identificada através de um nome, e é usada para guardar um valor. O conteúdo de uma variável pode ser alterado através de um comando de atribuição, ou seja, após uma atribuição a variável muda de valor. É interessante comentar que não há uma inicialização implícita na declaração, mas a variável pode ser inicializada na declaração

Todas as variáveis em C devem ser declaradas antes de serem usadas. A forma geral de uma declaração é:
tipo lista_de_variáveis;
Aqui tipo deve ser um tipo de dado válido em C e lista de variáveis pode consistir em um ou mais nomes de identificadores separados por vírgula. A declaração de variáveis pode ser feita em três lugares básicos: dentro de funções (variáveis locais), na definição dos parâmetros das funções (parâmetros formais) e fora de todas as funções (variáveis globais) [SCH 96].

Exemplos de declaração e atribuições de valores a variáveis:

```
# include <stdio.h>
void main()
{
   int a, b;                  // Declaração das variáveis "a" e "b"
   int SomaGeral;             // Declaração e inicialização da variável "SomaGeral"
   a = 3;                     // Atribuição do valor 3 à variável "a"
   b = a * 2;                 // Atribuição do dobro do valor da variável "a" para a variável "b"
   SomaGeral = a + b;         // "SomaGeral" recebe o resultado da soma de "a" e "b"
   :
}
```

Constantes são identificadores que não podem ter seus valores alterados durante a execução do programa. Constantes em C podem ser de qualquer um dos cinco tipos de dados básicos. Entretanto, a maneira como cada constante é representada depende do seu tipo. Constantes de caracteres são envolvidas por aspas simples (exemplo: 'a'). Constantes inteiras são especificadas como números sem componentes fracionários (exemplo: 10). Constantes em ponto flutuante requerem o ponto decimal seguido pela parte fracionária do número (exemplo: 11.98).

Para criar uma constante existe o comando #define que, em geral é colocado no início do programa-fonte, ou seja, no seu cabeçalho [PIN 99, SCH 96]. Exemplos:

```
#define LARGURA_MAXIMA          50
#define NRO_DE_DIAS_DA_SEMANA   7
#define NRO_DE_HORAS_DO_DIA     24
#define FALSO                   0
#define VERDADEIRO              1
#define UNIVERSIDADE            "Cambridge University"
#define TERMINO                 '/'
#define VALOR_DE_PI             3.1415  // Obs: não se coloca ponto-e-vírgula após o valor

void main ()
{
        int TotalDeHoras;
        TotalDeHoras = 10 * NRO_DE_DIAS_DA_SEMANA * NRO_DE_HORAS_DO_DIA;
        printf("Local: %s", UNIVERSIDADE);
        ......
}
```

2.5. Escopo de Variáveis

Variáveis *locais* são aquelas declaradas dentro de uma função. Em algumas literaturas de C, variáveis locais são referidas como variáveis "automáticas", porque em C pode-se usar a palavra reservada *auto* para declará-las. Variáveis locais só podem ser referenciadas por comandos que estão dentro do bloco no qual as variáveis foram declaradas, ou seja, elas não são reconhecidas fora de seu próprio bloco de código, que começa com { e termina com }. Exemplo:

```
:
void func1(void)
{
        int x;          // Variável local
        x = 10;
}

void func2(void)
{
        int x;          // Variável local (não tem relação com a anterior)
        x = -199;
}
:
```

A maioria dos programadores declara todas as variáveis usadas por uma função imediatamente após o abre-chaves e antes de qualquer outro comando. Porém, as variáveis locais podem ser declaradas dentro de qualquer bloco de código. Por exemplo:

```
void f(void)
{
        int t;                          // Variável local
        scanf("%d", &t);                // Valor digitado é atribuído a t

        if (t == 1)
        {                               // início do bloco if
                char s[80];             // s é criada apenas na entrada deste bloco
                printf("Digite o nome: ");
                gets(s);
                ....
        }                               // fim do bloco if, s é "destruída"
}
```

A principal vantagem em declarar uma variável local dentro de um bloco condicional, é que a memória para ela só será alocada se necessário, pois elas não existirão até que o bloco em que são declaradas seja iniciado.

É importante lembrar que deve-se declarar todas as variáveis locais no início do bloco em que são definidas, antes de qualquer comando do programa. Por exemplo, a função a seguir está tecnicamente incorreta e não será compilada na maioria dos compiladores.

```
/* Esta função está errada */
void f(void)
{
        int i;
        i = 10;

        int j;      // Esta linha irá provocar um erro
        j = 20;
}
```

Porém, se "*j*" fosse declarada dentro de seu próprio bloco de código ou antes do comando "*i=10;*", a função teria sido aceita. Por exemplo, as próximas duas versões do código anterior estão sintaticamente corretas [SCH 96]:

```
/* Define j dentro de seu próprio bloco de código */
void f(void)
{
        int i;
        i = 10;
        {
                int j;
                j = 20;
        }
}

/* Define j no início do bloco da função */
void f(void)
{
        int i;
        int j;
        i = 10;
        j = 20;
}
```

Variáveis *globais* são aquelas reconhecidas pelo programa inteiro, isto é, que podem ser usadas por qualquer pedaço de código e que guardam seus valores durante toda a execução do programa. Para criar variáveis globais basta declará-las fora de qualquer função.

No programa seguinte, a variável "*cont*" foi declarada fora de todas as funções. Embora sua declaração ocorra antes da função *main*, ela poderia ter sido colocada em qualquer lugar anterior ao seu primeiro uso, desde que não estivesse em uma função. No entanto, é melhor declarar variáveis globais no início do programa.

```
#include <stdio.h>

int cont;          // Variável global

void func1(void);
void func2(void);

void main (void) {
        cont = 100;
        func1();
}

void func1(void) {
        int temp;
        temp = cont;
        func2();
        printf("cont eh %d", cont);          // Exibirá 100
}

void func2(void) {
        int cont;
        for (cont=1; cont<10; cont++)
                putchar('.');
}
```

O armazenamento de variáveis globais encontra-se em uma região fixa da memória, separada para esse propósito pelo compilador C. Variáveis globais são úteis quando o mesmo dado é usado em muitas funções, no entanto deve-se evitar ao máximo o uso de variáveis globais desnecessárias. Elas ocupam memória durante todo tempo em que o programa está executando, deixam as funções menos genéricas e, além disso, podem levar a erros no programa por causa de desconhecidos e indesejáveis efeitos colaterais [SCH 96].

2.6. Classes de Variáveis

A inclusão do especificador ***extern*** nas declarações de variáveis diz ao compilador que os tipos e nomes de variável que o seguem foram declarados em outro arquivo. Em outras palavras, *extern* deixa o compilador saber o tipo e o nome das variáveis globais sem realmente criar armazenamento para ela novamente. Quando o *linkeditor* unir os módulos, todas as referências a variáveis externas são resolvidas.

Uma vez que C permite que módulos de um programa grande sejam compilados separadamente para então serem *linkeditados* juntos, uma forma de aumentar a velocidade de compilação e ajudar no gerenciamento de grandes projetos, utiliza-se, então, o especificador *extern* para notificar todos os arquivos sobre as variáveis globais solicitadas pelo sistema. Uma solução muito utilizada para o problema das variáveis globais quando se está trabalhando com vários arquivos diferentes é declarar todas as suas variáveis globais em um arquivo e usar declarações *extern* nos outros, como mostra o código a seguir:

```
// Arquivo 1

int x, y;
char ch;

main (void) {
:
}

func1() {
   x = 123;
}
```

```
// Arquivo 2

extern int x, y;
extern char ch;

func22(void) {
   x = y/10;
}

func23() {
   y = 10;
}
```

Variáveis ***static*** são variáveis permanentes dentro de sua própria função ou arquivo. Ao contrário das variáveis globais, elas não são reconhecidas fora de sua função ou arquivo, mas mantêm seus valores entre chamadas. Essa característica é útil para funções genéricas e funções de biblioteca que podem ser usadas por outros programadores.

O especificador *static* tem efeitos diferentes em variáveis locais e em variáveis globais. Quando aplicado a uma variável local, o compilador cria armazenamento permanente para ela quase da mesma forma como cria armazenamento para uma variável global. A diferença fundamental é que a variável local *static* é reconhecida apenas no bloco em que está declarada. Resumindo, uma variável local *static* é uma variável local que retém seu valor entre chamadas da função. Um exemplo de função que requer uma variável local *static* é um gerador de série de números que produz um novo número baseado no anterior. Nesse caso, usar uma variável global tornaria essa função difícil de ser colocada em uma biblioteca de funções. A melhor solução é declarar a variável que retém o número gerado como *static*, como mostra o exemplo a seguir:

```
// Exemplo de declaração/uso de variáveis static locais
#include <stdio.h>
#include <conio.h>
void func1(void);
int series(void);

void main (void) {
        clrscr();
        printf("Numero de serie: %d \n", series());
        func1();
        printf("Numero de serie: %d \n", series());
        getch();
}
```

```
void func1(void) {
        printf("Teste... \n");
}

int series(void) {
        static int series_num = 100;      // valor de inicialização da variável (atribuído apenas uma vez)
        series_num = series_num + 23;
        return(series_num);
}
```

Por outro lado, aplicar o especificador *static* a uma variável global informa ao compilador para criar uma variável global que é reconhecida apenas no arquivo no qual a mesma foi declarada. Isso significa que embora a variável seja global, funções em outros arquivos não podem reconhecê-la ou alterar seu conteúdo diretamente; assim, a variável não fica sujeita a efeitos colaterais. Então, para as poucas situações onde uma variável local *static* não possa fazer o trabalho, pode-se criar um pequeno arquivo que contenha apenas as funções que precisam da variável global *static*, e compilar separadamente esse arquivo sem medo de efeitos colaterais.

Para ilustrar, uma variável global *static*, o exemplo anterior for alterado de forma que um valor somente inicialize a série por meio de uma chamada a uma segunda função denominada "*series_start()*". Neste caso, o arquivo ficaria da seguinte maneira:

```
// Exemplo de declaração/uso de variáveis static globais
// Todo este código deve ficar em um único arquivo, preferencialmente isolado

#include <stdio.h>
#include <conio.h>

static int series_num;

void series_start(int seed);
int series(void);

void series_start (int seed)    // Inicializa a variável static series_num
{
      series_num = seed;
}

int series(void)              // Cria novo número de série
{
   series_num = series_num + 23;
   return(series_num);
}
```

Declarar uma variável com o especificador ***register*** significa, segundo determinação do padrão C ANSI, que "o acesso a ela é o mais rápido possível". Na prática, caracteres e inteiros são armazenados nos registradores da CPU ao invés da memória, onde as variáveis normais são armazenadas. Isso faz com que as operações ocorram mais rapidamente, pois como o valor dessas variáveis é conservado na CPU, não é necessário acesso à memória para determinar ou modificar seus valores. Objetos maiores, tais como matrizes, obviamente não podem ser armazenados em um registrador, mas podem receber um tratamento diferenciado, que depende da implementação do compilador C e de seu ambiente operacional.

O especificador *register* só pode ser aplicado a variáveis locais e a parâmetros formais em uma função. Assim, variáveis globais *register* não são permitidas. O próximo trecho de código exemplifica

como declarar uma variável *register* do tipo inteira e usá-la para controlar um laço. Essa função calcula o resultado de M^e para inteiros.

```
// Exemplo declaração/uso de variáveis register

#include <stdio.h>
#include <conio.h>

int_pwr(register int, register int);

void main (void)
{
        int X;
        clrscr();
        X = int_pwr(3, 2);
        printf("X = %d", X);
        getch();
}

int_pwr(register int m, register int e) // "m" e "e" declaradas como register porque são usadas num laço
{
        register int temp;
        temp = 1;
        for (; e; e--)
                temp = temp * m;
        return temp;
}
```

O número de variáveis em registradores dentro de qualquer bloco de código é determinado pelo ambiente e pela implementação específica de C. Não deve-se declarar muitas variáveis *register* porque o compilador C automaticamente transforma estas variáveis em variáveis comuns quando o limite for alcançado (isso é feito para assegurar a portabilidade do código em C por meio de uma ampla linha de processadores). Normalmente, pelo menos duas variáveis *register* do tipo *char* ou *int* podem de fato ser colocadas em registradores da CPU.

Obs.: variáveis do tipo *const* recebem um valor inicial, mas não podem ser modificadas pelo programa. Por exemplo, *const int a=10;*, cria uma variável inteira com um valor inicial 10, que o programa não pode modificar [SCH 96].

2.7. Operadores

O que é um operador é algo trivial. Por exemplo, o sinal de mais (+) é um operador que adiciona dois valores, e o sinal de menos (-) é um operador que subtrai dois valores. Estes e outros operadores semelhantes são denominados operadores binários, porque operam sobre dois argumentos. Alguns outros, tais como o operador de negação (!), são unários, pois eles requerem apenas um único argumento [SWA 93].

C é uma das linguagens com maior número de operadores, pois além de possuir todos os operadores comuns de uma linguagem de alto nível, também possui os operadores mais usuais a linguagens de baixo nível. Este operadores são agrupados em quatro classes em C (descritas a seguir): aritméticos, relacionais, lógicos e *bit* a *bit*. Além disso, ainda possui alguns operadores especiais para tarefas particulares.

A operação de **atribuição** é a operação mais simples do C, e consiste em atribuir o valor de uma expressão a uma variável. O operador de atribuição, =, pode ser usado dentro de qualquer expressão

válida de C, o que não acontece na maioria das linguagens, onde ele é tratado como um caso especial de comando.

Conversão de tipos refere-se à situação em que variáveis de um tipo são misturadas com variáveis de outro tipo. Em um comando de atribuição, a regra de conversão de tipos é muito simples: o valor do lado direito (lado da expressão) de uma atribuição é convertido no tipo do lado esquerdo (variável destino). Exemplos:

```
int x;
char ch;
float f;
ch = x;  // ch recebe os 8 bits menos significativos de x; se x está entre 256 e 0, então x=ch
x = f;   // x recebe a parte inteira de f
f = ch;  // f converte o valor inteiro(8 bits) armazenado em ch no mesmo valor em formato de ponto flutuante
f = x;   // f converte o valor inteiro(16 bits) armazenado em x no mesmo valor em formato de ponto flutuante
```

Quando se converte de inteiros para caracteres, inteiros longos para inteiros, etc., a regra básica é que a quantidade apropriada de bits significativos será ignorada. A conversão de um *int* em um *float* ou *float* em *double*, etc., não aumenta a precisão ou exatidão, apenas mudam a forma em que o valor é representado.

C também permite que seja atribuído o mesmo valor a muitas variáveis usando atribuições múltiplas em um único comando. Exemplo:

```
x = y = z = 0;
```

Torna-se importante destacar que é possível forçar uma expressão a ser de um determinado tipo usando um ***cast***. A forma genérica de um *cast* é:

```
<tipo> <expressão>
```

onde <tipo> é qualquer tipo de dados válido em C. Por exemplo, para garantir que a expressão "*x/2*" resulte em um valor do tipo *float*, sabendo-se que "*x*" é um inteiro, digita-se:

```
(float) x/2;
```

Casts são operadores tecnicamente, e como operador, é unário e possui a mesma precedência que qualquer outro operador unário. Embora não sejam muito usados, eles podem ser muito úteis quando necessário. Por exemplo, suponha que seja necessário usar um inteiro para controlar uma repetição, mas as operações sobre o valor exigem uma parte fracionária, como mostra o programa seguinte:

```
#include <stdio.h>
void main(void) // imprime i e i/2 com frações
{
        int i;
        for (i=1; i<=100; ++i)
                printf("%d / 2  eh: %f \n", i, (float) i/2);
}
```

Os operadores **aritméticos**, que funcionam em C da mesma forma em que na maioria das outras linguagens, são:

Operador	Ação	Precedência
- - , ++	Decremento e Incremento	Maior
-	Menos unário	↓
* , /	Multiplicação e Divisão	↓
%	Módulo (resto de divisão inteira)	↓
- , +	Subtração e Adição	Menor

Os operadores aritméticos podem ser aplicados em quase qualquer tipo de dado interno permitido em C. Deve-se apenas atentar para o fato de que quando "/" é aplicado a um inteiro ou caractere, qualquer resto é truncado, e "%" não pode ser usado nos tipos de ponto flutuante. Exemplos:

```
:
int x, y;
x = 5;
y = 2;
printf("%d", x/y);          // mostrará 2
printf("%d", x%y);          // mostrará 1, o resto da divisão inteira
x = 1;
y = 2;
printf("%d  %d", x/y, x%y);         // mostrará 0  1
:
```

Dois operadores úteis, geralmente não encontrados em outras linguagens, são os de incremento (++) e decremento (--). O primeiro soma ao seu operando e o segundo subtrai. Por exemplo: "*x = x+1;*" é o mesmo que "*++x;*" ou "*x++;*", assim como "*x = x-1;*" é o mesmo que "*--x;*" ou "*x--;*". Apesar dos operadores de incremento e decremento poderem ser utilizados como prefixo ou sufixo do operando, há uma diferença quando são usados em expressões. Quando um operador de incremento ou decremento precede seu operando, C executa a operação de incremento ou decremento antes de usar o valor do operando. Se o operador estiver após seu operando, C usará o valor do operando antes de incrementá-lo ou decrementá-lo. Por exemplo:

```
x = 10;
y = ++x;          // y recebe 11, x recebe 11
x = 10;
y = x++;          // y recebe 10, x recebe 11
:
int a, b, c, i = 3;          // a: ?   b: ?   c: ?   i: 3
a = i++;                     // a: 3   b: ?   c: ?   i: 4
b = ++i;                     // a: 3   b: 5   c: ?   i: 5
c = --i;                     // a: 3   b: 5   c: 4   i: 4
:
```

No termo operador **relacional**, relacional refere-se às relações que os valores podem ter uns com os outros. No termo operador **lógico**, lógico refere-se às maneiras como essas relações podem ser conectadas. Uma vez que os operadores lógicos e relacionais freqüentemente trabalham juntos, eles serão discutidos em conjunto.

A idéia de verdadeiro e falso é a base dos conceitos dos operadores lógicos e relacionais. Em C, verdadeiro é qualquer valor diferente de zero. Falso é zero. As expressões que usam operadores relacionais ou lógicos devolvem zero para falso e um para verdadeiro. Os operadores lógicos e relacionais são:

Operador	Ação	Precedência
!	Negação (ou NOT)	Maior
>, >=, <, <=	Maior que, Maior ou igual que, Menor que, Menor ou igual que	↓
==, !=	Igual, Diferente	↓
&&	Condição "E" (ou AND)	↓
||	Condição "OU" (ou OR)	Menor

A tabela verdade dos operadores lógicos é mostrada a seguir:

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

C permite que sejam combinadas diversas operações em uma expressão, como por exemplo:

```
10>5 && !(10<9) || 3 <=4
```

Neste caso, o resultado é verdadeiro.

Ao contrário de muitas outras linguagens, C suporta um completo conjunto de operadores ***bit a bit***. Uma vez que C foi projetada para substituir a linguagem *assembly* na maioria das tarefas de programação, era importante que ela tivesse a habilidade de suportar muitas das operações que poderiam ser feitas em tal linguagem. Operação *bit* a *bit* refere-se a testar, atribuir ou deslocar os bits efetivos em um *byte* ou uma palavra, que correspondem aos tipos de dados *char* e *int* e variantes do padrão C. Tais operações não podem ser usadas em *float*, *double*, *void* ou outros tipos mais complexos. Os operadores *bit* a *bit* são:

Operador	Ação
&	AND
|	OR
^	OR exclusivo (XOR)
~	Complemento de um (NOT)
>>	Deslocamento à esquerda
<<	Deslocamento à direita

As operações *bit* a *bit* AND, OR e NOT são governadas pela mesma tabela verdade de seus equivalentes lógicos, exceto por trabalharem *bit* a *bit*. O OR exclusivo (^) tem a seguinte tabela verdade:

p	q	p^q
0	0	0
1	0	1
1	1	0
0	1	1

Como a tabela indica, o resultado de um XOR é verdadeiro apenas se exatamente um dos operandos for verdadeiro; caso contrário, será falso.

Operadores *bit* a *bit* encontram aplicações mais freqüentemente em *drivers* de dispositivos – como em programas de modems, rotinas de arquivos em disco e rotinas de impressora – porque as operações *bit* a *bit* mascaram certos *bits*, como o *bit* de paridade. O *bit* de paridade confirma se o restante dos *bits* em um *byte* não se modificaram. É geralmente o *bit* mais significativo em cada *byte*.

Exemplos:

```
:
// Uma função simples de criptografia
char encode(char ch)
{
        return(~ch);      // complementa
}
// para decriptografar, basta fazer um novo complemento
:
:
// Programa que exemplifica o deslocamento de bits
#include <stdio.h>

void main (void)
{
        unsigned int i;
        int j;

        i=1;

        // Deslocamentos à esquerda
        for (j=0; j<4; j++) {
                i = i << 1; // desloca i de 1 à esquerda, que é o mesmo que multiplicar por 2
                printf("Deslocamento à esquerda %d: %d \n", j, i);
        }
```

```
        // Deslocamentos à direita
        for (j=0; j<4; j++) {
                i = i >> 1; // desloca i de 1 à direita, que é o mesmo que dividir por 2
                printf("Deslocamento à direita %d: %d \n", j, i);
        }
}
```

Existem dois operadores que são usados para manipular **ponteiros**: & e *. Como se sabe, um ponteiro é um endereço de memória de uma variável. Uma variável de ponteiro é uma variável especialmente declarada para guardar um ponteiro para seu tipo especificado. Em C ponteiros têm três funções principais. Eles podem fornecer uma maneira rápida de referenciar elementos de uma matriz, permitem que as funções em C modifiquem seus parâmetros de chamada, e suportam listas encadeadas e outras estruturas dinâmicas de dados.

O primeiro operador de ponteiro é &. Ele é um operador unário que devolve o endereço na memória de seu operando. Por exemplo:

```
m = &count;
```

põe o endereço de memória da variável "*count*" em "*m*". Esse endereço é a posição interna da variável no computador, e não tem nenhuma relação com o valor de "*count*". Pode-se imaginar & como significando "o endereço de". Desta forma, a sentença de atribuição anterior significa "*m* recebe o endereço de *count*".

O segundo operador é *, que é o complemento de &. O * é um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se "*m*" contém o endereço da variável "*count*",

```
q = *m;
```

coloca o valor de "*count*" em "*q*". Pense no * como significando "no endereço de". Neste caso, a sentença poderia ser lida como "*q* recebe o valor do endereço de *m*".

No próximo exemplo os seguintes operadores * e & colocam o valor 10 na variável chamada "*target*". Como esperado, esse programa mostra o valor 10 na tela.

```
#include <stdio.h>
void main (void)
{
        int target, source;
        int *m;
        source = 10;
        m = &source;
        target = *m;
        printf("%d", target);
}
```

2.8. Precedência de Operadores

Em C operadores de mesmo nível de precedência são avaliados da esquerda para a direita. Obviamente, parênteses podem ser usados para alterar a ordem de avaliação. A próxima tabela mostra a lista de precedência de todos os operadores de C [SCH 96].

Precedência	Operador
Maior	() [] ->
↓	! ~ ++ -- - (tipo) * & sizeof
↓	* / %
↓	+ -
↓	<< >>
↓	< <= > >=
↓	== !=
↓	&
↓	^
↓	|
↓	&&
↓	||
↓	?:
↓	= += -= *= /= etc.
Menor	,