Faculdade de Ciências Aplicadas de Cascavel – FACIAP

Curso de Ciência da Computação

LINGUAGEM C/C++

CASCAVEL - PR
2004

SUMÁRIO

UNIDADE 1 – CONSIDERAÇÕES INICIAIS SOBRE A LINGUAGEM C 1

1.1 AS ORIGENS DO C/C++ 1
1.2 COMPILADORES E INTERPRETADORES 2
1.3 WARNINGS E ERROS 3
1.4 A LINGUAGEM C/C++ É CASE-SENSITIVE 3
1.5 FUNÇÕES EM C 3
1.5.1 FUNÇÕES COM ARGUMENTOS 4
1.5.2 FUNÇÕES QUE RETORNAM VALORES 5
1.5.3 A FORMA GERAL DE UMA FUNÇÃO 6
1.6 PALAVRAS RESERVADAS DA LINGUAGEM C 6

UNIDADE 2 – TIPOS DE DADOS, CONSTANTES, OPERADORES E EXPRESSÕES 7

2.1 NOMES DE IDENTIFICADORES 7
2.2 TIPOS DE DADOS 7
2.2.1 MODIFICADORES DE TIPOS 8
2.3 DEFINIÇÃO DE VARIÁVEIS 9
2.4 DEFINIÇÃO DE CONSTANTES 9
2.4.1 CONSTANTES HEXADECIMAIS E OCTAIS 10
2.4.2 CONSTANTES STRING 11
2.4.3 CÓDIGOS DE BARRA INVERTIDA 11
2.5 OPERADORES 11
2.5.1 OPERADOR DE ATRIBUIÇÃO 12
2.5.2 OPERADORES ARITMÉTICOS 12
2.5.3 OPERADORES RELACIONAIS 12
2.5.4 OPERADORES LÓGICOS 12
2.5.5 MANIPULAÇÃO DE BITS 13
2.5.6 OPERADORES DE ASSINALAMENTO 13
2.5.7 OPERADORES DE PRÉ E PÓS-INCREMENTO 14
2.5.8 OPERADORES DE ENDEREÇO 14
2.6 TABELA DE OPERADORES DO C 14
2.7 EXPRESSÕES 15
2.7.1 CONVERSÕES DE TIPOS EM EXPRESSÕES 15
2.7.2 MODELADORES (CASTS) 16
2.7.3 ESPAÇAMENTO E PARÊNTESES 16
2.8 ESQUELETO DE UM PROGRAMA EM C 17

UNIDADE 3 – ESTRUTURAS BÁSICAS DE PROGRAMAÇÃO (FLUXO DE CONTROLE) 18

3.1 COMANDOS E BLOCOS 18
3.2 PRINTF( ) 18
3.3 SCANF( ) 20
3.4 A DECLARAÇÃO if 23
3.4.1 USANDO A DECLARAÇÃO else 24
3.4.2 O ENCADEAMENTO if-else-if 25
3.4.3 A EXPRESSÃO CONDICIONAL 26
3.4.4 ifs ANINHADOS 27
3.5 A DECLARAÇÃO switch 28
3.5.1 A DECLARAÇÃO break 29
3.5.2 DECLARAÇÕES switch ANINHADAS 30
3.6 LAÇOS DE REPETIÇÃO 33
3.7 O LAÇO for 33
3.7.1 VARIÁVEIS DO LAÇO for 35
3.7.2 O LAÇO INFINITO 37

3.7.3 A INTERRUPÇÃO DE UM LAÇO for 37
3.7.4 USANDO LAÇOS for SEM CONTEÚDO 38
3.8 O LAÇO while 38
3.9 O LAÇO do-while 40
3.10 LAÇOS ANINHADOS 41
3.11 QUEBRANDO UM LAÇO 43
3.12 A DECLARAÇÃO CONTINUE 44
3.13 RÓTULOS E goto 45

UNIDADE 4 – VETORES, MATRIZES E STRINGS 47

4.1 INTRODUÇÃO 47
4.2 DECLARAÇÃO DE VETORES UNIDIMENSIONAIS 47
4.3 CADEIAS DE CARACTERES (STRINGS) 49
4.4 DECLARAÇÃO DE VETORES MULTIDIMENSIONAIS (MATRIZES) 51
4.5 VETORES DE CADEIAS DE CARACTERES 52
4.6 INICIALIZAÇÃO DE VETORES 53
4.7 LIMITES DE VETORES E SUA REPRESENTAÇÃO EM MEMÓRIA 54

UNIDADE 5 – APONTADORES (PONTEIROS) 56

5.1 PONTEIROS SÃO ENDEREÇOS 56
5.2 VARIÁVEIS PONTEIRO 57
5.3 OS OPERADORES DE PONTEIROS 57
5.4 IMPORTÂNCIA DO TIPO BASE 59
5.5 EXPRESSÕES COM PONTEIROS 59
5.5.1 ATRIBUIÇÃO DE PONTEIROS 59
5.5.2 ARITMÉTICA COM PONTEIROS 60
5.5.3 COMPARAÇÃO COM PONTEIROS 61
5.6 PONTEIROS E MATRIZES 61
5.6.1 INDEXANDO UM PONTEIRO 63
5.6.2 PONTEIROS E STRINGS 63
5.6.3 OBTENDO O ENDEREÇO DE UM ELEMENTO DA MATRIZ 64
5.6.4 MATRIZES DE PONTEIROS 64
5.6.5 PROGRAMA EXEMPLO: TRADUTOR INGLÊS-PORTUGUÊS 65
5.7 PONTEIROS PARA PONTEIROS 69
5.8 INICIALIZANDO PONTEIROS 70
5.9 PROBLEMAS COM PONTEIROS 72

UNIDADE 6 – TIPOS DE DADOS DEFINIDOS PELO USUÁRIO 74

6.1 ESTRUTURAS 74
6.1.1 REFERENCIANDO OS CAMPOS DA ESTRUTURA 75
6.1.2 MATRIZES DE ESTRUTURAS 75
6.1.3 ATRIBUINDO ESTRUTURAS 76
6.1.4 PASSANDO ESTRUTURAS PARA FUNÇÕES 76
6.1.5 PONTEIROS PARA ESTRUTURAS 78
6.1.6 MATRIZES E ESTRUTURAS DENTRO DE ESTRUTURAS 81
6.2 UNIÕES 82
6.3 ENUMERAÇÕES 83
6.4 typedef 86

UNIDADE 7 – ESTUDO DETALHADO DE FUNÇÕES 87

7.1 FORMA GERAL DE UMA FUNÇÃO 87
7.2 A DECLARAÇÃO return 87
7.2.1 RETORNANDO DE UMA FUNÇÃO 88
7.2.2 RETORNANDO UM VALOR 88
7.3 FUNÇÕES RETORNANDO VALORES NÃO-INTEIROS 90
7.3.1 FUNÇÕES QUE RETORNAM PONTEIROS 92

7.3.2 FUNÇÕES DO TIPO void ___ 93
7.4 REGRAS DE ESCOPO DE FUNÇÕES ___ 93
7.4.1 VARIÁVEIS LOCAIS ___ 94
7.4.2 PARÂMETROS FORMAIS ___ 96
7.4.3 VARIÁVEIS GLOBAIS ___ 96
7.5 ARGUMENTOS E PARÂMETROS DE FUNÇÕES ___ 98
7.5.1 CHAMADA POR VALOR, CHAMADA POR REFERÊNCIA ___ 98
7.5.2 CRIANDO UMA CHAMADA POR REFERÊNCIA ___ 99
7.5.3 CHAMANDO FUNÇÕES COM MATRIZES ___ 100
7.5.4 OS ARGUMENTOS argc, argv PARA A FUNÇÃO main( ): ___ 103
7. 6 FUNÇÕES RECURSIVAS ___ 104

UNIDADE 8 – ARQUIVOS ___ 107

8.1 INTRODUÇÃO ___ 107
8.2 O PONTEIRO DE ARQUIVO ___ 107
8.3 ABRINDO UM ARQUIVO ___ 108
8.4 ESCREVENDO UM CARACTERE NUM ARQUIVO ___ 110
8.5 LENDO UM CARACTERE DE UM ARQUIVO ___ 110
8.6 USANDO A FUNÇÃO feof( ) ___ 110
8.7 FECHANDO UM ARQUIVO ___ 111
8.8 ferror() E rewind() ___ 111
8.9 USANDO fopen( ), getc( ), putc( ) E fclose( ) ___ 112
8.10 fgets( ) E fputs( ) ___ 113
8.11 fread( ) E fwrite( ): ___ 113
8.12 ACESSO RANDÔMICO A ARQUIVOS: fseek( ) ___ 116
8.13 fprintf( ) E fscanf( ) ___ 116
8.14 APAGANDO ARQUIVOS: remove( ) ___ 118
8.15 PROGRAMA EXEMPLO: MANIPULAÇÃO DE ARQUIVOS ___ 119

UNIDADE 9 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++ ___ 126

9.1 INTRODUÇÃO ___ 126
9.1.1 OBJETOS ___ 126
9.1.2 POLIMORFISMO ___ 127
9.1.3 HERANÇA ___ 127
9.2 CONSIDERAÇÕES FUNDAMENTAIS SOBRE O C++ ___ 127
9.3 COMPILANDO UM PROGRAMA C++ ___ 128
9.4 INTRODUÇÃO A CLASSES E OBJETOS ___ 128
9.5 SOBRECARGA DE FUNÇÕES ___ 132
9.6 SOBRECARGA DE OPERADOR ___ 134
9.7 HERANÇA ___ 134
9.8 CONSTRUTORES E DESTRUTORES ___ 138
9.9 FUNÇÕES FRIEND ___ 140
9.10 A PALAVRA RESERVADA this ___ 144
9.11 SOBRECARGA DE OPERADOR – MAIORES DETALHES ___ 145

UNIDADE 1 – CONSIDERAÇÕES INICIAIS SOBRE A LINGUAGEM C

1.1 AS ORIGENS DO C/C++

A história do C/C++ começa nos anos 70 com a invenção da linguagem C. A linguagem C foi inventada e implementada pela primeira vez por Dennis Ritchie em um DEC PDP-11, usando o sistema operacional UNIX. A linguagem C é o resultado do processo de desenvolvimento iniciado com outra linguagem, chamada BCPL, desenvolvida por Martin Richards. Esta linguagem influenciou a linguagem inventada por Ken Thompson, chamado B, a qual levou ao desenvolvimento da linguagem C.

A linguagem C tornou-se uma das linguagens de programação mais usadas. Flexível, ainda que poderosa, a linguagem C tem sido utilizada na criação de alguns dos mais importantes produtos de software dos últimos anos. Entretanto, a linguagem encontra seus limites quando o tamanho de um projeto ultrapassa um certo ponto. Ainda que este limite possa variar de projeto para projeto, quanto o tamanho de um programa se encontra entre 25.000 e 100.000 linhas, torna-se problemático o gerenciamento, tendo em vista que é difícil compreende-lo como um todo. Para resolver este problema, em 1980, enquanto trabalhava nos laboratórios da Bell, em Murray Bill, New Jersey, Bjarne Stroustrup acrescentou várias extensões à linguagem C e chamou inicialmente esta nova linguagem de "C com classes". Entretanto, em 1983, o nome foi mudado para C++.

Muitas adições foram feitas pós-Stroustrup para que a linguagem C pudesse suportar a ***programação orientada a objetos***, às vezes referenciada como ***POO***. Stroustrup declarou que algumas das características da orientação a objetos do C++ foram inspiradas por outra linguagem orientada a objetos chamada de Simula67.

A linguagem C é freqüentemente referenciada como uma linguagem de nível médio, posicionando-se entre o assembler (baixo nível) e o Pascal (alto nível)[1]. Uma das razões da invenção da linguagem C foi dar ao programador uma linguagem de alto nível que poderia ser utilizada como uma substituta para a linguagem assembly.

Como você provavelmente sabe, a linguagem assembly utiliza a representação simbólica das instruções executadas pelo computador. Existe um relacionamento de um para um entre cada instrução da linguagem assembly e a instrução de máquina. Ainda que este relacionamento possibilite escrever programas altamente eficientes, isso torna a programação um tanto quanto tediosa e passível de erro. Por outro lado, linguagens de alto nível, como Pascal, estão extremamente distantes da representação de máquina. Uma instrução em Pascal não possui essencialmente nenhum relacionamento com a seqüência de instruções de máquina que são executadas.

Entretanto, ainda que a linguagem C possua estruturas de controle de alto nível, como é encontrado na Pascal, ela também permite que o programador manipule bits, bytes e endereços de uma maneira mais proximamente ligada à máquina, ao contrário da abstração apresentadas por outras linguagens de alto nível. Por esse motivo, a linguagem C tem sido ocasionalmente chamada de "código assembly de alto nível". Por sua natureza dupla, a linguagem C permite que sejam criado programas rápidos e eficientes sem a necessidade de se recorrer a linguagem

---

[1] SCHILDT, H. **Turbo C++**: guia do usuário. São Paulo : Makron Books,1992.

assembly.

A filosofia que existe por trás da linguagem C é que o programador sabe realmente o que está fazendo. Por esse motivo, a linguagem C quase nunca "coloca-se no caminho" do programador, deixando-o livre para usar (ou abusar) dela de qualquer forma que queira. Existe uma pequena verificação de erro de execução ***runtime error***. Por exemplo, se por qualquer motivo você quiser sobrescrever a memória na qual o seu programa está atualmente residindo, o compilador nada fará para impedi-lo. O motivo para essa "liberdade na programação" é permitir ao compilador C criar códigos muito rápidos e eficientes, já que ele deixa a responsabilidade da verificação de erros para você. Em outras palavras, a linguagem C considera que você é hábil o bastante para adicionar suas próprias verificações de erro quando necessário.

Quando C++ foi inventado, Bjarne Stroustrup sabia que era importante manter o espírito original da linguagem C, incluindo a eficiência, a natureza de nível médio e a filosofia de que o programador, não a linguagem, está com as responsabilidades, enquanto, ao mesmo tempo, acrescentava o suporte à programação orientada a objetos. Assim, o C++ proporciona ao programador a liberdade e o controle da linguagem C junto com o poder dos objetos. As características da orientação a objetos em C++, usando as palavras de Stroustrup, "permite aos programas serem estruturados quanto à clareza e extensibilidade, tornando fácil a manutenção sem perda de eficiência".

1.2 COMPILADORES E INTERPRETADORES

Os termos ***compilador*** e ***interpretador*** referem-se à maneira pela qual um programa é executado. Na teoria, qualquer linguagem de programação pode ser tanto compilada como interpretada, mas algumas linguagem são usualmente executadas de uma forma ou de outra. Por exemplo, o BASIC geralmente é interpretado e o C++, compilado. A maneira como um programa é executado não é definida pela linguagem na qual ele é escrito. Tanto compiladores como interpretadores são simplesmente programas sofisticados que agem sobre o código-fonte do seu programa.

Um ***interpretador*** lê o código-fonte do seu programa uma linha por vez e executa uma instrução específica contida naquela linha. O interpretador deverá estar presente toda vez que o programa estiver sendo executado.

Um ***compilador*** lê o programa inteiro e converte-o em um código executável. O código executável também é referenciado como código objeto, binário ou de máquina. Entretanto, neste contexto, o termo ***objeto*** não tem nenhuma relação com os objetos definidos em um programa C++. Uma vez o programa compilado, uma linha de código-fonte está significativamente distante do código executável. O compilador não é necessário para executar o programa, desde que ele já esteja compilado.

Existem dois termos que você verá freqüentemente: ***tempo de compilação*** e ***tempo de execução***. O termo tempo de compilação refere-se aos eventos que acontecem durante o processo de compilação e tempo de execução, aos eventos que ocorrem enquanto o programa está sendo executado. Infelizmente, você verá constantemente esses termos relacionados a mensagens de erros, como em ***erros de tempo de compilação*** e ***erros de tempo de execução***. Mas, por sorte, após adquirir experiência na programação em C/C++, você verá essas mensagens raras vezes.

1.3 WARNINGS E ERROS

A linguagem C foi projetada para ser uma linguagem bastante tolerante e para permitir que qualquer coisa que sintaticamente esteja correta seja compilada. Entretanto, algumas declarações, ainda que sintaticamente corretas, são suspeitas. Quando o compilador C/C++ encontra uma situação dessas, imprime um ***warning*** (advertência). Você, como programador, decide, então, se a suspeita é justificada. Algumas vezes, mensagens de advertência falsas são geradas como um efeito colateral de um erro real, mas você indubitavelmente encontrará muitas advertências enquanto continuar a escrever programas em C.

1.4 A LINGUAGEM C/C++ É CASE-SENSITIVE

É importante saber que as letras maiúsculas e minúsculas são tratadas como caracteres distintos. Por exemplo, em algumas linguagens, os nomes de variáveis ***count***, ***Count*** e ***COUNT*** são três maneiras de se especificar a mesma variável. Entretanto na linguagem C/C++, serão três variáveis diferentes. Então, quando você digitar programas em C/C++ seja cuidadoso na utilização correta das letras.

1.5 FUNÇÕES EM C

A linguagem C é baseada no conceito de blocos de construção. Os blocos de construção são chamados de ***funções***. Um programa em C é uma coleção de uma ou mais funções. Para escrever um programa, você primeiro cria funções e, então, coloca-as juntas.

Em C, uma função é uma sub-rotina que contém uma ou mais declarações e realiza uma ou mais tarefas. Em programas bem-escritos, cada função realiza somente uma tarefa. Cada função tem um nome e uma lista de argumentos que receberá. Em geral, pode-se dar qualquer nome a uma função, com exceção de **main**, que é reservada para a função que começa a execução dos programas.

Uma convenção notacional tornou-se padrão quando se escreve sobre a linguagem C. Haverá parênteses depois do nome da função. Por exemplo, se o nome de uma função é **max( )**, ela será escrita como **max( )** quando referenciada no texto. Essa notação ajudará você a distinguir nomes de variáveis de nomes de funções neste texto.

Segue um primeiro programa exemplo em linguagem C:

```
#include <stdio.h>

void main(){
   printf (“Meu primeiro programa em C!!!\n”);
}
```

Neste exemplo temos a função **main( )** que, ao ser executada, exibirá uma mensagem na tela.

A impressão da mensagem é executada pela função **printf( )** que recebe como parâmetro a mensagem “Meu primeiro programa em C!!!\n”.

A constante “\n” ao final da mensagem significa que o cursor deverá retornar ao início de uma nova linha.

Um novo programa, com o mesmo objetivo, pode ser escrito da seguinte forma:

```
#include <stdio.h>

void hello(){
   printf (“Meu primeiro programa em C!!!\n”);
}

void main(){
   hello();
}
```

Neste novo programa criamos uma função **hello( )**, responsável por exibir a mensagem. Note que a função **hello( )** foi chamada a partir da função **main( )**.

1.5.1 FUNÇÕES COM ARGUMENTOS

Um argumento de função é simplesmente um valor que é passado para a função no momento em que ela é chamada. Você já viu uma função que leva argumento: **printf( )**. Você também pode criar funções que passam argumentos. Por exemplo, a função **sqr( )**, neste programa, leva um argumento inteiro e mostra o seu quadrado:

```
#include <stdio.h>

int sqr (int x){ //o parâmetro foi declarado dentro dos parênteses
   printf (“%d elevado ao quadrado é %d\n”, x, x * x);
}

void main(){
   int num;

   num = 100;
   sqr(num); //chama sqr com o parâmetro num
}
```

Como você pode ver na declaração de **sqr( )**, a variável **x** que receberá o valor passado por **sqr( )** é declarada dentro dos parênteses que seguem o nome da função. Funções que não passam argumentos não necessitam de quaisquer variáveis, de maneira que os parênteses ficam vazios.

Quando **sqr( )** é chamada, o valor de **num** – neste caso 100 – é passado para **x**. Desse modo, a linha “100 elevado ao quadrado é 10000” é apresentada. Você deve executar este programa para se convencer do que ele realmente faz, operando como esperado.

É importante fixar dois termos. Primeiro, o ***argumento*** refere-se ao valor que é usado para chamar uma função. A variável que recebe o valor dos

argumentos usados na chamada da função é ***um parâmetro formal*** da função. Na verdade, funções que levam argumentos são chamados de ***funções parametrizadas***. O importante é que a variável usada como argumento em uma chamada de função não tenha nada a ver com o parâmetro formal que recebe o seu valor.

Um outro exemplo de uma função parametrizada é mostrado aqui. A função **mul( )** imprime o produto de dois argumentos inteiros. Note que os parâmetros para a função **mul( )** são declarados usando-se uma lista separada por vírgula.

```
#include <stdio.h>

void mul (int a, int b){
   printf ("%d", a * b);
}

void main(){
   mul (10, 11);
}
```

Lembre-se que o tipo de argumento usado para chamar uma função deverá ser igual ao do parâmetro formal que está recebendo o dado argumento. Por exemplo, você não deve tentar chamar a função **mul( )** com dois argumentos de ponto flutuante.

1.5.2 FUNÇÕES QUE RETORNAM VALORES

Antes de deixar a discussão sobre funções, é necessário mencionar as funções que retornam valores. Muitas das funções de biblioteca que você usará retornam valores. Em C, uma função pode retornar um valor para uma rotina chamadora usado a palavra reservada ***return***. Para ilustrar, o programa anterior, que imprime o produto de dois números, pode ser reescrito como o programa seguinte. Note que o valor de retorno é associado a uma variável, colocando-se a função do lado direito de uma declaração de atribuição.

```
#include <stdio.h>
/* um programa com função que retorna valores*/

int mul (int a, int b){
   return (a * b);
}

void main(){
   int resposta;

   resposta = mul (10, 11); //atribui o valor de retorno
   printf ("A resposta é %d\n", resposta);
}
```

Nesse exemplo, a função **mul( )** retorna o valor de ***a*** * ***b*** usando a

declaração ***return***. Então, esse valor é atribuído a ***resposta***. Isto é, o valor retornado pela declaração ***return*** torna-se o valor de **mul( )** na chamada de rotina.

Assim como existem tipos diferentes de variáveis, existem tipos diferentes de valor de retorno. Certifique-se de que a variável que recebe o valor de retorno de uma função seja do mesmo tipo que o valor retornado pela função. O tipo retornado pela rotina **mul( )** é ***int***. Quando não explicitamos o tipo retornado por uma função, a linguagem C assume que é um ***int***.

É possível fazer com que uma função retorne usando-se a declaração ***return*** sem qualquer valor associada a ela, tornando o valor retornado indefinido. Também pode haver mais que um ***return*** numa função.

1.5.3 A FORMA GERAL DE UMA FUNÇÃO

A forma geral de uma função é mostrada aqui:

```
tipo_de_retorno nome_da_função (lista de parâmetros) {
   corpo do código da função
}
```

Para funções sem parâmetros, não haverá lista de parâmetros.

1.6 PALAVRAS RESERVADAS DA LINGUAGEM C

A linguagem C, como todas as outras linguagens de programação, consiste de palavras reservadas e regras de sintaxe que se aplicam a cada palavra reservada. Uma palavra reservada é essencialmente um comando e, na maioria das vezes, as palavras reservadas de uma linguagem definem o que pode ser feito e como pode ser feito.

O conjunto de palavras reservadas especificado pelo padrão ANSI C são as seguintes:

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

Todas as palavras reservadas na linguagem C estão em letras minúsculas. Como definido anteriormente, a linguagem C faz diferenciação entre letras maiúsculas e minúsculas: portanto, ***else*** é uma palavra reservada, ***ELSE*** não. Uma palavra reservada não pode ser usada para outro propósito em um programa C. Por exemplo, ela não pode ser usada como nome de uma variável.

UNIDADE 2 – TIPOS DE DADOS, CONSTANTES, OPERADORES E EXPRESSÕES

Variáveis e constantes são manipuladas pelos operadores para formar expressões. Estes são os alicerces da linguagem C.

Ao contrário de outras linguagens de programação, notavelmente BASIC, que possui uma abordagem simples (e limitada) para variáveis, operadores e expressões, a linguagem C oferece muito maior poder e importância a esses elementos.

2.1 NOMES DE IDENTIFICADORES

A linguagem C chama o que é usado para referenciar variáveis, funções, rótulos e vários outros objetos definidos pelo usuário de ***identificadores***.

Quanto aos nomes ou identificadores usados na declaração de variáveis, deve-se considerar as seguintes regras:

- nomes de variáveis começam com uma letra ('A'..'Z', 'a'..'z') ou pelo underscore ('_');
- após podem ser seguidos dígitos, letras e underscores;
- evite o uso do '_' no primeiro caractere do identificador de uma variável, pois este tipo de identificadores é de uso do sistema;
- normalmente ao declararmos uma variável esta será inicializada com zero. Não se deve, no entanto, contar que isto sempre seja verdadeiro, portanto inicialize sempre as variáveis.

Aqui estão alguns exemplos de nomes de identificadores corretos e incorretos:

Correto	**Incorreto**
count	1count
test23	Olá! Aqui
high_balance	high..balance

Em C os primeiros 32 caracteres de um nome de identificador são significantes. Isso quer dizer que duas variáveis com os 32 primeiros caracteres em comum, diferindo somente no 33º, são consideradas iguais.

Como você deve lembrar, em C, letras maiúsculas e minúsculas são tratadas como diferentes e distintas umas das outras. Por isso, **count**, **Count** e **COUNT** são três identificadores distintos.

Um identificador não pode ser o mesmo que uma palavra reservada e não devem ter o mesmo nome de uma função – tanto uma função que você tenha escrito como uma função de biblioteca da linguagem C.

2.2 TIPOS DE DADOS

Em C, os tipos básicos de dados são:

Tipo	Tamanho (em bits)	Intervalo
char	8	-128 a 127
int	16	-32768 a 32767
float	32	3,4E-38 a 3,4E+38
double	64	1,7E-308 a 1,7E+308
void	0	sem valor

Ao contrário do que ocorre nas maioria das linguagens de programação, como Pascal, C não possui um tipo para cadeias de caracteres (***strings***). Para utilizar ***strings*** em C é necessário declarar uma variável como sendo um vetor de caracteres. Desta forma para armazenar um nome de 30 caracteres usa-se o seguinte trecho de programa:

```
char nome[31]; /*Não se esqueça que além dos 30 caracteres,
                 deve-se reservar espaço para o final
                 de string: 0, '\0' ou NULL */
```

2.2.1 MODIFICADORES DE TIPOS

Com a exceção do tipo **void**, os tipos básicos de dados podem ter vários modificadores precedendo-os. Um modificador é usado para alterar o significado do tipo base para adequá-los melhor às necessidades das várias situações. Uma lista de modificadores é mostrada aqui:

signed
unsigned
long
short

Os modificadores **signed**, **unsigned**, **long** e **short** podem ser aplicados aos tipos de base caractere e inteira. Entretanto, **long**, também pode ser aplicado ao tipo **double**. A tabela a seguir mostra todas as combinações permitidas dos tipos básicos e dos modificadores de tipo.

Tipo	Tamanho (em bits)	Intervalo
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	3,4E-38 a 3,4E+38
double	64	1,7E-308 a 1,7E+308
long double	80	3,4E-4932 a 1,1E+4932

A linguagem C permite uma notação simplificada para declaração de inteiro **unsigned**, **short** ou **long**. Você pode simplesmente usar a palavra **unsigned**, **short** ou **long** sem o **int**. O **int** está implícito. Por exemplo:

```
unsigned x;
unsigned int x;
```

declaram igualmente variáveis inteiras sem sinal.

Variáveis do tipo **char** podem ser usadas para armazenar valores outros que são simplesmente o conjunto de caracteres ASCII. Uma variável **char** pode ser usada também como um “pequeno” inteiro com intervalo de –128 a 127, e no lugar de um inteiro quando a situação não requer números grandes.

2.3 DEFINIÇÃO DE VARIÁVEIS

Para declarar uma variável ou um conjunto de variáveis, basta especificar o tipo e a seguir a lista de variáveis:

```
<tipo> <nomev> [ {, <nomev> } ] ;
```

como por exemplo:

```
float juros;
int val = 10; /*declara e atribui o valor 10 à variável val*/
char a = ‘x’; /*declara e atribui o caractere x à variável a*/
double d1, d2, w, z;
```

Variáveis podem ser **globais** ou **locais**. Variáveis **globais** são aquelas declaradas fora do escopo das funções. Variáveis **locais** são aquelas declaradas no início de um bloco e seus escopos estão restritos aos blocos em que foram declaradas.

A declaração de variáveis locais deve obrigatoriamente ser a primeira parte de um bloco, ou seja, deve vir logo após um caractere de “abre chaves”, '**{**'; e não deve ser intercalada com instruções ou comandos.

```
{
int val; /* A declaração de variáveis é a primeira coisa que*/
         /* deve ser feita dentro de um bloco*/
⋮
}
```

2.4 DEFINIÇÃO DE CONSTANTES

Constantes no C podem ser definidas com a palavra reservada **#define**, com a seguinte sintaxe:

```
#define <nome_da_constante> valor
```

Observe que da mesma forma como nas outras linguagens, uma constante não faz parte do código, ou seja, **não gera código**. Na verdade uma constante é como se fosse um comando “substituir” existente em quase todos os editores de texto. Por exemplo:

```
#define PI 3,1415926536
#define ARQUIVO "lixo.dat"
```

Observe que na definição de uma constante não há o “;” no final. Se for colocado, este fará parte do valor associado à constante. Veja exemplos a seguir:

```
/*Código digitado-programador*/

#define PI 3,1415926536
#define ARQUIVO "lixo.dat"
#define SOMA 100+120;
⋮
a = PI;
printf("%s\n", ARQUIVO);
x = SOMA;
⋮
```

```
/*Código que será compilado, depois
do pré-processador do C resolver as
constantes*/
⋮
a = 3,1415926536;
printf("%s\n", "lixo.dat");
x = 100+120;;

/*Erro na linha acima!!!!!*/
```

Pelo exemplo fica claro como o C trata um constante. Como foi colocado um “;” no final da constante SOMA, esta passa a ser parte do valor da mesma, causando um erro na penúltima linha do código, erro que será difícil descobrir, pois, a primeira vista, esta linha está perfeita (o compilador C informará o erro existente na penúltima linha e não na definição de constantes). Muito cuidado com o uso de constantes no C (assim como em outras linguagens).

2.4.1 CONSTANTES HEXADECIMAIS E OCTAIS

Em programação algumas vezes é comum usar um sistema de numeração baseado em 8 ou 16 em vez de 10. O sistema numérico baseado em 8 é chamado ***octal*** e usa os dígitos de 0 a 7. Em octal, o número 10 é o mesmo que 8 em decimal. O sistema numérico de base 16 é chamado ***hexadecimal*** e usa os dígitos de 0 a 9 mais as letras de A até F, que equivalem a 10, 11, 12, 13, 14 e 15. Por exemplo, o número hexadecimal 10 é 16 em decimal. Por causa da freqüência com que estes dois sistemas numéricos são usados, a linguagem C permite que você especifique constantes inteiro em hexadecimal ou octal em vez de decimal, se preferir. Uma constante hexadecimal deve começar com “**0x**” (um zero seguido de um x), seguido pela constante em formato hexadecimal. Uma constante octal começa com um zero. Aqui estão alguns exemplos:

```
hex = 0xFF;  /* 255 em decimal */
oct = 011;     /* 9 em decimal */
```

2.4.2 CONSTANTES STRING

A linguagem C suporta outro tipo de constante em adição àquelas dos tipos predefinidos: a string. Uma string é um conjunto de caracteres entre aspas. Por exemplo, **“este é um teste”** é uma string.

Não confunda strings com caractere. Uma constante caractere simples fica entre dois apóstrofos, por exemplo **‘a’**. Entretanto **“a”** é uma string que contém somente uma letra.

2.4.3 CÓDIGOS DE BARRA INVERTIDA

Colocar todas as constantes caractere entre aspas funciona para muitos caracteres, mas alguns, como o retorno de carro, são impossíveis de serem inseridos em uma string a partir do teclado. Por isso, a linguagem C fornece ***constantes caractere mais barra invertida*** especiais. Estes códigos são mostrados na tabela a seguir.

Código	Significado
\b	Retrocesso
\f	Alimentação de formulário
\n	Nova linha
\r	Retorno de carro
\t	Tabulação horizontal
\”	Aspas
\’	Apóstrofo
\0	Nulo
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro
\N	Constante octal
\xN	Constante hexadecimal

Você usa um código de barra invertida exatamente da mesma maneira como usa qualquer outro caractere. Por exemplo:

```
ch = ‘\t’;
printf (“este é um teste\n”);
```

Esse fragmento de código primeiro atribui uma tabulação a ***ch*** e, então, imprime “este é um teste” na tela, seguido de uma nova linha.

2.5 OPERADORES

A linguagem C é muito rica em operadores internos. Um operador é um símbolo que diz ao compilador para realizar manipulações matemáticas e lógicas específicas. A linguagem C possui três classes gerais de operadores: ***aritméticos***, ***relacionais e lógicos*** e ***bit-a-bit***.

2.5.1 OPERADOR DE ATRIBUIÇÃO

O operador “=“ atribui um valor ou resultado de uma expressão contida a sua direita para a variável especificada a sua esquerda. Exemplos:

```
a = 10;
b = c * valor + getval(x);
a = b = c = 1;  /*Aceita associação sucessiva de valores*/
```

2.5.2 OPERADORES ARITMÉTICOS

Operam sobre números e expressões, resultando valores numéricos.

Operador	Ação
+	soma
-	subtração
*	multiplicação
/	divisão
%	módulo da divisão (resto da divisão inteira)
-	sinal negativo (operador unário)

2.5.3 OPERADORES RELACIONAIS

Operam sobre expressões, resultando valores lógicos de TRUE ou FALSE.

Operador	Ação
>	Maior
>=	maior ou igual
<	Menor
<=	menor ou igual
==	Igual
!=	não igual (diferente)

Cuidado! Não existem os operadores relacionais: “=<“, “=>“ e “<>“. Não confunda a atribuição (“=“) com a comparação (“==“).

2.5.4 OPERADORES LÓGICOS

Operam sobre expressões, resultando valores lógicos de TRUE ou FALSE. Possuem a característica de “***short circuit***”, ou seja, sua execução é curta e só é executada até o ponto necessário.

Operador	Ação
&&	operação AND
||	operação OR
!	operador de negação NOT (operador unário)

Exemplos de “short circuit”:

```
(a == b) && (b == c) /*Se a != b não avalia o resto da expressão*/
(a == b) || (b == c) /*Se a == b não avalia o resto da expressão*/
```

2.5.5 MANIPULAÇÃO DE BITS

A manipulação é feita em todos os bits da variável a qual não pode ser do tipo float ou double.

Operador	Ação
&	bit and
|	bit or
^	bit xor - exclusive or
<<	shift left
>>	shift right
~	bit not (complemento)

Observação: x << n irá rotacionar n vezes a variável x à esquerda.

2.5.6 OPERADORES DE ASSINALAMENTO

É expresso da seguinte forma: (operadores combinados)

```
var = var op expr          →          var op= expr
```

Onde tempos ***op*** como um dos seguintes operadores:

Operador	Ação
+	soma
-	subtração
*	multiplicação
/	divisão
%	módulo (resto da divisão)
>>	shift right
<<	shift left
&	and
^	xor - exclusive or
|	or

Exemplo de aplicação:

```
i += 2; /* É equivalente a: i = i + 2 */
j -= 3; /* É equivalente a: j = j – 3 */
k >>= 3; /* É equivalente a: k = k >> 3;
z &= flag; /* É equivalente a: z = z & flag;
```

2.5.7 OPERADORES DE PRÉ E PÓS-INCREMENTO

As operações abaixo podem ser representadas assim:

```
i = i + 1;              →        i = ++i;        →        ++i;
i = i - 1;              →        i = --i;        →        --i;
z = a; a = a + 1;       →        z = a++;
z = a; a = a - 1;       →        z = a--;
a = a + 1; z = a;       →        z = ++a;
a = a - 1; z = a;       →        z = --a;
```

2.5.8 OPERADORES DE ENDEREÇO

Usados com ponteiros (pointers), para acesso a endereços de memória.

Operador	Significado
&	endereço de uma variável
*	conteúdo do endereço especificado

Exemplos:

```
int var, *x;
:
   x = &var;
:
   var = *x;
```

2.6 TABELA DE OPERADORES DO C

Operador	Função	Exemplo “C”	Exemplo PASCAL
-	menos unário	a = -b	a := -b
+	mais unário	a = +b	a := +b
!	negação lógica	! flag	not flag
~	bitwise not	a = ~b	a := not b
&	endereço de	a = &b	a := ADDR(B)
*	referência a ptr	a = *ptr	a := ptr^
sizeof	tamanho de var	a = sizeof(b)	a := sizeof(b)
++	incremento	++a ou a++	a := succ(a)
--	decremento	--a ou a--	a := pred(a)
*	multiplicação	a = b * c	a := b * c
/	divisão inteira	a = b / c	a := b div c
/	divisão real	a = b / c	a := b / c
%	resto da divisão	a = b % c	a := b mod c
+	soma	a = b + c	a := b + c
-	subtração	a = b - c	a := b - c
>>	shift right	a = b >> n	a := b shr n

Operador	Função	Exemplo “C”	Exemplo PASCAL
<<	shift left	a = b << n	a := b shl n
>	maior que	a > b	a > b
>=	maior ou igual a	a >= b	a >= b
<	menor que	a < b	a < b
<=	menor ou igual a	a <= b	a <= b
==	igual a	a == b	a = b
!=	diferente de	a != b	a <> b
&	bitwise AND	a = b & c	a := b and c
|	bitwise OR	a = b | c	a := b or c
^	bitwise XOR	a = b ^ c	a := b xor c
&&	logical AND	flag1 && flag2	flag1 and flag2
||	logical OR	flag1 || flag2	flag1 or flag2
=	assinalamento	a = b	a := b
OP=	assinalamento	a OP= b	a := a OP b

2.7 EXPRESSÕES

Operadores, constantes e variáveis constituem ***expressões***. Uma expressão em C é qualquer combinação válida dessas partes. Uma vez que muitas expressões tendem a seguir as regras gerais da álgebra, estas regras são freqüentemente consideradas. Entretanto, existem alguns aspectos das expressões que estão especificamente relacionadas com a linguagem C e serão discutidas agora.

2.7.1 CONVERSÕES DE TIPOS EM EXPRESSÕES

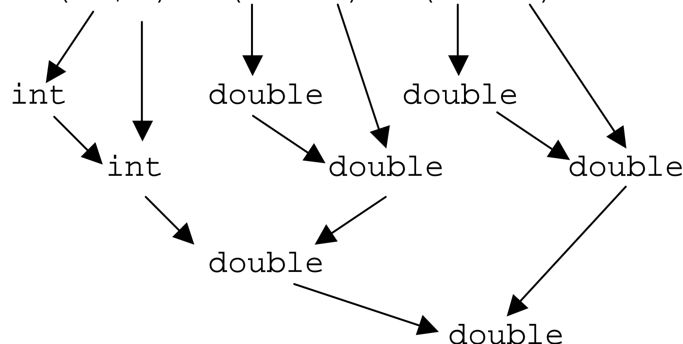
Quando constantes e variáveis de tipos diferentes são misturadas em uma expressão, elas são convertidas para o mesmo tipo. O compilador C converterá todos os operandos para o tipo do operando maior. Isso é feito na base de operação a operação, como descrito nestas regras de conversão de tipos:

- Todos os ***char***s e ***shor int***s são convertidos para ***int***s. Todos os ***float***s são convertidos para ***double***s;
- Para todos os pares de operandos, se um deles é um ***long double***, o outro operando é convertido para uma ***long double***. Se um dos operandos é ***double***, o outro é convertido para ***double***. Se um é ***long***, o outro é convertido para ***long***. Se um é ***unsigned***, o outro é convertido para ***unsigned***.

Uma vez que essas regras de conversão tenham sido aplicadas, cada par de operandos será do mesmo tipo e o resultado de cada operação terá o mesmo tipo dos dois operandos. Note que a regra 2 tem muitas condições que devem ser aplicada em seqüência.

Por exemplo, considere a conversão de tipos que ocorre na expressão a seguir. Primeiro, o caractere ***ch*** é convertido para um inteiro e ***float f*** é convertido para ***double***. Então, o resultado de ***ch/i*** é convertido para um ***double***, já que ***f * d*** é um ***double***. O resultado final é um ***double***, tendo em vista que, neste caso, os dois operandos são ***double***.

```
char ch;
int i;
float f;
double d;

result = (ch/i) + (f * d) - (f + i);
```



2.7.2 MODELADORES (CASTS)

É possível forçar uma expressão a ser de um tipo específico usando-se uma construção chamada de ***modelador***. A forma geral de um modelador é:

```
(tipo) expressão
```

onde ***tipo*** é um dos tipos dado-padrão da linguagem C. Por exemplo, se ***x*** é um inteiro e você quer certificar-se de que a expressão ***x/2*** resulta em um tipo ***float***, garantindo um componente fracionário, pode escrever

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

Aqui, o modelador ***(float)*** é associado com o ***x***, o que faz com que o 2 seja elevado ao tipo ***float*** e o resultado seja um ***float***. Entretanto, tome cuidado: se você escrever a expressão acima como segue, o componente fracionário não será considerado:

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

Neste caso, uma divisão de inteiros é levada a cabo e o resultado é transformado em ***float***.

Modeladores são freqüentemente considerados como operadores. Como um operador, um modelador é unário e tem a mesma precedência de qualquer outro operador unário.

2.7.3 ESPAÇAMENTO E PARÊNTESES

Você pode colocar espaços numa expressão ao seu critério para torná-la mais legível. Por exemplo, as duas expressões seguintes são as mesmas:

```
x=645/(num_entry)-y*(3217/balance);

x = 645 / (num_entry) - y * (3127 / balance);
```

O uso de parênteses redundantes ou adicionais não causará erros ou diminuirá a velocidade de execução da expressão. Você é estimulado a usar parênteses para tornar clara e exata a ordem de avaliação, tanto para você como para outros que precisarem entender o seu programa mais tarde. Por exemplo, qual das duas expressões seguintes é mais fácil de ler?

```
x=y/3-24*temp-127;
x = (y/3) - (34*temp) - 127;
```

2.8 ESQUELETO DE UM PROGRAMA EM C

Todo o programa em C deve conter a função **main( )**. Esta função é responsável pelo início da execução, de forma semelhante ao bloco BEGIN/END. do Pascal.

No C só existeM funções e não existe o conceito de procedimento, ou seja, todas devem retornar algo a quem a chamou, mesmo que o retorno seja do tipo ***void*** (sem valor).

Os comentários no C são feitos através do par "/*" e "*/", sendo um "/*" usado para abrir um comentário e um "*/" para encerrá-lo. Um bloco de comandos é delimitado por chaves ("{" e "}").

Exemplo:

```
#include <stdio.h>  /* No início do programa, declara-se as
                       bibliotecas usadas */

int a_global;       /* Declaração de variáveis globais */

/* Declaração de funções do programador, se for o caso */

int main(){         /* Declaração da função principal. Sempre
                       necessária */
   int conta;    /* Variáveis locais a esta função */
   /* comandos ... */
}
```

UNIDADE 3 – ESTRUTURAS BÁSICAS DE PROGRAMAÇÃO (FLUXO DE CONTROLE)

Os comandos de fluxo de controle de uma linguagem especificam a ordem em que a computação é feita.

3.1 COMANDOS E BLOCOS

Uma expressão tal como x = 0 ou i++ ou printf(...) torna-se um ***comando*** quando seguida por um ponto-e-vírgula, como em:

```
x = 0;
i++;
printf ("Olá !");
```

Em C, o ponto-e-vírgula é um terminador de comandos, e não um separador como em linguagens do tipo Pascal.

As chaves ***{*** e ***}*** são usadas para agruparem declarações e comandos num ***comando composto*** ou ***bloco*** de modo que são sintaticamente equivalentes a um único comando. Não há um ponto-e-vírgula após a chave direita que termina um bloco.

3.2 PRINTF( )

Quase todos os programas exemplo vistos até agora, que realizam saída para a tela, usaram a função ***printf( )***. Vamos dar uma olhada mais formal nela agora.

A forma geral da função ***printf( )*** é:

```
printf ("string de controle", lista de argumentos);
```

Na função ***printf( )***, a string de controle contém tanto caracteres para serem impressos na tela como códigos de formato que especificam como apresentar o restante dos argumentos. Existem códigos de formato que você já deve ter aprendido até aqui:

Código	Significado
%c	Exibe um caractere
%d	Exibe um inteiro em formato decimal
%i	Exibe um inteiro
%e	Exibe um número em notação científica (com e minúsculo)
%E	Exibe um número em notação científica (com E maiúsculo)
%f	Exibe um ponto flutuante em formato decimal
%g	Usa %e ou %f, o que for menor
%G	O mesmo que %g, só que um E maiúsculo é usado se o formato %e for escolhido
%o	Exibe um número em notação octal

Código	Significado
%s	Exibe uma string
%u	Exibe um decimal sem sinal
%x	Exibe um número em hexadecimal com letras minúsculas
%X	Exibe um número em hexadecimal com letras maiúsculas
%%	Exibe um sinal de %]
%p	Exibe um ponteiro

Os comandos de formato podem ter modificadores que especificam o tamanho do campo, o número de casas decimais e um indicador de justificação à esquerda. Um inteiro colocado entre o sinal ***%*** e o comando de formato atua como um ***especificador de largura mínima do campo***. Esse especificador preenche a saída com brancos ou zeros para assegurar que o campo esteja com pelo menos um certo comprimento mínimo. Se a string ou o número é maior que o mínimo, será impresso por completo, mesmo se o mínimo for ultrapassado. O preenchimento-padrão é feito com espaços. Se você quiser preencher com zeros coloque um 0 (zero) antes do especificador de comprimento de campo. Por exemplo, ***%05d*** preencherá um número de menos de cinco dígitos com zeros de maneira que seu tamanho total seja 5.

Para especificar o número de casas decimais impressas num número em ponto flutuante, coloque um ponto decimal depois do especificador de tamanho de campo seguido pelo número de casas decimais que deseja mostrar. Por exemplo, ***%10.4f*** exibirá um número de, no mínimo, dez caracteres de comprimento com quatro casas decimais. Quando isso é aplicado a strings ou inteiros, o número seguinte ao ponto especifica o comprimento máximo do campo. Por exemplo, ***%5.7s*** exibirá uma string que terá, no mínimo, cinco caracteres de comprimento e não excederá sete. Se for maior que o campo de tamanho máximo, a string será truncada.

Por definição, toda saída é ***justificada à direita***: se a largura do campo é maior que o dado impresso, o dado será colocado na extremidade direita do campo. Você pode forçar a informação a ser ***justificada à esquerda*** colocando um sinal de menos diretamente depois do ***%***. Por exemplo, ***%-10.2f*** justificará à esquerda um número em ponto flutuante, com duas casas decimais, em um campo de 10 caracteres.

Existem dois modificadores de comando de formato que permitem à função ***printf( )*** exibir inteiros ***short*** e ***long***. Esses modificadores podem ser aplicados aos especificadores ***d***, ***i***, ***o***, ***u*** e ***x***. O modificador ***l*** diz à função ***printf( )*** que um tipo de dados ***long*** o segue. Por exemplo, ***%ld*** indica que um ***long int*** deve ser exibido. O modificador ***h*** instrui a função ***printf( )*** a exibir um ***short int***. Portanto, ***%hu*** indica que o dado é do tipo ***short unsigned int***.

O modificador ***l*** também pode prefixar os comandos em ponto flutuante de ***e***, ***f*** e ***g*** para indicar que um ***double*** o segue. O modificador ***L*** especifica um ***long double***.

Com a função ***printf( )***, você pode enviar virtualmente qualquer formato de dado que desejar. Veja os exemplos abaixo.

```
printf (“%-6.2f”, 123.234)    //saída   |123.23|
(“%6.2f”, 3.234)              //saída   |  3.23|
(%10s”, “Alô”)                //saída   |       Alô|
(“%-10s”, “Alô”)              //saída   |Alô       |
(“%5.7s”, “123456789”)        //saída   |1234567|
```

3.3 SCANF( )

A função ***scanf( )*** é uma das funções de entrada de dados da linguagem C. Ainda que possa ser usada para ler virtualmente qualquer tipo de dado inserido por meio do teclado, freqüentemente você a usará para a entrada de números inteiros ou de ponto flutuante. A forma geral da função ***scanf( )*** é:

```
scanf (“string de controle”, lista de argumentos);
```

Os especificadores de formato de entrada são precedidos por um sinal ***%*** e dizem à função ***scanf( )*** qual tipo de dado deve ser lido em seguida. Esses códigos são listados na tabela a seguir. Por exemplo, ***%s*** lê uma string enquanto ***%d*** lê um inteiro.

Código	Significado
%c	Lê um único caractere
%d	Lê um decimal inteiro
%i	Lê um decimal inteiro (não pode ser octal ou hexadecimal)
%u	Lê um decimal sem sinal
%e	Lê um número em ponto flutuante com sinal opcional
%f	Lê um número em ponto flutuante com ponto opcional
%g	Lê um número em ponto flutuante com expoente opcional
%o	Lê um número em base octal
%s	Lê uma string
%x	Lê um número em base hexadecimal
%p	Lê um ponteiro

Os caracteres de conversão ***d***, ***i***, ***o***, ***u*** e ***x*** podem ser precedidos por ***h*** para indicarem que um apontador para ***short*** ao invés de ***int*** aparece na lista de argumentos, ou por ***l*** (letra ele) para indicar que um apontador para ***long*** aparece na lista de argumentos. Semelhantemente, os caracteres de conversão ***e***, ***f*** e ***g*** podem ser precedidos por ***l*** (letra ele) para indicarem que um apontador para ***double*** ao invés de ***float*** está na lista de argumentos.

A cadeia de formato geralmente contém especificações de conversão, que são usadas para controlar a conversão da entrada. A cadeia de formato pode conter:

- espaços, tabulações e novas linhas, que serão ignorados;
- caracteres comuns (não %), que devem combinar com o próximo caractere não espaço do fluxo de entrada;
- especificações de conversão, consistindo no caractere ***%***, um caractere ***** opcional de supressão de atribuição, um número opcional especificando um tamanho máximo do campo, um ***h*** ou ***l*** opcional indicando o tamanho do destino, e um caractere de conversão.

Um caractere que não seja um espaço em branco faz com que a função ***scanf( )*** leia e descarte o caractere correspondente. Por exemplo, ***“%d,%d”*** faz com que a função ***scanf( )*** leia um inteiro, então, leia uma vírgula (que será descartada) e, finalmente, leia outro inteiro. Se o caractere especificado não é encontrado, a função ***scanf( )*** terminará.

```
scanf("%d,%d", &a, &b); //entrada válida 10,15
```

Todas as variáveis usadas para receber valores por meio da função ***scanf( )*** deverão ser passadas pelos seus endereços. Por exemplo, para ler um inteiro em uma variável ***count***, você poderia usar a seguinte chamada à função ***scanf( )***:

```
scanf("%d", &count);
```

As strings serão lidas em vetores (cadeias de caracteres) e o nome do vetor é o endereço do primeiro elemento do vetor. Então, para ler uma string no vetor de caracteres ***nome***, você deve usar o seguinte comando:

```
scanf("%s", nome);
```

Nesse caso, ***nome*** já em um endereço e não precisa ser precedido pelo operador ***&***.

Os itens de dados de entrada devem ser separados por espaços, tabulações ou novas linhas. Pontuações como vírgula, ponto-e-vírgula e semelhantes não contam como operadores. Isso significa que

```
scanf("%d%d", &r, &c);
```

aceitará uma entrada dos números ***10 20***, mas falhará com ***10,20***. Como na função ***printf( )***, os códigos de formato da função ***scanf( )*** devem ser correspondidos na ordem com as variáveis que estão recebendo a entrada na lista de argumento.

Um * colocado depois do % e antes do código de formato lerá um dado de um tipo especificado, mas suprimirá a sua atribuição. Assim,

```
scanf("%d%*c%d", &x, &y);
```

dando-se a entrada ***10/20***, colocará o valor 10 em ***x*** descartando o sinal de divisão, e dará a ***y*** o valor 20.

Os comandos de formato podem especificar um campo modificador de comprimento máximo. Esse modificador é um número inteiro colocado entre o sinal ***%*** e o código de comando de formato, que limita o número de caracteres lidos para qualquer campo. Por exemplo, para ler não mais que 20 caracteres em ***str***, você pode escrever:

```
scanf("%20s", str);
```

Se o apontador de entrada é maior do que 20 caracteres, uma chamada subseqüentes para entrada começará onde ela pára. Por exemplo, se você digitar

ABCDEFGHIJKLMNOPQRSTUVWXYZ

em resposta à chamada ***scanf( )***, nesse exemplo, somente os 20 primeiros caracteres, até o "T", serão colocados em ***str*** por causa do especificador de tamanho máximo. Isso significa que os caracteres restantes "UVWXYZ" não são usados. Se uma outra chamada à função ***scanf( )*** é feita, tal como:

```
scanf("%s", str);
```

então “UVWXYZ” é colocado em ***str***. A entrada para o campo pode ser terminada, antes que o campo de comprimento máximo seja alcançado, se um caractere de espaço em branco é encontrado. Nesse caso, a função ***scanf( )*** move-se para o próximo campo.

Ainda que espaços, tabulações e novas linhas sejam usados como separadores de campos, quando da leitura de um único caractere, esses últimos são lidos como qualquer outro caractere. Por exemplo, com uma entrada de ***“x y”***;

```
scanf("%c%c%c", &a, &b, &c);
```

retornará com o caractere “x” em ***a***, um espaço em ***b*** e o caractere “y” em ***c***.

Você não pode usar a função ***scanf( )*** para exibir uma mensagem ao usuário. Portanto, todas as mensagens devem ser feitas explicitamente antes da chamada à função ***scanf( )***.

A função ***scanf( )*** inclui também uma característica muito poderosa chamada ***scanset***. Um scanset define uma lista de caracteres que serão correspondidos por ***scanf( )***. A função ***scanf( )*** continuará a ler caracteres enquanto eles estiverem no scanset. Assim que um caractere entrado não corresponder a qualquer um do scanset, a função ***scanf( )*** segue para o próximo especificador de formato (se existir). Um scanset é definido colocando-se uma lista de caracteres que você quer que seja examinada entre chaves. A chave inicial deve ser prefixada por um sinal de porcentagem. Por exemplo, este scanset diz à função ***scanf( )*** para ler somente os dígitos de 0 a 9:

```
%[1234567890]
```

O argumento correspondente a scanset deve ser uma variável string. Após retornar de função ***scanf( )***, a variável conterá uma string terminada com um nulo com os caracteres lidos. Para ver como isso funciona, considere este programa:

```
#include <stdio.h>

void main(){
   char s1[80], s2[80];

   scanf ("%[1234567890]%s", s1, s2);
   printf ("\n%s| |%s", s1, s2);
}
```

Experimente esse programa usando a entrada

“123456789abcdefg987654”

seguida por um retorno de carro. O programa exibirá, então:

```
123456789| |abcdefg987654
```

Uma vez que “a” não faz parte do scanset, a função ***scanf( )*** pára de ler os caracteres em ***s1*** quando ele é encontrado e os caracteres restantes são colocados em ***s2***.

Você pode especificar um intervalo dentro de um scanset usando um hífen. Por exemplo, isto diz à função ***scanf( )*** para aceitar os caracteres de A a Z:

```
%[A-Z]
```

Você pode especificar mais de um intervalo dentro de um scanset. Por exemplo, este programa lê dígitos e letras. Ele também ilustra que você pode usar o especificador de campo máximo como um scanset.

```
#include <stdio.h>

void main(){
   char str[80];

   printf (“Informe dígitos e letras: ”);
   scanf (“%78[a-z0-9]”, str);
   printf (“\n%s, str);
}
```

Você pode especificar um conjunto invertido se o primeiro caractere é um **^**. Quando está presente, o **^** instrui a função ***scanf( )*** a aceitar quaisquer caracteres que ***não estão*** definidos no scanset.

Um ponto importante a lembrar é que o scanset difere letras minúsculas de maiúsculas. Portanto, se você quiser examinar tanto letras maiúsculas como minúsculas, deve especificá-las individualmente.

3.4 A DECLARAÇÃO if

A forma geral da declaração ***if*** é

```
if (condição){
   comandos;
}
else{
   comandos;
}
```

A cláusula ***else*** é opcional. Se ***condição*** for verdadeira (qualquer coisa diferente de 0), o bloco que forma o destino de ***if*** será executado; caso contrário o bloco que forma o destino de ***else*** será executado (desde que exista ***else***).

Lembre-se que somente o código associado a ***if*** ou o código associado a ***else*** será executado, nunca os dois. Tenha em mente também que o destino dos dois, ***if*** e ***else***, pode ser um comando simples ou um bloco de comandos.

Para demonstrar o comando ***if*** vamos escrever um programa simples que converte bases numéricas. Este programa será capaz de apresentar as seguintes

conversões:

- Decimal para Hexadecimal;
- Hexadecimal para Decimal.

O programa permitirá que primeiro seja selecionado o tipo de conversão a partir de um menu e, então, solicitará o número a ser convertido.

```
/* Programa de conversão de base numérica
    decimal     ---> hexadecimal
    hexadecimal ---> decimal
*/
#include <stdio.h>

void main(){
   int opcao;
   int valor;

   printf ("Converter: \n");
   printf ("1: decimal para hexadecimal\n");
   printf ("2: hexadecimal para decimal\n");
   printf ("\nInforme sua opção: ");
   scanf ("%d", &opcao);

   if (opcao == 1){
      printf ("\nInforme o valor em decimal: ");
      scanf ("%d", &valor);
      printf ("%d em hexadecimal e: %x", valor, valor);
   }

   if (opcao == 2){
      printf ("\nInforme o valor em hexadecimal: ");
      scanf ("%x", &valor);
      printf ("%x em decimal e: %d", valor, valor);
   }
}
```

3.4.1 USANDO A DECLARAÇÃO else

É possível associar um ***else*** com qualquer ***if***. Se a expressão condicional associada a ***if*** é verdadeira, a instrução ou bloco de instruções associada será executada. Se for falsa, então a instrução ou bloco de instruções do ***else*** será executada. O programa seguinte demonstra este princípio fundamental.

```
/* Um exemplo de if-else */
#include <stdio.h>
```

```
void main(){
   int i;

   printf (“Informe um número: ”);
   scanf (“%d”, &i);

   if (i < 0) printf (“O número é negativo”);
   else printf (“O número é positivo ou nulo”);
}
```

3.4.2 O ENCADEAMENTO if-else-if

Uma construção comum em programação é o encadeamento ***if-else-if***. O seguinte exemplo ilustra esta construção:

```
if (condição){
   comandos;
}
else if (condição){
   comandos;
}
else if (condição){
   comandos;
}
else{
   comandos;
}
```

As expressões condicionais serão avaliadas de cima para baixo. Assim que uma condição verdadeira é encontrada, o bloco associado a ela será executado, e o resto do encadeamento é ignorado. Se nenhuma das condições for verdadeira, então o ***else*** final será executado.

Se o ***else*** final não estiver presente e todas as outras condições forem falsas, então nenhuma ação será realizada.

Pode-se usar o encadeamento ***if-else-if*** para implementar o programa de conversão de base numérica desenvolvido anteriormente. Na versão original, cada declaração ***if*** era avaliada sucessivamente, mesmo se uma das declarações anteriores tivesse êxito. Ainda que não haja grande significado neste caso, a avaliação redundante de todos os ***if***s não é muito eficiente ou elegante. O seguinte programa resolve este problema. Nessa versão de encadeamento ***if-else-if***, tão logo uma declaração ***if*** é satisfeita, o resto das declarações é ignorado.

```
/* Programa de conversão de base numérica – if-else-if
    decimal     ---> hexadecimal
    hexadecimal ---> decimal
*/
#include <stdio.h>
```

```
void main(){
   int opcao;
   int valor;

   printf ("Converter: \n");
   printf ("1: decimal para hexadecimal\n");
   printf ("2: hexadecimal para decimal\n");
   printf ("\nInforme sua opção: ");
   scanf ("%d", &opcao);

   if (opcao == 1){
      printf ("\nInforme o valor em decimal: ");
      scanf ("%d", &valor);
      printf ("%d em hexadecimal e: %x", valor, valor);
   }

   else if (opcao == 2){
      printf ("\nInforme o valor em hexadecimal: ");
      scanf ("%x", &valor);
      printf ("%x em decimal e: %d", valor, valor);
   }
   else {
      printf ("\nA opção escolhida é inválida.")
   }
}
```

3.4.3 A EXPRESSÃO CONDICIONAL

Algumas vezes, iniciantes na linguagem C confundem-se pelo fato de que qualquer expressão válida na linguagem C pode ser usada para controla a declaração ***if***. Isto é, o tipo de expressão não precisa se restringir àquelas envolvendo operadores relacionais e lógicos. Só é requerido que a expressão resulte em um valor zero ou não zero. Por exemplo, este programa lê dois inteiros do teclado e mostra o quociente. Para evitar um erro de divisão por zero, uma declaração ***if*** é usada para controlar o segundo número.

```
/* Divide o primeiro número pelo segundo*/
#include <stdio.h>

void main(){
   int a, b;

   printf ("Informe dois números: ");
   scanf ("%d%d", &a, &b);

   if (b) printf ("%d\n", a/b);
   else printf ("Não posso dividir por zero\n");
}
```

Essa abordagem funciona porque, se b for zero, a condição controlando o ***if*** é falsa e a instrução ***else*** é executada. Caso contrário, a expressão é verdadeira (não zero) e a divisão é realizada. Não é necessário escrever uma declaração ***if*** como esta

```
   if (b != 0) printf (“%d\n”, a/b);
```

porque é redundante.

3.4.4 ifs ANINHADOS

Um dos muitos aspectos que causam confusão na declaração ***if***, em qualquer linguagem de programação, são os ***if***s aninhados. Um ***if*** aninhado é uma declaração ***if*** que é objeto de um ***if*** ou um ***else***. Os ***if***s aninhados são incômodos por poderem dificultar saber qual ***else*** está associado a qual ***if***. Considere este exemplo:

```
if (x)
   if (y) printf (“1”);
   else printf (“2”);
```

A qual ***if*** o ***else*** se refere? Felizmente, a linguagem C fornece uma regra muito simples para resolver essa questão. Em C, o ***else*** é ligado ao ***if*** mais próximo dentro do mesmo bloco de código que já não tenha uma declaração ***else*** associada a ele. Neste caso o ***else*** é associado à declaração ***if(y)***. Para fazer com que ***else*** seja associado à declaração ***if(x)***, deve-se usar chaves para sobrepor a sua associação normal, como mostrado aqui:

```
if (x){
   if (y) printf (“1”);
}
else printf (“2”);
```

O ***else*** agora está associado ao ***if(x)***, já que ele não é parte do bloco de código do ***if(y)***.

Exercícios:

1) Escrever um trecho de código que receba duas variáveis inteiras e atribua a uma terceira variável o maior valor.

2) O mesmo problema, em forma de função, retornando o maior valor.

3) Fazer uma função que receba uma variável do tipo inteiro e retorne: 1, se o valor recebido for maior do que 0; 0, se o valor recebido for igual a zero; e -1, se o valor recebido for menor do que 0. Use o comando ***if***.

4) Escreva um programa que lê um par de coordenadas (x,y) inteiras e imprime uma mensagem informando em qual quadrante está o ponto. O programa deve identificar se o ponto está sobre algum dos eixos.

5) Escrever uma função que recebe dois inteiros (a e b) e um código inteiro (cod) e retorne: a + b, se cod >= 0; e |a - b|, se cod < 0.

3.5 A DECLARAÇÃO switch

Ainda que o encadeamento ***if-else-if*** possa realizar testes de múltipla escolha, ele quase nunca é elegante. O código pode ser muito difícil de acompanhar e pode confundir até mesmo o seu autor. Por esse motivo, a linguagem C tem internamente uma declaração de decisão de múltipla escolha chamada de ***switch***. Na declaração ***switch***, a variável é sucessivamente testada contra uma lista de inteiros ou constantes caractere. Quando uma associação é encontrada, o conjunto de comandos associado com a constante é executado. As constantes não precisam sequer estar em qualquer ordem especial. A forma geral da declaração ***switch*** é:

```
switch (variável){
   case constante1:
      commandos;
      break;
   case constante2:
      commandos;
      break;
   case constante3:
      commandos;
      break;
   ⋮
   default
      commandos;
}
```

onde a declaração ***default*** é executada se nenhuma correspondência for encontrada. O ***default*** é opcional e, se ele não estiver presente, nenhuma ação será realizada se todas as correspondências falharem. Quando uma correspondência é encontrada, os comandos associados a ***case*** são executados até que o ***break*** seja encontrado ou até que se encontre o final do ***switch***.

Há três coisas importantes a saber sobre a declaração ***switch***:

1) Ele difere do ***if***, já que o ***switch*** pode testar somente igualdades, enquanto a expressão condicional ***if*** pode ser de qualquer tipo;
2) Nunca duas constantes ***case*** no mesmo ***switch*** podem ter valores iguais. Obviamente, uma declaração ***switch*** dentro de outra declaração ***switch*** pode ter as mesmas constantes ***case***;
3) Uma declaração ***switch*** é mais eficiente que um encadeamento ***if-else-if***.

É possível especificar comandos no ***switch*** que serão executados caso nenhuma correspondência seja encontrada. Basta adicionar uma declaração ***default***. A declaração ***default*** é uma boa maneira de direcionar qualquer final livre que possa ficar pendente na declaração ***switch***. Por exemplo, no programa de conversão de base numérica, pode-se usar uma declaração ***default*** para informar ao usuário que uma resposta inválida foi dada e para tentar outra vez.

Usa-se freqüentemente o ***switch*** para desviar uma seleção de menu para a rotina apropriada. Seguindo essa linha, pode-se usá-la para fazer um melhoramento adicional ao programa de conversão de base numérica. A versão mostrada aqui elimina as séries anteriores de ***if***s e substitui-as por uma clara declaração ***switch***.

```
/* Programa de conversão de base numérica – switch
    decimal     ---> hexadecimal
    hexadecimal ---> decimal
*/
#include <stdio.h>
void main(){
   int opcao;
   int valor;

   printf ("Converter: \n");
   printf ("1: decimal para hexadecimal\n");
   printf ("2: hexadecimal para decimal\n");
   printf ("\nInforme sua opção: ");
   scanf ("%d", &opcao);

   switch(opcao){
      case 1:
         printf ("\nInforme o valor em decimal: ");
         scanf ("%d", &valor);
         printf ("%d em hexadecimal e: %x", valor, valor);
         break;
      case 2:
         printf ("\nInforme o valor em hexadecimal: ");
         scanf ("%x", &valor);
         printf ("%x em decimal e: %d", valor, valor);
         break;
      default:
         printf ("\nOpção inválida. Tente outra vez.")
   }
}
```

3.5.1 A DECLARAÇÃO break

Ainda que as declarações ***break*** sejam geralmente necessárias dentro de um ***switch***, sintaticamente são opcionais. Elas são usadas para terminar a seqüência de comandos associada a cada constante. Entretanto, se a declaração ***break*** é omitida, a execução continuará pela próxima declaração ***case*** até que um ***break*** ou o final do ***switch*** seja encontrado. Observe o programa a seguir.

```
/* Um programa simples com switch e break */
#include <stdio.h>
```

```
void main(){
   int t;

   for (t = 0; t < 10; t++)
      switch (t){
         case 1:
           printf ("Este");
           break;
         case 2:
            printf ("é");
         case 3:
            printf ("o");
            printf ("momento para todos os homens bons\n");
            break;
         case 5:
         case 6:
            printf ("para");
            break;
         case 7:
         case 8:
         case 9:
            printf (".");
      }
}
```

Quando executado, o resultado seguinte é produzido:

```
Este é o momento para todos os homens bons
o momento para todos os homens bons
para para...
```

Este programa também ilustra o fator de que se pode ter declarações ***case*** vazias. Isso é útil quanto muitas condições usam a mesma seqüência de comandos. A habilidade de os ***case***s serem executados juntos quando nenhum ***break*** está presente permite que programas bastante eficientes sejam escritos, evitando-se as injustificáveis duplicações de código.

3.5.2 DECLARAÇÕES switch ANINHADAS

É possível ter um ***switch*** como parte da seqüência de declaração de um ***switch*** externo. Mesmo que as constantes ***case*** do ***switch*** interno e externo contenham valores comuns, nenhum conflito surgirá. Por exemplo, o seguinte fragmento de código é perfeitamente aceitável.

```
switch(x){
   case 1:
      switch(y){
```

```
        case 0:
           printf (“erro de divisão por zero”);
           break;
        case 1:
           process (x, y);
      }
      break;
   case 2:
   ⋮
```

Como outro exemplo, um programa de banco de dados bastante simples mostrado a seguir ilustra como se pode usar uma declaração ***switch*** aninhada. Este programa solicita ao usuário a região e a letra inicial do nome do vendedor e, então, mostra o demonstrativo de vendas do mesmo. ***Switch***s aninhados são requeridos uma vez que muitos vendedores têm a mesma letra inicial. Note que uma nova função, ***toupper( )***, é introduzida. Ela retorna a letra maiúscula correspondente do seu argumento caractere. Ela é usada neste programa para permitir que o usuário insira informações tanto em letra maiúscula como em letra minúscula. A função complementar de ***toupper( )*** é ***tolower( )***, que converte caracteres maiúsculos em minúsculos. Ambas as funções estão contidas na biblioteca ***ctype.h***.

```
/* Um banco de dados simples de vendedores por região */
#include <stdio.h>
#include <ctype.h>

void main(){
   char regiao, vendedor;

   printf (“As regiões são: Leste, Oeste e Norte\n”);
   printf (“Informe a primeira letra da região: ”);
   regiao = getche();
   regiao = toupper(regiao); //converte para maiúsculas
   printf (“\n”);

   switch (regiao){
      case ‘L’:
         printf (“Vendedores são: Rafael, João e Maria \n”);
         printf (“Informe a primeira letra do vendedor: ”);
         vendedor = toupper(getche());
         printf (“\n”);

         switch (vendedor){
            case ‘R’:
               printf (“Vendas: R$%d\n”, 10000);
               break;
            case ‘J’:
               printf (“Vendas: R$%d\n”, 12000);
               break;
```

```
            case ‘M’:
               printf (“Vendas: R$%d\n”, 14000);
         }
         break;

      case ‘O’:
         printf (“Vendedores são: Ronaldo, Lisa e Hilton\n”);
         printf (“Informe a primeira letra do vendedor: ”);
         vendedor = toupper(getche());
         printf (“\n”);

         switch (vendedor){
            case ‘R’:
               printf (“Vendas: R$%d\n”, 10000);
               break;
            case ‘L’:
               printf (“Vendas: R$%d\n”, 9500);
               break;
            case ‘H’:
               printf (“Vendas: R$%d\n”, 13000);
         }
         break;

      case ‘N’:
         printf (“Vendedores são: Tomás, João e Raquel\n”);
         printf (“Informe a primeira letra do vendedor: ”);
         vendedor = toupper(getche());
         printf (“\n”);

         switch (vendedor){
            case ‘R’:
               printf (“Vendas: R$%d\n”, 5000);
               break;
            case ‘J’:
               printf (“Vendas: R$%d\n”, 9000);
               break;
            case ‘T’:
               printf (“Vendas: R$%d\n”, 14000);
         }
         break;
   }
}
```

Para ver como o programa funciona , selecione a região Oeste, digitando ***O***. Isso indica que ***case ‘O’*** é selecionado pela declaração ***switch*** externa. Para ver o total de vendas de Hilton, digite ***H***. Isso faz com que o valor 13000 seja apresentado.

Note que a declaração ***break*** em um ***switch*** aninhado não tem efeito no

switch externo.

Exercício:

1) Escreva um programa que pede para o usuário entrar um número correspondente a um dia da semana e que então apresente na tela o nome do dia. utilizando o comando switch.

3.6 LAÇOS DE REPETIÇÃO

Os ***laços de repetição*** permite que um conjunto de instruções seja repetido até que uma condição seja encontrada. A linguagem C suporta os mesmos tipos de laços que outras linguagens estruturadas modernas. Os laços em C são o ***for***, o ***while*** e o ***do-while***. Cada um deles será examinado separadamente.

3.7 O LAÇO for

A forma geral do laço ***for*** é

```
for (inicialização; condição; incremento){
   comandos;
}
```

Na sua forma simples, a ***inicialização*** é um comando de atribuição usado para inicializar a variável de controle do laço.

A ***condição*** é usualmente uma expressão relacional que determina quando o laço terminará pelo teste da variável de controle do laço contra algum valor.

O ***incremento*** usualmente define como a variável de controle do laço mudará cada vez que a repetição for realizada.

Estas três maiores divisões devem ser separadas por ponto-e-vírgula. O laço ***for*** continuará a execução do programa no comando seguinte a ***for***.

Para um exemplo simples, o seguinte programa imprime os números de 1 a 100 na tela:

```
#include <stdio.h>

void main(){
   int x;

   for (x = 1; x <= 100; x++){
      printf ("%d ", x);
   }
}
```

Neste programa, ***x*** é inicializado em 1. Desde que ***x*** seja menor ou igual a 100, a função ***printf( )*** é chamada. Depois que a função ***printf( )*** retorna, ***x*** é incrementado em 1 e testado para ver se é menor ou igual a 100. Este processo

repete-se até que ***x*** seja maior que 100, ponto este em que a repetição termina. Neste exemplo, ***x*** é a ***variável de controle do laço***, que é modificada e verificada cada vez que o laço é executado.

O laço ***for*** nem sempre precisa executar de maneira crescente. Um laço negativo é criado pelo decremento em vez do incremento da variável de controle do laço. Por exemplo, este programa imprime os números de 100 a 1 na tela.

```
#include <stdio.h>

void main(){
   int x;

   for (x = 100; x > 0; x--){
      printf ("%d ", x);
   }
}
```

Porém, você não está restrito a incrementar ou decrementar a variável de controle do laço. Você pode mudar esta variável da maneira que desejar. Por exemplo, este laço imprime os números de 0 a 100 de cinco em cinco.

```
#include <stdio.h>

void main(){
   int x;

   for (x = 0; x <= 100; x = x + 5){
      printf ("%d ", x);
   }
}
```

Usando-se um bloco de código, é possível que um ***for*** repita múltiplas comandos, como mostrado neste exemplo, que imprime o quadrado dos números de 0 a 99.

```
#include <stdio.h>

void main(){
   int i;

   for (i = 100; i < 100; i++){
      printf ("Este é o valor de i: %d ", i);
      printf (" e i ao quadrado: %d\n", i * i);
   }
}
```

Um ponto importante sobre o laço ***for*** é que o teste condicional é sempre realizado no início do laço. Isso significa que o código dentro do laço pode nunca ser

executado se a condição é falsa logo de início. Por exemplo:

```
x = 10;

for (y = 10; y != x; ++y){
   printf ("%d ", y);
}

printf ("%d", y);
```

Este laço nunca será executado, pois as variáveis ***x*** e ***y*** são de fato iguais quando o laço é testado pela primeira vez. Isso faz com que a expressão seja avaliada como falsa, nem o corpo do laço nem a parte de incremento do próprio laço será executado. Portanto, a variável ***y*** terá o valor 10 associado a ela e a saída será somente o número 10 impresso uma vez na tela.

3.7.1 VARIÁVEIS DO LAÇO for

A discussão mencionada descreve a forma mais comum do laço ***for***. Entretanto, muitas variações são permitidas, o que aumenta o seu poder, flexibilidade e aplicabilidade em certas situações em programação.

Uma das variações mais comuns é o uso de duas ou mais variáveis de controle do laço. Aqui está um exemplo que usa duas variáveis, ***x*** e ***y***, para controlar o laço.

```
#include <stdio.h>

void main(){
   int x, y;

   for (x = 0, y = 0; x + y < 100; x++, y++){
      printf ("%d", x + y);
   }
}
```

Este programa imprime os números de 0 a 98 de dois em dois. Note que as vírgulas são usadas para separar os comandos de inicialização e incremento. A vírgula é um operador da linguagem C que significa essencialmente “faça isto e isto”. A cada interação por meio do laço, tanto ***x*** como ***y*** são incrementadas e os dois devem estar com o valor correto para que o laço se encerre.

A expressão condicional não tem necessariamente que envolver teste de variáveis de controle do laço com algum valor-alvo. De fato, a condição pode ser qualquer expressão válida da linguagem C. Isso significa que é possível fazer o teste por meio de muitas condições de terminação. Por exemplo, este programa ajuda a ensinar crianças a somar. Se a criança cansa e quer parar, ela digita ***N*** quando perguntada se quer mais. Preste especial atenção à parte da condição do laço ***for***. A expressão condicional faz com que o ***for*** seja executado até 99 ou até que o usuário responda não à solicitação.

```
#include <stdio.h>
#include <ctype.h>

void main(){
   int i, j, resposta;
   char fim = ' ';

   for (i = 0, i < 100 && fim != 'N'; i++){
      for (j = 1; j < 10; j++){
         printf ("quanto é %d + %d? ", i, j);
         scanf ("%d", &resposta);

         if (resposta != (i + j)){
            printf ("errado\n");
         }
         else{
            printf ("certo\n");
         }
      }

      printf ("Mais ?");
      fim = toupper(getche());
      printf ("\n");
   }
}
```

Uma outra interessante variação do laço ***for*** é possível porque, atualmente, cada uma das três partes da declaração do ***for*** podem consistir de qualquer expressão válida da linguagem C. Elas não precisam ter nada que faça a seção ser usada de maneira padrão. Considerando isso, vejamos o seguinte exemplo:

```
#include <stdio.h>

void main(){
   int t;

   for (prompt(); t = readnum(); prompt()) {
     sqrnum(t);
   }
}

void prompt(){
   printf ("Informe um inteiro: ");
}
```

```
int readnum(){
   int t;

   scanf (“%d”, &t);
   return t;
}

void sqrnum (int num){
   printf (“%d\n”, num * num);
}
```

Esse programa primeiro apresenta uma solicitação e, então, aguarda pela resposta. Quando um número é informado, o seu quadrado é exibido e novamente é feita uma solicitação. Isso prossegue até que um 0 seja digitado. Se você olhar com atenção o laço ***for*** na função ***main( )***, verá que cada parte do ***for*** é compreendida por chamadas a funções que fazem a solicitação ao usuário e lêem um número do teclado. Se o número informado é zero, o laço termina, já que a expressão condicional será falsa; caso contrário, o quadrado do número é calculado. Assim, nesse laço ***for*** as partes de inicialização e incremento são usadas de modo não tradicional, mas perfeitamente válido.

Outra característica interessante do laço ***for*** é que pedaços da definição do laço não necessitam estar lá. Na verdade, não é necessário que uma extensão esteja presente em qualquer das partes – as expressões são opcionais. Por exemplo, este laço será executado até que o número 10 seja informado.

```
for (x = 0; x!= 10; ) scanf (“%d”, &x);
```

Note que a parte de incremento da definição do ***for*** está vazia. Isso significa que, cada vez que o laço é repetido, a variável ***x*** é testada para verificar se ela é igual a 10, mas ***x*** não é modificado de nenhuma forma. Entretanto, se você digital 10, a condição do laço torna-se falsa e o laço encerra-se.

3.7.2 O LAÇO INFINITO

Um dos usos mais interessantes do laço ***for*** é na criação de laços infinitos. Nenhuma das três expressões que formam o laço ***for*** são requeridas, portanto, é possível fazer um laço sem fim, deixando-se a expressão condicional vazia, como mostra este exemplo.

```
for ( ; ; ){
   printf (“Este laço será executado para sempre.\n”);
}
```

3.7.3 A INTERRUPÇÃO DE UM LAÇO for

A construção ***for (; ;)*** não cria necessariamente um laço infinito tendo em vista uma nova aplicação da declaração ***break*** da linguagem C. Quando encontrada

em algum lugar dentro do corpo de um laço, a declaração ***break*** faz com que o laço termine. O controle do programa segue no código posterior do laço, como mostrado aqui.

```
for ( ; ; ){
   ch = getche();         //lê um caractere
   if (ch = 'A') break;   //sai do laço
}

printf ("Você digitou a letra A");
```

Este laço é executado até que um caractere ***A*** seja digitado no teclado.

3.7.4 USANDO LAÇOS for SEM CONTEÚDO

Um bloco de comandos, como definido pela sintaxe da linguagem C, pode ser vazio. Isso significa que o conteúdo do ***for*** (ou qualquer outro laço) também pode ser vazio.

Essa característica pode ser usada para aumentar a eficiência de certos algoritmos, bem como para criar laços de tempo e espera. É assim que se cria um tempo de espera usando-se o ***for***.

```
for (t = 0; t < ALGUM VALOR; t++);
```

Exercícios:

1) Fazer uma função que receba um número e retorne o fatorial deste número, usando for. (Lembre que 0!=1 e 1!=1).
2) Fazer uma rotina para contar as ocorrências do caractere 'a' em uma string, usando for.
3) Fazer uma rotina para achar o maior valor de um vetor de 10 posições, usando for.
4) Fazer uma rotina para contar o número de ocorrências das letras de 'A' até 'Z' em um string.
5) Escrever um programa que gera e escreve os 5 primeiros números perfeitos. Um número perfeito é aquele que é igual a soma dos seus divisores. Por exemplo:
   6 = 1 + 2 + 3
   28 = 1 + 2 + 4 + 7 + 14
   etc.

3.8 O LAÇO while

O segundo laço disponível na linguagem C é o ***while***. A sua forma geral é

```
while (condição){
   comandos;
}
```

onde comandos, como mencionado anteriormente, pode ser um bloco vazio, um comando simples ou um bloco de comandos que deve ser repetido. ***Condição*** pode ser qualquer expressão válida. O laço é repetido enquanto a ***condição*** for verdadeira. Quando a ***condição*** torna-se falsa, o controle do programa passa para a linha seguinte ao código do laço.

O exemplo a seguir mostra uma rotina de entrada pelo teclado que se repete até que o caractere ***A*** seja digitado.

```
espera_por_caractere(){
   char ch;

   ch = '\0';
   while (ch != 'A') ch = getche();
}
```

A princípio, a variável ***ch*** é inicializada com nulo. Como uma variável local, o seu valor é desconhecido quando a rotina ***espera_por_caractere( )*** é executada. O laço ***while*** começa checando para ver se ***ch*** não é igual a ***A***. Uma vez que ***ch*** foi inicializado como nulo de antemão, o teste é verdadeiro e o laço tem início. Cada vez que uma tecla é pressionada, o teste se realiza novamente. Assim que um ***A*** é informado, a condição torna-se falsa, já que a variável ***ch*** é igual a ***A***, e o laço encerra-se.

Assim como os laços ***for***, os laços ***while*** verificam a condição do teste no início do laço, o que significa dizer que o código do laço pode não ser executado. A variável ***ch*** teve de ser inicializada no exemplo anterior para prevenir de acidentalmente conter um ***A***. Como o teste condicional é feito no início do laço, o ***while*** é bom em situações em que se pode não querer que o laço seja executado. Isto elimina a necessidade de um teste condicional separado antes do laço.

Por exemplo, a função ***centraliza( )***, no programa seguinte, usa um laço ***while*** para emitir o número correto de espaços para centralizar a linha em uma tela de oitenta colunas. Se ***tamanho*** é igual a zero, como é o correto se a linha a ser centralizada tiver 80 caracteres, o laço não é executado. Este programa usa outras duas funções chamadas ***gets( ) e strlen( )***. A função ***gets( )*** permite ler uma string diretamente do teclado; a função ***strlen( )*** retorna o tamanho da string argumento. Esta última função está na biblioteca ***string.h***.

```
#include <stdio.h>
#include <string.h>

void main(){
   char str[255];
   int tamanho;

   printf ("Insira uma string: ");
   gets (str);

   centraliza (strlen (str));
   printf (str);
}
void centraliza (int tamanho) {
```

```
   tamanho = (80 – tamanho)/2;

   while (tamanho > 0){
      printf (" ");
      tamanho--;
   }
}
```

Não há, em absoluto, a necessidade de qualquer declaração no corpo do laço ***while***. Por exemplo.

```
while (ch = getche() != 'A');
```

simplesmente será repetido até que o caractere A seja digitado. Se você se sentir pouco confortável com a atribuição dentro da expressão condicional ***while***, lembre-se de que o sinal de igual é somente um operador que especifica o valor do operando à sua direita.

Exercício:

1) Fazer uma função que receba um número e retorne o fatorial deste número, usando while. (Lembre que 0!=1 e 1!=1).
2) Fazer uma rotina para contar as ocorrências do caractere 'a' em uma string, usando while.
3) Fazer uma rotina para achar o maior valor de um vetor de 10 posições, usando while.

3.9 O LAÇO do-while

Ao contrário dos laços ***for*** e ***while***, que testam a condição do laço no início, o laço ***do-while*** verifica sua condição no final do laço. Isso significa que um laço ***do-while*** será executado pelo menos uma vez. A forma geral do laço ***do-while*** é:

```
do{
   comandos;
}while(condição);
```

Ainda que as chaves não sejam necessárias quando somente um comando será executado no laço, elas são usadas para aumentar a legibilidade do código do programa.

Este programa usa um laço ***do-while*** para ler números do teclado até que um deles seja menos que 100.

```
#include <stdio.h>

void main(){
   int num;
```

```
   do{
      scanf ("%d", &num);
   }while(num > 100);
}
```

Talvez o uso mais comum do laço ***do-while*** seja em uma rotina de seleção em menu. Já que você sempre precisará de uma rotina de seleção em menu para ser executada no mínimo uma vez, o laço ***do-while*** é uma opção óbvia. O seguinte fragmento mostra como adicionar um laço ***do-while*** no menu do programa de conversão numérica.

```
/* Assegura que o usuário especificou uma opção válida */

do{
   printf ("Converte:\n");
   printf ("     1: decimal para hexadecimal\n");
   printf ("     2: hexadecimal para decimal\n");
   printf ("informe sua opção: ");
   scanf ("%d", &opcao);
}while(opcao < 1 || opcao > 2);
```

Depois que a opção tiver sido apresentada, o programa se repetirá até que uma opção válida seja selecionada.

3.10 LAÇOS ANINHADOS

Quando um laço está dentro de outro, diz-se que o laço mais interno é ***aninhado***. Laços aninhados propiciam o meio de resolver alguns problemas interessantes de programação. Por exemplo, este pequeno programa exibe as quatro primeiras potências dos números de 1 a 9.

```
/* Exibe uma tabela das 4 primeiras potencies de 1 a 9 */
#include <stdio.h>

void main(){
   int i, j, k, temp;

   printf("    i     i^2     i^3     i^4\n");

   for (i = 1; i < 10; i++){      // laço externo
      for (j = 1; j < 5; j++) {  // primeiro aninhamento
         temp = 1;

         for (k = 0; k < j; k++) // segundo aninhamento
            temp = temp * i;
         printf ("%9d", temp);
```

```
      }
      printf ("\n");
   }
}
```

Algumas vezes, é importante determinar quantas interações o laço interno executa. Este número é conseguido multiplicando-se o número de vezes que o laço externo intera pelo número de vezes que o laço interno é repetido cada vez que é executado. No exemplo do programa de potência, o laço externo é repetido nove vezes e o segundo laço, quatro vezes; assim, o segundo laço interagirá 36 vezes. O laço interno é executado, em média, duas vezes; dessa forma, o número total de interações é 72.

Como no último exemplo, um melhoramento final para o programa de conversão de base numérica usando-se laços aninhados é mostrado aqui. O laço externo faz com que o programa seja executado até que o usuário diga para parar. o laço interno assegura que o usuário informará uma seleção válida do menu. Agora, em vez de simplesmente converter um número toda vez que for executado, o programa repete até que o usuário queira parar.

```
/* Programa de conversão de base numérica – versão final
    decimal     ---> hexadecimal
    hexadecimal ---> decimal*/
#include <stdio.h>

void main(){
   int opcao;
   int valor;

   //repete até que o usuário diga para terminar
   do{
      //assegura que o usuário especificou uma opção válida
      do{
         printf ("Converter: \n");
         printf ("1: decimal para hexadecimal\n");
         printf ("2: hexadecimal para decimal\n");
         printf ("3: fim\n");
         printf ("\nInforme sua opção: ");
         scanf ("%d", &opcao);
      }while(opcao < 1 || opcao > 3);

      switch(opcao){
         case 1:
            printf ("\nInforme o valor em decimal: ")
            scanf ("%d", &valor);
            printf ("%d em hexadecimal e: %x", valor, valor);
            break;
         case 2:
            printf ("\nInforme o valor em hexadecimal: ")
```

```
            scanf (“%x”, &valor);
            printf (“%x em decimal e: %d”, valor, valor);
            break;
         case 3:
         default:
            printf (“\nOpção inválida. Tente outra vez.”)
      }
      printf (“\n”);
   }while(opcao != 5);
}
```

3.11 QUEBRANDO UM LAÇO

A declaração ***break*** tem dois usos. O primeiro é terminar um ***case*** na declaração ***switch***. Esse uso é coberto na seção deste capítulo onde o ***switch*** foi inicialmente apresentado. O segundo uso é forçar a terminação imediata de um laço, passando sobre o teste condicional normal do laço. Este uso é examinado aqui.

Quando uma declaração ***break*** é encontrada dentro de um laço, este é imediatamente encerrado e o controle do programa é retornado na declaração seguinte ao laço. Por exemplo:

```
#include <stdio.h>

void main(){
   int t;

   for (t = 0; t < 100; t++){
      printf (“%d”, t);
      if (t == 10) break;
   }
}
```

Esse programa imprimirá os números de 0 a 10 na tela e, então, terminará devido à declaração ***break***, que fará com que haja uma saída imediata do laço, sobrepondo-se ao teste condicional ***t < 100*** construído dentro do laço.

É importante ressaltar que um ***break*** causará uma saída somente do laço mais interno. Por exemplo, considere este fragmento:

```
for (t = 0; t < 100; ++t){
   count = 1;
   for ( ; ; ){
      printf (“%d”, count);
      count++;
      if (count == 10) break;
   }
}
```

Ele imprimirá os números de 1 a 10 na tela cem vezes. Toda vez que a

declaração ***break*** é encontrada, o controle é devolvido para o laço ***for*** externo.

Um ***break*** usado em uma declaração ***switch*** afetará somente o dado ***switch*** e não qualquer outro laço em que o ***switch*** estiver.

3.12 A DECLARAÇÃO CONTINUE

A declaração ***continue*** funciona de maneira similar à declaração ***break***. Mas em vez de forçar a terminação, ***continue*** força a ativação da próxima interação do laço, pulando qualquer código que esteja entre eles. Por exemplo, o programa seguinte mostrará somente números pares.

```
#include <stdio.h>
void main(){
   int x;

   for (x = 0; x < 100, x++){
      if (x % 2) continue;
      printf ("%d", x);
   }
}
```

Cada vez que um número ímpar é gerado, a declaração ***if*** é executada, já que a divisão de um número ímpar por dois sempre resulta em resto igual a 1, o que significa verdadeiro. Assim, um número ímpar faz com que a declaração ***continue*** seja executada e o próxima interação ocorra, passando pelo comando ***printf( )***.

Em laços ***while*** e ***do-while***, uma declaração ***continue*** fará com que o controle vá diretamente para o teste condicional e, então, prossiga o processo do laço. No caso do ***for***, primeiro a parte do incremento do laço é realizada, em seguida o teste condicional é executado e, finalmente o laço continua.

Como você pode ver no exemplo seguinte, a declaração ***continue*** também pode ser usada para apressar a terminação de um laço, forçando o teste condicional a ser realizado logo que alguma condição de terminação seja encontrada. Considere este programa, que age como uma máquina de codificação.

```
/* Uma máquina de codificação simples */
#include <stdio.h>

void main(){
   printf ("Insira as letras que você quer codificar.\n");
   printf ("Digite um $ para parar.\n);
   code();
}

/* Codifica as letras */
void code(){
   char pronto, ch;

   pronto = 0;
```

```
   while (!pronto){
      ch = getche();
      if (ch == '$'){
         pronto = 1;
         continue;
      }
      printf ("%c", ch + 1); //desloca o alfabeto uma posição
   }
}
```

Você pode usar esta função para codificar uma mensagem, deslocando todos os caracteres uma letra acima; por exemplo, um ***a*** poderá tornar-se um ***b***. A função terminará quando um ***$*** for lido. Nenhuma interação adicional ocorrerá, uma vez que o teste condicional, que foi levado a efeito pelo ***continue***, encontrará a variável ***pronto*** como verdadeira e fará com que o laço se encerre.

3.13 RÓTULOS E goto

A declaração ***goto*** requer um ***rótulo*** para poder operar. Um rótulo é um identificador válido da linguagem C seguido de dois pontos. Além disso, o rótulo e o ***goto*** que o usa deve estar na mesma função. Por exemplo, um laço de 1 a 100 pode ser escrito usando-se uma declaração ***goto*** e um rótulo, como mostrado aqui.

```
x = 1;

laço1:
   x++;
   if (x < 100) goto laço1
```

Um bom uso para a declaração ***goto*** é quando se necessita sair de uma rotina profundamente aninhada. Por exemplo, considere o seguinte fragmento de código:

```
for (...){
   for (...){
      while (...){
         if(...) goto para;
          ⋮
      }
   }
}
para:
   printf ("erro no programa\n");
```

A eliminação do ***goto*** forçará a realização de um número de testes adicionais. Uma declaração ***break*** simples não pode funcionar aqui, já que ela só pode sair do laço mais interno. Se você substituir a verificação em cada laço, o código poderá, então, parecer-se com o seguinte:

```
pronto = 0;
for (...){
   for (...){
      while (...){
         if(...){
            pronto = 1;
            break;
         }
         ⋮
      }
      if (pronto) break;
   }
   if (pronto) break;
}
```

Se você usar a declaração ***goto***, deve fazê-lo restritamente. Mas, se o código ficar muito mais difícil de se entender sem ele ou se a velocidade de execução do código é crítica, então, certamente, use o ***goto***.

UNIDADE 4 – VETORES, MATRIZES E STRINGS

4.1 INTRODUÇÃO

Vetores são usados para tratamento de conjuntos de dados que possuem as mesmas características. Uma das vantagens de usar vetores é que o conjunto recebe um nome comum e elementos deste conjunto são referenciados através de índices.

Pelo nome vetor estaremos referenciando estruturas que podem ter mais de uma dimensão, como por exemplo matrizes de duas dimensões.

4.2 DECLARAÇÃO DE VETORES UNIDIMENSIONAIS

A forma geral da declaração de um vetor é:

```
tipo nome [tamanho];
```

onde tipo é um tipo qualquer dados, nome é o nome pelo qual o vetor vai ser referenciado e tamanho é o número de elementos que o vetor vai conter. Observar que, em C, o primeiro elemento tem índice ***0*** e o último ***(tamanho –1)***.

Exemplos de declarações de vetores são:

```
int numeros[1000];  /* conjunto de 1000 numeros inteiros */
float notas[65];    /* conjunto de 65 numeros reais */
char nome[40];      /* conjunto de 40 caracteres */
```

O espaço de memória, em bytes, ocupado por um vetor é igual a:

Espaço = tamanho * (número de bytes ocupado por tipo)

É importante notar que em C não há verificação de limites em vetores. Isto significa que é possível ultrapassar o fim de um vetor e escrever em outras variáveis, ou mesmo em trechos de código. É tarefa do programador fazer com que os índices dos vetores estejam sempre dentro dos limites estabelecidos pela declaração do vetor.

O exemplo, mostrado abaixo, ilustra como se declara um vetor, inicializa seus valores e imprime o conteúdo. Notar o uso da diretiva ***#define DIM 5*** para definir uma constante, que posteriormente foi usada para estabelecer o tamanho do vetor. Esta constante passa a ser usada nas referências ao vetor, observar o comando de geração do conjunto. Caso seja necessário trocar o tamanho do vetor basta alterar o valor da constante.

```
/*Este programa gera um vetor contendo números inteiros*/
#define DIM 5
#include <stdio.h>

void main(){
```

```
   int vetor[DIM];
   unsigned int i, num;

   printf (“Entre com o numero inicial do conjunto. ”);
   scanf (“%d”, &num);

   /* Geração do conjunto */
   for (i = 0 ; i < DIM; i++) vetor[i] = num++;

   /* Impressão do conjunto */
   for (i = 0; i < DIM; i++)
      printf (“Elemento %d = %d\n”, i, vetor[i]);
}
```

O exemplo, mostrado a seguir, calcula o produto escalar de dois vetores inteiros. Observar como na leitura dos elementos do vetor usa-se o operador de endereço **&** antes do nome de cada elemento.

```
/*Definições*/
#define DIM 5

#include <stdio.h>

void main(){
   int vetor1[DIM], vetor2[DIM], i, prod = 0;

   printf (“Entre com um vetor de %d elementos\n”, DIM);
   for (i = 0; i < DIM; i++){
      printf (“Elemento %d “, i);
      scanf (“%d”, &vetor1[i]);
   }

   printf(“Entre com um outro vetor de %d elementos\n”, DIM);
   for (i = 0; i < DIM; i++){
      printf (“Elemento %d “, i);
      scanf (“%d”, &vetor2[i]);
   }

   for (i = 0; i < DIM; i++) prod += vetor1[i] * vetor2[i];

   printf(“O produto vale %d”, prod);
}
```

O exemplo, mostrado a seguir ilustra o método da bolha para ordenação em ordem crescente de um vetor de inteiros. Neste método a cada etapa o maior elemento é movido para a sua posição.

```
/*Definições*/
#define DIM 5
#define FALSO 0
#define VERDADE 1

#include <stdio.h>

void main(){
   int vetor[DIM], i, trocou = FALSO, fim = DIM, temp;

   printf ("Entre com um vetor de %d elementos\n", DIM);
   for (i = 0; i < DIM; i++){
      printf ("Elemento %d ", i);
      scanf ("%d", &vetor[i]);
   }

   do {
      trocou = FALSO;
      for (i = 0; i < fim - 1; i++){
         if (vetor[i]>vetor[i+1]){
            temp = vetor[i];
            vetor[i] = vetor[i+1];
            vetor[i+1] = temp;
            trocou = VERDADE;
         }
      }
      fim--;
   } while (trocou);

   for (i = 0; i < DIM; i++) printf("%d\n", vetor[i]);
}
```

4.3 CADEIAS DE CARACTERES (STRINGS)

Um cadeia é um conjunto de caracteres terminado por um caractere nulo, que geralmente é especificado como '\0'. Para especificar um vetor para armazenar um cadeia deve-se sempre reservar um espaço para este caractere. Por exemplo, para armazenar um cadeia de 40 caracteres deve-se reservar um vetor de 41 de caracteres.

Em C é possível haver constantes cadeia, que são definidas como uma lista de caracteres entre aspas. Por exemplo,

```
"programando em C"
```

Não é necessário a colocação do caractere nulo ao final da cadeia. Em C não há tipo cadeia e portanto conjuntos de caracteres teriam de ser tratados como conjuntos de números inteiros, por exemplo. Para facilitar a programação foram

criadas algumas funções para manipular cadeias. As funções mais comuns são as seguintes:

Função de Cadeia	Descrição
`strcat (dest, orig)`	Concatena cadeia origem no final de destino.
`strncat (dest, orig, n)`	Concatena cadeia orig no final de dest, usando no máximo n caracteres de orig.
`strcmp (str1, str2)`	Compara os dois cadeias. Retorna zero se iguais, menor que 0 se str1 < str2, maior que 0 se str1 > str2.
`strcmpi (str1, str2)`	Compara os dois cadeias sem levar em conta maiúsculas e minúsculas.
`strlen (str)`	Calcula o comprimento da cadeia sem o caractere nulo.
`strlwr (str)`	Converte cadeia para minúsculas.
`strupr (str)`	Converte cadeia para maiúsculas.
`strcpy (dest, orig)`	Copia cadeia origem para destino.

Estas funções estão na biblioteca ***string.h***. O exemplo seguinte mostra exemplos de uso de algumas das funções de cadeia.

```
#include <string.h>
#include <stdio.h>

void main(){
   char c, nome[40], sobrenome[15];
   int i;

   printf ("Entre com um nome ");
   scanf ("%s", nome);
   printf (nome);

   printf ("Entre com um sobrenome ");
   scanf ("%s", sobrenome);
   printf (nome);

   strcat(nome, " ");
   strcat(nome, sobrenome);
   printf (nome);

   printf ("Qual caractere? ");
   c = getch();

   for (i =0; i < strlen(nome); i++){
      if (c == nome[i]) printf ("\n%d", i);
   }
}
```

4.4 DECLARAÇÃO DE VETORES MULTIDIMENSIONAIS (MATRIZES)

Em C existe há possibilidade de declararmos vetores de mais de uma dimensão. A forma geral da declaração é a seguinte:

```
tipo nome [dim1][dim2][dim3]...[dimN];
```

onde ***diml*** é o tamanho da dimensão I. Deve-se tomar cuidado com armazenamento de matrizes multidimensionais, por que a memória necessária para guardar estes dados é igual a ***sizeof(tipo) * dim1 * dim2 * dim3 * ... * dimN*** Por exemplo a declaração

```
int matriz[10][20];
```

define uma matriz quadrada de 10 linhas por 20 colunas, enquanto o comando

```
c = 2 * matriz[3][8];
```

armazena o dobro do elemento que está na terceira linha e oitava coluna na variável ***c***. Observar que o primeiro índice indica a linha e o segundo a coluna.

O exemplo abaixo mostra um programa que lê uma matriz de três linhas e cinco colunas e imprime os valores.

```
#define DIML 3
#define DIMC 5

#include<stdio.h>

void main(){
   int i, j;
   int matriz[DIML][DIMC];

   for (i = 0; i < DIML; i++){
      for (j = 0; j < DIMC; j++)
         scanf ("%d", &matriz[i][j]);
   }

   for (i = 0; i < DIML; i++){
      for (j = 0; j < DIMC; j++){
         printf ("%4d", matriz[i][j]);
      }
      printf("\n");
   }
}
```

A matriz é armazenada na memória linha a linha e a figura a seguir ilustra esta idéia com uma matriz de números inteiros de três por três. Estamos assumindo

que cada número inteiro ocupa dois bytes, o endereço aponta um byte e a matriz está armazenada a partir do endereço 1000.

Endereço	Elemento
1000	m[0][0]
1002	m[0][1]
1004	m[0][2]
1006	m[1][0]
1008	m[1][1]
1010	m[1][2]
1012	m[2][0]
1014	m[2][1]
1016	m[2][2]

4.5 VETORES DE CADEIAS DE CARACTERES

A declaração abaixo mostra uma matriz de cadeias de caracteres com 30 linhas de 80 caracteres.

```
char nome_turma[30][80];
```

O exemplo abaixo mostra um programa que lê uma matriz de nomes imprime os seus conteúdos.

```
#define DIML 5
#define DIMC 40

#include <stdio.h>

void main(){
   int i, j;
   int nomes[DIML][DIMC];

   for (i = 0; i < DIML; i++){
      printf (“Entre com a linha %d”, i);
      scanf (“%s”, nomes[i]);
   }

   for (i = 0; i < DIML; i++){
      printf(“O nome %d e\n”, i);
      printf (nomes[i]);
   }
}
```

4.6 INICIALIZAÇÃO DE VETORES

Em C é possível inicializar vetores no momento em que são declarados da mesma forma que variáveis. A forma de fazer isto é a seguinte:

```
tipo nome[dim1][dim2]...[dimN] = {Lista de valores};
```

A lista de valores é um conjunto de valores separados por vírgulas. Por exemplo, a declaração abaixo iniciliza um vetor inteiro de cinco posições.

```
int vetor[5] = { 10, 15, 20, 25, 30 };
```

Observe que nesta declaração é necessário que o tamanho do conjunto seja conhecido antecipadamente. No entanto, também é possível inicializar vetores em que não se conhece o seu tamanho. Observar que neste caso é importante que o programador preveja um modo de indicar o fim do vetor.

O exemplo a seguir mostra os dois casos ilustrados acima. Para descobrir como parar de processar o vetor apresentamos duas soluções possíveis. No primeiro caso a condição de fim do vetor é o número negativo -1, sendo assim uma posição do vetor é perdida para armazenar esta condição. No segundo caso é usado o operador sizeof para descobrir o tamanho do vetor. Observe que sizeof calcula o tamanho do vetor em bytes e por esta razão é necessário uma divisão pelo tamanho em bytes do tipo de cada elemento.

```
#define DIM 5
#include <stdio.h>

void main(){
   int vetor[DIM] = {10, 15, 20, 25, 30};
   int vetor1[] = {10, 20, 30, 40, 50, 60, -1};
   int vetor2[] = {3, 6, 9, 12, 15, 18, 21, 24};
   unsigned int i, tam;

   printf (“\nEste programa imprime um vetor contendo \n”);
   printf (“números inteiros e que foi inicializado \n”);
   printf (“ durante a sua declaração.\n”);

   /* Impressão dos conjuntos */
   printf (“\nVetor com tamanho pre-definido\n”);

   for (i = 0; i < DIM; i++)
      printf(“Elemento %d = %d\n”, i, vetor[i]);

   printf(“\nVetor terminando por -1\n”);

   for (i = 0; vetor1[i] > 0; i++)
      printf (“Elemento %d = %d\n”, i, vetor1[i]);
```

```
   tam = sizeof(vetor2) / sizeof(int);
   printf("\nDescobrindo o tamanho do Vetor\n");

   for (i=0; i < tam ; i++)
      printf("Elemento %d = %d\n", i, vetor2[i]);
}
```

É possível inicializar matrizes multidimensionais e neste caso é necessário especificar todas as dimensões menos a primeira para que o compilador possa reservar memória de maneira adequada. A primeira dimensão somente especifica quantos elementos o vetor irá armazenar e isto lendo a inicialização o compilador pode descobrir.

O exemplo abaixo ilustra a definição de um vetor de cadeia de caracteres, que nada mais é do que uma matriz de caracteres.

```
#include <stdio.h>

void main() {
   char disciplinas [][40] = {
      "disc 0: Computação para Informática",
      "disc 1: Banco de Dados I",
      "disc 2: Banco de Dados II",
      "disc 3: Arquitetura de Computadores I"
   };
   int i;

   printf ("Qual a disciplina? ");
   scanf ("%d", &i);
   printf (disciplinas[i]);
}
```

A declaração abaixo ilustra como declarar e inicializar uma matriz de três linhas por quatro colunas de números reais.

```
float mat[][4] = {1.0, 2.0, 3.0, 4.0,   // linha 1
                  8.0, 9.0, 7.5, 6.0,   // linha 2
                  0.0, 0.1, 0.5, 0.4};  // linha 3
```

4.7 LIMITES DE VETORES E SUA REPRESENTAÇÃO EM MEMÓRIA

Na linguagem C, devemos ter cuidado com os limites de um vetor. Embora na sua declaração, tenhamos definido o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento dentro do vetor ou não.

Por exemplo se declaramos um vetor como ***int valor[5]***, teoricamente só tem sentido usarmos os elementos ***valor[0], ..., valor[4]***. Porém, o C não acusa **erro** se usarmos ***valor[12]*** em algum lugar do programa. Estes testes de limite **devem** ser feitos **logicamente** dentro do programa.

Este fato se deve a maneira como o C trata vetores. A memória do

microcomputador é um espaço (físico) particionado em porções de ***1 byte***. Se declaramos um vetor como ***int vet[3]***, estamos reservando ***6 bytes*** (3 segmentos de 2 bytes) de memória para armazenar os seus elementos. O primeiro segmento será reservado para ***vet[0]***, o segundo segmento para ***vet[1]*** e o terceiro segmento para ***vet[2]***. O segmento inicial é chamado de segmento **base**, de modo que ***vet[0]*** será localizado no segmento base. Quando acessamos o elemento ***vet[i]***, o processador acessa o segmento localizado em ***(base + i)***. Se ***i*** for igual a 2, estamos acessando o segmento ***(base + 2)*** ou ***vet[2]*** (o último segmento reservado para o vetor). Porém, se ***i*** for igual a 7, estamos a acessando segmento ***(base + 7)*** que **não foi reservado** para os elementos do vetor e que provavelmente está sendo usado por uma outra variável ou contém informação espúria (lixo).

Observe que acessar um segmento fora do espaço destinado a um vetor pode **destruir informações** reservadas de outras variáveis. Estes erros são difíceis de detectar pois o compilador não gera nenhuma mensagem de erro... A solução mais adequada é sempre avaliar os limites de um vetor antes de manipulá-lo.

A princípio este fato poderia parecer um defeito da linguagem, mas na verdade trata-se de um recurso muito poderoso do C. Poder manipular sem restrições todos os segmentos de memória é uma flexibilidade apreciada pelos programadores.

UNIDADE 5 – APONTADORES (PONTEIROS)

A compreensão e o uso correto de ponteiros são críticos para criação de muitos programas de êxito na linguagem C. Há três motivos para isso:

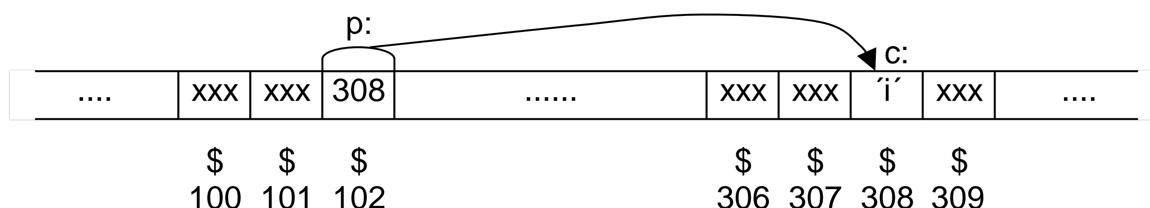
- Os ponteiros provêem o meio para que as funções possam modificar os seus argumentos de chamada;
- Eles são usados para suportar rotinas de alocação dinâmica da linguagem C;
- Podem ser substituídos pelas matrizes em muitas situações, proporcionando aumento de eficiência.

Além disso, muitas das características do C/C++ apóiam-se firmemente nos ponteiros; portanto, um completo entendimento de ponteiros é muito importante. Além de ser uma das características mais fortes da linguagem C, os ponteiros também são muito perigosos. Por exemplo, quando não-inicializados ou descuidados, eles podem causar o travamento do sistema. E pior: é fácil usar ponteiros incorretamente, o que causa “bugs” (erros) difíceis de serem encontrados.

5.1 PONTEIROS SÃO ENDEREÇOS

Um ponteiro é uma variável que guarda um endereço de memória de outro objeto. Mais comumente, esse endereço é a localização de outra variável na memória, embora ele possa ser o endereço de uma porta ou um endereço com propósito especial na RAM, como uma área auxiliar (buffer).

Se uma variável contém o endereço de uma outra variável, a primeira é conhecida como um ponteiro para a segunda. Essa situação é ilustrada na figura abaixo:



Na figura acima, ***p*** é um ponteiro para a variável ***c***. Ou seja, p contém o endereço de memória onde está armazenada a variável c.

Observe o código a seguir:

```
float mult = 315.2;
int sum = 32000;
char letter = ‘A’, /* equivalente a char letter = 65 */
int *set; /* declara o ponteiro set */
float *ptr1;

set = &sum; /* inicializa pointer set */
ptr1 = &mult; /* inicializa pointer ptr1 */
```

O código anterior gera o seguinte mapa de memória:

Endereço de memória: (hexadecimal)	Valor da variável na memória (decimal exceto indicação contrária):	
FA10	3.152 E 02	→ mult (4 bytes)
FA11		
FA12		
FA13		
FA14	32000	→ sum (2 bytes)
FA15		
FA16	65	→ letter (1 byte) ⇒ Valor ASCII do caracter ‘A’
FA17	FA14 (hex)	→ set (2 bytes) ⇒ set aponta para sum
FA18		
FA19	FA10 (hex)	→ ptr1 (2 bytes) ⇒ ptr1 aponta para mult
FA1A		

5.2 VARIÁVEIS PONTEIRO

Se uma variável vai armazenar um ponteiro, ela deve ser declarada como tal. Uma declaração de ponteiro consiste em um tipo base, um ***** e o nome da variável. A forma geral para declaração de uma variável ponteiro é:

```
tipo *nome-da-variável;
```

onde o tipo pode ser qualquer tipo válido da linguagem C e o nome-da-variável é o nome da variável ponteiro. O tipo base do ponteiro define qual tipo de variáveis o ponteiro pode apontar. Por exemplo, estas declarações criam um ponteiro caractere e dois ponteiros inteiros.

```
char *p;
int *temp, *início;
```

5.3 OS OPERADORES DE PONTEIROS

Existem dois operadores especiais de ponteiro: o ***&*** e o ***. O ***&*** é um operador unário que retorna o endereço de memória do seu operando. (Um operador unário requer somente um operando). Por exemplo,

```
cont_addr = &cont;
```

coloca em ***cont_addr*** o endereço de memória da variável ***cont***. Esse endereço é a localização interna da variável no computador. Ele não faz nada com o valor de ***cont***. A operação do operador ***&*** pode ser lembrada como “o retorno de endereço da variável que a ele sucede”. Portanto, a atribuição dada pode ser determinada como “a variável ***cont_addr*** recebe o endereço de variável ***cont***”.

Para entender melhor essa atribuição, assuma que a variável ***cont*** esteja localizada no endereço 2000. Então, após a atribuição, ***cont_addr*** terá o valor de 2000.

O segundo operador é o *. Este operador é o complemento do operador ***&***. Ele é um operador unário que retorna o valor da variável localizada no endereço que o segue. Por exemplo, se ***cont_addr*** contém o endereço de memória da variável ***cont***, então

```
val = *cont_addr;
```

colocará o valor de ***cont*** em ***val***. Se ***cont*** originalmente tem o valor 100, então ***val*** terá o valor de 100, porque este é o valor armazenado no endereço 2000, que é o endereço de memória que foi atribuído a ***cont_addr***. A operação do operador ***** pode ser lembrada como "valor contido no endereço". Neste caso, então, a declaração pode ser lida como "***val*** recebe o valor que está no endereço ***cont_addr***".

Com relação ao mapeamento de memória anterior, envolvendo as variáveis ***mult***, ***sum***, ***letter*** e os ponteiros ***set*** e ***ptr1***, se tivéssemos feito a declaração de uma outra variável inteira, digamos, de nome ***teste***, e fizéssemos a seguinte declaração:

```
teste = *set; //teste recebe o valor da variável apontada por set
```

teste receberia o valor de ***sum*** que é 32000.

De mesma maneira se ***teste*** fosse declarada como float e fizéssemos

```
teste = *ptr1;
```

teste receberia o valor de ***mult*** que é 315.2.

Infelizmente, o sinal de multiplicação e o sinal "valor no endereço" são os mesmos. Isso algumas vezes confunde iniciantes na linguagem C. Esses operadores não se relacionam um com o outro. Tanto ***&*** como ***** têm precedência maior que todos os outros operadores aritméticos, exceto o menos unário, com o qual eles se igualam.

Aqui está um programa que usa esses operadores para imprimir o número 100 na tela:

```
/* imprime 100 na tela */
#include <stdio.h>

void main() {
   int *cont_addr, cont, val;

   cont = 100;
   cont_addr= &cont; //pega o endereço de cont
   val = *cont_addr; //pega o valor armazenado no endereço cont_addr
   printf("%d", val); //exibe 100
}
```

5.4 IMPORTÂNCIA DO TIPO BASE

Como o compilador sabe quantos bytes copiar em ***val*** do endereço apontado por ***cont_addr***? Mais genericamente, como o compilador transfere o número adequado de bytes em qualquer atribuição usando um ponteiro?

A resposta é que o tipo base do ponteiro determina o tipo de dado que o compilador assume que o ponteiro está apontando. Nesse caso, uma vez que ***cont_addr*** é um ponteiro inteiro, 2 bytes de informação são copiados em ***val*** a partir do endereço apontado por ***cont_addr***. Se ele fosse um ponteiro double, então 8 bytes teriam sido copiados.

Você deve certificar-se de que as suas variáveis ponteiro sempre apontam para o tipo de dado correto. Por exemplo, quando você declara um ponteiro para ser do tipo ***int***, o compilador assume que qualquer endereço que ele armazena apontará para uma variável do tipo inteiro. A Linguagem C permite atribuir ***qualquer*** endereço para uma variável ponteiro. O código a seguir está incorreto mas será compilado.

```
#include <stdio.h>
/* Este programa não funciona adequadamente */

void main(){
   float x = 10.1, y;
   int *p;

   p = &x; //observar warning emitido pelo compilador
   y = *p;
   printf("%f",y);
}
```

Este programa não atribuirá o valor da variável ***x*** para ***y***. Tendo em vista que ***p*** é declarado para ser um ponteiro para um inteiro, somente 2 bytes de informação serão transferidos para ***y***, não os 4 bytes que normalmente formam um número de ponto flutuante.

5.5 EXPRESSÕES COM PONTEIROS

5.5.1 ATRIBUIÇÃO DE PONTEIROS

Como qualquer variável, um ponteiro pode ser usado no lado direito de uma declaração para atribuir seu valor para um outro ponteiro. Por exemplo:

```
#include <stdio.h>

void main(){
   int x;
   int *p1, *p2;

   x = 101;
```

```
   p1 = &x;
   p2 = p1;
   printf("Na localização %p ", p2); //imprime o endereço de x
   printf("está o valor %d\n", *p2); //imprime o valor de x
}
```

O endereço, em hexadecimal, de ***x*** é exibido usando-se outro dos códigos de formatação da função printf(). O ***%p*** especifica que um endereço de ponteiro é para ser exibido usando-se a notação hexadecimal.

5.5.2 ARITMÉTICA COM PONTEIROS

Somente duas operações aritméticas podem ser usadas com ponteiros: adição e subtração. Para entender o que ocorre na aritmética com ponteiros, consideremos que ***p1*** seja um ponteiro para um inteiro com o valor atual 2000. Depois da expressão

```
p1++;
```

o conteúdo de ***p1*** será 2002, não 2001. Cada vez que é incrementado, ***p1*** apontará para o próximo inteiro. O mesmo é válido para decrementos. Por exemplo,

```
p1--;
```

fará com que a variável ***p1*** tenha o valor 1998, assumindo-se que ela tinha anteriormente o valor 2000.

Cada vez que é incrementado, um ponteiro apontará para a próxima localização de memória do seu tipo base. Cada vez que é decrementado, apontará para a localização do elemento anterior. No caso de ponteiros para caracteres, parecerá ser uma aritmética normal. Porém, todos os outros ponteiros incrementarão ou decrementarão pelo tamanho do tipo dado para o qual apontam. Por exemplo, assumindo-se caracteres de 1 byte e inteiros de 2 bytes, quando um ponteiro caractere é incrementado, seu valor é incrementado de um; entretanto , quando um ponteiro inteiro é incrementado, seu valor é incrementado de dois bytes. Um ponteiro float seria incrementado de 4 bytes. Por exemplo:

```
char *chptr;
int *iptr;
float *flptr;

chptr = 0x3000;
iptr = 0x6000;
flptr = 0x9000;

chptr++; // ch = 0x3001
iptr++;  // i = 0x6002
flptr++; // flptr = 0x9004
```

Porém, você não está limitado a somente incrementar e decrementar. Pode-se adicionar ou subtrair inteiros para ou de ponteiros. A expressão:

```
p1 = p1+9;
```

fará ***p1*** apontar para o nono elemento do tipo base de ***p1*** considerando-se que ele está apontando para o primeiro. Por exemplo:

```
double *p1;
p1 = 0x1000; // p1 = 4096
p1 = p1+9    // p1 = 0x1048 = 4096 + 9*8
```

5.5.3 COMPARAÇÃO COM PONTEIROS

É possível comparar dois ponteiros em uma expressão relacional. Por exemplo, dados dois ponteiros ***p*** e ***q***, a seguinte declaração é perfeitamente válida:

```
if (p < q) printf("p aponta para um endereço menor que q\n");
```

Geralmente, comparações entre ponteiros devem ser usadas somente quando dois ou mais ponteiros estão apontando para um objeto em comum.

5.6 PONTEIROS E MATRIZES

Há uma relação entre ponteiros e matrizes. Considere este fragmento de código:

```
char str[80], *p1;
p1 = str;
```

Aqui, ***p1*** foi associado ao endereço do primeiro elemento da matriz em ***str***. Na linguagem C, um nome de matriz sem um índice é o endereço para o começo da matriz (em geral um ponteiro contém o endereço do início de uma área de armazenamento ou transferência de dados). O mesmo resultado, um ponteiro para o primeiro elemento da matriz ***str***, pode ser gerado com a declaração abaixo:

```
p1 = &str[0];
```

Entretanto, ela é considerada uma forma pobre por muitos programadores em C. Se você deseja acessar o quinto elemento em ***str***, pode escrever:

```
str[4];
```

ou

```
*(p1+4)
```

As duas declarações retornarão o quinto elemento.

LEMBRE-SE: Matrizes começam em índice zero, então um 4 é usado para indexar ***str***. Você também pode adicionar 4 ao ponteiro ***p1*** para obter o quinto elemento, uma vez que ***p1*** aponta atualmente para o primeiro elemento de ***str***.

A linguagem C permite essencialmente dois métodos para acessar os elementos de uma matriz. Isso é importante porque a aritmética de ponteiros pode ser mais rápida do que a indexação de matriz. Uma vez que a velocidade é freqüentemente importante em programação, o uso de ponteiros para acessar elementos da matriz é muito comum em programas em C.

Para ver um exemplo de como ponteiros podem ser usados no lugar de indexação de matriz, considere estes dois programas - um com indexação de matriz e um com ponteiros - , que exibem o conteúdo de uma string em letras minúsculas.

```
#include <stdio.h>
#include <ctype.h>

/* versão matriz */
void main(){
   char str[80];
   int i;

   printf("digite uma string em letra maiúscula: ");
   scanf ("%s", str);
   printf("aqui está a string em letra minúscula: ");
   for(i = 0; str[i]; i++) printf("%c", tolower(str[i]));
}
```

```
#include <stdio.h>
#include <ctype.h>

/* versão ponteiro */
void main() {
   char str[80], *p;
   printf("digite uma string em letra maiúscula: ");
   scanf ("%s", str);
   printf("aqui está a string em letra minúscula: ");
   p = str; /* obtém o endereço de str*/
   while (*p) printf("%c", tolower(*p++));
}
```

A versão matriz é mais lenta que a versão ponteiro porque é mais demorado indexar uma matriz do que usar o operador *.

Algumas vezes, programadores iniciantes na linguagem C cometem erros pensando que nunca devem usar a indexação de matrizes, já que ponteiros são muito mais eficientes. Mas não é o caso. Se a matriz será acessada em ordem estritamente ascendente ou descendente, ponteiros são mais rápidos e fáceis de usar. Entretanto, se a matriz será acessada randomicamente, a indexação da matriz

pode ser uma melhor opção, pois geralmente será tão rápida quanto a avaliação de uma expressão complexa de ponteiros, além de ser mais fácil de codificar e entender.

5.6.1 INDEXANDO UM PONTEIRO

Em C, é possível indexar um ponteiro como se ele fosse uma matriz. Isso estreita ainda mais o relacionamento entre ponteiros e matrizes. Por exemplo, este fragmento de código é perfeitamente válido e imprime os números de 1 até 5 na tela:

```
/* Indexando um ponteiro semelhante a uma matriz */
#include <stdio.h>

void main(){
   int i[5] = {1, 2, 3, 4, 5};
   int *p, t;

   p = i;
   for(t = 0; t < 5; t++) printf("%d ", *(p + t));
}
```

Em C, a declaração ***p[t]*** é idêntica a ****(p+t)***.

5.6.2 PONTEIROS E STRINGS

Uma vez que o nome de uma matriz sem um índice é um ponteiro para o primeiro elemento da matriz, o que está realmente acontecendo quando você usa as funções de string discutidas nos capítulos anteriores é que somente um ponteiro para strings é passado para função, e não a própria string. Para ver como isso funciona, aqui está uma maneira como a função ***strlen( )*** pode ser escrita:

```
int strlen(char *s){
   int i = 0;

   while(*s){ //strings em C terminam com ´\0´
      i++;
      s++;
   }
   return i;
}
```

Quando uma constante string é usada em qualquer tipo de expressão, ela é tratada como um ponteiro para o primeiro caractere na string. Por exemplo, este programa é perfeitamente válido e imprime a frase "Este programa funciona" na tela:

```
#include <stdio.h>
void main() {
```

```
   char *s;

   s = “Este programa funciona”;
   printf(s)
}
```

5.6.3 OBTENDO O ENDEREÇO DE UM ELEMENTO DA MATRIZ

É possível atribuir o endereço de um elemento específico de uma matriz aplicando-se o operador **&** para uma matriz indexada. Por exemplo, este fragmento coloca o endereço do terceiro elemento de ***x*** em *p*:

```
p = &x[2];
```

Essa declaração é especialmente útil para encontrar uma substring. Por exemplo, este programa imprimirá o restante de uma string, que é inserida pelo teclado, a partir do ponto onde o primeiro espaço é encontrado:

```
#include <stdio.h>
/* Exibe a string à direita depois que o primeiro espaço é
encontrado.*/

void main() {
   char s[80];
   char *p;
   int i;

   printf(“digite uma string: ”);
   gets(s);
   //encontra o primeiro espaço ou o fim da string
   for (i = 0; s[i] && s[i]!=' '; i++);
   p = &s[i];
   printf(p);
}
```

Esse programa funciona, uma vez que ***p*** estará apontando para um espaço ou para um nulo (se a string não contiver espaços). Se há um espaço, o restante da string será impresso. Se há um nulo, a função printf() não imprime nada. Por exemplo, se “Olá a todos” for digitado, “a todos” será exibido.

5.6.4 MATRIZES DE PONTEIROS

Podem existir matrizes de ponteiros como acontece com qualquer outro tipo de dado. A declaração para uma matriz de ponteiros, do tipo ***int***, de tamanho 10 é:

```
int *x[10];
```

Para atribuir o endereço de uma variável ponteiro chamada var ao terceiro elemento da matriz de ponteiros, você deve escrever:

```
x[2] = &var;
```

Para encontrar o valor de var, você deve escrever:

```
*x[2];
```

Um uso bastante comum de matrizes de ponteiros é armazenar ponteiros para mensagens de erros. Você pode criar uma função que emite uma mensagem, dado o seu número de código, como mostrado na função ***serror()*** abaixo:

```
void serror(int num){
   char *err[]={
      “Não posso abrir o arquivo\n”,
      “Erro de leitura\n”,
      “Erro de escrita\n”,
      “Falha na mídia\n”
   };

   printf(“%s”, err[num]);
}
```

Como você pode ver, a função ***printf()*** é chamada dentro da função ***serror()*** com um ponteiro do tipo caractere, que aponta para uma das várias mensagens de erro indexadas pelo número do erro passado para a função. Por exemplo, se ***num*** é passado com valor 2, a mensagem “Erro de escrita” é exibida.

Uma outra aplicação interessante para matrizes de ponteiros de caracteres inicializadas usa a função de linguagem C ***system()***, que permite ao seu programa enviar um comando ao sistema operacional. Uma chamada a ***system()*** tem esta forma geral:

```
system(“comando”);
```

onde comando é um comando do sistema operacional a ser executado, inclusive outro programa. Por exemplo, assumindo-se o ambiente Linux, esta declaração faz com que o diretório corrente seja exibido:

```
system(“ls -l”);
```

5.6.5 PROGRAMA EXEMPLO: TRADUTOR INGLÊS-PORTUGUÊS

A primeira coisa que você precisa é a tabela inglês-português mostrada aqui, entretanto, você é livre para expandI-la se assim o desejar.

```
char trans[][20] = {
   “is”, “é”,
```

```
   “this”, “isto”,
   “not”, “não”,
   “a”, “um”,
   “book”, “livro”,
   “apple”, “maçã”,
   “I”, “eu”,
   “bread”, “pão”,
   “drive”, “dirigir”,
   “to”, “para”,
   “buy”, “comprar”,
   ““,”“
}
```

Cada palavra em inglês é colocada em par com a equivalente em português. Note que a palavra mais longa não excede 19 caracteres.

A função ***main( )*** do programa de tradução é mostrada aqui junto com as variáveis globais necessárias:

```
char entrada[80];
char palavra[80];
char *p;

void main() {
   int loc;

   printf(“Informe a sentença em inglês; ”);
   gets(entrada);
   p = entrada; /* dá a p o endereço da matriz entrada */
   printf(“tradução rústica para o português: ”);
   pega_palavra(); /* pega a primeira palavra */

   /* Este é o laço principal. Ele lê uma palavra por vez da
      matriz entrada e traduz para o português.*/
   do {
      //procura o índice da palavra em inglês em trans
      loc = verifica(palavra);

      //imprime a palavra em português se foi traduzida
      if (loc != -1) printf(“%s “, trans[loc+1]);
      else printf(“<desconhecida> ”);

      pega_palavra(); //pega a próxima palavra
   }while(*palavra);  //repete até encontrar uma string nula
}
```

O programa opera desta maneira: primeiro, o usuário é solicitado a informar uma sentença em inglês. Essa sentença é lida na string entrada. O ponteiro p é, então, associado ao endereço do início de entrada e é usado pela função

pega_palavra( ) para ler uma palavra por vez da string entrada e colocá-la na matriz palavra. O laço principal verifica, então, cada palavra, usando a função ***verifica( )***, e retorna o índice da palavra em inglês ou -1 se a palavra não estiver na tabela. Adicionando-se um a este índice, a palavra correspondente em português é encontrada.

A função ***verifica( )*** é mostrada aqui:

```
/*Esta função retorna a localização de uma correspondência
  entre a string apontada pelo parâmetro s e a matriz trans.*/

int verifica(char *s){
   int i;

   for(i = 0; *trans[i]; i++){
      if(!strcmp(trans[i], s)) break;
   }

   if(*trans[i]) return (i);
   else return (-1);
}
```

Você chama ***verifica( )*** com um ponteiro para a palavra em inglês e a função retorna o seu índice se ela estiver na tabela e -1 se não for encontrada.

A função ***pega_palavra( )*** é mostrada a seguir. Da maneira como a função foi concebida, as palavras são delimitadas somente por espaços ou por terminador nulo.

```
/*Esta função lerá a próxima palavra da matriz entrada. Cada
  palavra é considerada como sendo separada por um espaço ou
  pelo terminador nulo. Nenhuma outra pontuação é permita.
  A palavra retornada será uma string de tamanho nulo quando o
  final da string entrada é encontrado.*/

int pega_palavra(){
   char *q;

   /*recarrega o endereço da palavra toda vez que a função é
     chamada*/
   q = palavra;

   /* pega a próxima palavra*/
   while(*p && *p!=' ') {
      *q = *p;
      p++;
      q++;
   }
   if(*p == ' ') p++;
   *q = '\0'; //termina cada palavra com um terminador nulo
}
```

Assim que retorna da função ***pega_palavra( )*** a variável global conterá ou a próxima palavra em inglês na sentença ou um nulo.

O programa completo da tradução é mostrado aqui:

```
/* Um tradutor (bastante) simples de inglês para português.*/

#include <stdio.h>
#include <string.h>

char trans[][20] = {
 "is", "é",
 "this", "isto",
 "not", "não",
 "a", "um",
 "book", "livro",
 "apple", "maçã",
 "I", "eu",
 "bread", "pão",
 "drive", "dirigir",
 "to", "para",
 "buy", "comprar",
 "",""
 };

char entrada[80];
char palavra[80];
char *p;

/*Esta função retorna a localização de uma correspondência
  entre a string apontada pelo parâmetro s e a matriz trans.*/

int verifica(char *s){
   int i;

   for(i=0; *trans[i]; i++){
      if(!strcmp(trans[i], s)) break;
   }

   if (*trans[i]) return (i);
   else return (-1);
}

/*Esta função lerá a próxima palavra da matriz entrada. Cada
  palavra é considerada como sendo separada por um espaço ou
  pelo terminador nulo. Nenhuma outra pontuação é permitida.
  A palavra retornada será uma string de tamanho nulo quando o
  final da string entrada é encontrado.*/
```

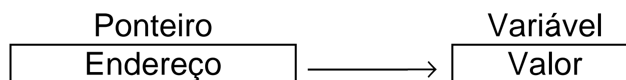
```
pega_palavra(){
   char *q;

   /*Recarrega o endereço da palavra toda vez que a função é
     chamada*/
   q = palavra; //palavra é global

   //pega a próxima palavra
   while(*p && *p!=' '){ //p é global
      *q = *p;
      p++;
      q++;
   }

   if (*p == ' ') p++;
   *q = '\0'; //termina cada palavra com um terminador nulo
}

void main(){
   int loc;

   printf("Informe a sentença em inglês: ");
   gets(entrada);
   p = entrada; //dá a p o endereço da matriz entrada
   printf("Tradução rústica para o português: ");
   pega_palavra(); //pega a primeira palavra

   /* Este é o laço principal. Ele lê uma palavra por vez da
      matriz entrada e traduz para o português.*/
   do {
      //procura o índice da palavra em inglês em trans
      loc = verifica(palavra);

      //imprime a palavra em português se foi traduzida
      if (loc != -1) printf("%s ", trans[loc+1]);
      else printf("<desconhecida> ");

      pega_palavra(); //pega a próxima palavra
   }while(*palavra); //repete até encontrar uma string nula
}
```

5.7 PONTEIROS PARA PONTEIROS

Um ponteiro para ponteiro é uma forma de ***múltipla indireção***, ou uma cadeia de ponteiros. Como você pode ver na figura a seguir, no caso de um ponteiro normal, o valor de um ponteiro é o endereço da variável que contém o valor desejado. No caso de um ponteiro para um ponteiro, o primeiro ponteiro contém o

endereço do segundo ponteiro que aponta para uma variável com o valor desejado.

Indireção simples:



Indireção múltipla:

Indireção múltipla pode ser encadeada até que se queira, mas há poucos casos onde mais de um ponteiro para um ponteiro é necessário ou, efetivamente, seja conveniente usar. Indireção excessiva é difícil de ser seguida e propensa a erros conceituais.

Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal. Isso é feito colocando-se um asterisco adicional na frente do seu nome. Por exemplo, esta declaração diz ao compilador que ***novobalanço*** é um ponteiro para um ponteiro do tipo ***float***:

```
float **novobalanço;
```

É importante entender que ***novobalanço*** não é um ponteiro para um número de ponto flutuante, mas, antes, um ponteiro para um ponteiro ***float***.

Para acessar o valor desejado apontado indiretamente por um ponteiro para um ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no pequeno exemplo a seguir:

```
#include <stdio.h>
void main(){
   int x, *p, **q;

   x = 10;
   p = &x;
   q = &p;
   printf("%d", **q); //imprime o valor de x
}
```

Aqui, ***p*** é declarado como um ponteiro para um inteiro e ***q*** como um ponteiro para um ponteiro para um inteiro. A chamada a ***printf( )*** imprimirá o número 10 na tela.

5.8 INICIALIZANDO PONTEIROS

Depois que é declarado, mas antes que tenha sido associado a um valor, um ponteiro conterá um valor desconhecido. Se tentarmos usar um ponteiro antes de dar a ele um valor, provavelmente não só travará o programa como também o sistema operacional - um tipo de erro muito desagradável!

Por convenção, um ponteiro que está apontando para nenhum lugar deve ter valor nulo. Entretanto, somente o fato de um ponteiro ter um valor nulo não o torna “seguro”. Se usarmos um ponteiro nulo no lado esquerdo de uma declaração de atribuição, correremos o risco de travar o programa e o sistema operacional.

Como um ponteiro nulo é considerado inútil, podemos usá-lo para tornar muitas rotinas de ponteiros fáceis de codificar e mais eficientes. Por exemplo, podemos usar um ponteiro nulo para marcar o final de uma matriz de ponteiros. Se isso é feito, a rotina que acessa aquela matriz saberá que chegou ao fim quando um valor nulo é encontrado.Esse tipo de abordagem é ilustrado pelo laço ***for*** mostrado aqui:

```
/* Observe um nome assumindo o último elemento de p como um
   nulo. */

   for(t=0; p[t]; ++t){
      if(!strcmp(p[t], nome)) break;
   }
```

O laço será executado até que seja encontrada uma correspondência ou um ponteiro nulo. Uma vez que o fim da matriz está marcado com um nulo, a condição que controla o laço falhará quando ele for encontrado.

É uma prática muito comum em programas profissionais escritos na linguagem C inicializar strings. Vimos dois exemplos anteriores neste capítulo na seção de matrizes de ponteiros. Outra variação desse tema é o seguinte tipo de declaração de string:

```
char *p = “Alô mundo\n”;
```

Esse tipo de declaração coloca o endereço da string “Alô mundo” no ponteiro ***p***. Por exemplo, o seguinte programa é perfeitamente válido:

```
#include <stdio.h>
#include <string.h>

char *p=“Alô mundo”;

void main() {
   int t;
   /* imprime a string normal e inversa*/

   printf(p);
   for(t = strlen(p) - 1; t > -1; t--) printf(“%c”, p[t]);
}
```

Entretanto, seu programa não deve fazer atribuições à tabela de strings pelo ***p***, uma vez que ela pode se tornar corrompida.

5.9 PROBLEMAS COM PONTEIROS

Um problema com um ponteiro é difícil de ser encontrado. O ponteiro por si só não é um problema; o problema é que, cada vez que for realizada uma operação usando-o, pode-se estar lendo ou escrevendo para algum lugar desconhecido da memória. Se você ler, o pior que lhe pode acontecer é ler informação inválida, "lixo". Entretanto, se você escrever para o ponteiro, estará escrevendo sobre outros pedaços de código ou dados. Isso pode não ser mostrado mais tarde na execução do programa, levando-o a procurar pelo erro em lugar errado. Pode ser pouco ou não certo sugerir que o ponteiro é um problema. Esse tipo de erro tem feito com que programadores percam muitas noites de sono.

Tendo em vista que erros com ponteiros podem se transformar em pesadelos, você deve fazer o máximo para nunca gerar um! Alguns dos erros mais comuns são discutidos aqui, começando com o exemplo clássico de erro com ponteiro: o ponteiro-não-inicializado. Considere o seguinte:

```
/* Este programa está errado. Não o execute.*/
void main(){
   int x, *p;

   x = 10;
   *p = x;
}
```

Esse programa atribui o valor 10 a alguma localização desconhecida na memória. Ao ponteiro ***p*** nunca foi dado um valor; portanto ele contém um valor, "lixo". Embora o compilador C emita uma mensagem de aviso neste exemplo, o mesmo tipo de problema surge quando um ponteiro está simplesmente apontado para um lugar indevido. Por exemplo, você pode acidentalmente atribuir um ponteiro para um endereço errado. Esse tipo de problema freqüentemente passa despercebido quando o seu programa é muito pequeno, por causa da probabilidade de ***p*** conter um endereço "seguro" - um endereço que não esteja no seu código, na área de dados ou no sistema operacional. Entretanto, à medida que o seu programa cresce, a probabilidade de ***p*** apontar para algum lugar fundamental aumenta. Eventualmente, o seu programa pára de funcionar. Para evitar esse tipo de problema, certifique-se sempre de que um ponteiro esteja apontado para alguma posição válida antes que seja usado.

Um segundo erro é causado pelo mal-entendido sobre como usar um ponteiro. Considere o seguinte:

```
#include <stdio.h>
/* Este programa está incorreto. Não o execute. */
void main(){
   int x, *p;

   x = 10;
   p = x;
   printf("%d", *p);
}
```

A chamada à função ***printf( )*** não imprimirá o valor de x, que é 10, na tela. Imprimirá algum valor desconhecido. O motivo para isso é que a atribuição

```
p = x;
```

está errada. Aquela declaração atribui o valor 10 para o ponteiro ***p***, que supostamente contém um endereço, não um valor. Para tornar o programa correto, você deve escrever:

```
p = &x;
```

Neste exemplo, o compilador C avisará você sobre o erro no programa. Entretanto, nem todos erros desse tipo geral podem ser verificados pelo compilador.

UNIDADE 6 – TIPOS DE DADOS DEFINIDOS PELO USUÁRIO

A linguagem C permite criar diversos tipos diferentes de dados particulares. Podemos citar a ***estrutura***, a ***união***, o ***tipo enumerado*** e o ***typedef***. A ***estrutura*** é um agrupamento de variáveis sobre um nome, que algumas vezes é chamado de ***conglomerado*** ou ***agregado*** de tipos de dados. A ***união*** habilita um mesmo pedaço de memória a ser definido como dois ou mais tipos de dados diferentes. O ***tipo enumerado*** é uma lista de símbolos. O ***typedef*** cria um novo nome para um tipo existente.

6.1 ESTRUTURAS

Na linguagem C, uma ***estrutura*** é uma coleção de variáveis referenciadas sobre um nome, provendo um meio conveniente de manter informações relacionadas juntas. Uma ***declaração de estrutura*** forma uma fonte que pode ser usada para criar variáveis de estruturas. As variáveis que compreendem a estrutura são chamadas de ***campos***. (As estruturas em C equivalem aos registros em Pascal).

Em geral, todos os campos na estrutura estarão logicamente relacionados uns aos outros. Por exemplo, a informação sobre o nome e o endereço em uma lista de endereços deve ser normalmente representada em uma estrutura. O seguinte fragmentos de código declara uma estrutura-fonte que define os campos nome e endereço de tal estrutura. A palavra reservada ***struct*** diz ao compilador que uma estrutura está sendo definida.

```
struct endereço{
   char nome[30];
   char rua[40];
   char cidade[20];
   char estado[3];
   unsigned long int cep;
};
```

Note que a declaração termina com um ponto-e-vírgula. É por isso que uma estrutura é uma declaração. Ainda, a etiqueta ***endereço*** identifica essa estrutura de dado particular e é o seu especificador de tipo.

Neste ponto do código, nenhuma variável foi declarada. Somente a forma dos dados foi definida. Para declarar uma variável com essa estrutura, você deve escrever:

```
struct endereço info_adr;
```

Isso declarará uma variável estrutura do tipo ***endereço*** chamada ***info_adr***. Quando declaramos uma estrutura, estamos definindo um tipo de variável complexa composto por campos. Até que se declare uma variável desse tipo, ela não existe.

Você também pode declarar uma ou mais variáveis enquanto declara uma estrutura. Por exemplo:

```
struct endereço{
   char nome[30];
   char rua[40];
   char cidade[20];
   char estado[3];
   unsigned long int cep;
} info_adr, binfo, cinfo;
```

Isso definirá um tipo de estrutura chamada ***endereço*** e declarará as variáveis ***info_adr***, ***binfo*** e ***cinfo*** como desse tipo.

6.1.1 REFERENCIANDO OS CAMPOS DA ESTRUTURA

Campos da estrutura são referenciados pelo uso do operador de seleção de campo: o ***ponto***. Por exemplo, o seguinte código atribuirá o cep 12345777 ao campo ***cep*** da variável estrutura ***info_adr*** declarada anteriormente:

```
info_adr.cep = 12345777;
```

O nome da variável estrutura seguido por um ponto e o nome do campo referenciarão um campo individual da estrutura. Todos os campos da estrutura são acessados da mesma maneira. A forma geral é:

```
nome_da_variável_estrutura.nome_do_campo
```

Portanto, para imprimir o cep na tela, você pode escrever

```
printf ("%lu", info_adr.cep);
```

Isso imprimirá o cep contido no campo ***cep*** da variável estrutura ***info_adr***.

Da mesma maneira, a matriz de caracteres ***info_adr.nome*** pode ser usada em uma chamada à função ***gets( )***, como mostrada aqui:

```
gets (info_adr.nome);
```

Isso passará um ponteiro para caractere para o começo do campo ***nome***. Se você quisesse acessar os elementos individuais de ***info_adr.nome***, poderia indexar ***nome***. Por exemplo, você pode imprimir o conteúdo da ***info_adr.nome***, um caractere por vez, usando este código:

```
int t;
for (t = 0; info_adr.nome[t]; ++t) putchar (info_adr.nome[t]);
```

6.1.2 MATRIZES DE ESTRUTURAS

Talvez o uso de ***matrizes de estruturas*** seja o mais comum das estruturas. Para declarar uma matriz de estruturas, você deve primeiro definir uma

estrutura e, então, declarar uma variável matriz daquele tipo. Por exemplo, para declarar uma matriz de 100 elementos de estrutura do tipo ***endereço*** (definida anteriormente), você deve escrever:

```
struct endereço info_adr[100];
```

Isso cria 100 conjuntos de variáveis que são organizados como definido na estrutura ***endereço***.

Para acessar uma estrutura específica, o nome da estrutura é indexado. Por exemplo, para imprimir o código cep na estrutura 3, você deve escrever

```
printf ("%lu", info_adr[2].cep);
```

Como todas as variáveis matrizes, as matrizes de estruturas começam sua indexação em zero.

6.1.3 ATRIBUINDO ESTRUTURAS

Se duas variáveis são do mesmo tipo, você pode atribuir uma a outra. Nesse caso, todos os elementos da estrutura no lado esquerdo da atribuição receberão os valores dos elementos correspondentes da estrutura do lado direito. Por exemplo, este programa atribui o valor da estrutura ***um*** para a estrutura ***dois*** e exibe o resultado 10 98.6.

```
#include <stdio.h>

void main(){
   struct exemplo{
      int i;
      double d;
   } um, dois;

   um.i = 10;
   um.d = 98.6;
   dois = um;   //atribui uma estrutura a outra
   printf ("%d %lf", dois.i, dois.d);
}
```

LEMBRE-SE: Você não pode atribuir uma estrutura para outra se elas são de tipos diferentes – mesmo que compartilhem certos elementos.

6.1.4 PASSANDO ESTRUTURAS PARA FUNÇÕES

Até aqui, todas as estruturas e matrizes de estruturas dos exemplos são assumidas como globais ou definidas dentro da função que as usa. Nesta seção, será dada atenção especial à passagem das estruturas e seus elementos para funções.

Passando Campos de uma Estrutura para Funções:

Quando passamos um campo de uma variável estrutura para uma função, você está passando o valor daquele elemento para a função. Portanto, você está passando uma simples variável. Por exemplo, considere esta estrutura:

```
//observe que neste exemplo a estrutura não possui nome.
struct{     //isso é permitido quanto declaramos apenas uma
   char x;  //variável com o formato da estrutura definida.
   int y;
   float z;
   char s[10];
} exemplo;
```

Aqui estão exemplos de cada campo sendo passado para uma função:

```
func1 (exemplo.x);     //passa o valor do caractere em x
func2 (exemplo.y);     //passa o valor do inteiro em y
func3 (exemplo.z);     //passa o valor do float em x
func4 (exemplo.s);     //passa o endereço da string em s
func5 (exemplo.s[2]); //passa o valor do caractere em s[2]
```

Contudo, se você quiser passar o endereço de um campo individual da estrutura, criando uma passagem de parâmetro por referência, deve colocar o operador ***&*** antes do nome da variável. Por exemplo, para passar o endereço dos campos na variável estrutura ***exemplo***, você deve escrever isto:

```
func1 (&exemplo.x);     //passa o endereço do caractere em x
func2 (&exemplo.y);     //passa o endereço do inteiro em y
func3 (&exemplo.z);     //passa o endereço do float em x
func4 (exemplo.s);      //passa o endereço da string em s
func5 (&exemplo.s[2]); //passa o endereço do caractere em s[2]
```

Note que o operador ***&*** precede o nome da variável estrutura, não o nome do campo individual. Note também que o elemento string ***s*** já significa o endereço, assim, o operador ***&*** não é requerido.

Passando Estruturas Inteiras para Funções:

Quando uma estrutura é usada como argumento para uma função, a estrutura inteira é passada usando-se o método padrão de chamada por valor (passagem de parâmetro por cópia). Isso, obviamente, significa que qualquer mudança feita no conteúdo de uma estrutura dentro da função para qual ela é passada não afeta a estrutura usada como argumento.

A consideração mais importante para se ter em mente quando se usa uma estrutura como parâmetro é que o tipo do argumento deve corresponder ao tipo do parâmetro. Por exemplo, este programa declara o argumento ***arg*** e o parâmetro ***parm*** para serem do mesmo tipo de estrutura:

```
#include <stdio.h>

//define um tipo de estrutura
struct exemplo{
   int a, b;
   char ch;
};

void fl(struct exemplo parm){
   printf ("%d", parm.a);
}

void main(){
   struct exemplo arg;

   arg.a = 1000;
   fl(arg);
}
```

Esse programa, como é fácil de se ver, imprimirá o número 1000 na tela. Como ele mostra, é melhor definir um tipo estrutura globalmente e, então, usar o seu nome para declarar variáveis estrutura e parâmetros, quando necessário. Isso ajuda a assegurar a correspondência entre os argumentos e os parâmetros. Também, mostram para outras pessoas que estiverem lendo o programa que ***parm*** e ***arg*** são do mesmo tipo.

6.1.5 PONTEIROS PARA ESTRUTURAS

A linguagem C permite ponteiros para estruturas da mesma maneira que aceita ponteiros para qualquer outro tipo de variável. Contudo, existem alguns aspectos especiais dos ponteiros para estruturas que você deve saber.

Declarando um Ponteiro para Estrutura:

Ponteiros para estruturas são declarados colocando-se o * na frente do nome da variável. Por exemplo, assumindo-se a estrutura ***endereço*** definida previamente, a linha abaixo declara ***ponteiro_endereço*** para ser um ponteiro para dados daquele tipo:

```
struct endereço *ponteiro_endereço;
```

Usando Ponteiros para Estruturas:

Existem muitos usos para ponteiros para estruturas. Um deles é na obtenção de uma chamada por referência para uma função. Outro é na criação de estruturas dinâmicas (listas, filas, pilhas, árvores, etc.).

Existe uma desvantagem maior em passar tudo, até a mais simples estrutura, para funções: o esforço necessário para levar (e trazer) todos os campos

da estrutura para a pilha. Em estruturas simples, com poucos campos, esse esforço não é tão importante, mas, se muitos campos são usados e se algum deles é matriz, a performance de tempo de execução pode se tornar inaceitável. A solução para esse problema é passar somente um ponteiro para estrutura.

Quando um ponteiro para estrutura é passado para uma função, somente o endereço da estrutura é empurrado (e puxado) para a pilha. Isso significa que uma chamada de função extremamente rápida pode ser executada. Também, como estará referenciando a estrutura em si e não uma cópia, a função terá condições de modificar o conteúdo dos campos da estrutura usada na chamada.

Para encontrar o endereço de uma variável estrutura, o operador ***&*** é colocado antes do nome da variável estrutura. Por exemplo, dado este fragmento

```
struct sal{
   float saldo;
   char nome[80];
} cliente;

struct sal *p;   // declaração de um ponteiro para estrutura
```

coloca o endereço de ***cliente*** no ponteiro ***p***. Para acessar o campo ***saldo***, você pode escrever

```
(*p).saldo
```

Você raramente verá, se vir, referências feitas a um campo de uma estrutura com uso explícito do operador *, como mostrado no exemplo anterior. Uma vez que o acesso a um campo de uma estrutura por meio de um ponteiro para a dada estrutura é tão comum, um operador especial definido pela linguagem C realiza esta tarefa. Ele é o ***->***, chamada de operador ***seta***. Ele é formado utilizando-se o sinal de menos seguido por um sinal de maior que. A seta é usada no lugar do operador ponto, quando se acessa um campo da estrutura utilizando-se um ponteiro para a variável estrutura. Por exemplo, a declaração anterior é usualmente escrita assim:

```
p->saldo
```

Para ver como um ponteiro para uma estrutura pode ser usado, examine este programa simples que imprime horas, minutos e segundos na sua tela, usando uma rotina de espera cronometrada:

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

struct estrut_horas{
   int horas;
   int minutos;
   int segundos;
};
```

```
void espera(){
   long int t;

   for (t = 1; t < 128000; ++t);
}

void atualiza (struct estrut_horas *t){
   t->segundos++;

   if (t->segundos == 60){
      t->segundos = 0;
      t->minutos++;
   }

   if (t->minutos == 60){
      t->minutos = 0;
      t->horas++;
   }

   if (t->horas == 24) t->horas = 0;

   espera();
}

void exibe(struct estrut_horas *t){
   printf ("%d:", t->horas);
   printf ("%d:", t->minutos);
   printf ("%d\n", t->segundos);
}

void main(){
   struct estrut_horas tempo;

   tempo.horas = 0;
   tempo.minutos = 0;
   tempo.segundos = 0;

   for ( ; !kbhit(); ){
      atualiza (&tempo);
      exibe (&tempo);
   }
}
```

A cronometragem desse programa é ajustada variando-se o contador do laço na função ***espera( )***.

Como você pode ver, a estrutura global, chamada ***estrut_hora***, é definida, mas nenhuma variável é declarada. Dentro da função ***main( )***, a estrutura ***tempo*** é declarada e inicializada como 00:00:00. Isso significa que ***tempo*** é

conhecida diretamente apenas na função ***main( )***.

Às duas funções – ***atualiza( )***, que atualiza a hora, e ***exibe( )***, que imprime a hora – é passado o endereço de tempo. Nas duas funções, o argumento é declarado para ser da estrutura ***estrut_hora***. Isso é necessário para que o compilador saiba como referenciar os campos da estrutura.

A referência a cada campo da estrutura é feita pelo uso de um ponteiro. Por exemplo, se você quisesse retornar as horas para zero quando chegasse 24:00, poderia escrever:

```
if (t->horas == 24) t->horas = 0
```

Essa linha de código diz ao compilador para pegar o endereço da variável ***t*** (que é o tempo na função ***main( )***) e atribuir zero ao seu campo chamado ***horas***.

LEMBRE-SE: Use o operador ponto para acessar campos da estrutura quando estiver operando na própria estrutura. Quando você tiver um ponteiro para uma estrutura, o operador seta deve ser usado. Lembre-se também de que você tem de passar o endereço da estrutura para uma função usando o caractere ***&***.

6.1.6 MATRIZES E ESTRUTURAS DENTRO DE ESTRUTURAS

Campos de estruturas podem ser de qualquer tipo de dado válido na linguagem C, incluindo-se matrizes e estruturas. Você já viu um exemplo de um campo matriz: a matriz de caracteres usada em ***info_adr***.

Um exemplo de estrutura que é uma matriz é tratado como você viu nos exemplos anteriores. Por exemplo, considere a estrutura:

```
struct x{
   int a[10][10];  //matriz 10x10 de inteiros
   float b;
} y;
```

Para referenciar o inteiro em [3][7] na variável ***a*** da estrutura ***y***, você deve escrever:

```
y.a[3][7]
```

Quando uma estrutura é um campo de outra estrutura, ela é chamada de estrutura ***aninhada***. Por exemplo, o elemento da variável estrutura ***info_adr*** do tipo ***endereço*** está aninhado em ***emp*** neste exemplo:

```
struct emp{
   struct endereço info_adr;
   float salário;
} funcionário;
```

Aqui ***endereço*** é a estrutura definida anteriormente e ***emp*** foi definida como tendo dois elementos. O primeiro elemento é a estrutura do tipo ***endereço***,

que conterá um endereço de um empregado. O segundo é ***salário***, que armazena o salário do empregado. O seguinte fragmento de código atribui o cep 98765777 para o campo ***cep*** do ***endereço*** do ***funcionário***:

```
Funcionário.endereço.cep = 98765777;
```

Como você pode ver, os elementos de cada estrutura são referenciados da esquerda para a direita do mais externo para o mais interno.

6.2 UNIÕES

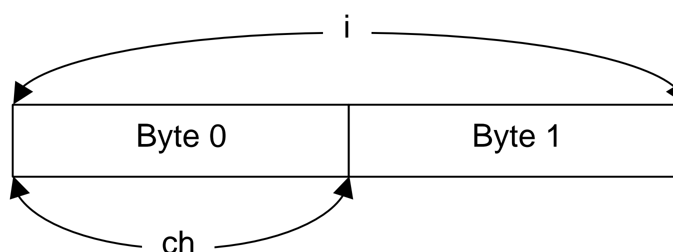
Em C, uma ***union*** é uma localização de memória usado por muitos tipos de variáveis diferentes. A declaração de uma ***union*** é similar ao de uma estrutura, como mostrado neste exemplo:

```
union u_tipo{
   int i;
   char ch;
};
```

Assim como com estruturas, essa declaração não declara qualquer variável. Pode-se declarar uma variável tanto colocando-se o nome dela no fim da declaração como pelo uso de uma declaração em separado. Para declarar uma variável ***union***, chamada ***cnvt***, do tipo ***u_tipo***, usando a declaração dada anteriormente, você pode escrever:

```
union u_tipo cnvt;
```

Em ***cnvt***, o inteiro ***i*** e o caractere ***ch*** compartilham a mesma localização de memória. A Figura a seguir mostra como ***i*** e ***ch*** compartilham o mesmo endereço.



Quando uma variável ***union*** é declarada, o compilador cria automaticamente uma variável grande o bastante para armazenar o tipo da variável de maior tamanho na ***union***.

Para acessar um elemento ***union*** deve-se usar a mesma sintaxe das estruturas: os operadores ponto e seta. Quando se opera diretamente na ***union***, usa-se o operador ponto. Se a variável ***union*** é acessada por meio de um ponteiro, usa-se o operador seta. Por exemplo, para atribuir o inteiro 10 ao elemento ***i*** de ***cnvt***, deve-se escrever

```
cnvt.i = 10;
```

Uniões são usadas freqüentemente quando são necessárias conversões de tipos, uma vez que elas permitem que se considere uma região de memória de mais de uma maneira.

6.3 ENUMERAÇÕES

Uma ***enumeração*** é um conjunto de constantes inteiras que especifica todos os valores legais que uma variável de um tipo específico pode ter. As enumerações não são incomuns no dia-a-dia. Por exemplo, uma enumeração das moedas usadas nos Estados Unidos é:

```
penny, nickel, dime, quarter, half_dollar, dollar
```

As enumerações são definidas muito semelhantemente às estruturas com a palavra reservada ***enum*** usada para indicar o início de um tipo enumerado. A forma geral é mostrada aqui:

```
enum nome_do_tipo_enumerado {lista_de_enumeração} lista de variáveis;
```

A lista de enumeração é uma lista de nomes separados por vírgulas que representam os valores que uma variável da enumeração pode ter. Tanto o nome do tipo enumerado como a lista de variáveis são opcionais. Assim como com estruturas, o nome do tipo enumerado é usado para declarar variáveis do seu tipo. O fragmento seguintes define uma enumeração chamada ***moeda*** e declara ***dinheiro*** para ser desse tipo:

```
enum moeda {penny, nickel, dime, quarter,
            half_dollar, dollar};

enum moeda dinheiro;
```

Dadas essas declarações, as seguintes utilizações são perfeitamente válidas:

```
dinheiro = dime;

if (dinheiro == quarter) printf ("É um quarter\n");
```

O ponto-chave para entender as enumerações é que cada um dos símbolos é representado por um valor inteiro. Assim sendo, eles podem ser usados em qualquer expressão válida com inteiros. A menos que seja inicializado de outra maneira, o valor do primeiro símbolo da enumeração é zero, o valor do segundo símbolo é 1, e assim por diante. Portanto,

```
printf ("%d %d", penny, dime);
```

exibe ***0 2*** na tela.

É possível especificar o valor de um ou mais símbolos usando-se um

inicializador. Isso é feito seguindo-se o símbolo com um sinal de igual e um valor inteiro. Sempre que um inicializador é usado, aos símbolos que aparecem depois dele são associados valores maiores que o valor de inicialização anterior. Por exemplo, a linha seguintes atribui o valor 100 para ***quarter***.

```
enum moeda {penny, nickel, dime, quarter = 100,
            half_dollar, dollar};
```

Agora, os valores desses símbolos são os seguintes:

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

Uma suposição comum, mas errônea, feita a respeito das enumerações é que os símbolos podem ser atribuídos e exibidos diretamente. Esse não é o caso. Por exemplo, o seguinte fragmento de código não procederá como desejado.

```
//isto não funciona
dinheiro = dollar;
printf ("%s", dinheiro);
```

LEMBRE-SE: O símbolo ***dollar*** é simplesmente um nome para um inteiro; não é uma string. Pelo mesmo motivo, não é possível usar este código para se obter os resultados desejados.

```
//este código está incorreto
gets(s);
strcpy (dinheiro, s);
```

Isto é, uma string que contém o nome de um símbolo não é convertida automaticamente naquele símbolo.

Criar um código para entrada e saída de símbolos enumerados é bastante monótono (a menos que se esteja disposto a tornar claro os seus valores inteiros). Por exemplo, o código seguinte é necessário para exibir, em palavras, o tipo de moeda que ***dinheiro*** contém:

```
switch (dinheiro){
   case penny: printf ("penny");
               break;
   case nickel: printf ("nickel");
                break;
   case dime: printf ("dime");
              break;
   case quarter: printf ("quarter");
                 break;
```

```
   case half_dollar: printf ("half_dollar");
                     break;
   case dollar: printf ("dollar");
                break;
}
```

Algumas vezes, é possível declarar uma matriz de strings e usar o valor de uma enumeração como índice na transformação de uma valor enumerado na sua string correspondente. Por exemplo, este código também produzirá a string apropriada:

```
char nome[][20] = {
   "penny",
   "nickel",
   "dime",
   "quarter",
   "half_dollar",
   "dollar"
};
:
printf ("%s", nome[dinheiro]);
```

Esse código funcionará se nenhuma inicialização de símbolos for usada, uma vez que a matriz de strings deve ser indexada começando em zero. Por exemplo, este programa imprime os nomes das moedas:

```
#include <stdio.h>

enum moeda {penny, nickel, dime, quarter,
            half_dollar, dollar};

char nome[][20] = {
   "penny",
   "nickel",
   "dime",
   "quarter",
   "half_dollar",
   "dollar"
};

void main(){
   enum moeda dinheiro;

   for (dinheiro = penny; dinheiro <= dollar; dinheiro++){
      printf ("%s", nome[dinheiro]);
   }
}
```

Como os valores enumerados devem ser convertidos manualmente nos valores correspondentes em português para E/S no console, eles encontram grande uso em rotinas que não fazem tais conversões. É comum ver uma enumeração ser usada para definir uma tabela de símbolos de um compilador, por exemplo, que não requer interações com usuários.

6.4 typedef

A linguagem C permite definir explicitamente novos nomes de tipos de dados usando-se a palavra reservada ***typedef***. Não se cria um novo tipo de dado, mas, ao contrário, define-se um novo nome para um tipo existente. Este processo pode auxiliar programas dependentes de máquinas a serem portáveis; somente a declaração ***typedef*** tem de ser modificada. Ele também auxilia na documentação do seu código por permitir nomes descritivos para tipos de dados padrões. A forma geral da declaração ***typedef*** é:

```
typedef tipo_nome;
```

onde ***tipo*** é qualquer tipo de dado permitido e ***nome*** é o novo nome para esse tipo. O novo nome definido é uma adição, não uma substituição, ao nome existente.

Por exemplo, pode-se criar um novo nome para o tipo ***float*** usando-se

```
typedef float flutuante;
```

Essa declaração diz ao compilador para reconhecer ***flutuante*** como um outros nome para ***float***. A seguir, pode-se criar uma variável ***float*** usando-se ***flutuante***:

```
flutuante atraso;
```

Aqui, ***atraso*** é uma variável de ponto flutuante do tipo ***flutuante***, que é um sinônimo para ***float***.

Pode-se usar ***typedef*** para criar nomes para tipos mais complexos também. Por exemplo:

```
typedef struct cliente_tipo{
   float divida;
   int atraso;
   char nome[40];
} cliente;

cliente clist[NUM_CLIENTES]; //define uma matriz de estruturas
                             //do tipo cliente
```

Nesse exemplo, ***cliente*** não é uma variável do tipo ***cliente_tipo*** mas, ao contrário, um outro nome para ***struct cliente_tipo***.

A utilização de ***typedef*** pode ajudas a tornar códigos mais fáceis de serem lidos e portados para outro computador. Mas, lembre-se: não está sendo criado qualquer tipo de dado novo.

UNIDADE 7 – ESTUDO DETALHADO DE FUNÇÕES

As funções são blocos de construção da linguagem C. Após uma breve introdução realizada na seção 1.5, você tem usado funções de uma maneira mais ou menos intuitiva. Neste capítulo, você as estudará em detalhes, aprendendo coisas tal como fazer uma função modificar seus argumentos, retornar diferentes tipos de dados e usar protótipos de funções.

7.1 FORMA GERAL DE UMA FUNÇÃO

A forma geral de uma função é:

```
especificador_de_tipo nome_da_função (declaração de parâmetros){
   corpo da função
}
```

O especificador de tipo determina o tipo de valor que a função retornará usando a declaração ***return***. Ele pode ser qualquer tipo válido. Se nenhum tipo é especificado então, por definição, a função retorna um resultado inteiro. A lista de declaração de parâmetros é uma lista de tipos de variáveis e nomes, separados por vírgula e que receberão os valores dos argumentos quando a função for chamada. Uma função pode estar sem parâmetro, caso em que a lista de parâmetros estará vazia. Contudo, mesmo se não houver parâmetros, os parênteses ainda são requeridos.

É importante entender algo sobre a lista de declaração de parâmetros de uma função. Ao contrário da declaração de variáveis, na qual muitas variáveis podem ser declaradas para serem de um tipo comum, usando-se uma lista de nomes de variáveis separados por vírgula, todos os parâmetros de uma função devem incluir tanto o tipo como o nome da variável. Isto é, a lista de declaração de parâmetros para uma função toma esta forma geral:

```
f (tipo nomevar1, tipo nomevar2, ..., tipo nomevarN)
```

Por exemplo, esta é uma declaração de parâmetros correta:

```
f (int x, int y, float z)
```

Entretanto, a linha abaixo é incorreta, já que cada parâmetro deve incluir o seu próprio tipo:

```
f (int x, y, float z)
```

7.2 A DECLARAÇÃO return

A declaração ***return*** tem dois importantes usos. Primeiro, ela fará com que haja uma saída imediata da função corrente. Isto é, uma declaração ***return*** fará

com que a execução do programa retorne para o código chamador assim que ele é encontrado. Segundo, ela pode ser usada para retornar um valor. Esses dois recursos serão examinados aqui.

7.2.1 RETORNANDO DE UMA FUNÇÃO

Existem duas maneiras de uma função terminar a execução e retornar ao seu chamador. A primeira é quando a última declaração na função é executada e, conceitualmente, o finalizador da função ***}*** é encontrado.

Por exemplo, esta função simples imprime uma string inversa na tela:

```
void pr_inverso(char *s){
   int t;

   for(t = strlen(s)- 1; t > -1; t--) printf("%c", s[t]);
}
```

Uma vez que a string tiver sido exibida, não há nada para a função fazer a seguir, então ela retorna ao lugar de onde foi chamada.

A segunda maneira de uma função poder retornar é pelo uso da declaração ***return***. A declaração ***return*** pode ser usada sem qualquer valor associado a ela. Por exemplo, a seguinte função imprime o resultado de um número elevado a uma potência positiva. Se o expoente é negativo, a declaração ***return*** faz com que a função termine antes que a chave final seja encontrada, mas nenhum valor é retornado.

```
void potência(int base, int exp){
   int i

   if (exp < 0) return; //não calcula com expoente negativo

   i = 1;

   for( ;exp ; exp--, i = base * i);

   printf("A resposta é: %d ", i);
}
```

7.2.2 RETORNANDO UM VALOR

Para retornar um valor de uma função, você deve colocar o valor a ser retornado após a declaração ***return***. Por exemplo, esta função retorna o maior valor entre dois argumentos:

```
max (int a, int b){
   int temp;

```

```
   if (a > b) temp = a;
   else temp = b;

   return temp;
}
```

Note que uma função retorna um valor inteiro. Esse é o tipo de retorno padrão de uma função se nenhum outro é explicitamente especificado na definição da função.

É possível uma função conter duas ou mais declarações ***return***. Por exemplo, a função ***max( )*** é melhor escrita como mostrado aqui:

```
max (int a, int b){
   if (a > b) return a;
   else return b;
}
```

Outro exemplo de função que usa duas declarações ***return*** é a ***encontra_substr( )***, mostrada aqui. Ela retorna o índice de início de uma substring dentro de uma string ou –1, se nenhuma correspondência é encontrada. Sem o uso da segunda declaração return, uma variável temporária e código extra precisariam ser usados.

```
encontra_substr (char *sub, char *str){
   int t;
   char *p, *p2;

   for(t=0; str[t]; t++){
      p = &str[t]; //pega o ponto de início de str
      p2 = sub;

      while (*p2 && *p2 == *p) { //enquanto não for final de
         p++;  //sub e for verificado igualdade entre os
         p2++; //caracteres de sub e str, avança.
      }

      if (!*p2) return t; //se está no final de sub, então a
                          //correspondência foi encontrada
   }
   return -1;
}
```

Se essa função é chamada com a substring contendo “dos” e a string principal contendo “Olá a todos”, a função retornará o valor 8.

Todas as funções, exceto aquelas declaradas para serem do tipo ***void***, retornam um valor. Esse valor é explicitamente especificado pela declaração ***return*** ou desconhecido, se nenhuma declaração ***return*** é especificada. Isso significa que uma função pode ser usada como um operando em qualquer expressão válida em

C. Portanto, cada uma das seguintes expressões são válidas em C:

```
x = abs(y);
if (max(x, y) > 100) printf("excelente");
for (ch = getchar(); isdigit(ch); ) ...;
```

Entretanto, uma função não pode ser alvo de uma atribuição. Uma declaração como

```
swap(x, y) = 100 ; /* declaração incorreta */
```

é errada. O compilador C sinalizará com um erro e não compilará um programa que contenha tal declaração.

Normalmente as funções, exceto aquelas do tipo ***void***, retornam valores. Entretanto, você não tem que necessariamente usá-los para alguma coisa. Uma questão bastante comum com relação a valores de retorno de funções é: " Eu tenho que atribuir este valor para alguma variável, uma vez que ele está sendo retornado?". A resposta é não. Se não há atribuição especificada, o valor retornado é descartado. Considere o seguinte programa que usa a função ***mul( )***.

```
#include <stdio.h>

int mul (int a, int b){
   return (a * b)
}

void main(){
   int x, y, z;

   x = 10;
   y = 20;
   z = mul(x, y);              //linha 1
   printf ("%d", mul (x, y)); //linha 2
   mul (x, y);                 //linha 3
}
```

Na linha 1, o valor de retorno de ***mul( )*** é associado a ***z***. Na linha 2, o valor de retorno não é associado, porém ele é usado pela função ***printf( )***. Finalmente, na linha 3, o valor de retorno é perdido porque ele não é associado a outra variável nem usado como parte de uma expressão.

7.3 FUNÇÕES RETORNANDO VALORES NÃO-INTEIROS

Quando o tipo de uma função não é explicitamente declarado, ele é automaticamente definido como ***int***.

Como exemplo introdutório de uma função retornando um tipo não-inteiro, aqui está um programa que usa a função ***sum( )***, que retorna um valor do tipo ***double*** que é a soma de seus dois argumentos ***double***:

```
#include <stdio.h>

double sum(double a, double b);
/* Declaração inicial necessária quando a função não retorna int.
Aqui está sendo definido o protótipo da função. Dessa forma, a
função sum poderá ser escrita após a função main(), sem gerar
erro de compilação e/ou execução. */

void main() {
   double primeiro, segundo;

   primeiro = 1023.23;
   segundo = 990.0;
   printf("%lf", sum(primeiro, segundo));
}

double sum(double a, double b){  //retorna um float
   return a + b;
}
```

O protótipo antes de ***main( )*** diz ao compilador que a função ***sum( )*** retornará um tipo de dado ***double*** em ponto flutuante.

Para um exemplo mais prático, aqui está um programa que calcula a área de um círculo, dado o seu raio:

```
#include <stdlib.h>
#include <math.h>

float area(float raio); //protótipo da função area

void main(){
   float r,res;

   printf("Informe o raio: ");
   scanf("%f", &r);
   res = area(r);
   printf("A área é: %f\n", res);
}

float area(float raio){
   return 3.14159265 * pow(raio, 2);
}
```

Observação: A declaração dos argumentos no protótipo antes de ***main( )*** é opcional.

7.3.1 FUNÇÕES QUE RETORNAM PONTEIROS

Ainda que funções que retornam ponteiros sejam manipuladas exatamente como qualquer outro tipo de função, uns poucos conceitos importantes precisam ser discutidos.

Por exemplo, aqui está uma função que retorna um ponteiro em uma string no lugar onde uma correspondência de um caractere é encontrada:

```
/* Retorna um ponteiro para o primeiro caractere em s que
   corresponde a c */

char *corresp (char c, char *s){
   int count = 0;

   //verifica uma correspondência ou o terminador nulo
   while ((c != s[count]) && (s[count] != '\0')) count++;

   //se houver correspondência, retorna o ponteiro para loca-
   //zação; caso contrário, retorna um ponteiro nulo
   if (s[count]) return (&s[count]);
else return (char *) '\0');
}
```

A função ***corresp( )*** tentará retornar um ponteiro para o lugar na string onde a primeira correspondência com ***c*** é encontrada. Se nenhuma correspondência é encontrada, um ponteiro para o terminador nulo será retornado. Um pequeno programa que usa a função ***corresp( )*** é mostrado aqui:

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

char *corresp(); //protótipo da função corresp()

void main(){
   char alfa[80];
   char ch, *p;

   printf("\nDigite uma string: ");
   gets(alfa);
   printf("Digite um caractere: ");
   ch = getche();
   p = corresp(ch, alfa);

   if (p)  //existe uma correspondência
      printf("%s", p);
   else printf("\nNenhuma correspondência encontrada.");
}
```

Esse programa lê uma string e um caractere. Se o caractere está na string, o programa imprime a string a partir do ponto em que houve a correspondência. Caso contrário, ele imprime a mensagem “nenhuma correspondência encontrada”. Por exemplo, se você digitar “Olá a todos” e ***t*** como caractere, o programa responderá: “todos”.

7.3.2 FUNÇÕES DO TIPO void

Funções que não retornam valor podem sem declaradas como ***void***. Fazendo isso, impede-se o seu uso em qualquer expressão e ajuda a evitar um uso incorreto acidental. Por exemplo, a função ***imprime_vertical( )*** imprime o seu argumento string verticalmente no lado da tela. Uma vez que não retorna um valor, ela é declarada como ***void***.

```
void imprime_vertical (char *s){
   while (*str) printf (“%c\n”, *str++);
}
```

Você deve incluir um protótipo para a função ***imprime_vertical( )*** em qualquer programa que for usá-la. Se não o fizer, o compilador C assumirá que ela está retornando um inteiro. Assim, quando o compilador encontrar a função, ele emitirá a mensagem de tipo sem correspondência. Este programa mostra um exemplo correto do uso desta função:

```
#include <stdio.h>
void imprime_vertical(char *str);

void main(){
   imprime_vertical (“Alô!”);
}

void imprime_vertical (char *str){
   while (*str) printf(“%c\n”, *str++);
}
```

7.4 REGRAS DE ESCOPO DE FUNÇÕES

As ***regras de escopo*** de uma linguagem são aquelas que governam se um pedaço de código conhece ou tem acesso a outro pedaço de código ou dado.

Na linguagem C, cada função é um bloco de código discreto. Um código de uma função é privativo daquela função e não pode ser acessado por qualquer declaração em qualquer outra função, exceto por uma chamada àquela função.O código que compreende o corpo de uma função está escondido do resto do programa e, a menos que use variáveis ou dados globais, ele não afetará nem poderá ser afetado por outras partes do programa, exceto quando especificado. Em outras palavra, o código e o dado definidos dentro de uma função não podem interagir com o código ou o dado definidos em outra função, a menos que explicitamente especificado, já que as duas funções têm um escopo diferente.

Existem três tipos de variáveis: ***variáveis locais***, ***parâmetros*** e ***variáveis globais***. As regras de escopo comandam como cada um desses tipos pode ser acessado por outras partes do seu programa e os seus tempos de vida. A discussão seguintes aborda mais detalhadamente as regras de escopo.

7.4.1 VARIÁVEIS LOCAIS

Variáveis declaradas dentro de uma função são chamadas de ***variáveis locais***. Entretanto, a linguagem C suporta um conceito mais largo de variáveis locais do que o visto até agora. Uma variável pode ser declarada dentro de qualquer bloco de código e ser local a ele. As variáveis locais só podem ser referenciadas por comandos que estão dentro do bloco onde as variáveis são declaradas. Definindo de outra maneira, as variáveis locais não são conhecidas fora do seu próprio bloco de código; o seu escopo é limitado ao bloco em que elas são declaradas. Lembre-se de que um bloco de código é iniciado quando uma chave é aberta e finalizado quando a chave é fechada.

Uma das coisas mais importantes para se entender sobre as variáveis locais é que elas existem somente enquanto o bloco de código onde são declaradas está sendo executado. Isto é, uma variável é criada na entrada do seu bloco e destruída na saída.

O bloco de código mais comum, onde variáveis locais são declaradas, é a função. Por exemplo, considere as duas funções seguintes:

```
void fund1(void){
   int x;

   x = 10;
}

void func2(void){
   int x;

   x = -199;
}
```

A variável inteira foi declarada duas vezes, uma vez em ***func1( )*** e uma vez em ***func2( )***. O ***x*** em ***func1( )*** não tem relação em, ou relacionamento com, o ***x*** na ***func2( )***. O motivo é que cada ***x*** conhecido somente no código que está dentro do mesmo bloco da declaração da variável.

Para demonstrar isso, considere este pequeno programa:

```
#include <stdio.h>

void f(void);

void main(){
   int x = 10;
```

```
   printf (“x no main é %d\n”, x);
   f();
   printf (“x no main é %d\n”, x);
}

void f(void){
   int x = 100;

   printf (“O x em f() é %d\n”, x);
}
```

É prática comum declarar todas as variáveis necessárias a uma função no começo do bloco de código daquela função. Isso é feito geralmente para facilitar a qualquer um que esteja lendo o código saber quais variáveis são usadas. Entretanto, isso não é necessário, já que as variáveis locais podem ser declaradas no início de qualquer bloco de código. Para entender como funcionada, considere a seguinte função:

```
void f(void){
   char ch; //esta variável é local na função f()

   printf (“Continuar (s/n): ”);
   ch = getche();

   if (ch == ‘s’){
      char s[80];  //esta variável é local neste bloco

      printf (“Informe o nome: ”);
      gets(s);
      process_it(s);  //função que realizaria algum processo
   }
}
```

Aqui a variável local ***s*** será criada na entrada do bloco de código ***if*** e destruída na saída. Além disso, ***s*** somente é conhecida dentro do bloco ***if*** e não pode ser referenciada em outro lugar – mesmo em outras partes da função que a contém.

Como as variáveis locais são criadas e destruídas a cada entrada e saída de um bloco, os seus conteúdos são perdidos quando o bloco é deixado. Isso deve ser especialmente lembrado quando se considera uma chamada de função. Ao chamar uma função, suas variáveis locais são criadas e, no seu retorno, destruídas. Isso significa que variáveis locais não podem reter os seus valores entre chamadas.

A menos que seja predeterminado de outra forma, as variáveis locais são armazenadas na pilha. O fato da pilha ser uma região de memória dinâmica e em constante mudança explica porque as variáveis locais não podem, em geral, reter seus valores entre chamadas de funções.

7.4.2 PARÂMETROS FORMAIS

Se uma função usa argumentos, ela deve declarar as variáveis que receberão os valores daqueles argumentos. Fora o fato de receberem argumentos de entrada de uma função, elas comportam-se como quaisquer outras variáveis locais dentro da função. Essas variáveis são chamadas de ***parâmetros formais*** das funções.

LEMBRE-SE: Certifique-se que os parâmetros formais declarados são do mesmo tipo que os argumentos que você usou para chamar a função. Além disso, ainda que essas variáveis realizem a tarefa especial de receber os valores dos argumentos passados para a função, elas podem ser usadas como qualquer outra variável local.

7.4.3 VARIÁVEIS GLOBAIS

Ao contrário das variáveis locais, as ***variáveis globais*** são conhecidas por meio do programa inteiro e podem ser usadas por qualquer pedaço de código. Essencialmente, seu escopo é global ao programa. Além disso, elas manterão seus valores durante a execução inteira do programa. Variáveis globais são criadas declarando-as fora de qualquer função. Elas podem ser acessadas por qualquer expressão, não importando em qual função esteja.

No programa seguintes, você pode ver que a variável ***count*** foi declarada fora de qualquer função. A sua declaração foi definida antes da função ***main( )***. Porém, ela poderia ter sido colocada em qualquer lugar, contando que não fosse em uma função anterior à sua primeira utilização. As práticas comuns ditam que é melhor declarar variáveis globais no topo do programa.

```
#include <stdio.h>
   int count;  //count é uma variável global
   void func1(void), func2(void); //protótipos de funções

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

void func1(void){
   func2();
   printf (“O valor de count é %d”, count); //imprimirá 100
}

void func2(void){
   int count;

   for (count = 1; count < 10; count++) printf (“.”);
}
```

Olhando este programa mais de perto, deve ficar claro que, embora a

variável ***count*** não tenha sido declarada nem em ***main( )*** nem em ***func1( )***, as duas funções podem usa-la. Entretanto, ***func2( )*** tem declarado uma variável local chamada ***count***. Quando referencia a variável ***count***, ***func2( )*** está referenciando somente a sua variável local, não a global. É muito importante lembrar que, se uma variável global e uma local tiverem o mesmo nome, todas as referências àquele nome de variável dentro da função onde a variável local é declarada se referirão somente a ela e não terão efeito na variável global. Isso pode ser uma conveniência, mas esquecer-se dessa característica pode levar o seu programa a agir de maneira estranha, mesmo que pareça correto.

Variáveis globais são muito úteis quando o mesmo dados é usado em muitas funções no seu programa. Entretanto, deve-se evitar a utilização de variáveis globais desnecessárias por três razões:

- Elas ocupam a memória durante todo o tempo em que o programa está executando, não somente quando necessário;
- Usar uma variável global onde haverá uma variável local torna menos geral uma função, porque ela vai depender de algo que deve ser definido fora dela;
- A utilização de um grande número de variáveis globais pode levar o projeto a erros por causa dos efeitos colaterais desconhecidos e não desejáveis.

Um dos pontos principais da linguagem estruturada é a compartimentação do código e dos dados. Na linguagem C, a compartimentação é obtida pelo uso de funções e variáveis locais. Por exemplo, aqui estão duas maneiras de se escrever a função ***mul( )***, que calcula o produto de dois números inteiros:

Geral

```
mul (int x, int y){
   return (x * y);
}
```

Específico

```
int x, y;
int mul(){
   return (x * y);
}
```

As duas funções retornam o produto das duas variáveis ***x*** e ***y***. Entretanto, a versão generalizada, ou ***parametrizada***, pode ser usada para retornar o produto de ***quaisquer*** dois números, enquanto a versão específica pode ser usada somente para encontrar o produto das variáveis globais ***x*** e ***y***.

O exemplo a seguir demonstra o escopo de variáveis:

```
/* Um programa com vários escopos */

#include <stdio.h>
#include <string.h>

int count; //variável global – reconhecida em todo o programa
void exec (char * p); //protótipo de uma função

void main(){
   char str[80];  //variável local na função main()

   printf ("Informe uma string: ");
   gets (str);
```

```
   exec (str);
}

void exec (char *p){ //p é uma variável local na função exec
   if (!strcmp (p, "soma")){
      int a, b;  //a e b são variáveis locais neste bloco if

      printf("\nInforme dois inteiros: ");
      scanf ("%d%d", &a, &d);
      printf ("%d\n", a + b);
   }
   else{  //int a, b não são conhecidas neste bloco
      if (!strcmp(p, "beep")) printf ("%c", 7);
   }
}
```

7.5 ARGUMENTOS E PARÂMETROS DE FUNÇÕES

7.5.1 CHAMADA POR VALOR, CHAMADA POR REFERÊNCIA

Podemos passar argumentos para sub-rotinas de uma entre duas maneiras. A primeira é chamada de ***chamada por valor*** (ou passagem de parâmetro por cópia). Esse método copia o ***valor*** de um argumento para um parâmetro formal de uma sub-rotina. Assim, as mudanças feitas nos parâmetros na sub-rotina não têm efeito nas variáveis usadas para chamá-las.

A ***chamada por referência*** (passagem de parâmetro por referência) é a segunda maneira. Nesse método, o ***endereço*** de um argumento é copiado no seu parâmetro. Dentro da sub-rotina, o endereço é usado para acessar o argumento usado na chamada. Isso significa que as mudanças feitas nos parâmetros afetarão a variável usada para chamar a rotina.

A linguagem C usa chamada por valor para passar argumentos. Isso quer dizer que não se pode alterar as variáveis usadas para chamar a função. Considere a seguinte função:

```
#include <stdio.h>

int sqr(int x){
   x = x * x;
   return(x);
}

void main(){
   int t = 10;

   printf ("%d %d", sqr(t), t);
}
```

Nesse exemplo, o valor do argumento para ***sqr( )***, 10, é copiado no parâmetro ***x***. Quando a atribuição ***x = x * x*** é executada, a única coisa modificada é a variável local ***x***. A variável ***t***, usada para chamar ***sqr( )***, ainda terá o valor 10. Por isso o resultado será 100 10.

LEMBRE-SE: É uma cópia do valor do argumento que é passado para uma dada função. O que ocorre dentro da função não terá efeito na variável usada na chamada.

7.5.2 CRIANDO UMA CHAMADA POR REFERÊNCIA

Ainda que a convenção de passagem de parâmetros da linguagem C seja a chamada por valor, é possível criar uma chamada por referência passando-se um ponteiro para o argumento. Como isso fará o endereço do argumento ser passado para a função, será possível mudar o valor do argumento fora da função.

Os ponteiros são passados para funções como qualquer outro valor. Obviamente é necessário declarar o parâmetro como um ponteiro. Por exemplo, considere a função ***troca( )***, que intercambia o valor dos seus dois argumentos inteiros, mostrada aqui:

```
void troca(int *x, int *y){
   int temp;

   temp = *x;  //salva o valor armazenado no endereço x
   *x = *y;    //coloca y em x
   *y = temp;  //coloca x em y
}
```

O operador * é usado para acessar a variável apontada pelo seu operando. Por isso, o conteúdo das variáveis usadas para chamar a função será trocado.

É importante lembrar que a função ***troca( )*** deve ser chamada com o ***endereço do argumento***. Esse programa mostra a maneira correta de chamar a função ***troca( )***:

```
#include <stdio.h>
void troca (int *x, int *y){
   int temp;

   temp = *x;
   *x = *y;
   *y = temp;
}

void main(){
   int x,y;

   printf("Digite dois números: ");
```

```
   scanf("%d%d", &x, &y);
   printf("Valores iniciais: x = %d y = %d\n", x, y);
   troca(&x, &y);
   printf("Valores trocados: x = %d y = %d\n", x, y);
}
```

Nesse exemplo, a função troca é chamada como o endereço de ***x*** e ***y***. O operador unário ***&*** é usado para obter o endereço das variáveis. Desse modo, os endereços ***x*** e ***y***, e não os seus valores, são passados para a função ***troca( )***.

Com isto, você deve ter entendido porque tinha de ser colocado o operador ***&*** na frente dos argumentos da função ***scanf( )***, que receberam valores. Realmente, você estava passando os endereços para que a variável de chamada pudesse ser modificada.

7.5.3 CHAMANDO FUNÇÕES COM MATRIZES

Quando uma matriz é usada como argumento para uma função, somente o endereço da matriz é passado para a função e não a matriz inteira. Em outras palavras quando chamamos uma função com um argumento matriz, um ponteiro para o primeiro elemento da matriz é passado para a função. Lembre-se que em C um nome de matriz/vetor sem qualquer índice é um ponteiro para o primeiro elemento da matriz/vetor. Isto significa que a declaração do parâmetro deve ser de um tipo ponteiro compatível.

Existem três maneiras de se declarar um parâmetro que irá receber um ponteiro de uma matriz. Primeiro, ele pode ser declarado como uma matriz do mesmo tipo e tamanho daquela usada para chamar a função, como mostrado aqui:

```
#include <stdio.h>
void exibe(int num[100]); //protótipo de função

void main(){
   int t[10], i;

   for (i = 0; i < 10; ++i) t[i] = i;
   exibe(t);
}

void exibe(int num[10]){
   int i;

   for (i = 0; i < 10; i++) printf("%d ", num[i]);
}
```

Ainda que o parâmetro ***num*** seja declarado uma matriz de 10 elementos, o compilador o converterá automaticamente em um ponteiro para um inteiro. Isso é necessário porque nenhum parâmetro pode receber uma matriz inteira. Uma vez que somente um ponteiro para matriz será passado, um parâmetro ponteiro deve estar lá para recebe-lo.

A segunda maneira de declarar um parâmetro para uma matriz é especificando-o como uma matriz sem tamanho, como mostrado aqui:

```
void imprime(int num[]){
   int i;

   for (i = 0; i < 10; i++) printf("%d ", num[i]);
}
```

Aqui, ***num*** é declarado como uma matriz de inteiros de tamanho desconhecido. Uma vez que a linguagem C não provê checagem de limites em matrizes, o tamanho real da matriz é irrelevante para o parâmetro (mas não para o programa, obviamente). Esse método de declaração define a variável ***num*** como um ponteiro inteiro.

O modo final como a variável ***num*** pode ser declarada, e a forma mais comum em programas profissionais escritos na linguagem C, é um ponteiro, como mostrado aqui:

```
void imprime(int *num){
   int i;

   for (i = 0; i < 10; i++) printf("d ", num[i]);
}
```

Isso é permitido já que qualquer ponteiro pode ser indexado usando-se ***[ ]*** como se fosse numa matriz.

Reconheça que todos os três métodos de declaração de parâmetro matriz produzirão o mesmo resultado: um ponteiro.

É importante entender que, quando uma matriz é usada como argumento de função, seu endereço é passado para a função. Essa é uma exceção para a convenção de passagem de parâmetro por chamada de valor da linguagem C. Isso significa que o código na função estará operando sobre, e potencialmente alterando, o conteúdo atual da matriz usada para chamar a função. Por exemplo, considere a função ***imprime_maiúscula( )***, que imprime o seu argumento string em letra maiúscula.

```
/* Imprime uma string em letra maiúscula*/
#include <stdio.h>
#include <ctype.h>

void main(){
   char s[80];

   printf("informe uma string: ");
   gets(s);
   imprime_maiuscula(s);
   printf("\nA string original é alterada: %s", s);
}
```

```
void imprime_maiuscula(char *string){
   int t;

   for (t = 0; string[t]; ++t) {
      string[t] = toupper(string[t]);
      printf("%c", string[t]);
   }
}
```

Depois de chamada à função ***imprime_maiúscula( )***, o conteúdo da matriz ***s*** em ***main( )*** será mudado para letras maiúsculas. Se você não quer que isso aconteça, pode escrever assim:

```
/* Imprime uma string em letra maiúscula */
#include <stdio.h>
#include <ctype.h>

void main(){
   char s[80];

   printf("informe uma string: ");
   gets(s);
   imprime_maiuscula(s);
   printf("\na string original nao é modificada: %s",s);
 }

void imprime_maiuscula(char *string){
   int t;

   for(t = 0; string[t]; ++t) {
      printf("%c", toupper(string[t]));
   }
}
```

Nessa versão, o conteúdo da matriz ***s*** permanece inalterado uma vez que os seus valores são os mesmos.

Um exemplo clássico de passagens de matrizes em funções é encontrado na função de biblioteca ***strcat( )***. Ainda que ***strcat( )***, na biblioteca padrão, seja algo diferente, a função mostrada aqui lhe dará uma idéia de como ela funciona. Para evitar confusão com a função-padrão, ela será chamada ***strcat2( )***.

```
/* Demonstração da função strcat2() */
#include <stdio.h>

char *strcat2(char *s1, char *s2){
   char *temp;

   temp = s1;
```

```
   while(*s1) s1++;  //encontra o final de s1

   while(*s2){ //adiciona s2
      *s1 = *s2;
      s1++;
      s2++;
   }

   *s1 = '\0'; //acrescenta o terminador nulo
   return temp;
}

void main(){
   char s1[80], s2[80];

   printf("Informe duas strings:\n");
   gets(s1);
   gets(s2);
   strcat2(s1, s2);
   printf("Concatenado: %s", s1);
}
```

A função ***strcat2( )*** deve ser chamada com duas matrizes de caracteres, que, por definição, são ponteiros para caracteres. Logo na entrada, a função ***strcat2( )*** busca o final da primeira string, onde acrescenta a segunda string. Depois, é colocado o terminador nulo na primeira string.

7.5.4 OS ARGUMENTOS argc, argv PARA A FUNÇÃO main( ):

Algumas vezes, é muito útil passar informações para um programa quando ele é executado. O método geral é passar informações para a função ***main( )*** por meio do uso de ***argumentos de linha de comando***. Um argumento de linha de comando é a informação que segue o nome do programa na linha de comando do sistema operacional.

Existem parâmetros internos na função ***main( )***: ***argc*** e ***argv***. Estes parâmetros são usados para receber argumentos da linha de comando.

O parâmetro ***argc*** armazena o número de argumentos digitados na linha de comando e é um inteiro. Ele sempre será, no mínimo, um, já que o próprio nome do programa é considerado como o primeiro argumento. O parâmetro ***argv*** é um ponteiro para uma matriz de strings. Cada elemento desta matriz unidimensional aponta para um argumento digitado na linha de comando.

Todos os argumentos de linha de comando são strings - qualquer número digitado deverá ser convertido por ***atof( )***, ***atoi( )*** ou ***atol( )***.

O pequeno programa a seguir imprimirá na tela a string "Alô" seguido pelo seu nome, se você a digitar corretamente depois do nome do programa:

```
#include <stdio.h>
#include <process.h>
```

```
void main(int argc, char *argv){
   if (argc != 2){
      printf (“Você se esqueceu de digitar o seu nome. \n”);
      exit (0);
   }
   printf (“Alô %s\n”, argv[1]);
}
```

Um programa interessante e útil que utiliza argumentos de linha de comando é mostrado a seguir. Ele executa uma série de comandos do Sistema Operacional, entrados na linha de comando, usando a função de biblioteca ***system( )***. Essa função executa o comando do sistema operacional correspondente ao argumento utilizado na função. Assumindo que a string contém um comando válido do sistema operacional, o dado comando é executado e, então, o programa recomeça.

```
/*Este programa executa qualquer comando do Sistema
Operacional que é especificado na linha de comando*/
#include <stdio.h>
#include <stdlib.h>

void main (int argc, char *argv){
   int i;

   for (i = 1; i < argc; i+_){
      system (argv[i]);
   }
}
```

Assumindo-se que esse programa seja chamado de ***comline***, a seguinte linha de comando faz com que os comandos do sistema operacional ***clear*** e ***ls – l***, sejam executados na seqüência.

```
comline clear ls-l
```

Basicamente ***argc*** e ***argv*** são usados para se obter os argumentos iniciais para o programa a ser executado.

Note que cada argumento da linha de comando deve ser separado do seguinte por um espaço ou uma tabulação (não são válidos a vírgula ou ponto e vírgula). Se for necessário passar um argumento de linha de comando contendo espaços, deve-se colocá-lo entre aspas.

7. 6 FUNÇÕES RECURSIVAS

Em C, as funções podem chamar a si próprias. Uma função é ***recursiva*** se um comando no corpo da função chama ela mesma. Algumas vezes chamada de ***definição circular***, a recursividade é o processo de definição de algo em termos de si mesmo.

Exemplos de recursividade existem em grande número. Uma maneira de definir um número inteiro sem sinal por meio de recursividade é utilizando-se os dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 mais ou menos outro número inteiro. Por exemplo, o número 15 é o número 7 mais o número 8; 21 é 9 mais 12 e 12 é 9 mais 3.

Para uma linguagem ser recursiva, uma função deve estar apta a chamar a si própria. O exemplo clássico de recursividade é mostrado na função ***fatorial_recursivo( )***, que calcula o fatorial de um número inteiro. O fatorial de um número ***N*** é o produto de todos os números inteiros entre 1 e ***N***. Por exemplo, o fatorial de 3 é 1 x 2 x 3, ou 6. Tanto a função ***fatorial_recursivo( )*** como sua equivalente interativa são mostradas aqui:

```
#include <stdlib.h>
#include <stdio.h>

unsigned long int fatorial_recursivo (int n){
   unsigned long int resposta;

   if ((n == 1) || (n == 0))return(1);

   resposta = n * fatorial_recursivo(n - 1);
   return(resposta);
}

void main(){
   unsigned long f;
   int n;

   printf("Digite um número: ");
   scanf("%d",&n);
   f = fatorial_recursivo(n);
   printf("O fatorial de %d é %ld\n", n, f);
}
```

```
#include <stdlib.h>
#include <stdio.h>

unsigned long int fatorial (int n){
   unsigned long int t, resposta;

   resposta = 1;
   for (t = 1; t < n; t++) resposta = resposta * t;
   return(resposta);
}

void main(){
   unsigned long f;
   int n;
```

```
   printf("Digite um número: ");
   scanf("%d",&n);
   f = fatorial(n);
   printf("O fatorial de %d é %ld\n", n, f);
}
```

Quando uma função chama a si própria, as novas variáveis locais e os argumentos são alocados na pilha, e o código da função é executado com esses novos valores a partir do início. Uma chamada recursiva não faz uma nova cópia da função. Somente os argumentos e as variáveis são novos. Quando cada chamada recursiva retorna, as antigas variáveis locais e os parâmetros são removidos da pilha e a execução recomeça no ponto de chamada da função dentro da função.

Tendo em vista que o local para os argumentos de funções e para as variáveis locais é a pilha e que a cada nova chamada é criado uma cópia destas variáveis na pilha, é possível ocorrer overflow da pilha (stack overflow) e o programa terminar com um erro.

UNIDADE 8 – ARQUIVOS

8.1 INTRODUÇÃO

Na linguagem C, um arquivo é um conceito lógico que pode ser aplicado a tudo, desde arquivos em disco até terminais. Um fluxo é associado a um arquivo específico pela realização de uma operação de abertura. Uma vez que um arquivo esteja aberto, informações podem ser intercambiadas entre o arquivo e o seu programa.

Nem todos os arquivos têm as mesmas capacidades. Por exemplo, um arquivo em disco pode suportar acesso randômico enquanto um acionador de fita não pode. Isso ilustra um ponto importante sobre o sistema de E/S da linguagem C: todos os fluxos são os mesmos, mas nem todos os arquivos os são.

Se pode suportar acesso randômico, então uma abertura de um arquivo também inicializa um ***indicador de posição do arquivo*** no começo do arquivo. À medida que cada caractere é lido ou escrito para o arquivo, o indicador de posição é incrementado, assegurando, assim, a progressão através do arquivo.

Um arquivo é desassociado de um fluxo específico por meio de uma operação de fechamento. Em fluxos abertos para saída, o fechamento do fluxo faz com que o conteúdo, se houver, da área intermediária (buffer) seja descarregado para o dispositivo externo. Esse processo é geralmente referenciado como uma ***limpeza/descarga*** de apontador e garante que nenhuma informação será acidentalmente esquecida na área intermediária do disco. Todos os arquivos são fechados automaticamente quando o programa termina normalmente.

8.2 O PONTEIRO DE ARQUIVO

A linha comum que une o sistema de E/S de disco aos programas escritos em C é o ***ponteiro de arquivo***. Um ponteiro de arquivo é um ponteiro para uma área na memória (buffer) onde estão contidos vários dados sobre o arquivo a ler ou escrever, tais como o nome do arquivo, estado e posição corrente. O ***buffer*** apontado pelo ponteiro de arquivo é a área intermediária entre o arquivo no disco e o programa.

Este buffer intermediário entre arquivo e programa é chamado “fluxo”, e no jargão dos programadores é comum falar em funções que operam fluxos em vez de arquivos. Isto se deve ao fato de que um fluxo é uma entidade lógica genérica, que pode estar associada a uma unidade de fita magnética, um disco, uma porta serial, etc. Adotaremos esta nomenclatura aqui.

Um ponteiro de arquivo é uma variável ponteiro do tipo ***FILE*** que é definida em ***stdio.h***. Para ler ou escrever em um arquivo de disco, o programa deve declarar uma (ou mais de uma se formos trabalhar com mais de um arquivo simultaneamente) variável ponteiro de arquivo. Para obter uma variável ponteiro de arquivo, usa-se uma declaração semelhante a esta:

```
FILE *fp;
```

onde ***fp*** é o nome que escolhemos para a variável (podia ser qualquer outro).

As funções mais comuns do sistema de arquivo são:

As Funções mais Comuns do Sistema de Arquivo ANSI

Função	Operação
fopen( )	Abre um fluxo
fclose( )	Fecha um fluxo
putc( )	Escreve um caractere para um fluxo
getc( )	Lê um caractere para um fluxo
fseek( )	Procura por um byte especificado no fluxo
fprintf( )	É para um fluxo aquilo que printf( ) é para o console
fscanf( )	É para um fluxo aquilo que scanf( ) é para o console
feof( )	Retorna verdadeiro se o fim do arquivo é encontrado
ferror( )	Retorna verdadeiro se ocorreu um erro
fread( )	Lê um bloco de dados de um fluxo
fwrite( )	Escreve um bloco de dados para um fluxo
rewind( )	Reposiciona o localizador de posição de arquivo no começo do arquivo
remove( )	Apaga um arquivo

8.3 ABRINDO UM ARQUIVO

A função ***fopen( )*** serve a dois propósitos. Primeiro, ela abre um fluxo para uso e liga um arquivo com ele. Segundo, retorna o ponteiro de arquivo associado àquele arquivo. Mais freqüentemente, e para o resto desta discussão, o arquivo é um arquivo em disco. A função ***fopen( )*** tem este protótipo:

```
FILE *fopen(char *nome_de_arquivo, char *modo);
```

onde modo é uma string contendo o estado desejado para abertura. O nome do arquivo deve ser uma string de caracteres que compreende um nome de arquivo válido para o sistema operacional e onde possa ser incluída uma especificação de caminho (PATH).

Como determinado, a função ***fopen( )*** retorna um ponteiro de arquivo que não deve ter o valor alterado pelo seu programa. Se um erro ocorre quando se está abrindo um arquivo, ***fopen( )*** retorna um nulo.

Como a tabela abaixo mostra, um arquivo pode ser aberto ou em modo texto ou em modo binário. No modo texto, as seqüências de retorno de carro e alimentação de formulários são transformadas em seqüências de novas linhas na entrada. Na saída, ocorre o inverso: novas linhas são transformadas em retorno de carro e alimentação de formulário. Tais transformações não acontecem em um arquivo binário.

Os Valores Legais para Modo

Modo	Significado
“r”	Abre um arquivo para leitura
“w”	Cria um arquivo para escrita
“a”	Acrescenta dados para um arquivo existente
“rb”	Abre um arquivo binário para leitura
“wb”	Cria um arquivo binário para escrita
“ab”	Acrescenta dados a um arquivo binário existente
“r+”	Abre um arquivo para leitura/escrita

Modo	Significado
“w+”	Cria um arquivo para leitura/escrita
“a+”	Acrescenta dados ou cria um arquivo para leitura/escrita
“r+b”	Abre um arquivo binário para leitura/escrita
“w+b”	Cria um arquivo binário para leitura/escrita
“a+b”	Acrescenta ou cria um arquivo binário para leitura/escrita
“rt”	Abre um arquivo texto para leitura
“wt”	Cria um arquivo texto para escrita
“at”	Acrescenta dados a um arquivo texto
“r+t”	Abre um arquivo-texto para leitura/escrita
“w+t”	Cria um arquivo texto para leitura/escrita
“a+t”	Acrescenta dados ou cria um arquivo texto para leitura/escrita

Se você deseja criar um arquivo para escrita com o nome “teste” você deve escrever:

```
FILE *fp;
fp = fopen (“teste”, “w”);
```

Entretanto, você verá freqüentemente escrito assim:

```
FILE *fp;
if((fp = fopen(“test”, “w”)) == NULL){
   puts(“Não posso abrir o arquivo\n”);
   exit(1);
}
```

A macro NULL é definida em ***stdio.h***. Esse método detecta qualquer erro na abertura do arquivo como um arquivo protegido contra escrita ou disco cheio, antes de tentar escrever nele. Um nulo é usado porque no ponteiro do arquivo nunca haverá aquele valor. Também introduzido por esse fragmento está outra função de biblioteca: ***exit( )***. Uma chamada à função ***exit( )*** faz com que haja uma terminação imediata do programa, não importando de onde a função ***exit( )*** é chamada. Ela tem este protótipo (encontrado em ***stdlib.h***):

```
void exit(int val);
```

O valor de ***val*** é retornado para o sistema operacional. Por convenção, um valor de retorno ***0*** significa término com sucesso para o sistema operacional. Qualquer outro valor indica que o programa terminou por causa de algum problema.

Se você usar a função ***fopen( )*** para criar um arquivo, qualquer arquivo preexistente com o mesmo nome será apagado e o novo arquivo iniciado. Se não existirem arquivos com o nome, então será criado um. Se você quiser acrescentar dados ao final do arquivo, deve usar o modo ***“a”***. Para se abrir um arquivo para operações de leitura é necessário que o arquivo já exista. Se ele não existir, será retornado um erro. Finalmente, se o arquivo existe e é aberto para escrita/leitura, as operações feitas não apagarão o arquivo; entretanto, se não existir, o arquivo deverá ser criado.

8.4 ESCREVENDO UM CARACTERE NUM ARQUIVO

A função ***putc( )*** é usada para escrever caracteres para um fluxo que foi previamente aberto para escrita pela função ***fopen( )***. A função é declarada como:

```
int putc(int ch, FILE *fp);
```

onde ***fp*** é o ponteiro de arquivo retornado pela função ***fopen( )*** e ***ch***, o caractere a ser escrito. Por razões históricas, ***ch*** é chamado formalmente de ***int***, mas somente o caractere de mais baixa ordem é usado.

Se uma operação com a função ***putc( )*** for satisfatória, ela retornará o caractere escrito. Em caso de falha, um ***EOF*** é retornado. ***EOF*** é uma macro definição em ***stdio.h*** que significa fim do arquivo.

8.5 LENDO UM CARACTERE DE UM ARQUIVO

A função ***getc( )*** é usada para ler caracteres do fluxo aberto em modo de leitura pela função ***fopen( )***. A função é declarada como

```
int getc(FILE *fp)
```

onde ***fp*** é um ponteiro de arquivo do tipo ***FILE*** retornado pela função ***fopen( )***.

Por razões históricas, a função ***getc( )*** retorna um inteiro, porém o byte de mais alta ordem é zero.

A função ***getc( )*** retornará uma marca ***EOF*** quando o fim do arquivo tiver sido encontrado ou um erro tiver ocorrido. Portanto, para ler um arquivo-texto até que a marca de fim de arquivo seja mostrada, você poderá usar o seguinte código:

```
ch = getc(fp);

while (ch != EOF){
   ch = getc(fp);
}
```

8.6 USANDO A FUNÇÃO feof( )

A linguagem C também pode operar com dados binários. Quando um arquivo é aberto para entrada binária, é possível encontrar um valor inteiro igual à marca de EOF. Isso pode fazer com que, na rotina anterior, seja indicada uma condição de fim de arquivo, ainda que o fim de arquivo físico não tenha sido encontrado. Para resolver esse problema, foi incluída a função ***feof( )*** que é usada para determinar o final de um arquivo quando da leitura de dados binários. A função ***feof( )*** tem este protótipo:

```
int feof(FILE *fp);
```

O seu protótipo está em ***stdio.h***. Ela retorna verdadeiro se o final do arquivo tiver sido encontrado; caso contrário, é retornado um zero. Portanto, a

seguinte rotina lê um arquivo binário até que o final do arquivo seja encontrado.

```
while (!feof(fp)) ch = getc(fp);
```

Obviamente, o mesmo método pode ser aplicado tanto a arquivos textos como a arquivos binários.

8.7 FECHANDO UM ARQUIVO

A função ***fclose( )*** é usada para fechar um fluxo que foi aberto por uma chamada à função ***fopen( )***. Ela escreve quaisquer dados restantes na área intermediária (buffer) no arquivo e faz um fechamento formal em nível de sistema operacional do arquivo. Uma falha no fechamento de um arquivo leva a todo tipo de problemas, incluindo-se perda de dados, arquivos destruídos e a possibilidade de erros no seu programa.

A função ***fclose( )*** é declarada como:

```
int fclose(FILE *fp);
```

onde ***fp*** é o ponteiro do arquivo retornado pela chamada à função ***fopen( )***. Um valor de retorno igual a zero significa uma operação de fechamento com sucesso; qualquer outro valor indica um erro. Você pode usar a função padrão ***ferror( )*** para determinar e reportar quaisquer problemas. Geralmente, o único momento em que a função ***fclose( )*** falhará é quando um disquete tiver sido prematuramente removido do drive ou não houver mais espaço no disco.

8.8 ferror() E rewind()

A função ***ferror( )*** é usada para determinar se uma operação em um arquivo produziu um erro. Se um arquivo é aberto em modo texto e ocorre um erro na leitura ou na escrita, é retornado ***EOF***. Você usa a função ***ferror( )*** para determinar o que aconteceu. A função ***ferror( )*** tem o seguinte protótipo:

```
int ferror(FILE *fp);
```

onde ***fp*** é um ponteiro válido para um arquivo. ***ferror( )*** retorna verdadeiro se um erro ocorreu durante a última operação com o arquivo e falso, caso contrário. Uma vez que cada operação em arquivo determina uma condição de erro, a função ***ferror( )*** deve ser chamada imediatamente após cada operação com o arquivo; caso contrário, um erro pode ser perdido. O protótipo de função ***ferror( )*** está no arquivo ***stdio.h***.

A função ***rewind( )*** recolocará o localizador de posição do arquivo no início do arquivo especificado como seu argumento. O seu protótipo é:

```
void rewind(FILE *fp);
```

onde ***fp*** é um ponteiro de arquivo. O protótipo para a função ***rewind( )*** está no arquivo ***stdio.h***.

8.9 USANDO fopen( ), getc( ), putc( ) E fclose( )

O programa abaixo lê caracteres do teclado e escreve-os para um arquivo em disco até que um sinal “$” seja digitado. O nome do arquivo é especificado na linha de comando. Por exemplo, se você chamar o programa ***ktod***, digitando ***ktod teste***, poderá gravar linhas de texto no arquivo chamado ***teste***.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]){
   FILE *fp;
   char ch;

   if(argc!=3) {
      puts(“Você não informou nome do arquivo e o modo!”);
      exit(1);
   }

   //experimente usar os modos “wb” e “wt” e teste o “Enter”
   if ((fp = fopen(argv[1], argv[2])) == NULL){
      puts(“O arquivo não pode ser aberto!”);
      exit(1);
   }

   do {         //getche() não funcionaria aqui. getchar( )
      ch = getchar(); //pega os caracteres do teclado e vai
                      //armazenando no buffer até o “Enter”
      if (EOF == putc(ch, fp)){
         puts(“Erro ao escrever no arquivo!”);
         break;
      }
   }while (ch != '$');

   fclose(fp);
}
```

O programa complementar ***dtos***, mostrado a seguir, lerá qualquer arquivo ASCII e exibirá o seu conteúdo na tela.

```
/*Um programa que lê arquivos e exibe-os na tela.*/
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[]){
   FILE *fp;
   char ch;
   if(argc!=3) {
```

```
      puts("Você não informou o nome do arquivo e o modo!");
      exit(1);
   }

   if ((fp = fopen(argv[1], argv[2])) == NULL){
      //experimente os modos "rb" e "rt" e teste o "Enter"
      puts("O arquivo não pode ser aberto!");
      exit(1);
   }

   ch = getc(fp); //lê um caractere

   while (ch != EOF){
      putchar(ch); //imprime na tela
      ch = getc(fp);
   }
}
```

8.10 fgets( ) E fputs( )

Funcionam como ***gets( )*** e ***puts( )*** só que lêem e escrevem em fluxos. Os seus protótipos são:

```
char *fputs(char *str, FILE *fp);
char *fgets(char *str, int tamanho, FILE *fp);
```

A função ***fputs( )*** funciona como ***puts( )***, só que ela escreve a string em um fluxo especificado. A função ***fgets( )*** lê uma string de um fluxo especificado até que um caractere de nova linha ("\n") seja lido ou ***tamanho – 1*** caracteres tenham sido lidos. Se uma nova linha ("\n") é lida, ela fará parte da string. A string resultante terminará com nulo.

8.11 fread( ) E fwrite( ):

Esta funções permitem a leitura e a escrita de blocos de dados. Os seus protótipos são:

```
unsigned fread(void *buffer, int num_bytes, int count, FILE *fp);
unsigned fwrite(void *buffer, int num_bytes, int count, FILE *fp);
```

No caso da função ***fread( )***, ***buffer*** é um ponteiro para uma região de memória que receberá os dados lidos do arquivo.

No caso de ***fwrite( )***, ***buffer*** é um ponteiro para uma região de memória onde se encontram os dados a serem escritos no arquivo. O ***buffer*** usualmente é um ponteiro para uma matriz ou uma estrutura.

O número de bytes a ser lido ou escrito é especificado por ***num_bytes***. O argumento ***count*** determina quantos itens (cada um tendo ***num_bytes*** de tamanho)

serão lidos ou escritos. Finalmente, ***fp*** é um ponteiro para um arquivo de um fluxo previamente aberto por ***fopen( )***.

A função ***fread( )*** retorna o número de itens lidos, que pode ser menor que ***count*** caso o final de arquivo seja encontrado ou ocorra um erro.

A função ***fwrite( )*** retorna o número de itens escritos, que será igual a ***count*** a menos que ocorra um erro.

Quando o arquivo for aberto para dados binários, ***fread( )*** e ***fwrite( )*** podem ler e escrever qualquer tipo de informação. O programa a seguir escreve um ***float*** em um arquivo de disco chamado "teste.dat".

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(){
   FILE *fp;
   float f = 3.141592654;

   if ((fp = fopen("teste.dat", "wb")) == NULL){
      puts ("Não posso abrir arquivo!");
      exit(1);
   }

   if (fwrite(&f, sizeof(float), 1, fp)!=1){
      puts("Erro escrevendo no arquivo!");
   }

   fclose(fp);
}
```

Como o programa anterior ilustra, a área intermediária de armazenamento pode ser, e freqüentemente é, simplesmente uma variável.

Uma das aplicações mais úteis de ***fread( )*** e ***fwrite( )*** envolve leitura e escrita em matrizes e estruturas. Por exemplo, este fragmento de programa escreve o conteúdo da matriz em ponto flutuante ***exemplo*** no arquivo "exemplo.dat" usando uma simples declaração ***fwrite( )***.

```
#include <stdio.h>
#include <stdlib.h>

void main(){
   FILE *fp;
   float exemplo[10][10];
   int i, j;

   if ((fp = fopen("exemplo.dat", "wb")) == NULL){
      puts("Não posso abrir arquivo!");
      exit(1);
```

```
   }

   for (i = 0; i < 10; i++){
      for (j = 0; j < 10; j++){
         exemplo[i][j] = (float) i+j;
      }
   }

   //O código a seguir grava a matriz inteira num único passo
   if (fwrite(exemplo, sizeof(exemplo), 1, fp) != 1){
      puts("Erro ao escrever arquivo!");
      exit(1);
   }

   fclose(fp);
}
```

Note como ***sizeof*** é usado para determinar o tamanho da matriz ***exemplo***.

O próximo exemplo usa a função ***fread( )*** para ler a informação escrita pelo programa anterior. Ele exibe os números na tela para verificação.

```
#include <stdio.h>
#include <stdlib.h>

void main(){
   FILE *fp;
   float exemplo[10][10];
   int i,j;

   if ((fp = fopen("exemplo.dat", "rb")) == NULL){
      puts("Não posso abrir arquivo!");
      exit(1);
   }

   //O código a seguir lê a matriz inteira num único passo
   if (fread(exemplo, sizeof(exemplo), 1, fp) != 1){
      puts("Erro ao ler arquivo!");
      exit(1);
   }

   for (i = 0; i < 10; i++){
      for (j = 0; j < 10; j++){
         printf ("%3.1f ", exemplo[i][j]);
         printf("\n");
      }
   }
```

```
   fclose(fp);
}
```

8.12 ACESSO RANDÔMICO A ARQUIVOS: fseek( )

Na linguagem C é possível realizar operações de leitura e escrita randômica, com o auxílio da função ***fseek( )***, que posiciona o ponteiro do arquivo. O protótipo desta função é:

```
int fseek(FILE *fp, long numbytes, int origem);
```

Aqui, ***fp*** é um ponteiro para o arquivo retornado por uma chamada à ***fopen( )***, ***numbytes*** é o número de bytes, a partir da ***origem***, necessários para se conseguir chegar até a posição desejada.

A ***origem*** é uma das seguintes macros definidas em ***stdio.h***:

Origem	Nome da Macro	Valor atual
Começo do arquivo	SEEK_SET	0
Posição corrente	SEEK_CUR	1
Fim do arquivo	SEEK_END	2

Portanto, para procurar ***numbytes*** do começo do arquivo, ***origem*** deve ser ***SEEK_SET***. Para procurar da posição corrente em diante, use ***SEEK_CUR***. Para procurar ***numbytes*** do final do arquivo, de trás para diante, origem é ***SEEK_END***.

O seguinte fragmento lê o 235º byte do arquivo chamado “teste.num”:

```
FILE *fp;
char ch;

if ((fp = fopen("teste", "rb")) == NULL){
   puts("Não posso abrir arquivo!");
   exit(1);
}

fseek(fp, 234, SEEK_SET);
ch = getc(fp); //lê um caractere na posição 235º
```

A função ***fseek( )*** retorna zero se houve sucesso ou um valor diferente de zero se houve falha no posicionamento do ponteiro do arquivo.

8.13 fprintf( ) E fscanf( )

Estas funções comportam-se como ***printf( )*** e ***scanf( )*** só que escrevem e lêem de arquivos em disco. Todos os códigos de formato e modificadores são os mesmos. Os protótipos destas funções são:

```
int fprintf(FILE *fp, char *string_controle, lista_de_argumentos);
int fscanf(FILE *fp, char *string_controle, lista_de_argumentos);
```

Embora ***fprintf( )*** e ***fscanf( )*** sejam a maneira mais fácil de ler e escrever tipos de dados nos mais diversos formatos, elas não são as mais eficientes em termos de tamanho de código resultante e velocidade. Assim, se a velocidade e tamanho são uma preocupação, deve-se dar preferência as funções ***fread( )*** e ***fwrite( )***.

Para ilustrar o quão útil essa funções podem ser, o seguinte programa mantém uma lista de telefones em um arquivo em disco. Você pode inserir nomes e números ou verificar um número dado um nome.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "biblioteca.c" //biblioteca com funções de tela

void add_num (void);  //protótipos de funções
void consulta(void);
int menu(void);

void main(){
   char opcao;

   do{
      opcao = menu();

      switch (opcao){
         case 'A': add_num();
                   break;
         case 'C': consulta();
                   break;
      }
   }while (opcao != 'T')
}

int menu(void){
   char ch;

   do{
      printf ("(A)diciona, (C)onsulta, (T)ermina: ");
      ch = toupper(getche());
      printf("\n");
   }while (ch != 'T' && ch != 'A' && ch != 'C');

   return(ch);
}
```

```
void add_num(void){
   FILE *fp;
   Char nome[80];
   int prefixo, num;

   if (fp = fopen("fones.db", "a")) == NULL){
      printf ("O arquivo da lista não pode ser aberto!\n");
      exit(1);
   }

   printf ("Informe o nome e o número: ");
   fscanf (stdin, "%s%d%d", nome, &prefixo, &num);
   fscanf (stdin, "%*c"); //remove o CR do buffer do teclado
   fprintf (fp, "%s %d %d", nome, prefixo, num);
   fclose (fp);
}

void consulta(void){
   FILE *fp;
   char nome[80], nome2[80];
   int prefixo, num;

   if (fp = fopen("fones.db", "r")) == NULL){
      printf ("O arquivo da lista não pode ser aberto!\n");
      exit(1);
   }

   printf ("Nome? ");
   gets (nome);

   //consulta um número
    while (!feof(fp)){
      fscanf (fp, "%s%d%d", nome2, &prefixo, &num);

      if (!strcmp(nome, nome2)){
         printf ("%s: %d-%d\n", nome, prefixo, num);
         break;
      }
   }

   fclose(fp);
}
```

8.14 APAGANDO ARQUIVOS: remove( )

A função ***remove( )*** apaga o arquivo especificado. O seu protótipo é:

```
int remove(char *nome_arquivo);
```

A função retorna zero em caso de sucesso; não zero se falhar.

8.15 PROGRAMA EXEMPLO: MANIPULAÇÃO DE ARQUIVOS

O programa a seguir demonstra como utilizar as funções para manipulação de arquivos binários em C.

```
#include <stdio.h>
#include <ctype.h>
#include "biblioteca.c"

//--------------------- Estrutura Global ----------------------
struct pessoa{
   char nome[41];
   char cpf[15];
   char sexo;
   char fone[14];
   char endereco[41];
   char cidade[26];
   char uf[3];
};

//--------------------- Variável Global -----------------------
FILE *f;
//------------------ Sub-rotinas e Funções --------------------
void abrir_arquivo(){
   f = fopen ("Cadastro.db", "rb+"); //tenta abrir o arquivo
                                     //para leitura/escrita
   if (f == NULL){  //verifica se conseguiu abrir o arquivo?
      f = fopen ("Cadastro.db", "wb+");  //Se não abriu, cria.
   }
}
//-------------------------------------------------------------
void limpa(){
   char resposta;

   clrscr();
   printf ("* * *   L I M P A R   A R Q U I V O   * * *\n");
   printf ("Confirma Limpeza do Arquivo (S/N): ");
   resposta = toupper(getche());

   if (resposta == 'S'){
      fclose(f);
      remove ("Cadastro.db");
```

```
      abrir_arquivo();
   }
}
//---------------------------------------------------------------
void exclui(){
   struct pessoa reg;
   int i;
   char resposta;

   clrscr();
   printf ("* * *   E X C L U S Ã O   * * *\n");
   printf ("\nDigite o número do registro: ");
   scanf ("%d", &i);
   fseek (f, i * sizeof(struct pessoa), SEEK_SET);

   if (fread(&reg, sizeof(struct pessoa), 1, f) != 1){
      printf ("\n\nRegistro inexistente!!! Tecle <Enter>");
      fflush(stdout);
      getch();
   }
   else{
      printf ("Nome    : %s\n", reg.nome);
      printf ("CPF     : %s\n", reg.cpf);
      printf ("Sexo    : %c\n", reg.sexo);
      printf ("Fone    : %s\n", reg.fone);
      printf ("Endereço: %s\n", reg.endereco);
      printf ("Cidade  : %s\n", reg.cidade);
      printf ("Estado  : %s\n", reg.uf);
      fflush(stdout);
      printf ("\nConfirma exclusão (S/N): ");
      resposta = toupper(getche());

      if (resposta == 'S') {
         strcpy (reg.nome, "");
         strcpy (reg.cpf, "");
         reg.sexo = '\0';
         strcpy (reg.fone, "");
         strcpy (reg.endereco, "");
         strcpy (reg.cidade, "");
         strcpy (reg.uf, "");

         fseek (f, -1 * sizeof(struct pessoa), SEEK_CUR);
         fwrite (&reg, sizeof (struct pessoa), 1, f);
      }
   }
}
//---------------------------------------------------------------
```

```
void lista(){
   struct pessoa reg;
   int i, j;

   rewind (f);
   i = 0;

   while (fread(&reg, sizeof(struct pessoa), 1, f) == 1){
      clrscr();
      printf ("* * *   L I S T A G E M   G E R A L  * * *\n");
      printf ("\nRegistro %d\n\n", i);
      printf ("Nome    : %s\n", reg.nome);
      printf ("CPF     : %s\n", reg.cpf);
      printf ("Sexo    : %c\n", reg.sexo);
      printf ("Fone    : %s\n", reg.fone);
      printf ("Endereço: %s\n", reg.endereco);
      printf ("Cidade  : %s\n", reg.cidade);
      printf ("Estado  : %s\n", reg.uf);
      printf ("\n\nPressione qualquer tecla!");
      fflush(stdout);
      getch();
      i++;
   }
}
//---------------------------------------------------------------
void consulta(){
   struct pessoa reg;
   int i;

   clrscr();
   printf ("* * *   C O N S U L T A   * * *\n");
   printf ("\nDigite o número do registro: ");
   scanf ("%d", &i);
   fseek (f, i * sizeof(struct pessoa), SEEK_SET);

   if (fread(&reg, sizeof(struct pessoa), 1, f) != 1){
      printf ("\n\nRegistro inexistente!!! Tecle <Enter>.");
      fflush(stdout);
      getch();
   }
   else{
      printf ("Nome    : %s\n", reg.nome);
      printf ("CPF     : %s\n", reg.cpf);
      printf ("Sexo    : %c\n", reg.sexo);
      printf ("Fone    : %s\n", reg.fone);
      printf ("Endereço: %s\n", reg.endereco);
      printf ("Cidade  : %s\n", reg.cidade);
```

```
      printf ("Estado  : %s\n", reg.uf);
      fflush(stdout);
      printf ("\n\nPressione qualquer tecla!");
      getch();
   }
}
//----------------------------------------------------------------
void altera(){
   struct pessoa reg_lido, reg_alt;
   int i, campo;
   char temp;

   clrscr();
   printf ("* * *   A L T E R A C A O   * * *\n");
   printf ("\nDigite o número do registro: ");
   scanf ("%d", &i);
   fseek (f, i * sizeof(struct pessoa), SEEK_SET);

   if (fread(&reg_lido, sizeof(struct pessoa), 1, f) != 1){
      printf ("Registro inexistente!!! Tecle <Enter>.");
      fflush(stdout);
      getch();
   }
   else{
      printf ("Nome    : %s\n", reg_lido.nome);
      printf ("CPF     : %s\n", reg_lido.cpf);
      printf ("Sexo    : %c\n", reg_lido.sexo);
      printf ("Fone    : %s\n", reg_lido.fone);
      printf ("Endereço: %s\n", reg_lido.endereco);
      printf ("Cidade  : %s\n", reg_lido.cidade);
      printf ("Estado  : %s\n", reg_lido.uf);
      fflush(stdout);

      reg_alt = reg_lido;
      printf ("1 - Nome  2 - CPF  3 - Sexo  4 – Fone   ");
      printf ("5 - Endereço  6 - Cidade  7 – Estado");
      printf ("\n\nQual campo deseja alterar: ");
      scanf ("%d", &campo);
      temp = getchar();
      printf ("\n\n");

      switch (campo){
         case 1: printf ("Nome: ");
                 gets(reg_alt.nome);
                 break;

         case 2: printf ("CPF: ");
```

```
                 gets(reg_alt.cpf);
                 break;

         case 3: printf ("Sexo: ");
                 reg_alt.sexo = getche();
                 break;

         case 4: printf ("Fone: ");
                 gets(reg_alt.fone);
                 break;

         case 5: printf ("Endereço: ");
                 gets(reg_alt.endereco);
                 break;

         case 6: printf ("Cidade: ");
                 gets(reg_alt.cidade);
                 break;

         case 7: printf ("Estado: ");
                 gets(reg_alt.uf);
      }

      fseek (f, -1 * sizeof(struct pessoa), SEEK_CUR);
      fwrite (&reg_alt, sizeof (struct pessoa), 1, f);
   }
}
//---------------------------------------------------------------
void cadastra(){
   struct pessoa reg;

   clrscr();
   printf ("* * *   C A D A S T R O   * * *\n");
   printf ("\nNome    : ");
   gets(reg.nome);
   printf ("\nCPF     : ");
   gets(reg.cpf);
   printf ("\nSexo    : ");
   reg.sexo = getche();
   printf ("\nFone    : ");
   gets(reg.fone);
   printf ("\nEndereço: ");
   gets(reg.endereco);
   printf ("\nCidade  : ");
   gets(reg.cidade);
   printf ("\nEstado  : ");
   gets(reg.uf);
```

```
   fseek (f, 0, SEEK_END);
   fwrite (&reg, sizeof(struct pessoa), 1, f);
}
//---------------------------------------------------------------
int menu(){
   char opl[2];
   int op;

   do{
      clrscr();
      fflush(stdout);
      printf ("* * *   M E N U  * * *\n\n");
      printf ("1 - Cadastrar\n");
      printf ("2 - Alterar\n");
      printf ("3 - Consultar\n");
      printf ("4 - Listar Todos\n");
      printf ("5 - Excluir\n");
      printf ("6 - Limpar\n");
      printf ("7 - Sair\n");
      printf ("\nQual sua opcao: ");
      opl[0] = getche();
      opl[1] = '\0';
      op = atoi(opl);

      if ((op < 1) || (op > 7)){
         printf ("\nOpcao inválida !!! Tecle <Enter>.");
         op = getche();
      }

   }while ((op < 1) || (op > 7));


   return(op);
}
//---------------------------------------------------------------
int main(){
   int opcao;

   abrir_arquivo();

   do{
      opcao = menu();

      switch (opcao){
         case 1: cadastra();
                 break;
         case 2: altera();
```

```
                    break;
            case 3: consulta();
                    break;
            case 4: lista();
                    break;
            case 5: exclui();
                    break;
            case 6: limpa();
                    break;
            case 7: fclose(f);
                    clrscr();
        }
    }while (opcao != 7);
}
```

UNIDADE 9 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS EM C++

9.1 INTRODUÇÃO

A programação orientada a objetos é uma maneira de abordar a tarefa de programação. Com o decorrer dos anos os programas desenvolvidos tornaram-se por demais complexos; assim, fez-se necessário incorporar mudanças às linguagens de programação frente a esta complexidade crescente.

Nos aos 60 nasceu a programação ***estruturada***. Esse é o método estimulado por linguagens como C e Pascal. Usando-se linguagens estruturadas, foi possível, pela primeira vez, escrever programas moderadamente complexos de maneira razoavelmente fácil. Entretanto, com a programação estruturada, quando um projeto atinge um certo tamanho, ele torna-se incontrolável, pois a sua complexidade excede a capacidade dessa programação. A cada marco no desenvolvimento da programação, métodos foram criados para permitir ao programador tratar com complexidades incrivelmente grandes. Cada passo combinava os melhores elementos dos métodos anteriores com mais avanços. Atualmente, muitos projetos estão próximos ou no ponto em que o tratamento estruturado não mais funciona. Para resolver esse problema, a programação orientada a objetos foi criada.

A programação orientada a objetos aproveitou as melhores idéias da programação estruturada e combinou-as com novos conceitos poderosos para a arte da programação. A programação orientada a objetos permite que um problema seja mais facilmente decomposto em subgrupos relacionados. Então, usando-se a linguagem, pode-se traduzir esses subgrupos em unidades autocontidas chamadas ***objetos***.

Todas as linguagens de programação orientadas a objetos possuem três coisas em comum: ***objetos***, ***polimorfismo*** e ***herança***.

9.1.1 OBJETOS

A característica mais importante de uma linguagem orientada a objetos é o ***objeto***. De maneira simples, um ***objeto*** é uma entidade lógica que contém dados e o código para manipular esses dados. Dentro de um objeto, alguns códigos e/ou dados podem ser privados ao objeto e inacessíveis diretamente para qualquer elemento fora dele. Dessa maneira, um objeto evita significativamente que algumas partes não relacionadas de programa modifiquem ou usem incorretamente as partes privadas do objeto. Essa ligação dos códigos e dos dados é freqüentemente referenciada como ***encapsulação***.

Para todas as intenções e propósitos, um objeto é uma variável de um tipo definido pelo usuário. Pode parecer estranho à primeira vista encarar um objeto, que une tanto código como dados, como sendo uma variável. Contudo, em programação orientada a objetos, esse é precisamente o caso. Quando se define um objetos, cria-se implicitamente um novo tipo de dado.

9.1.2 POLIMORFISMO

Linguagens de programação orientada a objetos suportam ***polimorfismo***, o que significa essencialmente que um nome pode ser usado para muitos propósitos relacionados, mas ligeiramente diferentes. A intenção do polimorfismo é permitir a um nome ser usado para especificar uma classe geral de ações. Entretanto, dependendo do tipo de dado que está sendo tratado, uma instância específica de um caso geral é executada. Por exemplo, pode-se ter um programa que define três tipos diferentes de pilhas. Uma pilha é usada para valores inteiros, uma para valores de ponto flutuante e uma, para inteiros longos. Por causa do polimorfismo, pode-se criar três conjuntos de funções para essas pilhas, chamadas ***push( )*** e ***pop( )***, e o compilador selecionará a rotina correta, dependendo com qual tipo a função é chamada. Nesse exemplo, o conceito geral é pôr e tirar dados de e para a pilha. As funções definem a maneira específica como isso é feito para cada tipo de dado.

As primeiras linguagens de programação orientadas a objetos eram interpretadas; assim, o polimorfismo era, obviamente, suportado no momento da execução. Entretanto, o C++ é uma linguagem compilada. Portanto, em C++, é suportado tanto o polimorfismo em tempo de execução como em tempo de compilação.

9.1.3 HERANÇA

Herança é o processo em que um objeto pode adquirir as propriedades de outro objeto. Isso é importante, já que suporta o conceito de classificação. Por exemplo, uma deliciosa maça vermelha é parte da classificação ***maçã***, que por sua vez, faz parte da classificação ***frutas***, que está na grande classe ***comida***. Sem o uso de classificações, cada objeto precisaria definir explicitamente todas as suas características. Portanto, usando-se classificações, um objeto precisa definir somente aquelas qualidades que o tornam único dentro de sua classe. Ele pode herdar as qualidades que compartilha com a classe mais geral. É o mecanismo de herança que torna possível a um objeto ser uma instância específica de uma classe mais geral.

9.2 CONSIDERAÇÕES FUNDAMENTAIS SOBRE O C++

O C++ é um superconjunto da linguagem C; portanto, muitos programas em C também são implicitamente programas em C++. Isso significa que é possível escrever programas em C++ idênticos a programas em C. Entretanto, esse processo não é recomendado, pois não extrai todas as vantagens e capacidades do C++.

Ainda que o C++ permita escrever programas idênticos ao C, muitos programadores em C++ usam um estilo e certas características que são exclusivas do C++. Já que é importante aprender a escrever programas em C++ que se pareçam com programas em C++, estão seção introduz um pouco dessas características antes de entrar no “âmago” do C++.

Veja o exemplo a seguir.

```
#include <iostream.h> //biblioteca para operações de E/S em C++

main (void){
```

```
   int i;
   char str[80];

   cout << "C++ é fácil\n";
   printf ("Você pode usar printf() se preferir");

   //para ler um número use
   cout << "Informe um número: ";
   cin >> i;

   //para exibir um número use
   cout << "O seu número é: " << i << "\n";

   //para ler uma string use
   cout << "Informe uma string: ";
   cin >> str;

   //para imprimir uma string use
   cout << str;

   return (0);
}
```

O comando ***cout <<*** é utilizado para enviar dados para um dispositivo de saída, no caso a tela; você também poderia usar ***printf( )***, entretanto ***cout <<*** está mais de acordo com o C++.

O comando ***cin >>*** é utilizado para realizar uma entrada de valores via teclado. Em geral, você pode usar cin >> para carregar uma variável de qualquer um dos tipos básicos mais as strings. Você também poderia usar ***scanf( )*** ao invés de ***cin >>***. Entretanto, cin >> está mais de acordo com o C++.

9.3 COMPILANDO UM PROGRAMA C++

Os compiladores normalmente compilam tanto C como C++. Em geral, se um programa termina em ***.CPP***, ele será compilado como programa C++. Se ele termina com qualquer outras extensão, será compilado como um programa C. Portanto, a maneira mais simples de compilar seus programas C++ como tal é dar a eles a extensão .CPP.

9.4 INTRODUÇÃO A CLASSES E OBJETOS

Conhecidas algumas das convenções e características especiais do C++, é o momento de introduzir a sua característica mais importante: a ***class***. Em C++, para se criar um objeto, deve-se, primeiro, definir a sua forma geral usando-se a palavra reservada ***class***. Uma ***class*** é similar a uma estrutura. Vamos começar com um exemplo. Esta ***class*** define um tipo chamado ***fila***, que é usado para criar um objeto fila:

```
#include <iostream.h>

//isto cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   void init(void);
   void qput(int i);
   int qget(void);
};
```

Vamos examinar cuidadosamente esta declaração de ***class***.

Uma ***class*** pode conter tanto partes privadas como públicas. Por definição, todos os itens definidos numa ***class*** são privados. Por exemplo, as variáveis ***q***, ***sloc*** e ***rloc*** são privadas. Isso significa que não podem ser acessadas por qualquer função que não seja membro da ***class***. Essa é uma maneira de se conseguir a encapsulação – a acesso a certos itens de dados pode ser firmemente controlado mantendo-os privados. Ainda que não seja mostrado nesse exemplo, também é possível definir funções privadas que só podem ser chamadas por outros membros de ***class***.

Para tornar partes de uma ***class*** públicas, isto é, acessíveis a outras partes do seu programa, você deve declará-las após a palavra reservada ***public***. Todas as variáveis ou funções definidas depois de ***public*** estão disponíveis a todas as outras funções no programa. Essencialmente, o resto do seu programa acessa um objeto por meio das suas funções e dados ***public***. É preciso ser mencionado que embora se possa ter variáveis ***public***, filosoficamente deve-se tentar limitar ou eliminar seu uso. O ideal é tornar todos os dados privados e controlar o acesso a eles com funções ***public***.

As funções ***init( )***, ***qput( )*** e ***qget( )*** são chamadas de ***funções membros***, uma vez que fazem parte da ***class fila***. Lembre-se de que um objeto forma um laço entre código e dados. Somente aquelas funções declaradas na ***class*** têm acesso às partes privadas da ***class***.

Uma vez definida uma ***class***, pode-se criar um objeto daquele tipo usando-se o nome da ***class***. Essencialmente, o nome da ***class*** torna-se um novo especificador de tipo de dado. Por exemplo, a linha abaixo cria um objeto chamado ***intfila*** do tipo ***fila***:

```
fila intfila;
```

Você também pode criar objetos, quando define a ***class***, colocando os seus nomes após o fechamento das chaves, exatamente como faz com estruturas.

Para revisar, em C++ uma ***class*** cria um novo tipo de dado que pode ser usado para criar objetos daquele tipo.

A forma geral de uma declaração ***class*** é:

```
class nome_de_classe{
   dados e funções privados
```

```
public:
   dados e funções públicos
} lista de objetos;
```

Obviamente, a lista de objetos pode estar vazia.

Dentro da declaração de ***fila***, foram usados protótipos para as funções membros. É importante entender que, em C++, quando é necessário informar ao compilador a respeito de uma função, deve-se usar a forma completa do seu protótipo.

Quando chega o momento de codificar uma função que é membro de uma ***class***, deve ser dito ao compilador a qual ***class*** a função pertence, qualificando-se o nome da função com o nome da ***class*** onde ela é membro. Por exemplo, aqui está uma maneira de codificar a função ***qput( )***:

```
void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}
```

O ***::*** é chamado de ***operador de escopo de resolução***. Ele informa ao compilador que essa versão da função ***qput( )*** pertence a ***class fila*** ou, em outras palavras, que a função ***qput( )*** está no escopo de ***fila***. Em C++, muitas ***classes*** diferentes podem usar os mesmos nomes de função. O compilador sabe qual função pertence a qual classe por causa do operador de escopo de resolução e do nome da ***class***.

Para chamar uma função membro de uma parte do seu programa que não faz parte da ***class***, você deve usar o nome do objeto e o operador ponto. Por exemplo, a declaração abaixo chama a função ***init( )*** por meio do objeto ***a***:

```
fila a, b;
a.init();
```

É muito importante entender que ***a*** e ***b*** são dois objetos separados. Isso significa, por exemplo, que a inicialização de ***a*** não faz com que ***b*** também seja inicializado. O único relacionamento entre ***a*** e ***b*** é serem objetos do mesmo tipo.

Outro ponto interessante é que uma função membro pode chamar outra função membro diretamente sem usar o operador ponto. Somente quando uma função membro é chamada por um código que não faz parte da ***class*** é que o operador ponto deve ser usado.

O programa mostrado aqui coloca junto todas as partes e detalhes esquecidos e ilustra a ***class fila***:

```
#include <iostream.h>
```

```
//isto cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   void init(void);
   void qput(int i);
   int qget(void);
};

void fila::init(void){
   rloc = sloc = 0;
}

void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}

int fila::qget(void){
   if (rloc == sloc){
      cout << "A fila está vazia";
      return (0);
   }
   rloc++;
   return (q[rloc]);
}

main(void){
   fila a, b;   //cria dois objetos fila

   a.init();
   b.init();

   a.qput(10);
   b.qput(19);

   a.qput(20);
   b.qput(1);
   cout << a.qget() << " ";
   cout << a.qget() << " ";
   cout << b.qget() << " ";
   cout << b.qget() << " ";
```

```
   return (0);
}
```

LEMBRE-SE: As partes privadas de um objeto são acessíveis somente às funções membros daquele objeto. Por exemplo, uma declaração como:

```
a.rloc = 0;
```

não poderia estar na função ***main( )*** do programa anterior.

9.5 SOBRECARGA DE FUNÇÕES

Uma maneira do C++ obter polimorfismo é pelo uso de sobrecarga de funções. Em C++m duas ou mais funções podem compartilhar o mesmo nome, contanto que as suas declarações de parâmetros sejam diferentes. Nessa situação, as funções que compartilham o mesmo nome são conhecidas como ***sobrecarregadas*** e o processo é chamado de ***sobrecarga de funções***. Por exemplo, considere este programa:

```
#include <iostream.h>
//a função quadrado é sobrecarregada três vezes
int quadrado (int i);
double quadrado (double d);
long quadrado (long l);

main(void){
   cout << quadrado(10) << "\n";
   cout << quadrado(11.0) << "\n";
   cout << quadrado(9L) << "\n";
   return(0);
}

int quadrado(int i){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento inteiro.\n";
   return (i * i);
}

double quadrado(double d){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento double.\n";
   return (d * d);
}

long quadrado(long l){
   cout << "Dentro da função quadrado() que usa ";
   cout << "um argumento long.\n";
```

```
   return (l * l);
}
```

Esse programa cria três funções similares, porém diferentes, chamadas de ***quadrado( )***. Cada uma delas retorna o quadrado do seu argumento. Como o programa ilustra, o compilador sabe qual função usar em cada situação por causa do tipo de argumento.

O mérito da sobrecarga de funções é permitir que conjuntos relacionados de funções sejam acessados usando-se somente um nome. Nesse sentido, a sobrecarga de função deixa que se crie um nome genérico para algumas operações, com o compilador resolvendo exatamente qual função é necessária no momento para realizar a operação.

O que torna a sobrecarga de funções importante é o fato de ela poder ajudar no tratamento de complexidades. Para entender como, considere este exemplo. Muitos compiladores de linguagem C contêm funções como ***atoi( )***, ***atof( )*** e ***atol( )*** nas suas bibliotecas-padrões. Coletivamente, essas funções convertem uma string de dígitos em formatos internos de inteiros, double e long, respectivamente. Embora essas funções realizem ações quase idênticas, três nomes completamente diferentes devem ser usados em C para representar essas tarefas, o que torna a situação mais complexa do que é na realidade.

Ainda que o conceito fundamental de cada função seja o mesmo, o programador tem três coisas para se lembrar, e não somente uma. Entretanto, em C++, é possível usar o mesmo nome, ***atonum( )***, por exemplo, para todas as três funções. Assim, o nome ***atonum( )*** representa a ***ação geral*** que está sendo realizada. É de responsabilidade do compilador selecionar a versão ***específica*** para uma circunstância particular. Assim, o programador só precisa lembrar da ação geral que é realizada. Portanto, aplicando-se o polimorfismo, três coisas a serem lembradas foram reduzidas a uma. Ainda que esse exemplo seja bastante trivial, se você expandir o conceito, verá como o polimorfismo pode ajudar na compreensão de programas muito complexos.

Um exemplo mais prático de sobrecarga de funções é ilustrado pelo programa seguinte. Como você sabe, a linguagem C (e o C++) não contém nenhuma função de biblioteca que solicite ao usuário uma entrada, esperando, uma resposta. Este programa cria três funções, chamadas ***solicitacao( )***, que realizam essa tarefa para dados dos tipos ***int***, ***double*** e ***long***:

```
#include <iostream.h>

void solicitacao (char *str, int *i);
void solicitacao (char *str, double *d);
void solicitacao (char *str, long *l);

main(void){
   int i;
   double d;
   long l;

   solicitacao("Informe um inteiro: ", &i);
   solicitacao("Informe um double: ", &d);
   solicitacao("Informe um long: ", &l);
```

```
   return(0);
}

void solicitacao (char *str, int *i){
   cout << str;
   cin >> *i;
}

void solicitacao (char *str, double *d){
   cout << str;
   cin >> *d;
}

void solicitacao (char *str, long *l){
   cout << str;
   cin >> *l;
}
```

CUIDADO: Você pode usar o mesmo nome para sobrecarregar funções não relacionadas, mas não deve faze-lo. Por exemplo, você pode usar o nome ***quadrado( )*** para criar funções que retornam o quadrado de um ***int*** e a raiz quadrada de um ***double***. Entretanto, essas duas operações são fundamentalmente diferentes e a aplicação de sobrecarga de função, dessa maneira, desvirtua inteiramente o seu propósito principal. Na prática, você somente deve usar sobrecarga em operações intimamente relacionadas.

9.6 SOBRECARGA DE OPERADOR

Uma outra maneira de se obter polimorfismo em C++ é pela sobrecarga de operador. Como você sabe, em C++m pode-se usar os operadores << e >> para realizar operações de E/S em console. Isso é possível porque, no arquivo ***iostream.h***, esses operadores são sobrecarregados, tomando, assim, um significado adicional relativo a uma certa ***class***. Entretanto, ele ainda mantém o seu antigo significado (deslocamento de bits).

Em geral, você pode sobrecarregar qualquer um dos operadores do C++ definindo qual significado eles têm em relação a uma ***class*** específica. Por exemplo, recordando a ***class fila*** desenvolvida anteriormente neste capítulo, é possível sobrecarregar o operador **+** relativo aos objetos do tipo ***fila*** de maneira que ele acrescente o conteúdo de uma pilha em uma outra já existente. Contudo, o operador **+** ainda retém seu significado original em relação a outros tipos de dados. Mais a frente será apresentado um exemplo de sobrecarga de operador.

9.7 HERANÇA

Como definido, a herança é uma das principais características de uma linguagem de programação orientada a objetos. Em C++, a herança é suportada por permitir a uma ***class*** incorporar outra ***class*** na sua declaração. Para ver como isso funciona, vamos começar com um exemplo. Aqui está uma ***class***, chamada

veiculos_na_estrada, que define claramente veículos que trafegam nas estradas. Ela armazena o número de rodas que um veículo tem e o número de passageiros que pode transportar:

```
class veiculos_na_estrada{
   int rodas;
   int passageiros;
public:
   void fixa_rodas(int num);
   int obtem_rodas(void);
   void fixa_passageiros(int num);
   int obtem_passageiros(void);
};
```

Essa rude definição de uma estrada de rodagem pode ser usada para ajudar a determinar objetos específicos. Por exemplo, este código declara uma ***class*** chamada ***caminhão*** usando ***veiculos_na_estrada***:

```
class caminhao : public veiculos_na_estrada{
   int carga;
public:
   void fixa_carga(int tamanho);
   int obtem_carga(void);
   void exibe(void);
};
```

Note como ***veiculos_na_estrada*** é herdada. A forma geral para herança é mostrada aqui:

```
class nome_da_nova_classe : acesso classe_herdada{...
```

Aqui, ***acesso*** é opcional. Contudo, se estiver presente, ele deve ser ***public***, ***private*** ou ***protected***. A utilização de ***public*** significa que todos os elementos ***public*** do antepassado também serão ***public*** para a ***class*** que os herda.

Portanto, os membros da ***class caminhao*** têm acesso às funções membros de ***veiculos_na_estrada***, exatamente como se tivessem sido declarados dentro da ***class caminhao***. Entretanto, as funções membros ***não*** têm acesso às partes privadas da ***class veiculos_na_estrada***. Aqui está um programa que ilustra herança. Ele cria duas subclasses ***veiculos_na_estrada*** usando herança. Uma é ***caminhao***, a outra, ***automovel***.

```
#include <iostream.h>

class veiculos_na_estrada{
   int rodas;
   int passageiros;
public:
   void fixa_rodas(int num);
```

```
   int obtem_rodas(void);
   void fixa_passageiros(int num);
   int obtem_passageiros(void);
};

class caminhao : public veiculos_na_estrada{
   int carga;
public:
   void fixa_carga(int tamanho);
   int obtem_carga(void);
   void exibe(void);
};

enum tipo {carro, furgão, caminhão};

class automovel : public veiculos_na_estrada{
   enum tipo tipo_de_carro;
public:
   void fixa_tipo(enum tipo t);
   enum tipo obtem_tipo(void);
   void exibe(void);
};

void veiculos_na_estrada::fixa_rodas(int num){
   rodas = num;
}

int veiculos_na_estrada::obtem_rodas(void){
   return rodas;
}

void veiculos_na_estrada::fixa_passageiros(int num){
   passageiros = num;
}

int veiculos_na_estrada::obtem_passageiros(void){
   return passageiros;
}

void caminhao::fixa_carga(int num){
   carga = num;
}

int caminhao::obtem_carga(void){
   return carga;
}
void caminhao::exibe(void){
```

```
   cout << "rodas: " << obtem_rodas() << "\n";
   cout << "passageiros: " << obtem_passageiros() << "\n";
   cout << "capacidade de carga em metros cúbicos: " <<
        obtem_rodas() << "\n";
}

void automovel::fixa_tipo(enum tipo t){
   tipo_de_carro = t;
}

enum tipo automovel::obtem_tipo(void){
   return tipo_de_carro;
}

void automovel::exibe(void){
   cout << "rodas: " << obtem_rodas() << "\n";
   cout << "passageiros: " << obtem_passageiros() << "\n";
   cout << "tipo: ";

   switch(obtem_tipo(){
      case furgão:   cout << "furgão\n";
                     break;
      case carro:    cout << "carro\n";
                     break;
      case caminhão: cout << "caminhão\n";
   }
}

main(void){
   caminhao t1, t2;
   automovel c;

   t1.fixa_rodas(18);
   t1.fixa_passageiros(2);
   t1.fixa_carga(3200);

   t1.fixa_rodas(6);
   t1.fixa_passageiros(3);
   t1.fixa_carga(1200);

   t1.exibe();
   t2.exibe();

   c.fixa_rodas(4);
   c.fixa_passageiros(6);
   c.fixa_tipo(furgão);
   c.exibe();
```

```
   return(0);
}
```

Como esse programa mostra, a maior vantagem da herança é poder criar uma classificação base possível de ser incorporada por outras mais específicas. Dessa maneira, cada objeto pode representar precisamente a sua própria classificação.

Note que, tanto ***caminhao*** como ***automovel***, incluem a função membro chamada ***exibe( )***, que mostra informações sobre cada objeto. Esse é um outro aspecto do polimorfismo. Já que cada função ***exibe( )*** está relacionada à sua própria classe, o compilador pode facilmente dizer qual chamar em qualquer circunstância.

9.8 CONSTRUTORES E DESTRUTORES

É muito comum alguma parte de um objeto requerer inicialização antes de ser usada. Por exemplo, voltando à ***class fila*** desenvolvida neste capítulo, antes da fila poder ser usada, as variáveis ***rloc*** e ***sloc*** são fixadas em zero. Isso foi realizado por meio da função ***init( )***. Já que a exigência de inicialização é tão comum, o C++ permite aos objetos serem inicializados por si mesmos quando são criados. Essa inicialização automáticas é realizada pelo uso de uma função ***construtora***.

A construtora é uma função especial que é membro da ***class*** e tem o mesmo nome que a ***class***. Por exemplo, aqui está com a ***class fila*** fica quando convertida para usar uma função construtora para inicialização.

```
//O código seguinte cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   fila(void);        //construtor
   void qput(int i);
   int qget(void);
};
```

Note que o construtor ***fila( )*** não possui tipo de retorno especificado. Em C++, funções construtoras não podem retornar valores.

O construtor ***fila( )*** é implementado assim:

```
//Este é o construtor
fila::fila(void){
   sloc = rloc = 0;
   cout << "fila inicializada\n";
}
```

Tenha em mente que a mensagem ***fila inicializada*** é emitida como uma maneira de ilustrar o construtor. Na prática, muitas funções construtoras não emitirão nem receberão qualquer coisa.

Um construtor de objeto é chamado quando o objeto é criado. Isso significa que ele é chamado quando a declaração do objeto é executada. Ainda, para objetos locais, o construtor é chamado toda vez que a declaração do objeto é encontrada.

O complemento do construtor é o ***destrutor***. Em muitas circunstâncias, um objeto precisará realizar alguma ação quando é destruído. Por exemplo, um objeto pode precisar desalocar memória previamente alocada. Em C++, é a função destrutora que manipula desativações. O destrutor tem o mesmo nome que o construtor, mas é precedido por um **~**. Por exemplo, aqui está a ***class fila*** e as suas funções construtora e destrutora. Lembre-se que a classe ***fila*** não requer um destrutor, de modo que o destrutor mostrado aqui é somente para ilustração.

```
#include <iostream.h>

//o código seguinte cria a classe fila
class fila{
   int q[100];
   int sloc, rloc;
public:
   fila(void);   //construtor
   ~fila(void);  //destrutor
   void qput(int i);
   int qget(void);
};

//Esta é a função construtora
fila::fila(void){
   sloc = rloc = 0;
   cout << "Fila inicializada\n";
}

//Esta é a função destrutora
fila::~fila(void){
   cout << "Fila destruída\n";
}

void fila::qput(int i){
   if (sloc == 100){
      cout << "A fila está cheia";
      return;
   }
   sloc++;
   q[sloc] = i;
}

int fila::qget(void){
   if (rloc == sloc){
      cout << "A fila está vazia";
```

```
        return (0);
    }
    rloc++;
    return (q[rloc]);
}

main(void){
    fila a, b;   //cria dois objetos fila

    a.qput(10);
    b.qput(19);
    a.qput(20);
    b.qput(1);

    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << b.qget() << " ";
    cout << b.qget() << " ";
    return (0);
}
```

Esse programa exibe o seguinte:

```
fila inicializada
fila inicializada
10 20 19 1
fila destruída
fila destruída
```

9.9 FUNÇÕES FRIEND

É possível para uma função não-membro de uma ***class*** ter acesso a partes privadas de uma ***class*** pela declaração da função como uma ***friend*** da ***class***. Por exemplo, aqui, a função ***frd( )*** é declarada para ser uma função ***friend*** da ***class c1***:

```
class c1{
⋮
public:
    friend void frd(void);
⋮
};
```

Como você pode ver, a palavra reservada ***friend*** precede a declaração inteira da função, que é o caso geral.

Ainda que não tecnicamente necessárias, as funções ***friend*** são

permitidas em C++ para acomodar uma situação em que, por questões de eficiência, duas classes devam compartilhar a mesma função. Para ver um exemplo, considere um programa que define duas classes chamadas ***linha*** e ***box***. A ***class linha*** contém todos os dados e código necessários para desenhar uma linha horizontal tracejada de qualquer tamanho, começando nas coordenadas X, Y determinadas e usando uma cor especificada. A ***class Box*** contém todo código e dados necessários para desenhar um Box nas coordenadas especificadas pelo canto superior esquerdo e canto inferior direito em uma cor determinada. As duas ***classes*** usam a mesma cor. Essas ***classes*** são declaradas como mostrado aqui:

```
class linha;

class box{
   int cor;          //cor do box
   int upx, upy;     //canto superior esquerdo
   int lowx, lowy.   //canto inferior direito
public:
   friend int mesma_cor(linha l, box b)
   void indica_cor(int c);
   void define_box(int x1, int y1, int x2, int y2);
   void exibe_box(void);
};

class linha{
   int cor;
   int comecox, comecoy;
   int tamanho;
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_linha(int x, int y, int l);
   void exibe_linha();
};
```

A função ***mesma_cor( )***, que não é membro de nenhuma mas, sim, ***friend*** das duas, retorna verdadeiro se tanto o objeto ***linha*** como o objeto ***box***, que formam os seus argumentos, são desenhados na mesma cor; retorna zero, caso contrário. A função ***mesma_cor( )*** é mostrada aqui:

```
//retorna verdadeiro se a linha e o box têm a mesma cor
int mesma_cor(linha l, box b){
   if (l.cor == b.cor) return (1);
   return (0);
}
```

Como você pode ver, a função ***mesma_cor( )*** precisa ter acesso a partes privadas, tanto de ***linha*** como de ***box***, para realizar sua tarefa eficientemente.

Note a declaração vazia de ***linha*** no começo das declarações ***class***. Já

que a função ***mesma_cor( )*** em ***box*** referencia ***linha*** antes que ela seja declarada, ***linha*** deve ser referenciada de antemão. Se isso não for feito, o compilador não saberá o que ela é quando encontrá-la na declaração de ***box***. Em C++, uma referência antecipada a uma classe é simplesmente uma palavra reservada ***class*** seguida pelo nome da ***class***. Usualmente, as referências antecipadas só são necessárias quando as funções ***friend*** estão envolvidas.

Aqui está um programa que demonstra as classes ***linha*** e ***box*** e ilustra como uma função ***friend*** pode acessar partes privadas de uma ***class***. Este programa faz uso de várias funções de tela do compilador Borland C++.

```
#include <iostream.h>
#include <conio.h>

class linha;

class box{
   int cor;          //cor do box
   int upx, upy;     //canto superior esquerdo
   int lowx, lowy;   //canto inferior direito
public:
   friend int mesma_cor(linha l, box b)
   void indica_cor(int c);
   void define_box(int x1, int y1, int x2, int y2);
   void exibe_box(void);
};

class linha{
   int cor;
   int comecox, comecoy;
   int tamanho;
public:
   friend int mesma_cor(linha l, box b);
   void indica_cor(int c);
   void define_linha(int x, int y, int l);
   void exibe_linha();
};

//retorna verdadeiro se a linha e o box têm a mesma cor
int mesma_cor(linha l, box b){
   if (l.cor == b.cor) return (1);
   return (0);
}

void box::indica_cor(int c){
   cor = c;
}

void linha::indica_cor(int c){
```

```
   cor = c;
}

void box::define_box(int x1, int y1, int x2, int y2){
   upx = x1;
   upy = y1;
   lowx = x2;
   lowy = y2;
};

void box::exibe_box(void){
   int i;

   textcolor(cor);
   gotoxy(upx, upy);
   for (i = upx; i <= lowx; i++) cprintf("-");
   gotoxy (upx, lowy - 1);
   for (i = upx; i <= lowx; i++) cprintf("-");

   gotoxy(upx, upy);
   for (i = upy; i <= lowy; i++){
      cprintf("|");
      gotoxy(upx, i);
   }

   gotoxy(lowx, upy);
   for (i = upy; i <= lowy; i++){
      cprintf("|");
      gotoxy(lowx, i);
   }
}

void linha::define_linha(int x, int y, int l){
   comecox = x;
   comecoy = y;
   tamanho = l;
}

void linha::exibe_linha(void){
   int i;

   textcolor(cor);
   for (i = 0; i < tamanho; i++) cprintf("-");
}

main(void){
   box b;
```

```
   linha l;
   b.define_box(10, 10, 15, 15);
   b.indica_cor(3);
   b.mostra_box();
   l.define_linha(2, 2, 10);
   l.indica_cor(2);
   l.exibe_linha();

   if (!mesma_cor(l, b)) cout << "Não são da mesma cor";

   cout << "\nPressione qualquer tecla";
   getche();

   //agora, torne a linha e o box da mesma cor
   l.define_linha(2, 2, 10);
   l.indica_cor(3);
   l.exibe_linha();

   if (mesma_cor(l, b)) cout << "São da mesma cor";
   return(0);
}
```

9.10 A PALAVRA RESERVADA this

Para melhor compreender a sobrecarga de operadores, é necessário aprender a respeito de outra palavra reservada do C++, chamada ***this***, que é um ingrediente essencial para muitas sobrecargas de operadores.

Cada vez que se invoca uma função membro, automaticamente é passado um ponteiro para o objeto que a invoca. Você pode acessar esse ponteiro usando a palavra reservada ***this***. O ponteiro ***this*** é um parâmetro ***implícito*** para todas as funções membros.

Uma função membro pode acessar diretamente os dados ***private*** da sua ***class***. Por exemplo, dado esta ***class***:

```
class cl{
   int i;
⋮
};
```

uma função membro pode atribuir a ***i*** o valor 10 usando esta declaração:

```
i = 10;
```

A declaração anterior é uma abreviação para esta declaração:

```
this->i = 10;
```

Para ver como o ponteiro ***this*** funciona, examine o pequeno programa seguinte:

```
#include <iostream.h>
class cl{
   int I;
public:
   void carrega_i(int val){
      this->i = val;     //o mesmo que i = val
   }
   int obtem_i(void){
      return(this->i);   //o mesmo que return i
   }
};

void main(void){
   cl o;

   o.carrega_i(100);
   cout << o.obtem_i();
}
```

Esse programa exibe o número 100.

Ainda que o exemplo anterior seja trivial – de fato, ninguém deve usar o ponteiro ***this*** dessa maneira –, na próxima seção você verá porque o ponteiro ***this*** é tão importante.

9.11 SOBRECARGA DE OPERADOR – MAIORES DETALHES

Outra característica do C++ relacionada com sobrecarga de funções é a chamada ***sobrecarga de operadores***. Com pouquíssimas exceções, muitos dos operadores do C++ podem receber significados especiais relativos a classes específicas. Por exemplo, uma ***class*** que define uma lista ligada pode usar o operador **+** para adicionar um objeto à lista. Outras ***class*** pode usar o operador **+** de uma maneira inteiramente diferente. Quando um operador é sobrecarregado, nada do seu significado original é perdido; simplesmente é definida uma nova operação relativa a uma ***class*** específica. Portanto, sobrecarregar o operador **+** para manipular uma lista ligada não faz com que, por exemplo, o seu significado relativo aos inteiros (isto é, adição) seja mudado.

Para sobrecarregar um operador, você deve definir o que a dada operação significa em relação à ***class*** em que ela é aplicada. Para isso, crie uma função ***operator*** que defina sua ação. A forma geral da função ***operator*** é mostra aqui:

```
tipo nome_da_classe::operator#(lista de argumentos){
   //operação definida relativa à classe
}
```

Aqui, ***tipo*** é o tipo do valor retornado pela operação especificada. Freqüentemente, o valor de retorno é do mesmo tipo que a ***class*** (ainda que possa ser de qualquer tipo que você selecionar). Um operador sobrecarregado tem freqüentemente um valor de retorno do mesmo tipo de ***class*** para onde é sobrecarregado porque facilita o seu uso em expressões complexas.

Funções operator devem ser ou membros ou ***friends*** da ***class*** na qual estão sendo usadas. Para ver como a sobrecarga de operadores funciona, vamos começar com um exemplo simples que cria uma ***class***, chamada ***tres_d***, que mantém as coordenadas de um objeto no espaço tridimensional. Este programa sobrecarrega os operadores **+** e **=** relativos à ***class tres_d***. Examine-o cuidadosamente:

```
#include <iostream.h>

class tres_d{
   int x, y, z;      //coordenadas 3-D
public:
   tres_d operator+ (tres_d t);
   tres_d operator= (tres_s t);
   void mostra(void);
   void atribui(int mx, int my, int mz);
};

//sobrecarrega o +
tres_d tres_d::operator+ (tres_d t){
   tres_d temp;

   temp.x = x + t.x;
   temp.y = y + t.y;
   temp.z = z + t.z;
   return(temp);
}

//sobrecarrega o =
tres_d tres_d::operator= (tres_d t){
   x = t.x;
   y = t.y;
   z = t.z;
   return(*this);
}

//mostra as coordenadas x, y, z
tres_d tres_d::mostra(void){
   cout << x << ", ";
   cout << y << ", ";
   cout << z << "\n";
}
```

```
//atribui coordenadas
void tres_d::atribui(int mx, int my, int mz){
   x = mx;
   y = my;
   z = mz;
}

main(void){
   tres_d a, b, c;

   a.atribui(1, 2, 3);
   b.atribui(10, 10, 10);
   a.mostra();
   b.mostra();
   c = a + b;      //agora adiciona a e b
   c.mostra();
   c = a + b + c;  //agora adiciona a, b e c
   c.mostra();
   c = b = a;      //demonstra atribuição múltipla
   c.mostra();
   b.mostra();
   return(0);
}
```

Esse programa produz o seguinte:

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

Note que as duas funções operator têm somente um parâmetro cada, embora sobrecarreguem operações binárias. A razão para essa aparente contradição é que, quando um operador binário é sobrecarregado usando-se uma função membro, somente um argumento precisa ser passado para ele explicitamente. O outro argumento é passado implicitamente usando o ponteiro ***this***. Assim, na linha

```
temp.x = x + t.x;
```

o ***x*** refere-se a ***this.x***, que é o ***x*** associado ao objeto que solicitou a chamada para a função operator. Em todos os casos, é o objeto no lado esquerdo de uma operação que faz com que seja chamada a função operator. O objeto no lado direito é passado para a função.

Em geral, quando você está sobrecarregando uma função membro, nenhum parâmetro é necessário para sobrecarregar um operador unário e somente

um é requerido para sobrecarregar um operador binário. Em ambos os casos, o objeto que causa a ativação da função operator é implicitamente passado pelo ponteiro ***this***.

Para entender como a sobrecarga de operador funciona, vamos examinar esse programa cuidadosamente, começando como o operador sobrecarregado **+**. Quando dois objetos do tipo ***tres_d*** são operados pelo operador **+**, a amplitude das suas respectivas coordenadas são adicionadas, como mostrado na função ***operator+( )*** associada a essa ***class***. Note, entretanto, que essa função não modifica o valor de qualquer operando. Em vez disso, um objeto do tipo ***tres_d*** é retornado pela função que contém o resultado da operação. Esse é um ponto importante.

Para entender por que a operação **+** não deve mudar o conteúdo de qualquer objeto, pense a respeito da operação aritmética padrão ***10 + 12***. O resultado dessa operação é 22, porém nem 10 nem 12 são modificados pela operação.

Outro ponto-chave sobre como o operador **+** é sobrecarregado é que ele retorna um objeto do tipo ***tres_d***. Apesar da função poder retornar qualquer tipo válido do C++, o fato de ela retornar um objeto ***tres_d*** permite ao operador **+** ser usado em expressões mais complexas, tal como ***a + b + c***.

Contrastando com o **+**, o operador de atribuição, na verdade, faz com que um dos seus argumentos seja modificado. Já que a função ***operator=( )*** é chamada pelo objeto que ocorre no lado esquerdo da atribuição, é ele que é modificado pela operação de atribuição. Entretanto, até mesmo a operação de atribuição deve retornar um valor, haja visto que, em C++, a operação de atribuição produz um valor que ocorre no lado direito. Assim, para permitir uma declaração como:

```
a = b = c = d;
```

é necessário que a função ***operator+( )*** retorne o objeto apontado por ***this***, que será aquele que ocorre no lado esquerdo da declaração de atribuição. Isso permite que uma cadeia de atribuições seja feita.

Também é possível sobrecarregar operadores unários, como **++** e **--**. Como definido anteriormente, quando se sobrecarrega um operador unário, nenhum objeto é explicitamente passado para a função operator. Em vez disso, a operação é realizada no objeto que gera a chamada para a função pela passagem implícita do ponteiro ***this***. Por exemplo, aqui está uma versão expandida do exemplo anterior, que define as operações de incremento para os objetos do tipo ***tres_d***:

```
#include <iostream.h>

class tres_d{
   int x, y, z;       //coordenadas 3-D
public:
   tres_d operator+ (tres_d op2);  //op1 está implícito
   tres_d operator= (tres_s op2);  //op1 está implícito
   tres_d operator++ (void);       //op1 também está implícito
   void mostra(void);
   void atribui(int mx, int my, int mz);
};
```

```
tres_d tres_d::operator+ (tres_d op2){
   tres_d temp;

   temp.x = x + op2.x;   //estas são adições de inteiros
   temp.y = y + op2.y;   //e o + retém o seu significado
   temp.z = z + op2.z;   //original relativo a eles
   return(temp);
}

tres_d tres_d::operator= (tres_d op2){
   x = op2.x;            //estas são atribuições de inteiros
   y = op2.y;            //e o = retém o seu significado
   z = op2.z;            //original relativo a eles
   return(*this);
}

//sobrecarrega um operador unário
tres_d tres_d::operator++ (void){
   x++;
   y++;
   z++;
   return(*this);
}

//mostra as coordenadas x, y, z
tres_d tres_d::mostra(void){
   cout << x << ", ";
   cout << y << ", ";
   cout << z << "\n";
}

//atribui coordenadas
void tres_d::atribui(int mx, int my, int mz){
   x = mx;
   y = my;
   z = mz;
}

main(void){
   tres_d a, b, c;

   a.atribui(1, 2, 3);
   b.atribui(10, 10, 10);
   a.mostra();
   b.mostra();
   c = a + b;      //agora adiciona a e b
   c.mostra();
```

```
   c = a + b + c;  //agora adiciona a, b e c
   c.mostra();
   c = b = a;      //demonstra atribuição múltipla
   c.mostra();
   b.mostra();
   c++;            //incrementa c
   c.mostra();
   return(0);
}
```

Um ponto importante quando se sobrecarrega os operadores **++** ou **–** é que não é possível determinar de dentro da função ***operator*** se o operador precede ou segue seu operando. Isto é, a sua função ***operator*** não pode saber se a expressão que causa a chamada para a função é:

```
++OBJ;
```

ou

```
OBJ++;
```

onde ***OBJ*** é o objeto da operação.

A ação de um operador sobrecarregado, como é aplicada à ***class*** para a qual é definido, não precisa sustentar qualquer relacionamento com o uso do operador-padrão quando aplicado aos tipos próprios do C++. Por exemplo, **<<** e **>>** quando aplicados a ***cout*** e ***cin***, não têm nada em comum com quando aplicados a tipos inteiros. Contudo, para propósitos de estruturação e legibilidade do seu código, um operador sobrecarregado deve refletir, quando possível, o espírito do uso original do operador. Por exemplo, o operador **+** relativo a ***class tres_d*** é conceitualmente similar ao **+** relativo aos tipos inteiros. Há poucos benefícios, por exemplo, na definição do operador **+** relativo a alguma ***class*** de tal forma que atue de modo completamente inesperado. O conceito-chave aqui é que, enquanto se pode dar a um operador sobrecarregado qualquer significado que se deseja, é melhor, para efeito de esclarecimento, que o novo significado esteja relacionado com o original.

Existem algumas restrições aplicáveis aos operadores sobrecarregados. Primeiro, não se pode alterar a precedência de qualquer operador. Segundo, não se pode alterar o número de operandos requeridos pelo operador, ainda que a função ***operator*** possa optar por ignorar um operando. Finalmente, com exceção do operador **=**, os operadores sobrecarregados são herdados por qualquer ***class*** derivada. Cada ***class*** deve definir explicitamente o seu próprio operador **=** sobrecarregado, se for necessário.

Os únicos operadores que não podem ser sobrecarregados são mostrados aqui: “***.***”, “***::***”, “***.****”, “***?***”.