Centro Universitário de Maringá - CESUMAR

Cursos de AUTOMAÇÃO INDUSTRIAL e

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Linguagem de Programação

Linguagem C

Disciplina de Informática para Automação

Prof. EE. Carlos Henrique Z. Pantaleão

2005

SUMÁRIO

1 Programação em C

Atualmente, empregam-se cada vez mais sistemas computacionais na automatização de processos industriais. Os sistemas computacionais empregados variam desde um simples circuito lógico digital, passando por uma circuito composto por um microprocessador ou um CLP, até sistemas complexos envolvendo um ou mais microcomputadores ou até estações de trabalho. Um engenheiro que atua nesta área deve conhecer os sistemas computacionais disponíveis e ser capaz de selecionar o melhor equipamento para uma dada aplicação. Além disto, este profissional deve conseguir instalar este sistema, configurá-lo e acima de tudo programá-lo para que este execute a tarefa de automatização atendendo os requisitos industrias do sistema, como imunidade a falhas ou comportamento determinístico com restrições temporais (sistemas tempo-real). Neste contexto, a programação destes sistemas se faz de suma importância. Basicamente, a inteligência dos sistemas automatizados é implementada através de programas computacionais, comandando os componentes de hardware para executar a tarefa com o comportamento desejado.

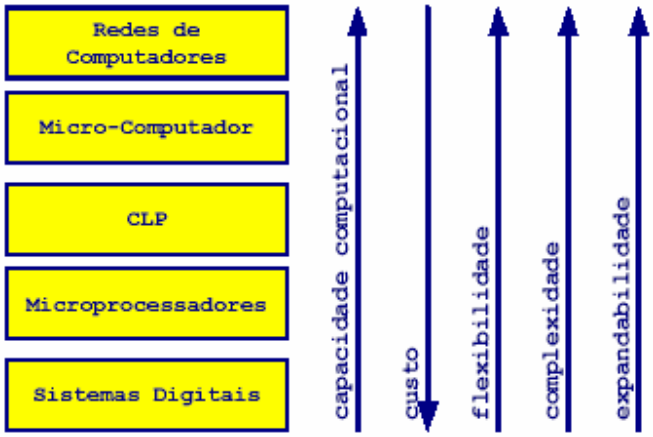

Figura 1 - Comparação entre os diversos sistemas computacionais para aplicações industriais.

Nas últimas décadas, o desenvolvimento em hardware permitiu que cada vez mais os processos industrias sejam automatizados e interligados através de sistemas computacionais. Entretanto, a evolução em software não se deu em tamanha velocidade como a de hardware. Desta

forma, um dos grandes paradigmas tecnológicos hoje é o desenvolvimento de programas para a realização de tarefas complexas e que exigem um alto grau de inteligência.

A maneira de se comunicar com um computador chama-se programa e a única linguagem que o computador entende chama-se *linguagem de máquina*. Portanto todos os programas que se comunicam com a máquina devem estar em *linguagem de máquina*.

Para permitir uma maior flexibilidade e portabilidade no desenvolvimento de software, foram implementados nos anos 50 os primeiros programas para a tradução de linguagens semelhantes à humana (linguagens de *"alto nível"*) em linguagem de máquina. A forma como os programas são traduzidos para a linguagem de máquina classifica-se em duas categorias:

• *Interpretadores*: Um interpretador lê a primeira instrução do programa, faz uma consistência de sua sintaxe e, se não houver erro converte-a para a linguagem de máquina para finalmente executá-la. Segue, então, para a próxima instrução, repetindo o processo até que a última instrução seja executada ou a consistência aponte algum erro. São muito bons para a função de depuração (*"debugging"*) de programas, mas são mais lentos. Ex.: BASIC Interpretado, Java.

• *Compiladores*: Traduzem o programa inteiro em linguagem de máquina antes de serem executados. Se não houver erros, o compilador gera um programa em disco com o sufixo **.OBJ** com as instruções já traduzidas. Este programa não pode ser executado até que sejam agregadas a ele rotinas em linguagem de máquina que lhe permitirão a sua execução. Este trabalho é feito por um programa chamado "*linkeditor*" que, além de juntar as rotinas necessárias ao programa **.OBJ**, cria um produto final em disco com sufixo **.EXE** que pode ser executado diretamente do sistema operacional.

Compiladores bem otimizados produzem código de máquina quase tão eficiente quanto aquele gerado por um programador que trabalhe direto em *Assembly*. Oferecem em geral menos facilidades de depuração que interpretadores, mas os programas são mais rápidos (na ordem de 100 vezes ou mais). Ex.: BASIC Compilado, FORTRAN, PASCAL, MÓDULA - 2, C, C++. Além da velocidade, outras vantagens podem ser mencionadas:

• é desnecessária a presença do interpretador ou do compilador para executar o programa já compilado e linkeditado;

• programas **.EXE** não podem ser alterados, o que protege o código-fonte.

Desta forma, os compiladores requerem o uso adicional de um editor de ligações (*"Linker"*), que combina módulos-objetos (*"Traduzidos"*) separados entre si e converte os módulos assim *"linkados"* no formato carregável pelo sistema operacional (programa ***.EXE***).

2 Conceitos Básicos da Programação C

2.1 Histórico de C

O compilador "C" vem se tornando o mais difundido em ambiente industrial. A linguagem "C" se originou das linguagens BCPL e B desenvolvidas em 1970. A primeira versão de "C" foi implementada para o sistema operacional UNIX pela *Bell Laboratories*, especificamente por Dennis M. Ritchie e Ken Thompson no início da década de 70, e rodava em um DEC PDP11 (*Digital Equipment Corporation*). A linguagem "C" foi utilizada para portar o UNIX para outros computadores. A linguagem "C" possui uma característica dual:

- É considerada linguagem estruturada de alto-nível;
- *Assembly* de alto-nível, que permite escrever programas muito próximos à linguagem de máquina, sendo usada para desenvolver muitas aplicações como compiladores, interpretadores, processadores de texto e mesmo sistemas operacionais. Ex: UNIX, MSDOS, TPW.

A linguagem de programação "C" tornou-se rapidamente uma das mais importantes e populares, principalmente por ser muito poderosa, portátil, pela padronização dos compiladores existentes (através da norma ANSI C) e flexível. Os programas em "C" tendem a ser bastante compactos e de execução rápida.

A linguagem "C" é baseada em um núcleo pequeno de funções e estruturas básicas, desta forma, todo programa é desenvolvido a partir deste núcleo básico. Isto implica na grande portabilidade de "C", haja vista que basta a implementação deste núcleo básico para um dado processador e automaticamente já estará disponível um compilador "C" para este processador. Por esta razão, existem compiladores "C" para a grande parte dos sistemas computacionais atualmente disponíveis. Devido também a este pequeno núcleo, um programador C é capaz de desenvolver

programas tão eficientes, pequenos e velozes quanto os programas desenvolvidos em *Assembly*. Por isso, diz-se que C é uma linguagem de alto nível, porém, próxima da linguagem de máquina.

2.2 Criando um Programa Executável

Primeiro, escreva o seu programa (arquivo em modo ASCII), com o auxílio de um programa denominado de compilador, e grave o programa em disco dando a ele um nome como sufixo **.C**. O programa gerado é chamado de *código fonte*. Na seqüência, compile o fonte seguindo as instruções do seu compilador, o que criará um programa com o sufixo **.OBJ** em disco. O programa gerado é chamado de *objeto*. Por fim, basta linkeditar o objeto seguindo as instruções do seu linkeditor o que criará um programa com sufixo **.EXE** em disco. O programa gerado é chamado de *executável*. A Figura 2 apresenta o processo de geração de um programa em C.

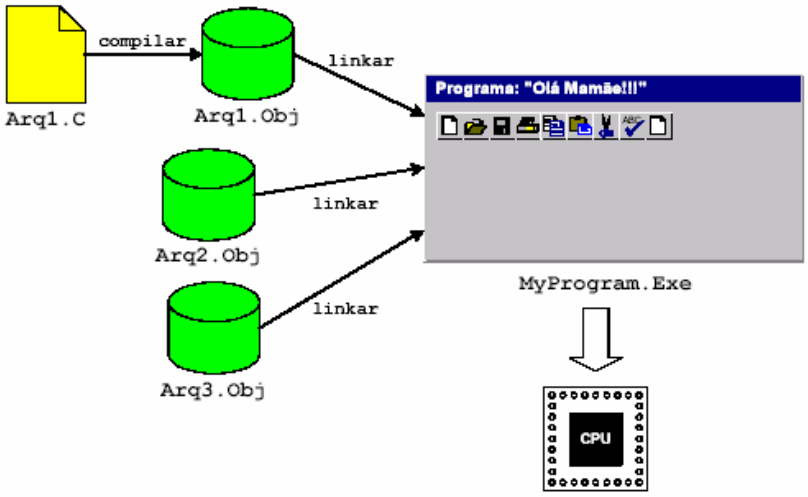

Figura 2 - Ilustração do processo de criação de um programa em C.

2.3 A Estrutura Básica de um Programa em C

A forma geral de um programa em "C" é a seguinte:

```
<diretivas do pré-processador>
<declarações globais>;
main()
{
      <declarações locais>;       /* comentários */
      <instruções>;
}
<outras funções>
```

Vamos começar por um programa bem simples em C. Você pode escrever este programa em um arquivo ASCII e salvá-lo com um nome terminando em ".C". O programa serve para escrever na tela a frase "Bom Dia!!!!".

```
/* Programa : Bom Dia! */
#include <stdio.h>
void main()
{
      printf("Bom Dia!!!!");
}
```

Na primeira linha, os símbolos **/*** e ***/** servem para delimitar um comentário do programa. É muito importante que os programas sejam comentados de forma organizada. Isto permite que outras pessoas possam facilmente entender o código fonte. Os comentários não são interpretados pelo compilador, servindo apenas para a documentação e esclarecimento do programador. Depois, segue-se com uma diretiva para o pré-processador "#include <stdio.h>".

Isto advém do fato de C ter um núcleo pequeno de funções básicas. Ao escrever esta linha de código, o pré-processador irá acrescentar ao programa todas as funcionalidades definidas na

biblioteca *"stdio"* e irá linká-la posteriormente ao programa. Desta forma, você poderá usufruir todos os serviços disponíveis nesta biblioteca.

Por fim, temos a função "main()". Esta função indica ao compilador em que instrução deve ser começada a execução do programa. Portanto, esta função deve ser única, aparecendo somente uma vez em cada programa. O programa termina quando for encerrada a execução da função main(). No caso deste programa exemplo, ela não recebe nenhum parâmetro e também não retorna parâmetro nenhum. Isto fica explícito através da palavra-chave void escrita na frente do programa. Se em vez de void tivéssemos escrito int, isto significaria que a função main() deveria retornar um valor do tipo inteiro ao final de sua execução. Como este é o valor retornado pela função main(), este também será o valor retornado pelo programa após a sua execução. As funções e as suas características serão apresentadas em detalhes nos próximos capítulos.

Todo o corpo de uma função em C é inicializado e finalizado através das chaves **{** e **}**. Estas chaves definem o bloco de instruções a serem executados por esta função. A primeira instrução dada dentro do programa é "printf ("Bom Dia!!!!");". Printf é uma função definida em "stdio.h" para escrever dados na janela console. Todas as instruções de programa têm que ser declaradas dentro de alguma função (na main() ou outra qualquer). Todas as instruções devem estar dentro das chaves que iniciam e terminam a função e são executadas na ordem em que as escrevemos. As instruções C são sempre encerradas por um ponto-e-vírgula (;). O ponto-e-vírgula é parte da instrução e não um simples separador. Esta instrução é uma chamada à função printf(), os parênteses nos certificam disso e o ponto-e-vírgula indica que esta é uma instrução. Nota-se que a função é chamada escrevendo-se o nome desta e colocando-se os parâmetros desta dentro dos parênteses. A final de cada instrução, faz-se necessário o acréscimo de um ponto-vírgula ";".

As variáveis em C podem estar dentro de uma função ou no início do arquivo fonte. Variáveis declaradas no início do arquivo fonte são consideradas "globais", isto é, são visíveis (acessíveis) para todas as funções do programa. Variáveis declaradas dentro de uma função são consideradas "locais", isto é, visíveis somente pela função onde são declaradas.

"C" distingue nomes de variáveis e funções em maiúsculas de nomes em minúsculas. Você pode colocar espaços, caracteres de tabulação e pular linhas à vontade em seu programa, pois o compilador ignora estes caracteres. Em C não há um estilo obrigatório. Entretanto, procure manter os programas tão organizados quanto for possível, pois isto melhora muito a legibilidade do programa, facilitando o seu entendimento e manutenção.

2.4 Variáveis

As variáveis são o aspecto fundamental de qualquer linguagem de computador. Uma variável em C é um espaço de memória reservado para armazenar um certo tipo de dado e tendo um nome para referenciar o seu conteúdo. O espaço de memória de uma variável pode ser compartilhado por diferentes valores segundo certas circunstâncias. Em outras palavras, uma variável é um espaço de memória que pode conter, a cada tempo, valores diferentes.

```
/* Programa : Exemplo de variáveis! */
#include <stdio.h>
void main()
{
      int num;                                    /* declaracao */
      num = 2;                                    /*atribui um valor*/
      printf("Este é o número dois: %d", num); /*acessa a variável*/
}
```

A primeira instrução (`int num`) é um exemplo de declaração de variável, isto é, apresenta um tipo, **int**, e um nome, **num**. A segunda instrução (`num = 2`) atribui um valor à variável e este valor será acessado através de seu nome. Usamos o operador de atribuição (=) para este fim. A terceira instrução chama a função printf() mandando o nome da variável como argumento. Esta lê o valor da variável e substitui na posição indicada por **%d**, compondo assim a frase apresentada na tela. O emprego da função printf() será apresentado em detalhe, posteriormente.

Em C todas as variáveis devem ser declaradas. Se você tiver mais de uma variável do mesmo tipo, poderá declará-las de uma única vez separando seus nomes por vírgulas. Exemplo:

```
int aviao, foguete, helicoptero;
```

2.5 Tipos de Dados

O tipo de uma variável informa a quantidade de memória, em bytes, que esta irá ocupar e a forma como o seu conteúdo será armazenado. Em C existem apenas 5 tipos básicos de variáveis, que são:

Identificador	Categoria
char	Caracter
int	Inteiro
float	Real de ponto flutuante
double	Real de ponto flutuante de dupla precisão
void	Sem valor.

Com exceção de **void**, os tipos de dados básicos podem estar acompanhados por *"modificadores"* na declaração de variáveis. Os *"modificadores"* de tipos oferecidos em C são:

Modificadores	Efeito
signed	Bit mais significante é usado como sinal
unsigned	Bit mais significante é parte do número (só +)
long	Estende precisão
short	Reduz precisão

Tipos de Dados Resultantes:

Tipo	Tamanho	Valores possíveis
(signed) char	1 Byte	-128 a +127
unsigned char	1 Byte	0 a 255
(short) (signed) int	2 Bytes	-32.768 a +32.767
(short) unsigned int	2 Bytes	0 a 65.535
(signed) long int	4 Bytes	-2.147.483.648 a +..647
unsigned long int	4 Bytes	0 a 4.294.967.295
float	4 Bytes	±3,4E-38 a ±3,4E+38
long float	8 Bytes	±1,7E-308 a ±1,7E+308
double	8 Bytes	±1,7E-308 a ±1,7E+308

Observação: As declarações que aparecem na tabela acima entre parênteses (), indicam que estas declarações são optativas. Por exemplo "short unsigned int" indica a mesma precisão que "unsigned int". O tipo **int** tem sempre o tamanho da palavra da máquina, isto é, em computadores de 16 bits ele terá 16 bits de tamanho.

Emprega-se o **complemento de dois** dos números positivos para o cálculo e representação dos números negativos. A escolha de nomes significativos para suas variáveis pode ajudá-lo a entender o que o programa faz e prevenir erros. Uma variável não pode ter o mesmo nome de uma palavra-chave de C. Em C, letras minúsculas e maiúsculas são diferentes.

Tabela de Palavras Chaves em C:

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

Exemplo de um programa que emprega as variáveis apresentadas.

```
void main()
{
      float y;              /*variável Real não inicializada*/
      int i;                /*variável Inteira não inicializada*/
      double x = 3.24;      /*var. Double inicializada com 3.24 */
      char c = 'a';         /*variável Char inicializada com a */
      i = 100;              /*variável i recebe o valor 100 */
      y = (float) i;        /*converte tipos */
}
```

Preste atenção na operação "y= (float) i;". Esta operação é muita empregada para conversão de tipos de variáveis diferentes. Suponha que você tenha uma variável 'x' de tipo A e queira

convertê-la para o tipo B e armazená-la em y deste tipo. Você pode executar esta operação através do operador:

```
y = ((B) x);
```

Atenção: Cuidado ao converter variáveis com precisão grande para variáveis com precisão pequena, ou seja, variáveis que utilizam um número diferente de bits para representar dados. Você pode perder informações importantes por causa desta conversão. Por exemplo, se você converter um float num int, você perderá todos os dígitos depois da vírgula (precisão).

```
void main()
{
      float y = 3.1415;
      int x = 0;
      x = (int) y;          /* Equivalente à: x = 3 */
}
```

2.6 Constantes

Um constante tem valor fixo e inalterável. No primeiro programa exemplo, mostramos o uso de uma cadeia de caracteres constante juntamente com a função printf():

```
printf("Bom Dia!!!!");
```

Há duas maneiras de declarar constantes em C:

a) usando a diretiva ***#define*** do pré-processador:

```
#define < nome da constante > < valor >
```

Esta diretiva faz com que toda aparição do nome da constante no código seja substituída antes da compilação pelo valor atribuído. Não é reservado espaço de memória no momento da declaração ***define***. A diretiva deve ser colocada no inicio do arquivo e tem valor global (isto é, tem valor sobre todo o código). Exemplo:

```
#define    size    400
#define    true    1
#define    false   0       /* não se utiliza ";" nem "=" */
```

b) utilizando a palavra-chave "***const***":

```
      const < tipo > < nome > = < valor >;
```

Esta forma reserva espaço de memória para uma variável do tipo declarado. Uma constante assim declarada só pode aparecer do lado direito de qualquer equação (isto equivale a dizer que não pode ser atribuído um novo valor àquela "variável" durante a execução do programa). Exemplo:

```
const char letra = 'a';
const int size = 400;
const double gravidade = 9.81;

/* Programa: Exemplo do uso de Constantes */
#define   Size   4
void main()
{
      const char c = 'c';
      const int num = 10;
      int val = Size;
}
```

Em C uma constante caractere é escrita entre aspas simples, uma constante cadeia de caracteres entre aspa duplas e constantes numéricas com o número propriamente dito. Exemplos de constantes:

a) caractere: 'a'

b) cadeia de caracteres: "Bom Dia !!!!"

c) número: -3.141523

2.7 Ponteiros

Uma das mais poderosas características oferecidas pela linguagem C é o uso de ponteiros. Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente. O mecanismo usado para isto é o endereço da variável. De fato, o endereço age como intermediário entre a variável e o programa que a acessa.

Basicamente, um ponteiro é uma representação simbólica de um endereço. Portanto, utiliza-se o endereço de memória de uma variável para acessá-la. Um ponteiro tem como conteúdo um endereço de memória. Este endereço é a localização de uma outra variável de memória. Dizemos que uma variável aponta para uma outra quando a primeira contém o endereço da segunda.

A declaração de ponteiros tem um sentindo diferente da de uma variável simples. A instrução:

```
int *px;
```

declara que *px é um dado do tipo **int** e que **px** é um ponteiro, isto é, **px** contém o endereço de uma variável do tipo **int**.

Para cada nome de variável (neste caso **px**), a declaração motiva o compilador a reservar dois bytes de memória onde os endereços serão armazenados. Além disto, o compilador deve estar ciente do tipo de variável armazenada naquele endereço; neste caso inteiro. O endereço de uma variável pode ser passado à um ponteiro através do operador &, como apresentado abaixo:

```
void main()
{
      int num, valor; /* declara as variáveis como inteiras */
      int *ptr;       /* declara um ponteiro para um inteiro */
      ptr = 0;        /* inicializa o ponteiro com o endereco '0' */
      ptr = &num;     /* atribui ao ptr o endereço da variável num*/
      num = 10;       /* atribui à variável inteira o valor '10' */
      valor = *ptr;   /* acessa o conteúdo apontado por 'ptr' e */
                      /* atribui a 'valor' */
}
```

Neste programa, primeiro declaram-se duas variáveis inteiras. Em seguida, declara-se um ponteiro para uma variável do tipo inteira. Este ponteiro tem o seu conteúdo inicializado com '0'. Este é um procedimento normal na manipulação de ponteiros, muito empregado para evitar o aparecimento de erros. Pois, enquanto o endereço de um ponteiro for nulo, isto indicará que este endereço contém um valor inválido e, portanto, o conteúdo representado por este endereço não deve ser acessado.

Na seqüência, emprega-se o operador & para obter o endereço da variável 'num' e armazenar este endereço em 'ptr'. Logo após, atribuímos a variável inteira 'num' o valor 10. Como 'ptr' contém o endereço de 'num', logo 'ptr' poderá já acessar o valor 10 armazenado na variável 'num'. Isto é feito na última linha, onde o conteúdo apontado por 'ptr' é acessado e copiado para a variável 'valor'. Outro exemplo da manipulação de ponteiros:

```
void main()
{
      int i, j, *ptr;       /* declara as variáveis */
      i = 1;                /* i recebe o valor '1' */
      j = 2;                /* j recebe o valor '2' */
      ptr = &i;             /* ptr recebe o valor do endereço de i */
      *ptr = *ptr + j;      /* equivale a: i = i + j */
}
```

Nosso objetivo neste momento não é apresentar todas as potencialidades dos ponteiros. Estamos aqui apresentando os ponteiros primeiramente como um tipo de dado especial. O importante aqui é entender o conteúdo de um ponteiro e como este pode ser empregado. Posteriormente, apresentaremos as funcionalidades dos ponteiros à medida que formos evoluindo no aprendizado de C.

2.8 Exercícios

2.1 Crie o seu programa “Hello, World!” ou “Olá, Mamãe!”;

2.2 Crie um programa que contenha todos os tipos de variáveis possíveis e as combinações dos modificadores possíveis. Inicialize estas variáveis com valores típicos. Use o método para a conversão de tipos para converter:

a) A variável do tipo char em todos os outros tipos de variáveis.

b) A variável do tipo double em todos os outros tipos de variáveis. Discuta o que acontece com a precisão das variáveis.

2.3 Crie um programa que exemplifique a utilização de ponteiros. Que contenha pelo menos a declaração de um ponteiro, sua inicialização com zero, a obtenção do endereço de uma variável com o operador ‘&’ e o acesso ao dado representado pelo ponteiro.

2.4 Utilize as diretivas #define para criar constantes e empregue estas constantes para inicializar as variáveis do programa acima.

2.5 Analise o seguinte programa e aponte onde está o erro.

```
#define Default_Y 2
void main()
{
      const int num = 10;
      int y = Default_Y;
      const int *ptr = 0;
      ptr = &num;
      *ptr = *ptr + y;
}
```

3 Entrada/Saída Console

As rotinas de entrada/saída do console se encontram nas bibliotecas "stdio.h" e "conio.h" e, por isso, estas bibliotecas devem ser incluídas nos programas aplicativos através da diretiva 'include':

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

Algumas das funções de entrada e saída para a console mais utilizadas são apresentadas a seguir:

3.1 Printf()

A função printf()é uma das funções de E/S (entrada e saída) que podem ser usadas em C. Ela não faz parte da definição de C mas todos os sistemas têm uma versão de printf() implementada. Ela permite a saída formatada na tela. Já vimos duas aplicações diferentes da função printf():

```
printf("Bom Dia!!!!");
printf("Este é o número dois: %d", num);
```

A função printf() pode ter um ou vários argumentos. No primeiro exemplo nós colocamos um único argumento: "Bom Dia !!!!". Entretanto, no segundo colocamos dois: "Este é o número dois: %d" que está à esquerda e o valor 2 à direita da vírgula que separa os argumentos. Sintaxe de printf():

```
printf("string-formatação", < lista de parâmetros >);
```

A string de formatação pode conter caracteres que serão exibidos na tela e códigos de formatação que indicam o formato em que os argumentos devem ser impressos. No nosso segundo exemplo, o código de formatação **%d** solicita a printf() para imprimir o segundo argumento em formato decimal na posição da string onde aparece o **%d**. Cada argumento deve ser separado por uma vírgula.

Além do código de formatação decimal (**%d**), printf() aceita vários outros. O próximo exemplo mostra o uso do código **%s** para imprimir uma cadeia de caracteres:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   printf("%s esta a %d milhoes de milhas \n do sol.", "Venus", 67);
}
```

A saída será:

Venus está a 67 milhoes de milhas

do sol.

Aqui, além do código de formatação, a expressão de controle de printf() contém um conjunto de caracteres estranho: **\n**. O **\n** é um código especial que informa a printf() que o restante da impressão deve ser feito em uma nova linha. A combinação de caracteres **\n** representa, na verdade, um único caractere em C, chamado de nova-linha (equivalente ao pressionamento da tecla 'Enter' em um editor de texto).

Os caracteres que não podem ser obtidos diretamente do teclado para dentro do programa (como a mudança de linha) são escritos em C, como a combinação do sinal \ (barra invertida) com outros caracteres. Por exemplo, \n representa a mudança de linha.

A ***string de formatação*** define a forma como os parâmetros serão apresentados e tem os seguintes campos:

"%[Flags] [largura] [.precisão] [FNlh] < tipo > [\Escape Sequence]"

onde:

Flags	Efeito
-	justifica saída a esquerda
+	apresenta sinal (+ ou -) do valor da variável
Em branco	apresenta branco se valor positivo, sinal de - se valor negativo
#	apresenta zero no início p/ octais apresenta Ox para hexadecimais apresenta ponto decimal para reais

largura = número máximo de caracteres a mostrar;

precisão = número de casas após a vírgula a mostrar

F = em ponteiros, apresentar como "Far" => base : offset (xxxx : xxxx)

N = em ponteiros, apresentar como "Near" => offset

h = apresentar como "short"

l = apresentar como "long"

Escape Sequence	Efeito
\\	Barra
\"	Aspas
\0	Nulo
\a	Tocar Sino (Bell)
\b	Backspace – Retrocesso
\f	Salta Página de Formulário
\n	Nova Linha
\o	Valor em Octal
\r	Retorno do Cursor
\t	Tabulação
\x	Valor em hexadecimal

Tipo	Formato
%c	Caracter
%d, %i	Inteiro decimal (signed int)
%e, %E	Formato científico
%f	Real (float)
%l, %ld	Decimal longo
%lf	Real longo (double)
%o	Octal (unsigned int)
%p	Pointer xxxx (offset) se Near, xxxx : xxxx (base: offset) se Far
%s	Apontador de string, emite caracteres até aparecer caracter zero (00H)
%u	Inteiro decimal sem sinal (unsigned int)
%x	Hexadecimal

A seguir são mostrados alguns exemplos da formatação apresentada acima:

```
#include <stdio.h>
int main(int argc, char* argv[ ])
{
      float x;
      double y = -203.4572345;
      int a, b;
      a = b = 12;
      x = 3.141523;
      printf("Bom dia");
      printf("\n\t Bom dia\n"); /*pula linha após escrever bom dia*/
      printf("O valor de x é %7.3f\n", x);
      printf("Os valores de i, j e y são: %d %d %lf \n", a, b, y);
}
```

Obs: Caso você queira imprimir na tela os caracteres especiais '\' ou '%', você deve escrevê-los na função printf() de forma duplicada, o que indicará ao compilador que este não se trata de um parâmetro da função printf() mas sim que deseja-se imprimir realmente este caractere. O exemplo abaixo apresenta este caso:

```
#include <stdio.h>
void main()
{
      int reajuste = 10;
      printf("O reajuste foi de %d%%. \n", reajuste);
}
```

a saída será:

O reajuste foi de 10%.

3.2 Cprintf()

Basicamente cprintf() é igual a printf(), mas usa as coordenadas atuais do cursor e da janela que forem ajustados anteriormente, bem como ajustes de côr de caracteres. Utilize esta função para escrever na tela em posições pré-definidas.

3.3 Scanf()

A função scanf() é outra das funções de E/S implementadas em todos os compiladores C. Ela é o complemento de printf() e nos permite ler dados formatados da entrada padrão (teclado). A função scanf() suporta a entrada via teclado. Um espaço em branco ou um CR/LF (tecla "Enter") definem o fim da leitura. Observe que isto torna inconveniente o uso de scanf() para ler strings compostas de várias palavras (por exemplo, o nome completo de uma pessoa).

Para realizar este tipo de tarefa, veremos outra função mais adequada na parte referente à strings. Sua sintaxe é similar à de printf(), isto é, uma expressão de controle seguida por uma lista de argumentos separados por vírgula.

Sintaxe:

```
scanf("string de definição das variáveis", <endereço das variáveis>);
```

A string de definição pode conter códigos de formatação, precedidos por um sinal % ou ainda o caractere * colocado após o % que avisa à função que deve ser lido um valor do tipo indicado pela especificação, mas não deve ser atribuído a nenhuma variável (não deve ter parâmetros na lista de argumentos para estas especificações).

A lista de argumentos deve consistir nos endereço das variáveis. O endereço das variáveis pode ser obtido através do operador '&' apresentado na secção sobre ponteiros. Exemplo:

```
#include <stdio.h>
void main()
{
      float x;
      printf ("Entre com o valor de x : ");
      scanf ("%f",&x);  /*lê o valor do tipo float e armazena em x*/
}
```

Esta função funciona como o inverso da função printf(), ou seja, você define as variáveis que deveram ser lidas da mesma forma que definia estas com printf(). Desta forma, os parâmetros de scanf() são em geral os mesmos que os parâmetros de printf(), no exemplo acima observa-se esta questão.

O código de formatação de scanf() é igual ao código de formatação de printf(). Por isso, veja as tabelas de formatação de printf() para conhecer os parâmetros de scanf(). Outro exemplo de scanf():

```
#include <stdio.h>
main()
{
      float anos, dias;
      printf("Digite a sua idade em anos: ");
      scanf("%f", &anos);
      dias = anos*365;
      printf("Sua idade em dias e´ %.0f.\n", dias);
}
```

3.4 Getch(), Getche() e Getchar()

Em algumas situações, a função scanf() não se adapta perfeitamente pois você precisa pressionar [enter] depois da sua entrada para que scanf() termine a leitura. Neste caso, é melhor usar getch(), getche() ou getchar(). A função getchar() aguarda a digitação de um caractere e quando o usuário apertar a tecla [enter], esta função adquire o caractere e retorna com o seu

resultado. Getchar() imprime na tela o caractere digitado. A função getch() lê um único caractere do teclado sem ecoá-lo na tela, ou seja, sem imprimir o seu valor na tela. A função getche() tem a mesma operação básica, mas com eco. Ambas funções não requerem que o usuário digite [enter] para finalizar a sua execução. Veja exemplo abaixo, junto à função putch().

3.5 Putch() ou Putchar()

A função putch() apresenta um único caractere na tela, recebendo-o como parâmetro. A função putchar() executa a mesma operação, com caracteres. Exemplo:

```
#include <stdio.h>
#include <conio.h>
void main()
{
      char c1, c2;
      c1 = getch();   /* lê caractere c1 mas não mostra na tela */
      c2 = getche();  /* lê caractere c2, escrevendo-o na tela */
      printf("\nO primeiro valor digitado foi: ");
      putch(c1);      /* escreve valor de c1 na tela */
      printf("\nO segundo valor digitado foi: ");
      putchar(c2);
}
```

3.6 Exercícios

3.1 Escreva um programa que contenha uma única instrução e imprima na tela:

```
Esta e' a linha um.
 Esta e' a linha dois.
```

3.2 Escreva um programa que imprima na tela:

3.3 Escreva um programa que peça os dados da conta bancária de um cliente e, posteriormente, escreva na tela os dados do cliente também de forma organizada.

3.4 Escreva um programa que exemplifique todos os tipos de formatações fornecidas para o printf().

3.5 Escreva um programa que peça ao usuário para entrar com um valor real (float), converta este valor para uma variável do tipo char e depois imprima o seu valor na tela em formato decimal e em formato caractere.

3.6 Faça um programa que peça ao usuário para entrar com um caractere, converta este número para um valor inteiro e apresente na tela o valor deste número em formato float e o endereço da variável que o contém. Utilize ponteiros para armazenar e acessar este valor.

4 Operadores

4.1 Operadores Aritméticos

C é uma linguagem rica em operadores, em torno de 40. Alguns são mais usados que outros, como é o caso dos operadores aritméticos que executam operações aritméticas. C oferece 6 operadores aritméticos binários (operam sobre dois operandos) e um operador aritmético unário (opera sobre um operando). São eles:

Binários:

=	Atribuição
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (devolve o resto da divisão inteira)

Unário:

-	Menos unário (indica a troca do sinal algébrico do valor)

O operador = já é conhecido dos exemplos apresentados anteriormente. Em C, o sinal de igual não tem a interpretação dada em matemática. Representa a atribuição da expressão à direita ao nome da variável à esquerda. Já os operadores + - / * representam as operações aritméticas básicas de soma, subtração, divisão e multiplicação. A seguir está um programa que usa vários operadores aritméticos e converte temperatura Fahrenheit em seus correspondentes graus Celsius.

```
#include <stdio.h>
void main()
{
      int ftemp = 0;
      int ctemp = 0;
      printf("Digite temperatura em graus Fahrenheit: ");
      scanf("%d", &ftemp);
```

```
        ctemp = (ftemp - 32)*5/9;
        printf("Temperatura em graus Celsius e´ %d", ctemp);
}
```

O operador módulo (%) aceita somente operandos inteiros. Resulta o resto da divisão do inteiro à sua esquerda pelo inteiro à sua direita. Por exemplo, 17%5 tem o valor 2 pois quando dividimos 17 por 5 teremos resto 2.

4.2 Operadores Relacionais

Os operadores relacionais são usados para fazer comparações. Em C não existe um tipo de variável chamada "booleana", isto é, que assuma um valor verdadeiro ou falso. O valor zero (0) é considerado falso e qualquer valor diferente de 0 é considerado verdadeiro e é representado pelo inteiro 1. Os operadores relacionais comparam dois operandos e retornam 0 se o resultado for falso e 1 se o resultado for verdadeiro. Os operadores relacionais disponíveis em C são:

Operador C	Função
&&	E
||	OU
!	NÃO
<	MENOR
<=	MENOR OU IGUAL
>	MAIOR
>=	MAIOR OU IGUAL
==	IGUAL
!=	DIFERENTE
?:	OPERADOR CONDICIONAL TERNÁRIO

O programa a seguir mostra expressões booleanas como argumento da função printf():

```
#include <stdio.h>
void main()
{
int verdade, falso;
verdade = (15 < 20);
falso = (15 == 20);
```

```
printf("Verdadeiro= %d, falso= %d\n", verdade, falso);
}
```

Note que o operador relacional "igual a" é representado por dois sinais de iguais. Um erro comum é o de usar um único sinal de igual como operador relacional. O compilador não o avisará que este é um erro, por quê? Na verdade, como toda expressão C tem um valor verdadeiro ou falso, este não é um erro de programa e sim um erro lógico do programador.

4.3 Operadores lógicos binários

Estes operadores são empregados para comparar os bits contidos em duas variáveis, por isso, são denominados operadores lógicos binários. Ou seja, estes operadores fazem uma comparação lógica entre cada bit dos operandos envolvidos. Os operadores binários disponíveis são:

Operador C	Função
&	E lógico
|	OU lógico
^	Ou Exclusivo
~	Não
<<	Desloca esquerda
>>	Desloca direita

A seguir temos um exemplo da aplicação destes operadores.

```
#include <stdio.h>
void main()
{
      int i, j, k;
      i = 1;
      j = 2;
      printf("\ti = %x (00000001)\n\tj = %x (00000010)", i, j);
      k = i & j;                  /* k = i AND j */
      printf("\n\t i & j => %x (00000000)", k);
```

```
        k = i ^ j;                  /* k = i XOR j */
        printf("\n\t i ^ j => %x (00000011)", k);
        k = i << 2;      /*k = i deslocando 2 bits à esquerda (4)*/
        printf("\n\t i << 2 => %x (00000100)", k);
}
```

Primeiro o programa faz uma comparação binária 'e' (&) entre o número 1 (0x00000001) e o número 2 (0x00000010). Por isso o resultado obtido é logicamente 0, ou seja, 0x00000000. Em seguida, o programa faz uma comparação binária 'ou' (^) entre estes dois números, resultando agora, logicamente, 3, ou seja, (0x00000011). Por fim, o programa desloca o número 1 em duas casas para a esquerda, representado no programa pelo operador << 2. Assim, o resultado é o número 4 ou, em binário, (0x00000100). A partir deste exemplo fica clara a utilização destes operadores.

4.4 Operadores de Ponteiros

Estes operadores já foram apresentados anteriormente na secção sobre ponteiros (ver 2.7). O primeiro operador server para acessar o valor da variável, cujo endereço está armazenado em um ponteiro. O segundo operador server para obter o endereço de uma variável.

Operador	Função
*	Acesso ao conteúdo do ponteiro
&	Obtém o endereço de uma variável

O programa abaixo apresenta novamente a aplicação destes operadores.

```
void main()
{
        int i, j, *ptr;
        ptr = &i;  /* atribui a ptr o endereço da variável i */
        j = *ptr; /*atribui a j o conteúdo do endereço definido por*/
                  /* ptr = valor de i ! */
}
```

4.5 Operadores Incrementais e Decrementais

Uma das razões para que os programas em C sejam pequenos em geral é que C tem vários operadores que podem comprimir comandos de programas. Neste aspecto, destacam-se os seguintes operadores:

Operador	Função
++	Incrementa em 1
- -	Decrementa em 1

O operador de incremento (++) incrementa de um seu operando. Este operador trabalha de dois modos. O primeiro modo é chamado pré-fixado e o operador aparece antes do nome da variável. O segundo é o modo pós-fixado em que o operador aparece seguindo o nome da variável.

Em ambos os casos, a variável é incrementada. Porém quando **++n** é usado numa instrução, **n** é incrementada antes de seu valor ser usado, e quando **n++** estiver numa instrução, **n** é incrementado depois de seu valor ser usado. O programa abaixo mostra um exemplo destes operadores, ressaltando esta questão.

```
#include <stdio.h>
void main()
{
      int n, m, x, y;
      n = m = 5;
      x = ++n;
      y = m++;
      printf("O valor de n=%d, x=++n=%d, m=%d e y=m++=%d;",n,x,m,y);
}
```

Vamos analisar duas expressões seguintes.

a) k = 3*n++;

Aqui, primeiro ‘n’ é multiplicado por 3, depois o resultado é atribuído a ‘k’ e, finalmente, ‘n’ é incrementado de 1.

b) k = 3*++n;

Primeiro ‘n’ é incrementado em 1, depois ‘n’ é multiplicado por 3 e, por fim, o resultado é atribuído a ‘k’.

Dica: Para evitar problemas, faça uso de parênteses para guiar a execução do programa evitando que o compilador não compreenda a ordem correta de executar a operação. Abaixo, temos um outro exemplo dos operadores incrementais apresentado várias aplicações destes operadores:

```
void main()
{
      int i = 10;
      int *ptr = 0;
      i = i + 1;            /* incrementa i */
      i++;                  /* incrementa i */
      i = i - 1;            /* decrementa i */
      i--;                  /* decrementa i */
      ptr = &i;             /* recebe o endereco de i */
      (*ptr)++;             /* incrementa valor de i */
      ptr++;                /* incrementa em 2 Bytes (1 inteiro */
                            /* ocupa 2 Bytes) o valor do endereço */
                            /* apontado pelo ponteiro ptr */
}
```

Atenção para a última instrução **ptr++**. Esta instrução irá altera o endereço armazenado em ptr. Se o valor deste ponteiro for acessado, o dado representado por este ponteiro não tem mais nenhuma relação com a variável ‘i’, podendo conter qualquer dado. Por isso, tome cuidado ao manipular ponteiros para não acessar variáveis com uso desconhecido. Isto poderá fazer com que o seu programa gere erros fatais para o sistema operacional.

4.6 Operadores de Atribuição

O operador básico de atribuição (=) pode ser combinado com outros operadores para gerar instruções em forma compacta. Cada um destes operadores é usado com um nome de variável à sua esquerda e uma expressão à sua direita. A operação consiste em atribuir um novo valor à variável que dependerá do operador e da expressão à direita. Se **x** é uma variável, **exp** uma expressão e **op** um operador, então:

x op= exp; equivale a **x = (x)op(exp);**

Operador C:	Função:
=	A = B
+=	A += B; ⇒ A = A + B;
-=	A -= B; ⇒ A = A - B;
*=	A *= B; ⇒ A = A * B;
/=	A /= B; ⇒ A = A / B;
%=	A %= B; ⇒ A = A % B;
>>=	A >>= B; ⇒ A = A >> B;
<<=	A <<= B; ⇒ A = A << B;
&=	A &= B; ⇒ A = A & B;
|=	A |= B; ⇒ A = A | B;
^=	A ^= B; ⇒ A = A ^ B;

Exemplo:

```
#include <stdio.h>
void main()
{
      int total = 0;
      int cont = 10;
      printf("Total=%d\n", total);
      total += 1;
      printf("Total=%d\n", total);
```

```
        total ^= 2;
        printf("Total=%d\n", total);
        total <<= 2;
        printf("Total=%d\n", total);
        total *= cont;
        printf("Total=%d\n", total);
}
```

4.7 O Operador Lógico Ternário

O operador condicional possui uma construção um pouco estranha. È o único operador em C que opera sobre três expressões. Sua sintaxe geral possui a seguinte construção:

```
exp1 ? exp2 : exp3
```

A exp1 é avaliada primeiro. Se seu valor for differente de zero (verdadeira), a exp2 é avaliada e seu resultado será o valor da expressão condicional como um todo. Se exp1 for zero, exp3 é avaliada e será o valor da expressão condicional como um todo. Na expressão:

```
max = (a>b)?a:b
```

a variável que contém o maior valor numérico entre a e b será atribuída a max. Outro exemplo:

```
abs = (x > 0) ? x : -x;
```

A expressão

```
printf(" %d é uma variável %s !", x, ((x%2)?"Impar":"Par");
```

imprime ímpar se x for um número "ímpar", caso contrário imprimirá "par".

4.8 Precedência

O operador ! é o de maior precedência, a mesma que a do operador unário. A tabela seguinte mostra as precedências dos operadores:

Operadores	Tipos
! - ++ --	Unários; não lógicos e menos aritméticos
* / %	Aritméticos
+ -	Aritméticos
< > <= >=	Relacionais
== !=	Relacionais
&&	Lógico &
||	Lógico OU
= += -= *= /= %=	Atribuição

4.9 Exercícios:

4.1 Qual é o resultado da seguinte expressão:

```
int a = 1, b = 2, c = 3;
int result = ++a/a&&!b&&c||b--||-a+4*c>!!b;
```

4.2 Escreva uma expressão lógica que resulte 1 se o ano for bissexto e 0 se o ano não for bissexto. Um ano é bissexto se for divisível por 4, mas não por 100. Um ano também é bissexto se for divisível por 400.

4.3 Faça um programa que solicite ao usuário o ano e imprima “Ano Bissexto” ou “Ano Não-Bissexto” conforme o valor da expressão do exercício anterior. Utilize o operador condicional.

4.4 Num cercado, há vários patos e coelhos. Escreva um programa que solicite ao usuário o total de cabeças e o total de pés e determine quantos patos e quantos coelhos encontram-se nesse cercado.

4.5 Uma firma contrata um encanador a 20.000,00 por dia. Crie um programa que solicite o número de dias trabalhados pelo encanador e imprima a quantia líquida que deverá ser paga, sabendo-se que são descontados 8% para imposto de renda.

4.6 Faça um programa que solicite um caractere do teclado por meio da função getch(); se for uma letra minúscula imprima-a em maiúsculo, caso contrário imprima o próprio caractere. Use uma expressão condicional.

4.7 Faça um programa que solicite ao usuário uma seqüência binária com 16 bits. Transforme esta seqüência de 0 e 1 em um número inteiro. Depois manipule este número inteiro para verificar se os bits 0, 3, 7, 14 estão habilitados (valor igual a 1). Peça a usuário se ele deseja modificar algum bit específico da palavra. Se ele quiser, modifique o bit desejado para o valor por ele fornecido.

5 Laços

Em C existem 3 estruturas principais de laços: o laço **for**, o laço **while** e o laço **do-while**.

5.1 O Laço For

O laço **for** engloba 3 expressões numa única, e é útil principalmente quando queremos repetir algo um número fixo de vezes. Sintaxe:

```
for(inicialização; teste; incremento)
      instrução;
```

Os parênteses, que seguem a palavra chave for, contêm três expressões separadas por ponto-e- vírgulas, chamadas respectivamente de : "expressão de inicialização", "expressão de teste" e "expressão de incremento". As 3 expressões podem ser compostas por quaisquer instruções válidas em C.

• Inicialização: executada uma única vez na inicialização do laço. Serve para iniciar variáveis.

• Teste: esta é a expressão que controla o laço. Esta é testada quando o laço é inciado ou reiniciado. Sempre que o seu resultado for verdadeiro, a instruções do laço serão executadas. Quando a expressão se tornar falsa, o laço é terminado.

• Incremento: Define a maneira como a variável de controle do laço será alterada cada vez que o laço é repetido (conta++). Esta instrução é executada, toda vez, imediatamente após a execução do corpo do laço. Exemplo:

```
#include <stdio.h>
void main()
{
      int conta;
      for(conta = 0; conta < 10; conta += 3)
```

```
        printf("conta=%d\n", conta);
}
```

Qualquer uma das expressões de um laço for pode conter várias instruções separadas por vírgulas. A vírgula é na verdade um operador C que significa "faça isto e isto". Um par de expressões separadas por vírgula é avaliado da esquerda para a direita.

```
#include <stdio.h>
void main()
{
        int x, y;
        for( x=0, y=0; x+y < 100; x = x+1, y++)
        printf("%d\n", x+y);
}
```

Podemos usar chamadas a funções em qualquer uma das expressões do laço. Qualquer uma das três partes de um laço for pode ser omitida, embora os pontos-e-vírgulas devam permanecer. Se a expressão de inicialização ou a de incremento forem omitidas, elas serão simplesmente desconsideradas. Se a condição de teste não está presente é considerada permanentemente verdadeira.

O corpo do laço pode ser composto por várias instruções, desde que estas estejam delimitadas por chaves **{ }**. O corpo do laço pode ser vazio, entretanto o ponto-e-vírgula deve permanecer para indicar uma instrução vazia. Quando um laço está dentro de outro laço, dizemos que o laço interior está aninhado. Para mostrar esta estrutura preparamos um programa que imprime tabuada.

```
#include <stdio.h>
void main()
{
        int i,j,k;
        printf("\n");
        for(k=0; k<=1; k++)
        {
```

```
            printf("\n");
            for(i=1; i<5; i++)
                  printf("Tabuada do %3d ", i+4*k+1);
            printf("\n");
            for(i=1; i<=9; i++)
            {
                  for(j = 2+4*k; j<=5+4*k; j++)
                  printf("%3d x%3d = %3d ", j, i, j*i);
                  printf("\n");
            }
      }
}
```

5.2 O Laço While

O laço while utiliza os mesmos elementos que o laço for, mas eles são distribuídos de maneira diferente no programa. Sintaxe:

```
while(expressão de teste)
instrução;
```

Se a expressão de teste for verdadeira (diferente de zero), o corpo do laço while será executado uma vez e a expressão de teste é avaliada novamente. Este ciclo de teste e execução é repetido até que a expressão de teste se torne falsa (igual a zero), então o laço termina. Assim como o for, o corpo do laço while pode conter uma instrução terminada por (;), nenhuma instrução desde que possua um (;) e um conjunto de instruções separadas por chaves { }. Abaixo temos um exemplo, onde um laço while substituiu um antigo laço for.

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

```
        int conta = 0;
        while(conta < 10)
        {
              printf("conta=%d\n", conta);
              conta++;
        }
}
```

Quando se conhece a priori o número de loops que o laço deve fazer, recomenda-se o uso do laço for. Caso o número de iterações dependa das instruções executadas, então se recomenda o uso do laço while. Os laços while podem ser aninhados como o laço for. C permite que no interior do corpo de um laço while, você possa utilizar um outro laço while.

5.3 O Laço Do-While

O laço Do-While cria um ciclo repetido até que a expressão de teste seja falsa (zero). Este laço é bastante similar ao laço while. A diferença é que no laço do-while o teste de condição é avaliado depois do laço ser executado. Assim, o laço do-while é sempre executado pelo menos uma vez. Sintaxe:

```
do
{
      instrução;
} while(expressão de teste);
```

Embora as chaves não sejam necessárias quando apenas uma instrução está presente no corpo do laçop do-while, elas são geralmente usadas para aumentar a legibilidade. Exemplo usando os laços while e do-while:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>             /* requerida para rand() */
void main()
{
      char ch, c;
      int tentativas;
      do
      {
            ch = rand()%26 + 'a';
            tentativas = 1;
            printf("\nDigite uma letra de 'a' a 'z':\n");
            while((c=getch())!= ch)
            {
                  printf("%c e´ incorreto. Tente novamente. \n\n",c);
                  tentativas++;
            }
            printf("\n%c e´ correto",c);
            printf("\nvoce acertou em %d tentativas", tentativas);
            printf("\nQuer jogar novamente? (s\\n): ");
      }while(getche()=='s');
}
```

5.4 Break e Continue

O comando break pode ser usado no corpo de qualquer estrutura do laço C. Causa a saída imediata do laço e o controle passa para o próximo estágio do programa. Se o comando break estiver em estruturas de laços aninhados, afetará somente o laço que o contém e os laços internos a este. Já o comando continue força a próxima interação do laço e pula o código que estiver abaixo. Nos while e do-while um comando continue faz com que o controle do programa vá diretamente para o teste condicional e depois continue o processo do laço. No caso do laço for, o computador

primeiro executa o incremento do laço e, depois, o teste condicional, e finalmente faz com que o laço continue.

5.5 Goto

O comando goto faz com que o programa pule para a instrução logo após o label passado como parâmetro. Este comando é desnecessário e desaconselhado. Foi implementado somente para manter a compatibilidade com outros compiladores. A instrução goto tem duas partes: a palavra goto e um nome. O nome segue as conveções de nomes de variáveis C. Por exemplo:

```
goto parte1;
```

Para que esta instrução opere, deve haver um rótulo (label) em outra parte do programa. Um rótulo é um nome seguindo por dois pontos.

```
parte1:
     printf("Análise dos Resultados.\n");
```

A instrução goto causa o desvio do controle do programa para a instrução seguinte ao rótulo com o nome indicado. Os dois pontos são usados para separar o nome da instrução.

5.6 Exercícios

5.1 Escreva um programa que imprima os caracteres da tabela ASCII de códigos de 0 a 255. O programa deve imprimir cada caractere, seu código decimal e hexadecimal. Monte uma tabela usando os parâmetros de formatação de printf().

5.2 Escreva um programa, utilizando um laço while, que leia caracteres do teclado. Enquanto o usuário não pressionar a tecla ESC de código 27, os caracteres lidos não são ecoados no

vídeo. Se o caractere lido for uma letra minúscula, o programa a imprime em maiúsculo e, se for qualquer outro caractere, ele mesmo é impresso.

5.3 Elabore um programa que solicite um número inteiro ao usuário e crie um novo número inteiro com os dígitos em ordem inversa. Por exemplo, o número 3567 deve ser escrito 7653.

5.4 Escreva um programa que desenhe uma moldura na tela do micro, desenhando um sol com as tuas iniciais no meio da tela e escrevendo “Feliz Natal” e o seu nome embaixo.

6 Comandos para Tomada de Decisão

Uma das tarefas fundamentais de qualquer programa é decidir o que deve ser executado a seguir. Os comandos de decisão permitem determinar qual é a ação a ser tomada com base no resultado de uma expressão condicional. Isto significa que podemos selecionar entre ações alternativas dependendo de critérios desenvolvidos no decorrer da execução do programa.

C oferece 3 principais estruturas de decisão: **if, if-else, switch.** Estas estruturas são o tema deste capítulo.

6.1 If

O comando if é usado para testar uma condição e caso esta condição seja verdadeira, o programa irá executar uma instrução ou um conjunto delas. Este é um dos comandos básicos para qualquer linguagem de programação. Sintaxe:

```
if(expressão de teste)
      instrução;
```

Como C não possui variáveis booleanas, o teste sobre a condição opera como os operadores condicionais, ou seja, se o valor for igual a zero (0) a condição será falsa e se o valor for diferente de zero, a condição será verdadeira. Como os laços, o comando if pode ser usado para uma única instrução ou para um conjunto delas. Caso se utilize para um conjunto de instruções, este conjunto deve ser delimitado por chaves { e }.

Um comando if pode estar dentro de outro comando if. Dizemos então que o if interno está aninhado. O programa abaixo mostra exemplos do uso e da sintaxe dos comandos if. O primeiro if mostra a forma aninhada do comando if, o segundo apresenta a sintaxe básica e o último um exemplo onde o comando if executa um bloco de instruções.

```
#include <stdio.h>
#include <conio.h>
void main()
{
      char ch;
      printf("Digite uma letra de 'a' a 'z':");
      ch = getche();
      if(ch >= 'a')         /* Dois comandos if aninhados */
            if(ch <= 'z')
                  printf("\n Voce digitou um caractere valido!");
      if(ch == 'p')    /* Forma basica de um comando if */
            printf("\nVoce pressionou a tecla 'p'!");
      if(ch != 'p')    /*if executando um bloco de instrucoes*/
      {
            printf("\nVoce nao digitou a tecla 'p'!”);
            printf(“\n Digite um caractere para finalizar.");
            ch = getche();
      }
}
```

6.2 If-Else

No exemplo anterior o comando if executará uma única instrução ou um grupo de instruções, se a expressão de teste for verdadeira. Não fará nada se a expressão de teste for falsa. O comando else, quando associado ao if, executará uma instrução ou um grupo de instruções entre chaves, se a expressão de teste do comando if for falsa. O if-else também permite o aninhamento de outros comandos if, ou if-else dentro do bloco de instruções do após o else. Sintaxe:

```
if(expressão de teste)
      instrução_1;
else
      instrução_2;
```

Exemplo:

```
#include <stdio.h>
void main()
{
      int x, y;
      for(y=1; y<24; y++)              /* passo de descida */
      {
            for(x=1; x<24; x++)        /* passo de largura */
            if(x==y || x==24-y)        /* diagonal? */
                  printf("\xDB");      /* cor escura */
            else
                  printf("\xB0");      /* cor clara */
            printf("\n");              /* Nova Linha */
      }
}
```

6.3 Switch

O comando switch permite selecionar uma entre várias ações alternativas. Embora construções if-else possam executar testes para escolha de uma entre várias alternativas, muitas vezes são deselegantes. O comando switch tem um formato limpo e claro. A instrução switch consiste na palavra-chave switch seguida do nome de uma variável ou de um número constante entre parênteses. O corpo do comando switch é composto de vários casos rotulados por uma constante e opcionalmente um caso default. A expressão entre parênteses após a palavra-chave switch determina para qual caso será desviado o controle do programa.

O corpo de cada caso é composto por qualquer número de instruções. Geralmente, a última instrução é break. O comando break causa a saída imediata de todo o corpo do switch. Na falta do comando break, todas as instruções após o caso escolhido serão executadas, mesmo as que pertencem aos casos seguintes. Sintaxe:

```
switch(variável ou constante)
{
      case constante1:
            instrução;
            instrução;
            break;
      case constante2:
            instrução;
            instrução;
            break;
      case constante3:
            instrução:
            instrução:
            break;
      default:
            instrução;
            instrução;
}
```

Você não poderá usar uma variável nem uma expressão lógica como rótulo de um caso. Pode haver nenhuma, uma ou mais instruções seguindo cada caso. Estas instruções não necessitam estar entre chaves. O corpo de um switch deve estar entre chaves.

Se o rótulo de um caso for igual ao valor da expressão do switch, a execução começa nele. Se nenhum caso for satisfeito e existir um caso default, a execução começará nele. Um caso default é opcional. Não pode haver casos com rótulos iguais. A seguir apresentamos um exemplo que calcula o dia da semana a partir de uma data. O ano deve ser maior ou igual a 1600, pois nesta data houve uma redefinição do calendário.

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

```
int D, M, A, i;
long int F = 0;
do
{
      do
      {
            printf("\nDigite uma data na forma dd/mm/aaaa: ");
            scanf("%d/%d/%d", &D, &M, &A);
      } while(A<1600 || M<1 || M>12 || D<1 || D>31);
      F = A + D + 3*(M-1) - 1;
      if(M<3)
      {
            A--;
            i = 0;
      }
      else
            i = .4*M+2.3;
      F+= A/4 - i;
      i = A/100 + 1;
      i *= .75;
      F -= i;
      F %= 7;
      switch(F)
      {
            case 0:
                  printf("\nDomingo");
                  break;
            case 1:
                  printf("\nSegunda-Feira");
                  break;
            case 2:
                  printf("\nTerca-Feira");
                  break;
            case 3:
                  printf("\nQuarta-Feira");
```

```
                    break;
               case 4:
                    printf("\nQuinta-Feira");
                    break;
               case 5:
                    printf("\nSexta-Feira");
                    break;
               case 6:
                    printf("\nSabado");
                    break;
          }
     }while(getche()!=27);   /*Tecla ESC nao for pressionada*/
}
```

6.4 Exercícios

6.1 Crie um programa que desenhe na tela um tabuleiro de xadrez.

6.2 Crie um programa que desenhe na tela o seu quadro de horários deste semestre.

6.3 Escreva um programa que solicite ao usuário três números inteiros a, b, e c onda a é maior que 1. Seu programa deve somar todos os inteiros entre b e c que sejam divisíveis por a.

6.4 A sequência de Fibonacci é a seguinte: 1, 1, 2, 3, 5, 8, 13, 21,... Os dois primeiros termos são iguais a 1. Cada termo seguinte é igual `a soma dos dois anteriores. Escreva um programa que solicite ao usuário o número do termo e calcule o valor do termo. Por exemplo: se o número fornecido pelo usuário for igual a 7, o programa deverá imprimir 13.

6.5 Implemente um jogo da velha, onde o programa desenha na tela um tabuleiro e você pede a dois usuários (A e B) na respectiva ordem, que jogada eles querem realizar e depois atualiza o desenho. O programa deve perceber quando o jogo acabou e anunciar o vencedor.

7 Funções

As funções servem para agrupar um conjunto de instruções de acordo com a tarefa que elas desempenham. Uma vez implementada corretamente esta tarefa, você não precisa mais se preocupar em implementar esta tarefa, basta usar a sua função. Por exemplo, quando usamos printf() para imprimir informações na tela, não nos preocupamos como o programa realiza esta tarefa, pois a função já fornecia este serviço de forma adequada.

As funções quando bem empregadas, facilitam bastante a organização modular do programa, permitem a reutilização de partes do programa e facilitam a sua manutenção. Uma função é uma unidade de código de programa autônoma desenhada para cumprir uma tarefa particular. Provavelmente a principal razão da existência de funções é impedir que o programador tenha que escrever o mesmo código repetidas vezes.

7.1 Sintaxe

A estrutura de uma função C é bastante semelhante à da função main(). A única diferença é que main() possui um nome especial pois a função main() é a primeira a ser chamada quando o programa é executado. Sintaxe:

```
<tipo retorno> <nome>(<tipo parâmetro> <nome parâmetro>, <..> <..>)
{
      <declarações locais>;
      <comandos>;
      return <expressão ou valor compatível com o tipo de retorno>;
}
```

O *tipo de retorno* define o tipo de dado o qual a função retornará com o resultado de sua execução. *Nome* indica qual é o nome da função. Em seguida temos a lista de argumentos da função inseridos entre os parênteses após o nome da função e separados por vírgulas. Os argumentos são declarados no cabeçalho da função com o *tipo parâmetro* e em seguida o *nome do*

parâmetro. Uma função pode ter tantos argumentos quanto você queira definir. Quando uma função não necessita de parâmetros, o nome desta deve ser seguido por parênteses ( ) sem nenhum parâmetro no seu interior. A lista de instruções a serem executadas pela função vem após a lista de parâmetros e deve ser delimitada por chaves { e }. As instruções utilizam os parâmetros para executarem a tarefa.

Dentro do bloco de instruções da função, primeiramente declara-se e inicializa-se as variáveis locais e posteriormente temos os comandos da função (instruções). O comando return encerra a função e retorna ao ponto do programa onde esta foi chamada. No entanto, return não precisa ser o último comando da função. Infelizmente, uma função em C só pode retornar um resultado e este é repassado através de return. Algumas linguagens como Matlab permitem que uma função retorne mais do que uma única variável contendo o resultado.

Funções com tipo de retorno void, não utilizam o comando return pois estas não retornam nenhum tipo de dado. O comando return é utilizado quando queremos finalizar esta função antes de executar todas as instruções. Neste caso, ele aparece sem nenhum parâmetro a sua direita, somente o ponto-e-vírgula. Quando uma função não tem um tipo de retorno definido, o compilador C considera que o tipo de retorno adotado é void.

Do mesmo modo que chamamos uma função de biblioteca C (printf(), getche(), ...) chamamos nossas próprias funções. Os parênteses que seguem o nome são necessários para que o compilador possa diferenciar a chamada a uma função de uma variável que você esqueceu de declarar. Visto que a chamada de uma função constitui uma instrução de programa, deve ser encerrada por ponto-e-vírgula. Entretanto, na definição de uma função, o ponto-e-vírgula não pode ser usado.

7.2 Exemplos

Abaixo apresentamos um exemplo do emprego das funções:

```
#include <stdio.h>
doisbeep()
{
      int k;
```

```
        printf("\x7");              /* Beep 1 */
        for(k=1; k<10000; k++)      /* Gera um pequeno delay */
            ;
        printf("\x7"); /* Beep 2 */
}

/* funçao principal */
void main()
{
        doisbeep();
        printf("Digite um caractere: ");
        getche();
        doisbeep();
}
```

A declaração da função doisbeep() é equivalente a seguinte declaração: void doisbeep(void). Isto acontece pois quando o tipo retornado ou o parâmetro for void (representando que nenhum dado será passado), estes podem ser omitidos já que C considera este tipo como default.

Em C, todos os argumentos de funções são passados "por valor". Isto significa que à função chamada é dada uma cópia dos valores dos argumentos, e ela cria outras variáveis temporárias para armazenar estes valores. A diferença principal é que, em C, uma função chamada não pode alterar o valor de uma variável da função que chama; ela só pode alterar sua cópia temporária.

C permite a criação de funções recursivas, isto é, uma função que dentro do seu corpo de instruções {} existe uma chamada de função a si própria. Este caso funciona como uma chamada para uma outra função qualquer. Basta que na definição da função você faça uma chamada à própria função e você já terá implementado uma função recursiva. Entretanto, tenha cuidado com funções recursivas para não fazer com que o programa entre em loops infinitos. Exemplo:

```
long int Fatorial(long int i)
{
        if(i > 1)
            return i*Fatorial(i-1);
```

```
      else return 1;
}
```

Cada chamada recursiva da função "fatorial" coloca mais uma variável ***i*** do tipo **long int** na pilha (stack). É, portanto, necessário ter cuidado com funções recursivas, pois estas podem causar um "estouro" da pilha. Exemplo de função para calcular a área da esfera:

```
#include <stdio.h>
float area(float r);       /* declaracao do prototipo da funcao */
float potencia(float num, int pot);
void main()
{
      float raio = 0;
      printf("Digite o raio da esfera: ");
      scanf("%f", &raio);
      printf("A area da esfera e' %.2f", area(raio));
}

/* função area () - retorna a area da funcao */
float area(float r)                        /* definicao da funcao */
{
      return (4*3.14159*potencia(r,2));     /* retorna float */
}
/* funcao que eleva a uma potencia positiva um parametro dado */
float potencia(float num, int pot)
{
      float result = 0;          /* declaracao de var. local */
      int i = 0;
      if(pot < 0)
            return 0;            /* Indica que houve erro */
      if(pot == 0)
            return 1;
      result = num;
```

```
        for(i = 1; i < pot; i++)
        result *= num;
        return result;
}
```

7.3 Prototipagem

Toda função que for utilizada em main() e que for declarada após main(), tem que ser prototipada. O ***protótipo*** consiste de uma versão simplificada do cabeçalho da função, na forma:

```
<tipo retorno> <nome da função> (<tipo dos parâmetros>);
```

Se main() for declarado por último, nenhuma prototipagem é necessária. Protótipos são também utilizados para escrever programas compostos de vários módulos, como veremos mais à frente. Feita a declaração do protótipo da função, esta poderá ser chamada ou definida em qualquer parte do seu programa, por isso é que o protótipo de uma função tem um papel importante para a organização e flexibilidade do seu código fonte. O programa anterior e o exemplo abaixo apresentam a forma de se criar o protótipo de uma função.

```
double Raiz(double);        /* protótipo */
main()
{
        double x,y;
        x = 4.0;
        y = Raiz(x);
}
double Raiz(double valor)
{
        /* <calculo da raiz do valor>; */
        return valor;
}
```

7.4 Classes de Armazenamento

Todas as variáveis e funções em C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos nós já conhecemos. As 4 classes de armazenamento serão vistas a seguir:

auto, extern, static e register.

7.4.1 Auto

As variáveis que temos visto em todos os nossos exemplos estão confinadas nas funções que as usam; isto é, são visíveis ou acessíveis somente às funções onde estão declaradas. Tais variáveis são chamadas "locais" ou "automáticas", pois são criadas quando a função é chamada e destruída quando a função termina a sua execução.

As variáveis declaradas dentro de uma função são automáticas por "default". Variáveis automáticas são as mais comuns dentre as 4 classes. A classe de variáveis automáticas pode ser explicitada usando-se a palavra auto. As duas declarações abaixo são equivalentes:

```
auto int n;
int n;
```

7.4.2 Extern

Todas as funções C e todas as variáveis declaradas fora de qualquer função têm a classe de armazenamento extern. Variáveis com este atributo serão conhecidas por todas as funções declaradas depois dela. A declaração de variáveis externas é feita da mesma maneira como declaramos variáveis dentro do bloco de uma função.

Uma variável definida fora de qualquer função é dita extern. A palavra extern indica que a função usará mais uma variável externa. Este tipo de declaração, entretanto, não é obrigatório se a definição original ocorre no mesmo arquivo fonte. A seguir temos um exemplo:

```
#include <stdio.h>
int teclanum;         /* Declaracao de variaveis extern */
void parimpar(void);  /* Declaracao de funcoes */
void main()
{
      extern teclanum;
      printf("Digite teclanum: ");
      scanf("%d", &teclanum);
      parimpar();
}
/* parimpar() - checa se teclanum e' par ou impar */
void parimpar(void)
{
      extern teclanum;
      if(teclanum % 2)
            printf("teclanum e' impar. \n");
      else
            printf("teclanum e' par. \n");
}
```

7.4.3 Static

Variáveis static de um lado se assemelham às automáticas, pois são conhecidas somente as funções que as declaram e de outro lado se assemelham às externas pois mantém seus valores mesmo quando a função termina. Declarações static têm dois usos importantes e distintos. O mais elementar é permitir a variáveis locais reterem seus valores mesmo após o término da execução do bloco onde estão declaradas. Exemplo:

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

```
        soma();
        soma();
        soma();
}
/* soma() - usa variavel static */
void soma()
{
        static int i = 0;
        i++;
        printf("i = %d\n", i);
}
```

Observe que i não é inicializada a cada chamada de soma(). Portanto, a saída será:

```
i = 0
i = 1
i = 2
```

7.4.4 Variáveis Estáticas Externas

O segundo e mais poderoso uso de static é associado a declarações externas. Junto a construções externas, permite um mecanismo de "privacidade" muito importante à programação modular.

A diferença entre variáveis externas e externas estáticas é que variáveis externas podem ser usadas por qualquer função abaixo de (a partir das) suas declarações, enquanto que variáveis externas estáticas somente podem ser usadas pelas funções de mesmo arquivo-fonte e abaixo de suas declarações. Isto é, você pode definir uma variável static em um arquivo fonte onde estão todas as funções que necessitam acesso a esta variável. Os demais arquivos fontes não terão acesso a ela.

7.4.5 Register

A classe de armazenamento register indica que a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápida chamada registrador. Um registrador da máquina é um espaço de memória junto ao microprocessador onde se podem armazenar variáveis do tipo inteiro ou char. Opera como uma variável do tipo auto, mas possui um tempo de acesso muito mais rápido, isto é, o programa torna-se mais rápido usando register. Variáveis que usualmente utilizam este tipo são as variáveis de laços e argumentos de funções, justamente, para que possamos aumentar a velocidade do programa.

Como os computadores possuem um número limitado de registradores (2 para o IBM-PC), o seu programa também só poderá utilizar um número limitado de variáveis register. Mas isto não nos impede de declarar quantas variáveis register quisermos. Se os registradores estiverem ocupados o computador simplesmente ignora a palavra register das nossas declarações.

```
register int i;
for(i = 0; i< 10; i++)
printf("Número: %d", i);
```

7.5 Exercícios

7.1 Um número primo é qualquer inteiro positivo divisível apenas por si próprio e por 1. Escreva uma função que receba um inteiro positivo e, se este número for primo, retorne 1, caso contrário retorne 0. Faça um programa que imprima na tela os 'n' primeiros números primos, onde 'n' será fornecido pelo usuário.

7.2 Crie um programa para gerar números aleatórios, utilizando variáveis static para armazenar o valor da semente.

7.3 Crie um programa para calcular o fatorial de um número, para tal implemente uma função recursiva.

7.4 Escreva um programa que solicite ao usuário um ano e imprima o calendário desse ano. Utilize os programas desenvolvidos nos exercícios 4.2 e 4.3. Organize o programa de forma adequada através de funções.

7.5 Escreva uma função recursiva de nome soma() que receba um número inteiro positivo n como argumento e retorne a soma dos n primeiros números inteiros. Por exemplo, se a função recebe n = 5, deve retornar 15, pois:

15 = 1 + 2 + 3 + 4 + 5

8 Diretivas do Pré-Processador

O pré-processador C é um programa que examina o código fonte em C e executa certas modificações no código, baseado em instruções chamadas diretivas. O pré-processador faz parte do compilador e pode ser considerado uma linguagem dentro da linguagem C. Ele é executado automaticamente antes da compilação. Diretivas do pré-processador seriam instruções desta linguagem. As instruções desta linguagem são executadas antes do programa ser compilado e têm a tarefa de alterar os códigos-fonte, na sua forma de texto.

Instruções para o pré-processador devem fazer parte do texto que criamos, mas não farão parte do programa que compilamos, pois são retiradas do texto compilador antes da compilação. Diretivas do pré-processador são instruções para o compilador propriamente dito. Mais precisamente, elas operam diretamente no compilador antes do processo de compilação ser iniciado. Linhas normais de programa são instruções para o microprocessador; diretivas do pré-processador são instruções para o compilador.

Todas as diretivas do pré-processador são iniciadas com o símbolo (#). As diretivas podem ser colocadas em qualquer parte do programa, mas é costume serem colocadas no início do programa, antes de main(), ou antes do começo de uma função particular.

8.1 Diretiva #define

A diretiva #define pode ser usada para definir constantes simbólicas com nomes apropriados. Como por exemplo, podemos criar uma constante para conter o valor de pi:

```
#define   PI   3.14159
```

e depois utilizar o valor desta constante no programa:

```
area_esfera = (4*raio*raio*PI);
```

Quando o compilador encontra #define, ele substitui cada ocorrência de PI no programa por 3.14159. O primeiro termo após a diretiva #define (PI), que será procurada, é chamada "identificador". O segundo termo (3.14159), que será substituída, é chamada "texto". Um ou mais espaços separam o identificador do texto.

Note que só podemos escrever um comando deste por linha. Observe também que não há ponto-e-vírgula após qualquer diretiva do pré-processador. A diretiva #define pode ser usada não só para definir constantes, como mostra o exemplo seguinte:

```
#include <stdio.h>
#define  ERRO  printf("\n Erro.\n")
void main()
{
      int i;
      printf("\nDigite um numero entre 0 a 100: ");
      scanf("%d", &i);
      if(i<0||i>100)
            ERRO;
}
```

8.2 Macros

A diretiva #define tem a habilidade de usar argumentos. Uma diretiva #define com argumentos é chamada de macro e é bastante semelhante a uma função. Eis um exemplo que ilustra como macro é definida e utilizada:

```
#include <stdio.h>
#define  PRN(n)  printf("%.2f\n",n);
void main()
{
      float num1 = 27.25;
      float num2;
```

```
        num2 = 1.0/3.0;
        PRN(num1);
        PRN(num2);
}
```

Quando você usar a diretiva #define nunca deve haver espaço em branco no identificador. Por exemplo, a instrução:

```
#define PRN (n) printf("%.2f\n",n)
```

não trabalhará corretamente, porque o espaço entre PR e (n) é interpretado como o fim do identificador. Para definir um texto "muito grande" devemos delimitar a linha anterior com uma barra invertida \ e prosseguir com a definição em outra linha.

Toda ocorrência de PRN(n) em seu programa é substituída por printf("%.2f\n",n). O **n** na definição da macro é substituído pelo nome usado na chamada a macro em seu programa; portanto, PRN(num1)será substituído por printf("%.2f\n",num1). Assim, **n** age realmente com um argumento. Macros aumentam a clareza do código, pois permitem a utilização de nomes sugestivos para expressões. Por segurança, coloque parênteses envolvendo o texto todo de qualquer diretiva #define que use argumentos, bem como envolvendo cada argumento. Por exemplo:

```
#define  SOMA(x,y)  x+y
ans = 10 * SOMA(3,4);
```

O complilador substituíra SOMA(3,4) por 3+4, e o computador executará:

```
ans = 10 * 3+4;
```

ou seja, um erro. Para evitar isto, você deve definir a macro da seguinte forma:

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

Como macros são simples substituições dentro dos programas, o seu código aparecerá em cada ponto do programa onde forem usadas. Assim, a execução do programa será mais rápida que a chamada a uma função toada vez que se fizer necessário. Em contra partida, o código do programa será aumentado, pois o código da macro será duplicado cada vez que a macro for chamada. Outra vantagem do uso de macros é a não necessidade de especificar o tipo do dado.

8.3 Diretiva #undef

A diretiva #undef remove a mais recente definição criada com #define.

```
#define  GRANDE  3
#define  ENORME  8
#define  SOMA(x,y)  ((x)+(y))
...
#undef  GRANDE       /* cancela a definição de GRANDE */
#define  ENORME  10  /* redefine ENORME para o valor 10 */
...
#undef  ENORME       /* ENORME volta a valer 8 */
#undef  ENORME       /* cancela a definição de ENORME */
#undef  SOMA         /* cancela a macro SOMA */
```

Observe que, para remover uma macro, somente o nome da macro deve constar na diretiva #undef. Não deve ser incluída a lista de argumento.

8.4 Diretiva #include

A diretiva #include causa a inclusão de um código fonte em outro. Aqui está o exemplo de sua utilidade: suponha que você escreveu várias fórmulas matemáticas para calcular áreas de diversas figuras.

Os arquivos de inclusão são textos escritos em caracteres ASCII normais, criados geralmente para incorporar definições de constantes, macros, protótipos de funções, definições de tipos de dados complexos e declarações de variáveis externas. Geralmente, os arquivos de inclusão têm nome terminado com o sufixo ".H" (header ou cabeçalho). Por exemplo, o arquivo conio.h contém o protótipo das funções getch() e getche() e é por esse motivo incluído nos programas que fazem uso destas funções.

Você pode colocar estas fórmulas em macros em um programa separado. No instante em que você precisar reescrevê-las para a utilização em seu programa use a diretiva #include para inserir este arquivo no seu código fonte. Exemplo:

```
#define PI 3.14159
#define AREA_CIRCULO(raio) (PI*(raio)*(raio))
#define AREA_RETANG(base,altura) ((base)*(altura))
#define AREA_TRIANG(base,altura) ((base)*(altura)/2)
```

Grave o programa acima como areas.h. Quando você quiser utilizar estas macros, basta incluir nos seus outros programas uma das diretivas:

```
#include <areas.h>    /* para arquivos localizados na biblioteca */
                      /* do compilador */

#include "areas.h"    /* para arquivos locais ou localizados */
                      /* na biblioteca do compilador */
```

O uso consciente das diretivas #define e #include podem melhorar e muito o desempenho e a organização dos seus projetos de software. Procure explorá-las para conhecer todo os seus potenciais.

8.5 Compilação Condicional

O pré-processador oferece diretivas que permitem a compilação condicional de um programa. Elas facilitam o desenvolvimento do programa e a escrita de códigos com maior portabilidade de uma máquina para outra ou de um ambiente a outro. São elas, as diretivas:

#if, #ifdef, #ifndef, #elif, #else e ***#endif***

Cada diretiva #if deve terminar pela diretiva #endif. Entre #if e #endif pode ser colocado qualquer número de #elif, mas só se permite uma única diretiva #else. A diretiva #else é opcional e, se estiver presente, deve ser a última anterior a #endif. Abaixo apresentamos um exemplo onde definimos o valor de uma constante com #define e depois a utilizamos para fazer uma compilação condicional:

```
#define DEBUG 1
...
#if DEBUG == 1
     printf("\nERRO = ", erro1);
#elif DEBUG == 2
     printf("\nERRO = ", erro2);
#else
     printf("\nERRO não documentado.");
#endif
```

Para testar constantes definidas com #define que não tenham valor nenhum, podemos utilizar #ifdef e #ifndef. Por exemplo:

```
#define VERSAO_DEMO
...
#ifdef VERSAO_DEMO
     #define NUM_REC 20
     #include "progdemo.h"
```

```
else
      #define NUM_REC MAXINT
      #include "program.h"
#endif
```

O último exemplo mostra como um único programa-fonte pode gerar dois executáveis diferentes: se a diretiva que define VERSAO_DEMO for inserida, o executável poderá manipular somente 20 registros e estará criada a versão demo de seu programa. Caso esta diretiva não esteja presente, o programa poderá manipular o número máximo de registros. A diretiva #ifndef verifica a não-definição da constante. Por exemplo:

```
#ifndef WINDOWS
#define VERSAO "\nVersão DOS"
#else
#define VERSAO "\nVersão Windows"
#endif
```

8.6 Operador defined

Uma alternativa ao uso de #ifdef e #ifndef é o operador defined.

```
#if defined(UNIX) && !defined(INTEL_486)
...
#endif
```

8.7 Diretiva #error

A diretiva #error provoca uma mensagem de erro do compilador em tempo de compilação.

```
#if TAMANHO > TAMANHO1
```

```
#error "Tamanho incompatível"
#endif
```

8.8 diretiva *#pragma*

`#pragma inline` - indica ao compilador C a presença de código *Assembly* no arquivo.

`#pragma warn` - indica ao compilador C para ignorar "warnings" (avisos)

8.9 Exercícios

8.1 Escreva uma macro que tenha valor 1 se o seu argumento for um número ímpar, e o valor 0 se for par.

8.2 Escreva uma macro que encontre o maior entre seus três argumentos.

8.3 Escreva uma macro que tenha valor 1 se o seu argumento for um caractere entre '0' e '9', e 0 se não for.

8.4 Escreva uma macro que converta um dígito ASCII entre '0' e '9' a um valor numérico entre 0 e 9.

8.5 Escreva um programa que solicite ao usuário uma letra e retorne o número equivalente desta letra na tabela ASCII em decimal e hexadecimal. Utilize a compilação condicional para gerar um programa com interface para diferentes línguas, por exemplo, uma para português e outra para inglês. Caso nenhuma das duas for definida, gere um erro de compilação. Dica, coloque o texto de cada língua em um header diferente. Por exemplo, os textos em português ficam definidos em *"progport.h"* e os textos em inglês em *"progingl.h"*.

8.6 Faça com que o programa do exercício 7.4, escreva o calendário de um ano qualquer dado pelo usuário em três línguas diferentes: alemão, inglês e português. Utilize a mesma idéia que o programa anterior.

9 Matrizes

Uma matriz é um tipo de dado usado para representar uma certa quantidade de variáveis de valores homogêneos. Imagine que você esteja precisando armazenar e manipular as notas de um ano inteiro de um conjunto de 40 alunos. Você certamente precisará de uma maneira conveniente para referenciar tais coleções de dados similares. Matriz é o tipo de dado oferecido por C para este propósito. Um matriz é uma série de variáveis do mesmo tipo referenciadas por um único nome, onde cada variável é diferenciada através de um número chamado "subscrito" ou "índice". Colchetes são usados para conter o subscrito. A declaração:

```
int notas[5];
```

aloca memória para armazenar 5 inteiros e anuncia que notas é uma matriz de 5 membros ou "elementos". Vamos agora fazer um programa que calcula a média das notas da prova do mês de Junho de 5 alunos.

```
#include <stdio.h>
#define  NUM_ALUNOS  5
void main()
{
      int notas[NUM_ALUNOS];     /* Declaracao de matrizes */
      int i, soma;
      for(i=0; i<NUM_ALUNOS; i++)
      {
            printf("Digite a nota do aluno %d: ", i);
            scanf("%d", &notas[i]); /* atribuindo valores a matriz */
      }
      soma = 0;
      for(i=0; i<NUM_ALUNOS; i++)
      soma = soma + notas[i]; /* lendo os dados da matriz */
      printf("Media das notas: %d.", soma/NUM_ALUNOS);
}
```

9.1 Sintaxe de Matrizes

As dimensões são definidas entre colchetes, após a definição convencional do tipo de variável.

```
<Tipo> <nome> [<dimensão1>] [<dimensão2>] ... ;
```

Em C, matrizes precisam ser declaradas, como quaisquer outras variáveis, para que o compilador conheça o tipo de matriz e reserve espaço de memória suficiente para armazená-la. Os elementos da matriz são guardados numa seqüência contínua de memória, isto é, um seguido do outro.

O que diferencia a declaração de uma matriz da declaração de qualquer outra variável é a parte que segue o nome, isto é, os pares de colchetes [e] que envolvem um número inteiro, que indica ao compilador o tamanho da matriz. A dimensão da matriz vai depender de quantos pares de colchetes a declaração da matriz tiver. Por exemplo:

```
int notas[5]; /* declaração de uma matriz unidimensional = vetor */
unsigned float tabela[3][2]; /* matriz bidimensional */
char cubo[3][4][6]; /* matriz tridimensional */
```

Analisando-se a primeira declaração: a palavra int declara que todo elemento da matriz é do tipo int, notas é o nome dado a matriz e [5] indica que a nossa matriz terá 5 elementos do tipo “int”. Por definição uma matriz é composta por elementos de um único tipo.

O acesso aos elementos da matriz é feito através do número entre colchetes seguindo o nome da matriz. Observe que este número tem um significado diferente quando referencia um elemento da matriz e na declaração da matriz, onde indica o seu tamanho. Quando referenciamos um elemento da matriz este número especifica a posição do elemento na matriz. Os elementos da matriz são sempre numerados por índices iniciados por 0 (zero). O elemento referenciado pelo número 2:

```
notas[2]
```

não é o segundo elemento da matriz mas sim o terceiro, pois a numeração começa em 0. Assim o último elemento da matriz possui um índice, uma unidade menor que o tamanho da matriz.

O acesso com notas[i] pode ser tanto usado para ler ou escrever um dado na matriz. Você pode tratar este elemento como um variável simples, no caso de notas[i], como uma variável inteira qualquer.

Como proceder caso você não conheça a quantidade de termos que você precisa para a tua matriz? Aplique a idéia apresentada no programa exemplo acima, com a diretiva:

```
#define  NUM_ALUNOS  5
```

para alocar um número maior de termos, de forma a possuir espaço suficiente alocado para conter todos os possíveis termos da matriz. **Atenção:** A linguagem C não realiza verificação de limites em matrizes; por isto nada impede que você vá além do fim da matriz. Se você transpuser o fim da matriz durante uma operação de atribuição, então atribuirá valores a outros dados ou até mesmo a uma parte do código do programa. Isto acarretará resultados imprevisíveis e nenhuma mensagem de erro do compilador avisará o que está ocorrendo.

Como programador você tem a responsabilidade de providenciar a verificação dos limites, sempre que necessário. Assim a solução é não permitir que o usuário digite dados, para elementos da matriz, acima do limite.

9.2 Inicializando Matrizes

Em C a inicialização de uma matriz é feita em tempo de compilação. Matrizes das classes extern e static podem ser inicializadas. Matrizes da classe auto não podem ser inicializadas. Lembre-se de que a inicialização de uma variável é feita na instrução de sua declaração e é uma maneira de especificarmos valores iniciais pré-fixados. O programa a seguir exemplifica a inicialização de uma matriz:

```
#include <stdio.h>
#define  LIM  5
void main()
{
```

```
        int d, valor, quant;
        static int tab[] = {50, 25, 10, 5, 1};
        printf("Digite o valor em centavos: ");
        scanf("%d", &valor);
        for(d=0; d<LIM; d++)
        {
              quant = valor/tab[d];
              printf("Moedas de %d centavos = %2d\n", tab[d], quant);
              valor %= tab[d];
        }
}
```

A instrução que inicializa a matriz é:

```
static int tab[] = {50, 25, 10, 5, 1};
```

A lista de valores é colocada entre chaves e os valores são separados por vírgulas. Os valores são atribuídos na seqüência, isto é, tab[0] = 50, tab[1] é 25, tab[2] é 10 e assim por diante. Note que não foi necessária a declaração da dimensão da matriz. Caso tivéssemos escrito explicitamente a dimensão da matriz, esta deveria se maior ou igual a dimensão fornecida pelo colchetes. Se há menos inicializadores que a dimensão especificada, os outros serão zero. É um erro ter-se mais inicializadores que o necessário.

Se em seu programa você deseja inicializar uma matriz, você deve criar uma variável que existirá durante toda a execução do programa. As classes de variáveis com esta característica são extern e static.

A linguagem C permite matrizes de qualquer tipo, incluindo matrizes de matrizes. Por exemplo, uma matriz de duas dimensões é uma matriz em que seus elementos são matrizes de uma dimensão. Na verdade, em C, o termo duas dimensões não faz sentido pois todas as matrizes são de uma dimensão. Usamos o termo mais de uma dimensão para referência a matrizes em que os elementos são matrizes.

As matrizes de duas dimensões são inicializadas da mesma maneira que as de dimensão única, isto é, os elementos (matrizes) são colocados entre as chaves depois do sinal de igual e separados por vírgulas. Cada elemento é composto por chaves e seus elementos internos separados por vírgulas. Como exemplo segue abaixo uma matriz bidimensional:

```
char tabela[3][5] = { {0, 0, 0, 0, 0},
                      {0, 1, 1, 1, 0},
                      {1, 1, 0, 0, 1} };
```

Cada elemento da matriz tabela na verdade será outra matriz, então, por exemplo:

```
tabela[2] == {1, 1, 0, 0, 1}
```

Da mesma maneira se aplica para matrizes de três ou mais dimensões:

```
int tresd[3][2][4] = { { {1, 2, 3, 4} , {5, 6, 7, 8} },
                       { {7, 9, 3, 2} , {4, 6, 8, 3} },
                       { {7, 2, 6, 3} , {0, 1, 9, 4} } };
```

A matriz de fora tem três elementos, cada um destes elementos é uma matriz de duas dimensões de dois elementos onde cada um dos elementos é uma matriz de uma dimensão de quatro números. Como podemos acessar o único elemento igual a 0 na matriz do exemplo anterior? O primeiro índice é [2], pois é o terceiro grupo de duas dimensões. O segundo índice é [1], pois é o segundo de duas matrizes de uma dimensão. O terceiro índice é [0], pois é o primeiro elemento da matriz de uma dimensão. Então, temo que a primitiva abaixo é verdade:

```
tresd[2][1][0] == 0
```

9.3 Matrizes como Argumentos de Funções

C permite passar um matriz como parâmetro para uma função. A seguir apresentamos como exemplo o método de Ordenação Bolha como exemplo de passagem de matrizes como argumentos. Este é um método famoso de ordenação de elementos em uma matriz devido a sua simplicidade e eficiência. A função começa considerando a primeira variável da matriz, list[0]. O objetivo é colocar o menor item da lista nesta variável. Assim, a função percorre todos os itens restantes da lista, de list[1] a list[tam-1], comparando cada um com o primeiro item. Sempre que encontrarmos um item menor, eles são trocados. Terminada esta operação, é tomado o próximo item, list[1]. Este item deverá conter o próximo menor item. E novamente são feitas as comparações e trocas. O processo continua até que a lista toda seja ordenada.

```
#include <stdio.h>
#define TAMAX 30
void ordena(int list[], int tam);
void main()
{
      int list[TAMAX], tam=0, d;
      do
      {
            printf("Digite numero: ");
            scanf("%d", &list[tam]);
      } while(list[tam++] != 0);
      ordena(list, --tam);
      for(d=0; d<tam; d++)
            printf("%d\n", list[d]);
}
/* ordena() - ordena matriz de inteiros */
void ordena(int list[], int tam)
{
      int register j, i;
      int temp;
```

```
        for(j = 0; j < tam-1; j++)
        for(i = j + 1; i < tam; i++)
        if(list[j] > list[i])
        {
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
        }
}
```

O único elemento novo no programa acima é a instrução:

```
ordena(list, --tam);
```

ela é uma chamada à função ordena() que ordena os elementos de uma matriz segundo o método bolha. Esta função tem dois argumentos: o primeiro é a matriz list e o segundo é a variável que contém o tamanho da matriz.

A matriz foi passada como argumento da função quando escrevemos somente o seu nome como parâmetro da função. Quando desejamos referenciar um elemento da matriz devemos acessá-lo através do nome da matriz seguida de colchetes, onde o índice do elemento é escrito entre os colchetes. Entretanto, se desejamos referenciar a matriz inteira, basta escrever o nome desta, sem colchetes. O nome de uma matriz desacompanhada de colchetes representa o endereço de memória onde a matriz foi armazenada.

O operador **&** não pode ser aplicado a matriz para se obter o endereço desta, pois, a endereço da matriz já está referenciado no seu nome. Portanto, a instrução abaixo seria inválida:

```
&list;
```

O operador & somente pode ser aplicado a um elemento da matriz. Portanto, a instrução:

```
int * ptr = &list[0];
```

será válido e irá retornar o endereço da primeira variável da matriz. Como a área de memória ocupada por uma matriz começa através do primeiro termo armazenado nesta, portanto, os endereços da matriz e do primeiro elemento serão equivalentes:

```
list == &list[0];
```

Assim, a função ordena() recebe um endereço de uma variável int e não um valor inteiro e deve declará-lo de acordo. O tipo :

```
int []
```

indica um endereço de uma variável int.

9.4 Chamada Por Valor e Chamada Por Referência

Quando um nome de uma simples variável é usado como argumento na chamada a uma função, a função toma o valor contido nesta variável e o instala em uma nova variável e em nova posição de memória criada pela função. Portanto as alterações que forem feitas nos parâmetros da função não terão nenhum efeito nas variações usadas na chamada. Este método é conhecido como **chamada por valor**.

Entretanto, as matrizes são consideradas um tipo de dado bastante grande, pois são formadas por diversas variáveis. Por causa disto, a linguagem C++ determina ser mais eficiente existir uma única cópia da matriz na memória, sendo, portanto, irrelevante o número de funções que a acessem.

Assim, não são passados os valores contidos na matriz, somente o seu endereço de memória. Desta forma, a função usa o endereço da matriz para acessar diretamente os elementos da própria matriz da função que chama. Isto significa que as alterações que forem feitas na matriz pela função afetarão diretamente a matriz original.

Quando se passa o nome de uma matriz para uma função, não se cria uma cópia dos dados desta mas sim, utiliza-se diretamente os dados da matriz original através do endereço passado. Esta tipo de passagem de parâmetros é denominado **passagem de parâmetros por referência**.

A seguir apresentamos um exemplo de passagem de uma matriz bidimensional como parâmetro de uma função. O programa exemplo avalia a eficiência de funcionários de uma loja, quanto ao número de peças vendidas por cada um. O primeiro índice da matriz indica o número de funcionários da loja e o segundo índice, o número de meses a serem avaliados.

```
#include <stdio.h>
#define FUNC 4        /* numero de funcionarios */
#define MES 3         /* numero de meses */
#define MAXBARRA 50   /* tamanho maximo da barra */
void grafico(int p[][MES], int nfunc); /* declaracao da funcao */

void main()
{
      int pecas[FUNC][MES]; /* declaracao de matriz bidimensional*/
      int i, j;
      for(i=0; i<FUNC; i++)
            for(j=0; j<MES; j++)
            {
                  printf("Numero de pecas vendidas pelo funcionario");
                  printf("%d no mes %d: ", i+1, j+1);
                  scanf("%d", &pecas[i][j]);
            }
      grafico(pecas, FUNC);
}

/* grafico() - desenha um histograma */
void grafico(int p[][MES], int nfunc)
{
      int register i, j;
      int max, tam = 0, media[FUNC];
```

```
        for(i=0; i<nfunc; i++)
        {
              media[i] = 0;
              for(j=0; j<MES; j++)
                    media[i] += p[i][j];
              media[i] /= MES;
        }
        max = 0;
        for(i = 0; i< nfunc; i++)
              if(media[i]>max)
                    max = media[i];
        printf("\n\n\n\n FUNC - MEDIA\n--------------\n");
        for(i=0;i<FUNC; i++)
        {
              printf("%5d - %5d : ", i+1, media[i]);
              if(media[i]>0)
              {
                    if((tam=(int)(((float)media[i])*MAXBARRA/max)) <= 0)
                    tam = 1;
              }
              else
                    tam = 0;
              for(; tam>0; --tam)
                    printf("%c", '>');
              printf("\n");
        }
}
```

O método de passagem do endereço da matriz para a função é idêntico ao da passagem de uma matriz de uma dimensão, não importando quantas dimensões tem a matriz, visto que sempre passamos o seu endereço.

```
grafico(pecas, FUNC);
```

Como não há verificação de limites, uma matriz com a primeira dimensão de qualquer valor pode ser passada para a função chamada. Mas uma função que recebe uma matriz bidimensional deverá ser informada do comprimento da segunda dimensão para saber como esta matriz está organizada na memória e assim poder operar com declarações do tipo:

```
pecas[2][1]
```

pois, para encontrar a posição na memória onde pecas[2][1] está guardada, a função multiplica o primeiro índice (2) pelo número de elementos da segunda dimensão (MES) e adiciona o segundo índice (1), para finalmente somar ao endereço da matriz (pecas). Caso o comprimento da segunda matriz não seja conhecido, será impossível saber onde estão os valores.

9.5 Strings

C não possui um tipo de dado especial para tratar strings. Desta forma, uma das aplicações mais importante de matrizes em C é justamente a criação do tipo string, caracterizado por uma sequência de caracteres armazenados terminados por '\0' (que possui o valor 0 na tablea ASCII) em uma matriz. Por esta razão é que C não fornece uma estrutura amigável para a manipulação de string comparada com outras linguagens como Pascal, por exemplo. Somente com o advento da orientação a objetos (C++) é que C veio a fornecer uma estrutura amigável e flexível para tratar strings. Esta estrutura será vista somente nos capítulos posteriores. Aqui, apresentaremos string na sua forma básica em C.

String é uma matriz tipo de char que armazena um texto formado de caracteres e sempre terminado pelo caractere zero ('\0'). Em outras palavras, string é uma série de caracteres, onde cada um ocupa um byte de memória, armazenados em seqüência e terminados por um byte de valor zero ('\0'). Cada caractere é um elemento independente da matriz e pode ser acessado por meio de um índice.

9.5.1 Strings Constantes

Qualquer seqüência de caracteres delimitadas por aspas duplas é considerada pelo compilador como uma string constante. O compilador aloca uma matriz com tamanho uma (1) unidade maior que o número de caracteres inseridos na string constante, armazenando estes caracteres nesta matriz e adicionando automaticamente o caractere delimitador de strings '\0' no final da matriz. Vários exemplos de strings constantes estão presentes ao longo de texto, exemplos como :

```
printf("Bom Dia!!!!");
```

O importante é lembrar que o compilador sempre adiciona o caractere '\0' a strings constantes antes de armazená-las na forma de matriz para poder marcar o final da matriz. Observe que o caractere '\0', também chamado de NULL, tem o valor zero decimal (tabela ASCII), e não se trata do caractere '0', que tem valor 48 decimal (tabela ASCII). A terminação '\0' é importante, pois é a única maneira que as funções possuem para poder reconhecer onde é o fim da string.

9.5.2 String Variáveis

Um das maneiras de receber uma string do teclado é através da função scanf() pelo formato %s. A função scanf() lê uma string até você apertar a tecla [enter], adiciona o caractere '\0' e armazena em uma matriz de caracteres. Perceba que o endereço da matriz e passado escrevendo-se somente o nome da matriz. Abaixo apresentamos um exemplo, nele fica claro que se você digitar uma frase com um número maior de caracteres do que a matriz alocada, (char[15]) um erro de memória irá ocorrer. Isto se deve ao fato de scanf() não testar a dimensão de matriz e simplesmente adquirir uma seqüência de caracteres até encontrar um caractere do [enter], espaço simples ou tabulação. Depois ela armazena a string na matriz fornecida, sem testa a dimensão da matriz. A definição de uma string segue o mesmo formato da definição de matrizes, entretanto o tipo empregado é o tipo char.

```
#include <stdio.h>
void main()
{
      char nome[15]; /* declara uma string com 15 caracteres */
      printf("Digite seu nome: ");
      scanf("%s", nome);
      printf("Saudacoes, %s.", nome);
}
```

Se você escrever Pedro Silva no programa acima, a função scanf() irá eliminar o segundo nome. Isto se deve por que scanf() utiliza qualquer espaço, quebra de linha ou tabulação para determinar o fim da leitura de uma string.

Neste caso, a função gets() se aplica de forma mais eficiente para a aquisição de strings. A função gets() adquire uma string até se pressionada a tecla [enter], adicionando o caractere '\0' para especificar o fim da string e armazená-la em uma matriz. Desta forma podemos utilizar gets() para ler strings com espaços e tabulações.

Para imprimir strings utilizamos as duas funções principais printf() e puts(). O formato de impressão de strings com prinf() nós já conhecemos, basta escrever o parâmetro %s no meio da string de formatação de printf() e posteriormente passar o nome da matriz de caracteres, terminada com '\0', como parâmetro.

Puts() tem uma sintaxe bem simples também. Chamando a função puts() e passando-se o nome da matriz de caracteres terminada por '\0' como parâmetro, teremos a string impressa na tela (saída padrão) até encontrar o caractere '\0'. Após a impressão, puts() executa uma quebra de linha automaticamente [enter], o que impossibilita a sua aplicação para escrever duas strings na mesma linha. Abaixo, temos um exemplo:

```
char nome[] = "Carlos Henrique";
puts(nome);
```

Em C, a forma formal da inicialização de strings ocorre da mesma maneira que uma matriz, ou seja, inicializamos termo por termo como a seguir:

```
char text[] = {'B', 'o', 'm', ' ', 'd', 'i', 'a', '!', '\0'};
```

C fornece ainda uma maneira bem simples, que foi apresentada no outro exemplo acima. Note que na forma formal de inicialização temos que adicionar explicitamente o caractere '\0' para determinar o fim da matriz. Já na forma simples, o compilador faz isto automaticamente para você.

9.5.3 Funções para Manipulação de Strings

Em C existem várias funções para manipular matrizes e tentar tornar a vida do programador um pouco mais fácil. Veremos aqui as 4 funções principais: strlen(), strcat(), strcmp() e strcpy(). Note que estas funções exigem que o caractere '\0' esteja delimitando o final da string. A função strlen() aceita um endereço de string como argumento e retorna o tamanho da string armazenada a partir deste endereço até um caractere antes de '0', ou seja, ela retorna o tamanho da matriz que armazena a string menos um, descontando assim o caractere de final da string '\0'. Sintaxe:

```
strlen(string);
```

A função strcat() concatena duas strings, isto é, junta uma string ao final de outra. Ela toma dois endereços de strings como argumento e copia a segunda string no final da primeira e esta combinação gera uma nova primeira string. A segunda string não é alterada. Entretanto, a matriz da primeira string deve conter caracteres livres (ou seja, caracteres alocados localizados após o delimitador de final de string '\0') suficientes para armazenar a segunda string. A função strcat() não verifica se a segunda string cabe no espaço livre da primeira. Sintaxe de strcat():

```
strcat(string1, string2);
```

A função strcmp() compara duas strings e retorna:

< 0	:	no caso da string 1 < que a string 2
0	:	no caso da string 1 = a string 2
> 0	:	no caso da string 1 > que a string2

Neste contexto, “menor que” (<) ou “maior que” (>) indica que, se você colocar string1 e string2 em ordem alfabética, o que aparecerá primeiro será menor que o outro. Sintaxe:

```
strcmp(string1, string2);
```

A função strcpy() simplesmente copia uma string para outra. Note, que a string1 deve conter espaço suficiente para armazenar os dados da string2. Sintaxe:

```
strcpy(string1, string2);
```

Abaixo vamos mostrar um exemplo que apagar um caractere do interior de uma string. Neste exemplo vamos implementar a função strdel() que move, um espaço à esquerda, todos os caracteres que estão a direita do caractere sendo apagado.

```
#include <stdio.h>
void strdel(char str[], int n);
void main()
{
      char string[81];
      int posicao;
      printf("Digite string, [enter], posicao\n");
      gets(string);
      scanf("%d", &posicao);
      strdel(string, posicao);
      puts(string);
}

/* strdel() - apaga um caractere de uma string */
void strdel(char str[], int n)
{
      strcpy(&str[n], &str[n+1]);
}
```

9.6 Exercícios

9.1 Crie um programa para calcular a matriz transposta de uma dada matriz. Aloque uma memória para uma matriz bidimensional com dimensão máxima de 10x10. Crie uma função para inicializar a matriz com zero. Depois questione o usuário para sob a dimensão da matriz que ele deseja calcular a transposta, considerando a dimensão máxima permitida. Depois, adquira os valores dos termos que compõem a matriz, solicitando ao usuário que forneça estes dados. Por fim, calcule a transporta da matriz e escreva na tela o resultado final da matriz.

9.2 Continue o programa anterior e agora calcule o tamanho de memória alocada para a matriz e que não está sendo utilizada. Ou seja, considerando o tipo de dado da matriz, que a matriz inicial tem tamanho 10x10 e o tamanho utilizado pelo usuário, quantos bytes de memória o computador alocou para a matriz e quantos bytes de memória o usuário realmente utilizou. Crie uma função que calcule este dado comparando duas matrizes de dimensões quaisquer. Reflita sobre este problema.

9.3 A decomposição LDU é uma decomposição famosa de matrizes aplicada para calcular inversas de matrizes e resolver problemas de equações lineares. Utilize o programa 9.1 e crie um programa que calcule a decomposição LDU de uma matriz, preferencialmente, com pivotamento.

9.4 Escreva uma função que indique quantas vezes aparece um determinado caractere em uma dada string.

9.5 Escreva uma função que localize um caractere em uma string, substituindo-a por outro.

9.6 Escreva uma função que insira um determinado caractere em uma determinada posição de uma string.

9.7 Escreva uma função que retire todos os caracteres brancos, tabulações ou nova linha [enter] de uma dada string.

9.8 Escreva um programa que converta todas os caracteres minúsculos de uma string para o correspondente caractere maiúsculo.

9.9 Refaça o seu exercício 6.2 para criar uma tabela com os seus horários ocupados e compromissos na semana. Armazene o valor de cada compromisso através de uma tabela de strings. Inicialize a tabela com valor 0, e solicite ao usuário que forneça o seu horário. Por fim, apresente na tela o resultado obtido.

10 Tipos Especiais de Dados

Em C podemos definir novos tipos de dados, adicionando complexidade aos tipos de dados já existentes. Com a finalidade de tornar a vida do programador mais fácil e permitir que este crie novos tipos de dados, compostos pelos tipos de dados já pré-existentes (char, int, float, double e matrizes destes tipos).

10.1 Typedef

Typedef nos apresenta a primeira forma de criar um novo tipo de dado. ***Typedef*** permite a definição de novos tipos de variáveis. Os novos tipos são sempre compostos de uma ou mais variáveis básicas agrupadas de alguma maneira especial. Sintaxe:

```
typedef <tipo> <definição>;
```

Exemplo da definição de um novo tipo de dado:

```
typedef double real_array [10];       /* novo tipo */
real_array x, y;           /* declara variáveis tipo real_array */
```

No exemplo acima, ambas ‘x’ e ‘y’ foram declaradas como sendo do tipo ‘real-array’ e, por isso, representam um vetor de valores reais (double) com capacidade para armazenar 10 dados deste tipo.

10.2 Enumerados (Enum)

Permitem atribuir valores inteiros seqüenciais à constantes. Tipos enumerados são usados quando conhecemos o conjunto de valores que uma variável pode assumir. A variável deste tipo é sempre int e, para cada um dos valores do conjunto, atribuímos um nome significativo para representá-lo. A palavra enum enumera a lista de nomes automaticamente, dando-lhes números em

seqüência (0, 1, 2, etc.). A vantagem é que usamos estes nomes no lugar de números, o que torna o programa mais claro. Abaixo são apresentados exemplos da aplicação de tipos enumerados e suas definições:

```
enum dias
{
      segunda,          /* atribui: segunda = 0; */
      terca,            /* atribui: terca = 1; */
      quarta,           /* atribui: quarta = 2; */
      quinta,           /* atribui: quinta = 3; */
      sexta,            /* atribui: sexta = 4; */
      sabado,           /* atribui: sabado = 5; */
      domingo           /* atribui: domingo = 6; */
};
enum cores
{
      verde = 1,
      azul = 2,
      preto, /*não foi dado valor, recebe valor anterior + 1 = 3*/
      branco = 7
};
enum boolean
{
      false,      /* false = 0 */
      true        /* true = 1 */
};
void main()
{
      enum dias dia1, dia2; /*declara variável "dias" do tipo enum*/
      enum boolean b1;      /*declara variável do tipo enum boolean*/
      dia1 = terca;
      dia2 = terca + 3;     /* dia2 = sexta */
      if (dia2 == sabado)
      b1 = true;
}
```

A palavra enum define um conjunto de nomes com valores permitidos para esse tipo e enumera esses nomes a partir de zero (default) ou, como em nosso programa, a partir do primeiro valor fornecido. Se fornecermos um valor a alguns nomes e não a outros, o compilador atribuirá o próximo valor inteiro aos que não tiverem valor. A sintaxe fica clara no exemplo acima.

Um variável de tipo enumerado pode assumir qualquer valor listado em sua definição. Valores não listados na sua definição não podem ser assumidos por esta variável. Tipos enumerados são tratados internamente como inteiros, portanto qualquer operação válida com inteiros é permitida com eles.

10.3 Estruturas (Struct)

Estruturas permitem definir estruturas complexas de dados, com campos de tipos diferentes. Observe que nas matrizes (arrays), todos os campos são do mesmo tipo. As structs permitem, entretanto, criar elementos semelhantes a arrays, mas composta de elementos com tipos diferentes de dados.

Por meio da palavra-chave struct definimos um novo tipo de dado. Definir um tipo de dado significa informar ao compilador o seu nome, o seu tamanho em bytes e o formato em que ele deve ser armazenado e recuperado na memória. Após ter sido definido, o novo tipo existe e pode ser utilizado para criar variáveis de modo similar a qualquer tipo simples. Abaixo apresentamos as duas sintaxes que podem ser empregadas para a definição de estruturas e para a declaração de variáveis deste tipo:

Sintaxe :	Exemplo :

struct <nome> /* Definição */
{
  <tipo> <nome campo>;
}[<variáveis deste tipo>];
 	
/* Declaração de Variáveis */
struct <nome> <nome variável>;
	struct Dados_Pessoais
	{
	  char Nome[81];
	  int Idade;
	  float Peso;
	} P1;
	
	/* Declaração de Variáveis */
	struct Dados_Pessoais P2, P3;

typedef struct <nome> /* Def.*/
{
  <tipo> <nome campo>;
} <nome tipo>;
 	
/* Declaração de variáveis/
<nome tipo> <nome variável>;
	typedef struct Dados_Pessoais
	{
	  char Nome [81];
	  int Idade;
	  float Peso;
	} Pessoa;
	
	/* Declaração de variáveis/
	Pessoa P1, P2;

Definir uma estrutura não cria nenhuma variável, somente informa ao compilador as características de um novo tipo de dado. Não há nenhuma reserva de memória. A palavra struct indica que um novo tipo de dado está sendo definido e a palavra seguinte será o seu nome. Desta forma, faz-se necessário a declaração de variáveis deste tipo. Na primeira sintaxe, mostramos no exemplo como podemos declarar um variável do tipo da estrutura "Dados_Pessoais" já na definição da estrutura (P1) e como podemos criar outras variáveis deste tipo ao longo do programa (P2 e P3). Já na segunda sintaxe, a forma da declaração de variáveis deve ser sempre explícita, como mostrado no exemplo.

Uma vez criada a variável estrutura, seus membros podem ser acessados por meio do operador ponto. O operador ponto conecta o nome de uma variável estrutura a um membro desta. Abaixo, fornecemos exemplos de acesso as variáveis definidas por meio de estruturas:

```
gets(P1.Nome);
P1.Idade = 41;
P3.Peso = 75.6;
```

A linguagem C trata os membros de uma estrutura como quaisquer outras variáveis simples. Por exemplo, P1.Idade é o nome de uma variável do tipo int e pode ser utilizada em todo lugar onde

possamos utilizar uma variável int. A inicialização de estruturas é semelhante à inicialização de uma matriz. Veja um exemplo:

```
struct data
{
      int dia;
      char mes[10];
      int ano;
};
data natal = { 25, "Dezembro", 1994};
data aniversario = { 30, "Julho", 1998};
```

Uma variável estrutura pode ser atribuída através do operador igual (=) a outra variável do mesmo tipo:

```
aniversario = natal;
```

Certamente constatamos que esta é uma estupenda capacidade quando pensamos a respeito: todos os valores dos membros da estrutura estão realmente sendo atribuídos de uma única vez aos correspondentes membros da outra estrutura. Isto já não é possível, por exemplo, com matrizes.

Estruturas não permitem operações entre elas, somente entre os seus membros. Após definirmos um tipo de dado através de uma estrutura, podemos incluir este tipo de dado como membro de uma outra estrutura. Isto cria o que chamamos tipos aninhados de estruturas, onde uma estrutura contém no seu interior como membro outra estrutura. Esta operação funciona assim como uma matriz bidimensional na verdade é uma matriz de matrizes unidimensionais.

Estruturas podem ser passadas para funções assim como qualquer outra variável, sendo declarada e utilizada da mesma forma. Funções podem também retornar uma estrutura como o resultado do seu processamento. Para tal, basta declarar o tipo de retorno de uma função como sendo um tipo declarado por uma struct. Esta é uma forma de fazer com que uma função retorne mais de um parâmetro.

Podemos criar uma matriz para estruturas assim como criamos uma matriz de um tipo qualquer, basta declarar o tipo da matriz como o tipo definido pela estrutura. A seguir apresentamos

um exemplo envolvendo matrizes de estruturas, onde trataremos de uma matriz de estruturas que contém os dados necessários para representar um livro. A forma de acesso aos membros da estrutura, armazenadas na matriz fica clara no exemplo. As funções atoi() e atof() apresentadas no exemplo abaixo servem para converter strings para um tipo inteiro ou float, respectivamente, e estão definidas em stdlib.

```
#include <stdio.h>
#include <stdlib.h>        /* para atof() e atoi() */
/* definiçoes de tipo */
struct list
{
      char titulo[30];
      char autor[30];
      int regnum;
      double preco;
};
/* declaracao de variaveis e funcoes*/
struct list livro[50];
int n = 0;
void novonome();
void listatotal();

void main()
{
      char ch;
      while(1)
      {
            printf("\nDigite 'e' para adicionar um livro");
            printf("\n'l' para listar todos os livros: ");
            ch = getche();
            switch(ch)
            {
                  case 'e' :
                        novonome();
```

```
                    break;
               case 'l':
                    listatotal();
                    break;
               default:
                    puts("\nDigite somente opcoes validas!!!!");
          }
     }
}
/* novonome() - adiciona um livro ao arquivo */
void novonome()
{
     char numstr[81];
     printf("\nRegistro %d. \nDigite titulo: ", n+1);
     gets(livro[n].titulo);
     printf("\nDigite autor: ");
     gets(livro[n].autor);
     printf("\nDigite o numero do livro (3 digitos): ");
     gets(numstr);
     livro[n].regnum = atoi(numstr); /* converte string p/ int */
     printf("\nDigite preco: ");
     gets(numstr);
     livro[n].preco = atof(numstr);
     n++;
}
/* listatotal() - lista os dados de todos os livros */
void listatotal()
{
     int i;
     if(!n)
          printf("\nLista vazia.\n");
     else
          for(i=0; i<n; i++)
          {
               printf("\nRegistro: %d.\n", i+1);
```

```
                printf("\nTitulo: %s.\n", livro[i].titulo);
                printf("\nAutor: %s.\n", livro[i].autor);
                printf("\nNr do registro: %3d.\n", livro[i].regnum);
                printf("\nPreco: %4.3f. \n", livro[i].preco);
            }
}
```

10.4 Uniões

A palavra union é usada, de forma semelhante a struct, para agrupar um número de diferentes variáveis sob um único nome. Entretanto, uma union utiliza um mesmo espaço de memória a ser compartilhado com um número de diferentes membros, enquanto uma struct aloca um espaço diferente de memória para cada membro.

Em outras palavras, uma union é o meio pelo qual um pedaço de memória ora é tratado como uma variável de um certo tipo, ora como outra variável de outro tipo. Por isso, uniões são usadas para poupar memória e o seu uso não é recomendado a não ser em casos extremamente necessários, pois a utilização irresponsável das uniões podem causar a perda de dados importantes do programa. Quando você declara uma variável do tipo union, automaticamente é alocado espaço de memória suficiente para conter o seu maior membro. Sintaxe:

```
union <nome>
{
      <tipo> <campo1>;
      :
      <tipo n> <campo n>;
};
```

Exemplo do emprego das uniões:

```
typedef unsigned char byte;
union Oito_bytes
{
      double x; /*um double de 8 bytes */
```

```
      int i[4]; /* um array com 4 inteiros = 8 bytes */
      byte j[8]; /* um array de 8 bytes */
} unionvar;

void main()
{
      unionvar.x = 2.7;
      unionvar.i[1] = 3; /* sobreescreve valor anterior ! */
}
```

Para verificar que o tamanho de uma variável union é igual ao tamanho do maior membro, vamos utilizar a função sizeof(). A função sizeof() resulta o tamanho em bytes ocupado por uma dada variável. Veja o teste realizado abaixo, onde apresentamos o valor em bytes ocupado por cada membro da união e o valor em bytes ocupado pela união inteira. Podemos verificar também que todos os membros da união ocupam a mesma posição da memória utilizando o operador & para obter a posição da memória onde estão armazenados os dados.

```
#include <stdio.h>
union num
{
      char str[20];
      int i;
      float f;
} x;                  // Cria a variavel do tipo union num.

void main()
{
  printf("Sizeof(str[20])=%d bytes, mem:%p.\n",sizeof(x.str),x.str);
  printf("Sizeof(i) = %d bytes, \tmem: %p.\n", sizeof(x.i), &(x.i));
  printf("Sizeof(f) = %d bytes, \tmem: %p.\n", sizeof(x.f), &(x.f));
  printf("Sizeof(x) = %d bytes, \tmem: %p.\n", sizeof(x), &(x));
}
```

A saída do programa será como esperado:

```
Sizeof(str[20]) = 20 bytes, mem: 50EF:0D66.
Sizeof(i) = 2 bytes, mem: 50EF:0D66.
Sizeof(f) = 4 bytes, mem: 50EF:0D66.
Sizeof(x) = 20 bytes, mem: 50EF:0D66.
```

10.5 Bitfields

Bitfields são um tipo especial de estrutura cujos campos tem comprimento especificado em bits. Estas são úteis quando desejamos representar dados que ocupam somente um bit. Sintaxe:

```
struct <nome>
{
     <tipo> <campo> : <comprimento em bits>;
};
```

Exemplo aplicando bitfields:

```
#define ON 1
#define OFF 0
struct Registro_Flags /* declara bitfield */
{
     unsigend int Bit_1 : 1;
     unsigend int Bit_2 : 1;
     unsigend int Bit_3 : 1;
     unsigend int Bit_4 : 1;
     unsigend int Bit_5 : 1;
     unsigend int Bit_6 : 1;
     unsigend int Bit_7 : 1;
     unsigend int Bit_8 : 1;
};
```

```
main( )
{
      struct Registro_Flags Flags;       /*declara variável Flags*/
      Flags.Bit_1 = ON;
      Flags.Bit_2 = OFF;
}
```

10.6 Exercícios

10.1 Escreva uma estrutura para descrever um mês do ano. A estrutura deve ser capaz de armazenar o nome do mês, a abreviação em letras, o número de dias e o número do mês. Escreva também um tipo enumerado para associar um nome ou uma abreviação ao número do mês.

10.2 Declare uma matriz de 12 estruturas descritas na questão anterior e inicialize-a com os dados de um ano não-bissexto.

10.3 Escreva uma função que recebe o número do mês como argumento e retorna o total de dias do ano até aquele mês. Escreva um programa que solicite ao usuário o dia, o mês e o ano. Imprima o total de dias do ano até o dia digitado.

10.4 Escreva um programa para cadastrar livros em uma biblioteca e fazer consultas a estes.

10.5 Refaça o programa do exercício 8.6 utilizando as estruturas aqui desenvolvidas.

10.6 Crie uma estrutura para descrever restaurantes. Os membros devem armazenar o nome, o endereço, o preço médio e o tipo de comida. Crie uma matriz de estruturas e escreva um programa que utilize uma função para solicitar os dados de um elemento da matriz e outra para listar todos os dados.

10.7 Crie um programa que faça o controle de cadastro de usuários e controle de alocação para uma locadora de fitas de vídeo. Utilize uma estrutura para os clientes e outra para as fitas.

11 Ponteiros e a Alocação Dinâmica de Memória

No capítulo sobre tipos de dados, definimos o que era ponteiro. Ponteiros são tipos especiais de dados que armazenam o endereço de memória onde uma variável normal armazena seus dados. Desta forma, empregamos o endereço dos ponteiros como uma forma indireta de acessar e manipular o dado de uma variável. Agora apresentaremos diversas aplicações de ponteiros, justificaremos por que este são intensivamente empregados na programação C e apresentaremos algumas de suas aplicações típicas.

11.1 Declaração de Ponteiros e o Acesso de Dados com Ponteiros

C permite o uso de ponteiros para qualquer tipo de dado, sendo este um dado típico de C (int, float, double, char) ou definido pelo programador (estruturas, uniões). Como havíamos visto anteriormente (2.7), os ponteiros são declarados da seguinte forma:

```
<tipo> * <nome da variável> = <valor inicial>;
float *p, float *r = 0;
```

A sintaxe basicamente é a mesma que a declaração de variáveis, mas o operador * indica que esta variável é de um ponteiro, por isso ***p** e ***r** são do tipo **float** e que **p** e **r** são ponteiros para variáveis **float**. Lembrando,

```
*p = 10;
```

faz com que a variável endereçada por p tenha o seu valor alterado para 10, ou seja, * é um operador que faz o acesso a variável apontada por um ponteiro. O operador & aplicado a uma variável normal retorna o valor do seu endereço, podemos dizer então:

```
int num = 15;
p = &num; /* p recebe o endereço de 'num' */
```

11.2 Operações com Ponteiros

A linguagem C oferece 5 operações básicas que podem ser executadas em ponteiros. O próximo programa mostras estas possibilidades. Para mostrar o resultado de cada operação, o programa imprimirá o valor do ponteiro (que é um endereço), o valor armazenado na variável apontada e o endereço da variável que armazena o próprio ponteiro.

```
#include <stdio.h>
void main()
{
      int x = 5, y = 6;
      int *px;
      int *py = 0; /* ponteiro inicializado com 0 */
      printf("Endereco contigo incialmente em :\n");
      printf("\tpx = %p \n\tpy = %p.\n", px, py);
      printf("Cuidado - ponteiros nao incializados como px!!!\n\n");
      px = &x;
      py = &y;
      if(px<py)
            printf("py - px = %u\n", py-px);
      else
            printf("px - py = %u\n", px-py);
      printf("px = %p, \t*px = %d, \t&px = %p\n", px, *px, &px);
      printf("py = %p, \t*py = %d, \t&py = %p\n\n", py, *py, &py);
      px++;
      printf("px = %p, \t*px = %d, \t&px = %p\n\n", px, *px, &px);
      py = px + 3;
      printf("py = %p, \t*py = %d, \t&py = %p\n", py, *py, &py);
      printf("py - px = %u\n", py-px);
}
```

Resultado do programa:

```
Endereço contido inicialmente em:
px = 21CD:FFFF
py = 0000:0000
Cuidado - ponteiros não inicializados como px!!!!
px – py = 1
px = 4F97:2126, *px = 5, &px = 4F97:2120
py = 4F97:2124, *py = 6, &py = 4F97:211C
px = 4F97:2128, *px = 20375, &px = 4F97:2120
py = 4F97:212E, *py = 20351, &py = 4F97:211C
py – px = 3
```

Primeiramente, o programa exemplo acima declara as variáveis inteiras 'x' e 'y', declarando dois ponteiros para inteiros também. Note que um ponteiro foi inicializado com '0' e o outro não. Para mostrar a importância da inicialização de ponteiros, resolvemos mostrar na tela o valor inicial destes. Caso, tentássemos acessar o valor da variável apontada pelo ponteiro 'py', o programa não executaria esta operação pelo fato do ponteiro conter um endereço inválido '0'. Entretanto, se tentarmos acessar o valor da variável apontada por 'px', o computador executará esta operação sem problema, pois o endereço está válido, e assim poderemos cometer um erro gravíssimo acessando áreas de memórias inadequadas ou proibidas.

Inicialize sempre os seus ponteiros com o valor zero ou 'NULL'. Em seguida, atribuímos um valor válido aos ponteiros utilizando o operador (&) para obter o endereço das variáveis 'x' e 'y'. O operador (*) aplicado na frente do nome do ponteiro, retornará o valor da variável apontada por este ponteiro.

Como todas as variáveis, os ponteiros variáveis têm um endereço e um valor. O operador (&) retorna a posição de memória onde o ponteiro está localizado. Em resumo:

• O nome do ponteiro retorna o endereço para o qual ele aponta.

• O operador & junto ao nome do ponteiro retorna o endereço onde o ponteiro está armazenado.

• O operador * junto ao nome do ponteiro retorna o conteúdo da variável apontada.

No programa acima, apresentamos esta idéia de forma clara, mostrando na tela estes valores para os dois respectivos ponteiros. Podemos incrementar um ponteiro através de adição regular ou pelo operador de incremento. Incrementar um ponteiro acarreta a movimentação do mesmo para o próximo tipo apontado. Por exemplo, se px é um ponteiro para um inteiro e px contém o endereço 0x3006. Após a execução da instrução:

```
px++;
```

o ponteiro px conterá o valor 0x3008 e não 0x3007, pois a variável inteira ocupa 2 bytes. Cada vez que incrementamos px ele apontará para o próximo tipo apontado, ou seja, o próximo inteiro. O mesmo é verdadeiro para decremento. Se px tem valor 0x3006 depois da instrução:

```
px--;
```

ele terá valor 0x3004. Você pode adicionar ou subtrair de e para ponteiros. A instrução:

```
py = px + 3;
```

fará com que py aponte para o terceiro elemento do tipo apontado após px. Se px tem valor 0x3000, depois de executada a instrução acima, py terá o valor 0x3006. Novamente, tenha cuidado na manipulação de ponteiros para não acessar regiões inválidas de memória ou ocupadas por outros programas.

Você pode encontrar a diferença entre dois ponteiros. Esta diferença será expressa na unidade tipo apontado, então, se py tem valor 0x4008 e px o valor 0x4006, a expressão py – px terá valor 1 quando px e py são ponteiros para int. Testes relacionais com >=, <=, > e < são aceitos entre ponteiros somente quando os dois operandos são ponteiros. Outro cuidado a se tomar é quanto ao tipo apontado pelo operandos. Se você comparar ponteiros que apontam para variáveis de tipo diferentes, você obterá resultados sem sentido.

Variáveis ponteiros podem ser testadas quanto à igualdade (==) ou desigualdade (!=) onde os dois operandos são ponteiros (px == py) ou um dos operandos NULL (px != NULL ou px != 0).

11.3 Funções & Ponteiros

Um das maneiras mais importantes para passar argumentos para uma função ou para que a função retorne um resultado é através de ponteiros. Ponteiros são empregados em funções principalmente quando necessitamos de umas das seguintes características:

• ponteiros permitem que uma função tenha acesso direto as variáveis da função chamadora, podendo efetuar modificações nestas variáveis;

• para tipos de dados complexos ou de grande dimensão (estruturas, matrizes, ...) é mais fácil e mais rápido acessar estes dados indiretamente através de um ponteiro do que realizar uma cópia destes para que a função possa utilizar o valor copiado.

• Podemos utilizar ponteiros de estruturas complexas ou matrizes como forma de fazer com que funções retornem mais de um valor como resultado;

• A passagem de parâmetro com ponteiros geralmente é mais rápida e eficiente do que com o próprio dado, haja visto que o compilador não faz uma cópia deste dado e um ponteiro necessita somente de 4 bytes para armazenar o seu valor.

Para tal, basta que em vez da função chamadora passar valores para a função chamada, esta passe endereços usando operadores de endereços (&). Estes endereços são de variáveis da função chamadora onde queremos que a função coloque os novos valores. Estes endereços devem ser armazenados em variáveis temporárias da função chamada para permitir posteriormente o seu acesso. No exemplo a seguir mostramos como ponteiros podem ser usados como parâmetros da função.

```
#include <stdio.h>
void altera(int *, int *);
void main()
{
      int x = 0, y = 0;
      altera(&x, &y);
      printf("O primeiro e %d, o segundo e %d.", x, y);
}
```

```
/* altera(), altera dois numeros da funcao que chama*/
void altera(int *px, int *py)
{
      if(px != 0)
            *px = 3;
      if(py != 0)
      *py = 5;
}
```

Note, que o objetivo da função é prover modificações nas duas variáveis x e y. Numa forma simples, só poderíamos efetuar esta operação utilizando-se uma estrutura para conter x e y, ou criando uma função para alterar cada parâmetro. Entretanto, passando-se o endereço das variáveis x e y permitimos a função chamada que altera-se diretamente os seus valores.

O valor do endereço das variáveis foi obtido com o operador &. A função necessitou apenas da declaração de um tipo ponteiro em seu cabeçalho para estar apta a receber endereços de variáveis como um de seus parâmetros.

Uma vez conhecidos os endereços e os tipos das variáveis do programa chamador, a função pode não somente colocar valores nestas variáveis como também tomar o valor já armazenado nelas. Ponteiros podem ser usados não somente para que a função passe valores para o programa chamador, mas também para que o programa chamador passe valores para a função.

Outro aspecto importante é o teste efetuado antes de utilizar a o ponteiro recebido como parâmetro:

```
if(px != 0)
*px = 3;
```

Isto serve para que a função verifique se o valor de endereço recebido é válido. Tome muito cuidado com a manipulação de ponteiros, um erro no programa ou em uma função do sistema operacional, pode gerar um ponteiro inválido e o acesso a área de memória representada por este ponteiro pode causar um erro fatal no seu programa. Este é um dos métodos para prever erros com ponteiros.

11.4 Ponteiros & Matrizes

Você não percebe, mas C trata matrizes como se fossem ponteiros. O tipo matriz é na verdade uma forma mais amigável que C fornece para tratar um ponteiro que aponta para uma lista de variáveis do mesmo tipo. O compilador transforma matrizes em ponteiros quando compila, pois a arquitetura do microcomputador entende ponteiros e não matrizes.

Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros. O nome de uma matriz é um endereço, ou seja, um ponteiro. Ponteiros e matrizes são idênticos na maneira de acessar a memória. **Na verdade, o nome de uma matriz é um ponteiro constante. Um ponteiro variável é um endereço onde é armazenado um outro endereço.** Suponha que declaramos uma matriz qualquer:

```
int table = {10, 1, 8, 5, 6}
```

e queremos acessar o terceiro elemento da matriz. Na forma convencional por matriz, fazemos isto através da instrução:

```
table[2]
```

mas a mesma instrução pode ser feita com ponteiros, considerando que o nome do vetor é na verdade um ponteiro para o primeiro elemento do vetor:

```
*(table+2)
```

A expressão (table+2) resulta no endereço do elemento de índice 2 da matriz. Se cada elemento da matriz é um inteiro (2 bytes), então vão ser pulados 4 bytes do início do endereço da matriz para atingir o elemento de índice 2. Em outras palavras, a expressão (table+2) não significa avançar 2 bytes além de table e sim 2 elementos da matriz: 2 inteiros se a matriz for inteira, 2 floats se a matriz for float e assim por diante. Ou seja,

```
*(matriz + indice) == matriz[indice]
```

Existem duas maneiras de referenciar o endereço de um elemento da matriz: em notação de ponteiros, matriz+indice, ou em notação de matriz, &matriz[indice]. Vamos mostrar um exemplo para esclarecer estes conceitos:

```
#include <stdio.h>
#define LIM 40
void main()
{
      float notas[LIM], soma = 0.0;
      int register i = 0;
      do
      {
            printf("Digite a nota do aluno %d: ", i);
            scanf("%f", notas+i);
            if(*(notas+i) > 0)
                  soma += *(notas+i);
      }while(*(notas+i++) > 0);
      printf("Media das notas: %.2f", soma/(i-1));
}
```

O operador de incremento ou decremento não pode ser aplicado a ponteiros constantes. Por exemplo, não podemos substituir o comando (notas + i++) por (notas++), haja visto que notas é um ponteiro constante. Para fazer isto, precisamos criar um ponteiro variável qualquer, atribuir a ele o endereço da matriz e depois incrementar o valor deste ponteiro:

```
float * ptr = 0;
ptr = notas;
(notas + i++) == (ptr++) /* são equivalentes */
```

Como, ptr aponta para uma matriz do tipo float, o operador (++) incrementa em 4 bytes a cada operação. Podemos passar matrizes como parâmetros para funções, passando-se como parâmetro um ponteiro para o primeiro elemento da matriz e depois utilizando este ponteiro para

acessar todos os elementos da matriz. Esta sintaxe funciona exatamente igual como se estivéssemos passando um ponteiro de uma variável normal como parâmetro. Veja a secção anterior para verificar esta sintaxe.

11.5 Ponteiros & Strings

Como strings são na verdade tratadas em C como matrizes simples de caracteres finalinazadas por '0' (caractere '\0'), podemos utilizar as facilidades fornecidas pelos ponteiros para manipular strings também. Abaixo mostramos um exemplo que ilustra este caso:

```
#include <stdio.h>
#include <conio.h>
char * procstr(char *, char);
void main()
{
      char *ptr;
      char ch, lin[81];
      puts("Digite uma sentenca: ");
      gets(lin);
      printf("Digite o caractere a ser procurado: ");
      ch=getche();
      ptr = procstr(lin, ch);
      printf("\nA string comeca no endereco %p.\n", lin);
      if(ptr) /* if(ptr != 0) */
      {
            printf("Primeira ocorrencia do caractere: %p.\n", ptr);
            printf("E a posicao: %d", ptr-lin);
      }
      else
            printf("\nCaractere nao existe.\n");
}
```

```
char * procstr(char * l, char c)
{
      if(l == 0)
            return 0;
      while((*l != c)&&(*l != '\0'))
            l++;
      if(*l != '\0')
            return l;
      return 0;
}
```

Primeiramente, fornecemos aqui um exemplo de função que retorna como resultado um ponteiro. Note que, antes de acessarmos o valor deste ponteiro, testamos se este é um ponteiro válido para não cometer nenhum erro acessando uma área imprópria da memória.

O programa utilizou a manipulação de ponteiros para localizar a posição de um caractere em uma string e fornecer a sua posição ao usuário. A maioria das funções em C manipulam strings como ponteiros e, por isso, são extremamente rápidas e otimizadas. Em vez de declararmos uma string como uma tabela, podemos faze-lo diretamente como sendo um ponteiro. O tipo (char *) é reconhecido em C como sendo um tipo string, ou seja, um ponteiro para uma seqüência de caracteres. Por isso, as inicializações abaixo são equivalentes:

```
char * salute = "Saudacoes";
char salute[] = "Saudacoes";
```

Entretanto, esta inicialização alocará memória somente para o correspondente número de caracteres da string passada mais um caractere '\0' para delimitar a string. Nenhum espaço a mais de memória será alocado neste caso. As duas formas provocam o mesmo efeito, mas são diferentes: a primeira declara salute como um ponteiro variável e a segunda como um ponteiro constante. O valor de um ponteiro constante não pode ser modificado, já um ponteiro variável pode ter seu valor modificado, por um incremento (++), por exemplo: **Podemos criar em C matrizes de ponteiros**

para strings. Esta é uma das maneiras mais econômicas para alocar memória para uma matriz de strings sem desperdiçar memória. Por exemplo, a inicialização:

```
static char * list[5] =
{          "Katarina",
           "Nigel",
           "Gustavo",
           "Francisco",
           "Airton"        };
```

Vai alocar uma matriz com 5 elementos, onde cada elemento conterá o valor de um ponteiro para uma lista de caracteres (string). Desta forma, o espaço de memória necessário para alocar será somente o tamanho estritamente necessário. Como a string é de ponteiros, basta alocar espaço para cada elemento de 2 bytes, enquanto que a seqüência de caracteres apontado por este ponteiro pode estar em qualquer posição de memória.

Se quiséssemos fazer a mesma inicialização utilizando uma matriz de vetores de caracteres, isto implicaria que teríamos que alocar uma tabela MxN para conter todos os caracteres e isto nos levaria a desperdiçar algumas posições da memória que não chegaram nunca a conter algum caractere, já que as strings se caracterizam por conter um comprimento diferentes de caracteres.

Por isto, uma das razões para se inicializar "strings" com ponteiros é a alocação mais eficiente de memória. Uma outra razão é a de obter maior flexibilidade para manipular matrizes de strings. Por exemplo, suponha que desejássemos reordenar as strings da matriz acima. Para fazer isto, não precisamos remover as strings de sua posição e escrevê-las em outra matriz na ordem correta, basta trocar de posição os ponteiros que apontam para as strings. Reordenando os ponteiros, obteremos a ordem desejada das strings sem ter que se preocupar em reescrever as strings em outra posição de memória.

11.6 Ponteiros para Ponteiros

A habilidade da linguagem C de tratar partes de matrizes como matrizes cria um novo tópico de C, ponteiros que apontam para ponteiros. Esta habilidade dá a C uma grande flexibilidade na

criação e ordenação de dados complexos. Vamos analisar um exemplo de acesso duplamente indireto de dados derivados de uma matriz de duas dimensões:

```
#include <stdio.h>
#define LIN 4
#define COL 5
void main()
{
      static int tabela[LIN][COL] =
      {          {13, 15, 17, 19, 21},
                 {20, 22, 24, 26, 28},
                 {31, 33, 35, 37, 39},
                 {40, 42, 44, 46, 48}       };
      int c = 10;
      int j, k;
      int * ptr = ( int *) tabela;
      for(j=0; j<LIN; j++)
            for(k=0; k<COL; k++)
                  *(ptr + j*COL + k) += c;
      for(j=0; j<LIN; j++)
      {
            for(k=0; k<COL; k++)
                  printf("%d ", *(*(tabela+j)+k));
            printf("\n");
      }
}
```

Neste exemplo, estamos utilizando o ponteiro tabela para acessar os termos da matriz bidimensional. Mas como fazer para acessar um termo posicionado em tabela[i][j]? Como tabela é uma matriz para inteiros, cada elemento ocupará dois bytes e cada coluna com 5 elementos ocupará 10 bytes. Abaixo, mostramos o armazenamento desta tabela na memória (os endereços estão em números decimais para facilitar o entendimento).

Tabela:	i\j	0	1	2	3	4
Mem\offset	i\offset	0	2	4	6	8
1000	0	13	15	17	19	21
1010	1	20	22	24	26	28
1020	2	31	33	35	37	39
1030	3	40	42	44	46	48

Vamos tentar agora acessar o elemento tabela[2][4], onde a tabela tem dimensão 4x5.

a) *(tabela + 2*5 + 4)

Note que, podemos tratar uma matriz mxn como sendo um vetor simples de inteiros, pois as linhas são posicionadas na memória uma após a outra. Então, se calcularmos a posição de memória ocupada por tabela[2][4], poderemos acessar o seu valor através de um ponteiro:

• int * ptr = tabela : contém o endereço inicial da matriz, no caso 1000.

• (ptr + 2*5) == (ptr + 10) : contém a posição incial da terceira linha da tabela (tabela[2]), no caso 1020 pois cada elemento ocupa 2 bytes (int). Isto fica claro na matriz acima. Sabendo-se que a terceira linha começa após 2 linhas com 5 elementos, basta adicionar o número de termos contidos nestas linhas ao ponteiro da matriz e obtemos um ponteiro para o início da linha 2.

• ((ptr + 2*5) + 4) == (ptr + 14) : contém a posição de memória onde está o termo tabela[2][4], no caso 1028. Com o ponteiro da linha, queremos agora acessar o termo 4 desta linha. Então, basta adicionar 4 ao ponteiro da linha, que obtemos um ponteiro para o referido termo.

• *(ptr + 2*5 + 4) == *(ptr + 14) : calculada a posição de memória onde está o referido termo, basta utilizar o operador (*) para acessá-lo.

b) *(*(tabela + 2) + 4)

Considere a declaração da seguinte tabela 4x5:

```
int tabela[4][5];
```

Queremos agora criar um vetor que contenha a terceira linha desta tabela. Podemos fazer isto, usando uma das duas declarações equivalentes:

```
int linha[5] = tabela[2];
int linha[5] = *(tabela + 2); /* eq. 1 */
```

Note que, uma matriz é na verdade um vetor contendo outras matrizes de ordem menor. Na declaração acima, fazemos a atribuição de um elemento da matriz tabela, isto é, um vetor com 5 inteiros, para uma matriz que contém 5 inteiros. A diferença das notações é que a segunda utiliza a manipulação de ponteiros na sua declaração.

Agora, vamos acessar o quinto termo de linha, isto é, linha[4]. Podemos fazer isto pelo método convencional de matrizes ou por ponteiros. Os dois métodos abaixo são equivalentes:

```
int num = linha[4];
int num = *(linha + 4);
```

Desta forma, conseguimos acessar indiretamente o termo tabela[2][4]. Primeiro atribuímos a terceira linha da tabela a um vetor simples e depois acessamos o quinto elemento deste vetor. Entretanto, C nos permite realizar o mesmo método de acesso, sem criar este passo intermediário, basta fazer:

```
int tabela[4][5];
int num = *(*(tabela + 2) + 4); /* tabela[2][4]; */
```

Assim, *(tabela+2) retorna o endereço da terceira linha da matriz tabela, equivalente a escrevermos &tabela[2][0]. Ao adicionarmos 4 a este endereço, calculamos o endereço do quinto elemento nesta linha. Portanto, o endereço do quinto elemento é (*(tabela+2)+4) e o conteúdo deste endereço é *(*(tabela+2)+4), que é 39. Esta expressão é m ponteiro para ponteiro. Este princípio é aplicado para matrizes de qualquer dimensão, em outras palavras:

```
tabela[j][k] = *(*(tabela + j) + k);
cubo[i][j][k] = *(*(*(cubo + i) + j) + k);
```

Desta maneira, C nos permite criar ponteiros para ponteiros e fazer estruturas tão complexas e flexíveis quanto desejarmos. O operador *(ptr) pode então ser aninhado para obter o valor final apontado pela seqüência de ponteiros. Esta técnica pode fornecer grande velocidade de execução e economia de memória, mas tenha cuidado para construir um código claro de forma a evitar problemas devido a um gerenciamento ruim destes ponteiros.

11.7 Argumentos da Linha de Comando

C permite que um programa receba uma lista de parâmetros através da função main(). A forma geral da função main é dada por:

```
int main(int argc, char* argv[]);
```

onde argc é o número de argumentos passados para a função, e argv é uma matriz de string que contém todos os argumentos passados. A função main() nesta forma geral necessita retornar um valor indicando o resultado do programa ao sistema operacional. No caso, o retorno do valor 0 indicará que o programa foi executado adequadamente.

Suponha que você tenha criado um programa chamado jogo e tenha executado ele no sistema operacional com a linha de comando:

```
C:> jogo xadrex.txt 2 3.14159
```

Ao executar o programa passo-a-passo, verificaremos que argc será igual a 4, indicando que o usuário forneceu 4 parâmetros ao programa. Sendo que os parâmetros estão armazenados em argv na forma de string e conterão os seguintes valores:

```
argv[0] == "jogo";
argv[1] == "xadrex.txt";
argv[2] == "2";
argv[3] == "3.14159";
```

Conhecendo como funciona os parâmetros da função main(), você já pode utilizar a atribuição abaixo para obter os valores fornecidos, bastando converter o parâmetro do formato ASCII para o tipo de dado requerido :

```
char * param = argv[3];
```

11.8 Ponteiros para Estruturas

Como já mostramos ponteiros são mais fáceis de manipular que matrizes em diversas situações, assim ponteiros para estruturas são mais fáceis de manipular que matrizes de estruturas. Várias representações de dados que parecem fantásticas são constituídas de estruturas contendo ponteiros para outras estruturas. O nosso próximo exemplo mostra como definir um ponteiro para estrutura e usá-lo para acessar os membros da estrutura.

```
#include <stdio.h>

struct lista /* declara estrutura */
{
      char titulo[30];
      char autor[30];
      int regnum;
      double preco;
};

int main(int argc, char * argv[])
{
      static struct lista livro[2] =
      {          { "Helena", "Machado de Assis", 102, 70.50 },
                 { "Iracema", "Jose de Alencar", 321, 63.25 }     };
      struct lista *ptrl = 0; /* ponteiro para estrutura */
      printf("Endereco #1: %p #2: %p\n", &livro[0], &livro[1]);
      ptrl = &livro[0];
      printf("Ponteiro #1: %p #2: %p\n", ptrl, ptrl + 1);
```

```
        printf("ptrl->preco: R$%.2f \t (*ptrl).preco:
                              R$%.2f\n", ptrl->preco, (*ptrl).preco);
        ptrl++; /* Aponta para a proxima estrutura */
        printf("ptrl->titulo: %s \tptrl->autor: %s\n",
                                        ptrl->titulo, ptrl->autor);
        return 0;
}
```

A declaração é feita como se estivéssemos declarando uma variável de qualquer tipo, adicionando-se o operador (*) na frente do nome da variável. Por isso, a declaração de um ponteiro para uma estrutura é feito na forma:

```
struct lista *ptrl;
```

O ponteiro ptrl pode então apontar para qualquer estrutura do tipo lista. A atribuição de um endereço a um ponteiro de estrutura funciona da mesma forma como uma variável qualquer, empregando-se o operador (&):

```
ptrl = &(livro[0]);
```

Vimos no capítulo sobre estruturas como fazer o acesso a um elemento de uma estrutura através do operador (.). Por exemplo, se quisermos ler o valor do preço da primeira estrutura da matriz livro, procederíamos da forma:

```
livro[0].preco = 89.95;
```

Mas como proceder com ponteiros? Podemos fazê-lo de duas formas. Primeiro, podemos utilizar o operador (*) para obter a estrutura apontada por um ponteiro e depois empregar o operador normal (.) para acessar um elemento desta estrutura. Aplicando no exemplo acima, teremos:

```
(*ptrl).preco = 89.95;
```

O segundo método utiliza o operador (->) que nos permite acessar um elemento de uma estrutura apontada por um dado ponteiro. Aplicando-se este operador no problema acima, temos:

```
ptrl->preco = 89.95;
```

Em outras palavras, um ponteiro para estrutura seguido pelo operador (->) trabalha da mesma maneira que o nome de uma estrutura seguido pelo operador (.). É importante notar que ptrl é um ponteiro, mas `ptrl->preco` é um membro da estrutura apontada. Neste caso, ptrl->preco é uma variável double. O operador (.) conecta a estrutura a um membro dela; o operador (->) conecta um ponteiro a um membro da estrutura.

11.9 Alocação Dinâmica de Memória

A linguagem C oferece um conjunto de funções que permitem a alocação ou liberação dinâmica de memória. Desta forma, podemos alocar memória para um programa de acordo com a sua necessidade instantânea de memória. A memória de trabalho do computador (RAM) usualmente é subdividida em vários segmentos lógicos dentro de um programa. Estes segmentos são:

• segmento de ***dados***, onde são alocadas as variáveis globais (extern), definidas em prerun-time;

• o segmento de ***código***, onde estão as instruções de máquina do programa em si;

• o segmento de ***pilha*** (“stack”), onde as funções alocam provisoriamente suas variáveis locais (auto). Este segmento também é usado para passagem de parâmetros;

• o segmento ***extra***, que pode conter mais variáveis globais;

Toda a área de memória restante entre o fim do programa e o fim da RAM livre é chamada de ***heap***. O ***heap*** é usado para a criação de variáveis dinâmicas, que são criadas em ***run-time*** (isto é, durante a execução do programa). Este tipo de variável é útil quando não se sabe de antemão quantas variáveis de um determinado tipo serão necessárias para a aplicação em questão.

Quando escrevemos um programa utilizando o método de declaração de variáveis visto anteriormente (alocação estática de memória), o programa ao ser executado alocará somente um

bloco fixo de memória para armazenar todos os seus dados. Isto resolve o problema de termos um espaço de memória alocado para podermos armazenar os dados do programa. Entretanto, como visto no capítulo de matrizes, este método não otimiza a utilização do espaço de memória alocado.

Por exemplo, imagine que você precise de uma matriz temporária para armazenar alguns dados temporários durante a execução de uma dada função de manipulação de matrizes. Para tal, você deverá declarar esta matriz na função, o que implicará que o computador irá alocar um bloco de memória para esta matriz. Este espaço de memória ficará alocado ao seu programa durante toda a execução deste, apesar do programa só utilizar uma vez esta matriz e, posteriormente, não precisar mais desta matriz e nem do espaço de memória alocado a esta.

Com a alocação dinâmica de memória, podemos, em tempo de execução, fazer com que um programa aloque um determinado espaço de memória, utilize este espaço por um determinado tempo e depois o libere, para que outros programas possam vir a utilizá-lo.

No caso do nosso exemplo, podemos fazer com que sempre que a função for chamada, ela alocará um espaço de memória para armazenar a referida matriz e após o seu uso, o programa liberará este bloco de memória para que outro programa o utilize. Desta forma, se executarmos esta função apenas uma vez, o programa irá liberar esta memória posteriormente, permitindo assim que outros programas façam um uso mais adequado desta. **Desta forma, a alocação dinâmica de memória é utilizada em programas para alocar e liberar blocos temporários de memórias durante a execução de um programa (por isso é chamado alocação dinâmica)**.

Este bloco de memória é solicitado ao sistema operacional que procura um espaço livre de memória para o programa. Se o sistema operacional achar um bloco de memória livre do tamanho do bloco solicitado, este passa o bloco de memória para o controle do programa e não irá permitir que nenhum outro programa utilize esta memória enquanto ela estiver alocada. No final do seu uso, o programa libera novamente esta memória ao sistema operacional.

Outro exemplo de aplicação da alocação dinâmica de memória é na utilização de matrizes quando não sabemos de antemão quantos elementos serão necessários. Desta forma, podemos utilizar a alocação dinâmica de memória para somente alocar a quantidade necessária de memória e no momento em que esta memória for requerida.

11.9.1 Malloc()

A função malloc() é utilizada para fazer a alocação dinâmica de um bloco de memória a um dado programa. A função malloc() toma um inteiro sem sinal como argumento. Este número representa a quantidade em bytes de memória requerida. **A função retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado. É importante verificar que o ponteiro retornado por malloc() é para um tipo void**. O conceito de ponteiro para void deve ser introduzido para tratar com situações em que seja necessário que uma função retorne um ponteiro genérico, i.e., que possa ser convertido em um ponteiro para qualquer outro tipo de dado. Este ponteiro void pode ser convertido para um ponteiro do tipo de dado desejado (int, float, struct, ...) empregando-se o método de conversão de tipos apresentado na secção sobre tipos de dados (ver 2.5). No próximo exemplo, mostraremos seu emprego novamente.

Quando a função malloc() não encontrar espaço suficiente de memória para ser alocado, esta retornará um ponteiro NULL, i.e., um ponteiro inválido. O exemplo abaixo mostra como a função malloc() opera. Este programa declara uma estrutura chamada xx e chama malloc() 4 vezes. A cada chamada, malloc() retorna um ponteiro para uma área de memória suficiente para guardar um nova estrutura.

```
#include <stdio.h>
struct xx
{
      int numl;
      char chl;
};
void main() {
      struct xx *ptr = 0;
      int j;
      printf("sizeof(struct xx) = %d\n", sizeof(struct xx));
      for(j=0; j<4; j++){
            ptr = (struct xx *) malloc(sizeof(struct xx));
            printf("ptr = %x\n", ptr);
      }
}
```

Note que em nenhuma parte do programa declaramos qualquer variável estrutura. De fato, a variável estrutura é criada pela função malloc(); o programa não conhece o nome desta variável; mas sabe onde ela está na memória, pois malloc() retorna um ponteiro para ela. As variáveis criadas podem ser acessadas usando ponteiros, exatamente como se tivessem sido declaradas no início do programa.

A cada chamada de malloc() devemos informá-la do tamanho da estrutura que queremos guardar. Nós podemos conhecer este tamanho adicionando os bytes usados por cada membro da estrutura, ou através do uso de um novo operador em C unário chamado sizeof(). Este operador produz um inteiro igual ao tamanho, em bytes, da variável ou do tipo de dado que está em seu operando. Por exemplo, a expressão:

```
sizeof(float)
```

retornará o valor 4, haja vista que um float ocupa 4 bytes. No programa exemplo, usamos sizeof() em printf() e ele retorna 3, pois a estrutura xx consiste em um caractere e um inteiro. Então, sizeof() forneceu o tamanho em bytes da estrutura para que malloc() pudesse alocar o espaço de memória requerido. Feito isto, malloc() retornou um ponteiro do tipo void, que foi convertido para o tipo struct xx através da expressão:

```
(struct xx *)
```

11.9.2 Calloc()

Uma outra opção para a alocação de memória é o uso da função calloc(). Há uma grande semelhança entre calloc() e malloc() que também retorna um ponteiro para void apontando para o primeiro byte do bloco solicitado.

A nova função aceita dois argumentos do tipo unsigned int. Um uso típico é mostrado abaixo:

```
long * memnova;
memnova = (long *) calloc(100, sizeof(long));
```

O primeiro argumento é o número de células de memórias desejadas e o segundo argumento é o tamanho de cada célula em bytes. No exemplo acima, long usa quatro bytes, então esta instrução alocará espaço para 100 unidades de quatro bytes, ou seja, 400 bytes. **A função calloc() tem mais uma característica: ela inicializa todo o conteúdo do bloco com zero**.

11.9.3 Free()

A função free()libera a memória alocada por malloc() e calloc(). Aceita, como argumento, um ponteiro para uma área de memória previamente alocada e então libera esta área para uma possível utilização futura.

Sempre que um espaço de memória for alocado, este deve ser necessariamente liberado após o seu uso. Se não for liberada, esta memória ficará indisponível para o uso pelo sistema operacional para outros aplicativos. A utilização consciente da alocação e liberação dinâmica de memória permite um uso otimizado da memória disponível no computador.

A função free()declara o seu argumento como um ponteiro para void. A vantagem desta declaração é que ela permite que a chamada à função seja feita com um argumento ponteiro para qualquer tipo de dado.

```
long * memnova;
memnova = (long *)calloc(100, sizeof(long));
/* usa memnova */
free(memnova); /* libera a memória alocada */
```

11.10 Exercícios

11.1 Escreva um programa que receba duas strigns como argumentos e troque o conteúdo de string1 como string2.

11.2 Escreva um programa que inverta a ordem dos caracteres de uma string. Por exemplo, se a string recebida é “Saudacoes” deve ser modificada para “seocaduaS”.

11.3 Reescreva o programa do exercício 9.1 utilizando alocação dinâmica de memória.

11.4 Reescreva o programa do exercício 9.3 utilizando alocação dinâmica de memória.

11.5 A lista encadeada se assemelha a uma corrente em que as estruturas estão penduradas seqüencialmente. Isto é, a corrente é acessada através de um ponteiro para a primeira estrutura, chamada cabeça, e cada estrutura contêm um ponteiro para a sua sucessora, e o ponteiro da última estrutura tem valor NULL (0) indicando o fim da lista. Normalmente uma lista encadeada é criada dinamicamente na memória. Crie um programa com uma lista encadeada para armazenar dados de livros em uma biblioteca.

11.6 Crie um programa com lista encadeada para catalogar clientes e fitas em uma vídeo locadora.

11.7 Crie uma estrutura para descrever restaurantes. Os membros devem armazenar o nome, o endereço, o preço médio e o tipo de comida. Crie uma lista ligada que apresente os restaurantes de um certo tipo de comida indexados pelo preço. O menor preço deve ser o primeiro da lista. Escreva um programa que peça o tipo de comida e imprima os restaurantes que oferecem este tipo de comida.

11.8 Escreva um programa para montar uma matriz de estruturas para armazenar as notas de 40 alunos. A primeira coluna da matriz deve conter o nome do aluno, a segunda o telefone, a terceira a data de nascimento, depois seguem as notas em lista e na última coluna deve ser calculada a média até o presente momento. A professora deve ser capaz de inserir e retirar alunos, e poder editar os dados dos alunos. A professora deve poder também listar os dados de todos alunos na forma de uma tabela na tela do computador. A lista de alunos deve ser indexada pelo nome destes. Utilize a idéia da lista ligada e da alocação dinâmica de memória.

12 Manipulação de Arquivos em C

Neste capítulo, veremos brevemente a manipulação de arquivos em C. Em um capítulo posterior, será apresentada novamente a manipulação de arquivos agora utilizando C++.

12.1 Tipos de Arquivos

Uma maneira de classificar operações de acesso a arquivos é conforme a forma como eles são abertos: em modo texto ou em modo binário. Arquivos em modo texto operam em dados armazenados em formato texto, ou seja, os dados são traduzidos para caracteres e estes caracteres são escritos nos arquivos. Por esta razão, fica mais fácil de compreender os seus formatos e localizar possíveis erros.

Arquivos em modo binário operam em dados binários, ou seja, os dados escritos neste formato são escritos na forma binária, não necessitando de nenhuma conversão do tipo do dado utilizado para ASCII e ocupando bem menos memória de disco (arquivos menores).

Uma outra diferença entre o modo texto e o modo binário é a forma usada para guardar números no disco. Na forma de texto, os números são guardados como cadeias de caracteres, enquanto que na forma binária são guardados como estão na memória, dois bytes para um inteiro, quatro bytes para float e assim por diante.

12.2 Declaração, abertura e fechamento

C declara um tipo especial de estrutura, chamada FILE, para operar com arquivos. Este tipo é definido na biblioteca "stdio.h", que deve ser incluída na compilação com a diretiva #include para permitir operações sobre arquivos. Os membros da estrutura FILE contêm informações sobre o arquivo a ser usado, tais como: seu atual tamanho, a localização de seus *buffers* de dados, se o arquivo está sendo lido ou gravado, etc. Toda operação realizada sobre um arquivo (abertura, fechamento, leitura, escrita) requer um apontador para uma estrutura do tipo FILE:

```
FILE *File_ptr;
```

A abertura de um arquivo é feita com a função fopen():

```
File_ptr = fopen("Nome do Arquivo","<I/O mode>);
```

onde as opções para a abertura do arquivo estão listadas abaixo:

I/O mode	Função:
r	Read, abre arquivo para leitura. O arquivo deve existir.
w	Write, abre arquivo para escrita. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
a	Append, abre arquivo para escrita. Os dados serão adicionados ao fim do arquivo se este existir, ou um novo arquivo será criado.
r+	Read, abre um arquivo para leitura e gravação. O arquivo deve existir e pode ser atualizado.
w+	Write, abre um arquivo para leitura e gravação. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
a+	Append, abre um arquivo para atualizações ou para adicionar dados ao seu final.
t	Text, arquivo contém texto dados em ASCII.
b	Binary, arquivo contém dados em binário.

A função fopen() executa duas tarefas. Primeiro, ela preenche a estrutura FILE com as informações necessárias para o programa e para o sistema operacional, assim eles podem se comunicar. Segundo, fopen() retorna um ponteiro do tipo FILE que aponta para a localização na memória da estrutura FILE.

A função fopen() pode não conseguir abrir um arquivo por algum motivo (falta de espaço em disco, arquivo inexistente, etc.) e por isso esta retornará um ponteiro inválido, isto é, contendo o valor NULL (0). Por isso, teste sempre se o ponteiro fornecido por fopen() é válido antes de utilizado, caso contrário o seu programa pode vir a ter uma falha séria.

Quando terminamos a gravação do arquivo, precisamos fechá-lo. O fechamento de um arquivo é feito com a função fclose():

```
fclose(File_ptr);
```

Quando fechamos um arquivo é que o sistema operacional irá salvar as suas modificações ou até mesmo criar o arquivo, no caso de ser um arquivo novo. Até então, o sistema operacional estava

salvando as alterações em um buffer antes de escrever estes dados no arquivo. Este procedimento é executado para otimizar o tempo de acesso ao disco empregado pelo sistema operacional. Por isso, não esqueça de fechar um arquivo, senão os seus dados podem ser perdidos e o arquivo não seja criado adequadamente.

Uma outra razão para fechar o arquivo é a deliberar as áreas de comunicação usadas, para que estejam disponíveis a outros arquivos. Estas áreas incluem a estrutura FILE e o buffer. Uma outra função que fecha arquivos é a função exit(). A função exit() difere da função fclose() em vários pontos. Primeiro, exit() fecha todos os arquivos abertos. Segundo, a função exit() também termina o programa e devolve o controle ao sistema operacional. A função fclose() simplesmente fecha o arquivo associado ao ponteiro FILE usado como argumento.

12.3 Leitura e escrita de caracteres

A função usada para ler um único caractere de um arquivo é getc() enquanto que a função putc() escreve um caractere em um arquivo. Abaixo, apresentamos exemplos destas funções:

Escrita

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
char mychar;
fileptr = fopen(filename,"w");
putchar(mychar,fileptr);
fclose(fileptr);
```

Leitura

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
int mychar;
int i = 0;
fileptr = fopen(filename,"r");
mychar = getchar(fileptr);
while(mychar != EOF)
{
   printf("%c", mychar);
   mychar = getchar(fileptr);
}
fclose(fileptr);
```

12.4 Fim de Arquivo (EOF)

EOF é um sinal enviado pelo sistema operacional para indicar o fim de um arquivo. O sinal EOF (Fim de Arquivo) enviado pelo sistema operacional para o programa C não é um caractere, e sim um inteiro de valor –1 e está definido em *stdio.h*.

Perceba que no exemplo anterior de leitura de caracteres, nós usamos uma variável inteira para guardar os caracteres lidos para que possamos interpretar o sinal de EOF. Se usarmos uma variável do tipo char, o caractere de código ASCII 255 decimal (0xFF em Hexa) será interpretado como EOF. Queremos usar todos os caracteres de 0 a 255 em nosso arquivo e uma variável inteira nos assegura isto. Neste exemplo, EOF é usado para ler todos os caracteres de um dado arquivo, quando não conhecemos de antemão a quantidade de caracteres deste arquivo.

A marca de fim de arquivo pode ser diferente para diferentes sistemas operacionais. Assim, o valor de EOF pode ser qualquer. O seu arquivo stdio.h define EOF com o valor correto para o seu sistema operacional; assim, em seus programas, use EOF para testar fim de arquivo. O fim de um arquivo pode também ser determinado utilizando-se a função feof(), que recebe como parâmetro um ponteiro válido para a estrutura FILE.

12.5 Leitura e escrita de strings

A função fputs() escreve uma string em um arquivo e, por isso, toma dois argumentos, sendo o primeiro a matriz de caracteres que será gravada e o segundo o ponteiro para a estrutura FILE do arquivo a ser gravado. Observe que a função fputs() não coloca automaticamente o caractere de nova-linha no fim de cada linha. No programa exemplo a seguir, fazemos isto explicitamente com o caractere '\n'.

A função gets() lê uma linha por vez de um arquivo texto. A função gets() toma 3 argumentos. O primeiro é um ponteiro para o buffer onde será colocada a linha lida. O segundo é um número inteiro que indica o limite máximo de caracteres a serem lidos. Na verdade, este número deve ser pelo menos um maior que o número de caracteres lidos, pois gets() acrescenta o caractere NULL ('\0') na próxima posição livre. O terceiro argumento é um ponteiro para a estrutura FILE do arquivo a ser lido. A função termina a leitura após ler um caractere de nova linha ('\n') ou um caractere de fim de arquivo (EOF).

Escrita:

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
char line[81];
fileptr = fopen(filename, "w");
fputs(line, fileptr);  /* fprintf(fileptr,"%s\n", line) */
fputs("\n", fileptr); /* pode ser usado aqui no lugar    */
fclose(fileptr);        /* dos dois fputs */
```

Leitura:

```
#include <stdio.h>
FILE *fileptr;
char filename[65];
char line[81];
fileptr = fopen(filename, "r");
fgets(line, 80, fileptr);   /* fscanf(fileptr, "%s", line); */
close(fileptr);       /* pode ser usado no lugar de fgets */
```

12.6 Arquivos Padrões

C define um conjunto de arquivos padrões utilizados para acessar alguns periféricos do computador (como a impressora) ou para ler da entrada padrão (normalmente o teclado) ou escrever para a saída padrão (normalmente a tela). Desta forma, _streams[] foi criada como uma matriz de estruturas FILE. Se você perder um tempo e analisar o seu arquivo stdio.h, encontrará várias constantes simbólicas definidas como:

```
#define stdin (&_streams[0])
#define stdout (&_streams[1])
#define stderr (&_streams[2])
#define stdaux (&_streams[3])
#define stdprn (&_streams[4])
```

Estas constantes podem ser usadas para acessar qualquer um dos 5 arquivos padrão que são predefinidos pelo MS-DOS e abertos automaticamente quando o seu programa inicia a sua execução e fechados ao seu fim.

Nome:	Periférico:
stdin	Standard input device (teclado)
stdout	Standard output device (tela)
stderr	Standard error device (tela)
stdaux	Standard auxiliary device (porta serial)
stdprn	Standard printing device (impressora paralela)

Cada uma destas constantes pode ser tratada como um ponteiro para uma estrutura FILE dos arquivos in, out, err, aux e prn respectivamente. Você pode usar os ponteiros FILE definidos em *stdio.h* para acessar os periféricos predefinidos pelo MS-DOS ou usar seus nomes e definir os ponteiros necessários. Como exemplo, a instrução:

```
fgets(string, 80, stdin);
```

lê uma string do teclado.

A instrução:

```
fputs(string, stdprn);
```

imprimirá uma string na impressora.

12.7 Gravando um Arquivo de Forma Formatada

Nos capítulos iniciais apresentamos a função printf() para imprimir na tela dados de forma formatada. Para realizar a mesma tarefa, entretanto não para escrever na tela, mas sim para um arquivo, foi criada a função fprintf(). Esta função é similar a printf() exceto que o ponteiro para FILE é tomado como primeiro argumento. Como em printf(), podemos formatar os dados de várias maneiras; todas as possibilidades de formato de printf() operam com fprintf().

Da mesma forma foi criada a função fscanf(), que como scanf(), lê um dado formatado. A diferença consiste que fscanf() lê um dado de um arquivo e recebe um ponteiro para FILE como primeiro argumento. Exemplo:

```
#include <stdio.h>
FILE *fptr;
int size = 0;
fptr = fopen("dados.txt", "rw");
fscanf(fptr, "%d", &size);
fprintf(fptr, "%s %d %f", "Casa Nova", 12, 13.45);
fclose(fptr);
```

12.8 Leitura e escrita de valores binários

Quando desejamos operar com arquivos no modo binário, basta adicionar o caractere 'b' no I/O Mode da função open(), como apresentado anteriormente. As funções apresentadas anteriormente podem ser usadas para ler e escrever no modo binário, entretanto apresentaremos aqui duas novas funções que facilitam este processo: fwrite() e fread(). Estas funções são empregadas para escrever/ler os dados armazenados em um bloco de memória (um *buffer* de memória) em um arquivo. Aplicações típicas e na escrita/leitura de dados complexos como matrizes e estruturas.

A função fwrite() toma 4 argumentos. O primeiro é um ponteiro do tipo void que aponta para a localização na memória do dado a ser gravado. O segundo argumento é um número inteiro que indica o tamanho do tipo de dado a ser gravado. Normalmente, pode-se utilizar o operador sizeof() para se obter este valor. O terceiro argumento é um número inteiro que informa a fwrite() quantos itens do mesmo tipo serão gravados. O quarto argumento é um ponteiro para a estrutura FILE do arquivo onde queremos gravar.

A função fread() toma também 4 argumentos. O primeiro é um ponteiro void para a localização da memória onde serão armazenados os dados lidos. O segundo indica também a quantidade de bytes do tipo de dado a ser lido. O terceiro argumento informa a quantidade de itens a serem lidos a cada chamada, e o quarto argumento é um ponteiro para a estrutura FILE do arquivo a ser lido.

A função fread() retorna o número de itens lidos. Normalmente este número deve ser igual ao terceiro argumento. Se for encontrado o fim do arquivo, o número será menor que o valor do terceiro argumento, podendo ser zero caso nenhum dado tenha sido lido. As funções fread()e

fwrite() trabalham com qualquer tipo de dado, incluindo matrizes e estruturas, e armazenam números em formato binário.

Escrita:
`fileptr = fopen(filename, "wb");` `fwrite(&dados, sizeof(dados),1 ,fileptr);`
Leitura:
`fileptr = fopen(filename, "rb");` `fread(&dados,sizeof(dados),1 ,fileptr);`

Se "filename" for inicializado com "prn", os dados são enviados a impressora. Exemplo:

```
fileptr=fopen(filename,"rb");
while(!feof(fileptr))
{
      fread(&dados, sizeof(dados),1,fileptr);
}
fclose(fileptr);
```

12.9 Exercícios

12.1 Escreva um programa que imprima um arquivo na tela de 20 em 20 linhas. O arquivo de entrada deve ser fornecido na linha de comando. A cada impressão de 20 linhas, o programa aguarda o pressionamento de uma tecla.

12.2 Escreva um programa que imprima o tamanho de um arquivo em bytes. O nome do arquivo deve ser fornecido na linha de comando.

12.3 Escreva um programa que criptografa um arquivo usando o operador de complemento de bit-a-bit (~). Quando o programa é executado para um arquivo já criptografado, o arquivo é recomposto e volta ao original.

12.4 Refaça o problema 11.5, agora salvando a lista de livros em um arquivo. Permita que o usuário possa retirar um livro desta lista, apagando-o do arquivo, ou adicionar um livro em uma posição determinada na lista, reescrevendo a lista no arquivo. Utilize o modo texto para a manipulação de arquivos.

12.5 Como o exemplo anterior, refaça o problema 11.6 mas agora utilizando o modo binário.