

INSTITUTO FEDERAL DO ESPÍRITO SANTO

Programação em C

```
printf("I: Algoritmos \n  
      II: Ferramentas da Linguagem C \n  
      III: Introdução às Estruturas de Dados");
```

CAMPUS SERRA

V2.0.0

APRESENTAÇÃO

Fico feliz em dizer que esta apostila, que originalmente foi construída pensando no curso de Engenharia de Controle e Automação do IFES/Serra, está, nesta nova versão, dando um passo a frente. Ela evoluiu. Pelo fato de ter sido montada de maneira razoavelmente genérica, mostrou-se, na prática, aplicável a outros cursos de computação, tanto como material base, como de apoio.

Originalmente, conforme Apresentação da versão 1.x.x, esta visava “reunir material didático dirigido a este curso [Eng. de Controle e Automação]. Sem qualquer intuito de ser vista como uma obra original, visa reunir, em um único meio, tópicos encontrados em diversos livros, apostilas e sites, cujas referências bibliográficas encontram-se presentes, àqueles que desejarem consultar.”

Hoje, o passo a frente que este material dá é a sua expansão, incluindo uma Introdução às Estruturas de Dados e um refinamento geral de todos os capítulos (revisão/adição de exemplos, explicações e detalhes adicionais, etc.). Por isso, houve também uma mudança no nome da apostila, que passando a ter uma nomenclatura mais genérica. No entanto, ela continua sendo aplicável ao curso de Engenharia, o qual, desde já, deixo autorizado o uso pelo professor responsável, se assim desejar.

Mas, como produzir um único material capaz de atender a cursos com focos distintos? De um lado, a Engenharia precisa do conhecimento de linguagem de programação como ferramenta. Por outro lado, os alunos de cursos de Computação tem, por obrigação, que aprofundar um pouco mais seu conhecimento, enxergar este campo como ciência, não apenas como ferramenta a ser usada.

Pois bem. Este era o meu novo desafio. Trabalhar em camadas, manter o mais separado possível assuntos básicos de detalhes, aspectos considerados avançados. Manter o conteúdo e o grau de aprofundamento separados, de tal forma que seja adequado aos diferentes cursos, ficando a critério do professor abordar ou não determinado tópico, sempre respeitando, é claro, a ementa do curso.

Por fim, esta apostila tem deixado de ser apenas uma coletânea de materiais presentes em outras obras (artigos, apostilas e livros), aproximando-se cada vez mais de uma obra “original”, visto que as adições/revisões/adaptações têm sido cada vez mais profundas e pouco a pouco trechos diretamente extraídos de livros/apostilas vêm sendo reescritos. Isso tem sido possível depois de todos os semestres em que ela vem sendo aplicada por mim (bem como por outros professores), em suas disciplinas. O aumento na qualidade vem diretamente desse aprimoramento progressivo ao longo do tempo e da experiência em sala de aula. Apesar de desejar manter exclusivo o direito de edição dessa apostila, convido a todos que estejam fazendo uso desse material (sejam alunos ou professores) que contribuam com seu desenvolvimento, com sugestões de aprimoramento, críticas ou novas ideias. Aproveito, neste momento, para agradecer a todos os professores da Coordenadoria de Informática que contribuíram com seus materiais, especialmente os usados na Parte III.

Pois bem, vamos aos trabalhos? ☺

Prof. Flávio Giralde

Engenheiro de Computação

M.Sc. Engenharia Elétrica (Proc. Digital de Sinais)

Coordenadoria de Informática - IFES/Serra

Sumário

Parte I.....	8
Cap. 1: Introdução.....	9
1.1 Algoritmos.....	9
1.1.1 Algoritmos vs. Programas de Computador	9
1.1.2 Aprendendo a Criar Algoritmos	9
1.1.3 Formas de Representação de Algoritmos	10
1.2 Linguagens de Programação	11
1.2.1 Interpretação e Compilação	11
1.2.2 Paradigma de Programação	12
1.3 A Escolha da Linguagem C para este Curso	12
Cap. 2: A Linguagem C.....	13
2.1 Histórico	13
2.2 Algumas Características do C.....	14
2.3 C vs. C++	15
2.4 A Forma de um Programa em C.....	16
2.4.1 Dois Primeiros Programas.....	17
Cap. 3: Expressões e Comandos Básicos em C.....	19
3.1 Os Cinco Tipos Básicos de Dados	19
3.1.1 Um Pouco Mais a Respeito do Tipo char	21
3.2 Variáveis.....	22
3.3 Constantes	23
3.3.1 Constantes String	24
3.3.2 Constantes Caractere de Barra Invertida	24
3.4 Operadores	25
3.4.1 Operador de Atribuição.....	25
3.4.1.1 Atribuições Múltiplas.....	25
3.4.2 Operadores Aritméticos.....	25
3.4.3 Operadores Relacionais e Lógicos.....	27
3.4.4 Outros Operadores.....	28
3.4.4.1 Operadores Bit a Bit.....	28
3.4.4.2 O Operador ?	28
3.5 Expressões.....	29
3.5.1 Ordem de Avaliação	30
3.5.2 Conversão de Tipos em Expressões	30
3.5.3 Expressões que Podem ser Abreviadas.....	30
3.5.4 Encadeando Expressões: o Operador ,.....	30
3.5.5 Modeladores (Casts)	31
3.6 Funções de Entrada e Saída	31
3.6.1 Saída de Texto	32
3.6.2 Entrada de Texto	32
Cap. 4: Comandos de Controle do Programa	34

4.1 Verdadeiro e Falso em C	34
4.2 Comandos de Seleção	34
4.2.1 if.....	34
4.2.1.1 A Expressão Condicional.....	36
4.2.1.2 if's Aninhados	36
4.2.1.3 O Operador ?	37
4.2.2 switch	37
4.3 Comandos de Iteração	38
4.3.1 for	38
4.3.2 while	40
4.3.3 do-while.....	41
4.3.4 Uma Discussão Sobre os Três Comandos de Iteração	42
4.4 Comandos de Desvio.....	43
4.4.1 break.....	43
4.4.2 continue	44
Cap. 5: Vetores, Matrizes e Strings.....	46
5.1 Vetores.....	46
5.2 Strings	47
5.2.1 gets().....	48
5.2.2 strcpy()	48
5.2.3 strcat()	49
5.2.4 strlen()	49
5.2.5 strcmp()	49
5.3 Matrizes	50
5.3.1 Matrizes Bidimensionais	50
5.3.2 Matrizes de Strings.....	51
5.3.3 Matrizes Multidimensionais.....	52
5.3.4 Inicialização	52
5.3.4.1 Inicialização Sem Especificação de Tamanho	52
5.4 Introdução à Pesquisa e Ordenação	53
5.4.1 Algoritmos Simples de Ordenação	53
5.4.1.1 Insertion Sort	53
5.4.1.2 Selection Sort.....	54
5.4.1.3 Bubble Sort	54
5.4.2 Pesquisa Binária	55
Cap. 6: Introdução à Modularização: Subprogramas.....	56
6.1 Planejamento	56
6.1.1 Saiba Onde Quer Chegar	56
6.1.2 Faça uma Especificação do Seu Programa	56
6.1.3 Planeje Seu Programa Pensando em Várias Coisas.....	57
6.1.4 Pesquise Sempre a Melhor Forma de Fazer.....	57
6.2 Modularização	59
6.2.1 Benefícios da Modularização	59
6.2.2 Unidades Básicas	60
6.2.3 O Princípio da Caixa Preta	60
6.2.4 Critérios Úteis na Modularização	61

6.2.4.1 Reusabilidade.....	61
6.2.4.2 Baixa Complexidade	61
6.2.4.3 Acoplamento	61
6.2.4.4 Coesão	62
6.2.4.5 Resumindo os Critérios.....	62
6.3 Funções em C: Aspectos Básicos.....	62
6.3.1 Argumentos.....	64
6.3.2 Retornando valores.....	64
6.4 Escopo de Variáveis	65
6.4.1 Variáveis Locais	65
6.4.2 Parâmetros Formais	67
6.4.3 Variáveis Globais	68
Parte II.....	70
Cap. 7: Criação, Compilação e Execução de Programas em C.....	71
7.1 A Biblioteca e a Linkedição	71
7.2 Compilação Separada.....	72
7.3 Compilando um Programa em C.....	72
7.3.1 O Modelo de Compilação da Linguagem C	72
7.3.1.1 O Pré-Processador	73
7.3.1.2 O Compilador.....	73
7.3.1.3 O Assembler.....	73
7.3.1.4 O Linker.....	73
7.3.1.5 A Utilização de Bibliotecas.....	73
7.4 O Mapa de Memória de C.....	73
7.5 O Pré-Processador do C	74
7.5.1 #define	75
7.5.1.1 Definindo Macros Semelhantes a Funções	76
7.5.2 #include.....	76
7.6 Uma Revisão de Termos	77
Cap. 8: Ponteiros.....	78
8.1 O Que São Ponteiros?	78
8.2 Declarando e Utilizando Ponteiros	79
8.3 Ponteiros e Vetores.....	81
8.3.1 Vetores como Ponteiros.....	81
8.3.2 Ponteiros como Vetores.....	83
8.3.3 Strings.....	83
8.3.4 Endereços de Elementos de Vetores	84
8.3.5 Vetores de Ponteiros.....	84
8.4 Inicializando Ponteiros.....	85
8.5 Indireção Múltipla (Ponteiros para Ponteiros)	85
8.6 Cuidados a Serem Tomados ao Se Usar Ponteiros	88
Cap. 9: Funções: Aspectos Avançados	89
9.1 A Função.....	89
9.2 O Comando return	89
9.3 Três Tipos de Função.....	90
9.4 Protótipos de Funções	91

9.5 O Tipo void	92
9.6 Arquivos-Cabeçalhos.....	93
9.7 Passagem de Parâmetros por Valor e Passagem por Referência	94
9.8 Matrizes como Argumentos de Funções	95
9.9 Os Argumentos argc e argv	97
9.10 Recursão.....	98
9.11 Ponteiros para Funções	99
9.12 Algumas Funções de Entrada/Saída Padronizadas	101
9.12.1 Lendo e Escrevendo Caracteres	101
9.12.1.1 getche() e getch().....	101
9.12.1.2 putchar()	102
9.12.2 Lendo e Escrevendo Strings	102
9.12.2.1 gets().....	102
9.12.2.2 puts().....	102
9.12.3 Entrada e Saída Formatada	102
9.12.3.1 printf().....	103
9.12.3.2 scanf()	105
9.12.3.3 sprintf() e sscanf().....	107
Cap. 10: Tipos de Dados Avançados e Definidos Pelo Usuário	108
10.1 Modificadores de Acesso	108
10.1.1 const.....	108
10.1.2 volatile.....	109
10.2 Conversão de Tipos.....	109
10.3 Alocação Dinâmica de Memória	109
10.3.1 Funções de Alocação Dinâmica do C.....	110
10.3.1.1 malloc()	110
10.3.1.2 calloc()	111
10.3.1.3 realloc().....	111
10.3.1.4 free()	112
10.3.2 Alocação Dinâmica de Vetores e Matrizes.....	113
10.3.2.1 Alocação Dinâmica de Vetores	113
10.3.2.2 Alocação Dinâmica de Matrizes.....	113
10.4 Estruturas.....	115
10.4.1 Referenciando Elementos de Estruturas.....	116
10.4.2 Estruturas Contendo Estruturas.....	116
10.4.3 Atribuição de Estruturas	117
10.4.4 Matrizes de Estruturas	117
10.4.5 Passando Estruturas para Funções	118
10.4.5.1 Passando Elementos de Estrutura para Funções	118
10.4.5.2 Passando Estruturas Inteiras para Funções.....	118
10.4.6 Ponteiros para Estruturas	119
10.4.6.1 Declarando um Ponteiro para Estrutura	119
10.4.6.2 Usando Ponteiros para Estruturas.....	119
10.4.6.3 Alocação Dinâmica de Estruturas	120
10.5 Enumerações.....	121
10.6 O Comando sizeof	121
10.7 O Comando typedef	122

Cap. 11: Entrada/Saída com Arquivos	123
11.1 Abrindo e Fechando um Arquivo	124
11.1.1 fopen().....	124
11.1.2 exit()	125
11.1.3 fclose().....	126
11.2 Principais Funções de Leitura/Escrita em Modo Texto.....	126
11.2.1 fprintf()	126
11.2.2 fscanf().....	126
11.3 Lendo e Escrevendo Caracteres em Arquivos.....	127
11.3.1 putc()	127
11.3.2 getc()	128
11.3.3 feof().....	128
11.3.4 Exemplo.....	128
11.4 Arquivos em Modo Binário	129
11.4.1 Funções para Salvar e Recuperar	129
11.5 Outros Comandos de Acesso a Arquivos	130
11.5.1 Arquivos Pré-definidos.....	130
11.5.2 fgets()	130
11.5.3 fputs()	131
11.5.4 ferror() e perror()	131
11.5.5 fseek().....	132
11.5.6 rewind().....	132
11.5.7 remove().....	132
11.5.8 Exemplo.....	133
11.6 Estruturação de Dados em Arquivo Texto	133
11.6.1 Acesso Caractere a Caractere	133
11.6.2 Acesso Linha a Linha.....	135
11.6.3 Acesso via Palavras Chaves	138
Parte III.....	139
Cap. 12: Abstração de Dados	140
12.1 Introdução.....	140
12.2 Abstração em Computação.....	141
12.3 TAD: Tipo Abstrato de Dados.....	142
12.3.1 Estrutura de Dados versus Tipo Abstrato de Dados	142
12.3.2 Exemplos	143
12.3.2.1 TAD Ponto	143
12.3.2.2 TAD Matriz.....	145
Cap. 13: TAD Lista	148
13.1 Listas Sequenciais.....	148
13.2 Listas Simplesmente Encadeadas	151
13.2.1 Função de Inicialização.....	152
13.2.2 Função de Inserção	152
13.2.3 Função Que Percorre Os Elementos Da Lista.....	153
13.2.4 Função Que Verifica Se Lista Está Vazia	153
13.2.5 Função de Busca.....	153
13.2.6 Função que Retira um Elemento da Lista	154

13.2.7 Função para Liberar a Lista	155
13.2.8 Manutenção da Lista Ordenada	155
13.2.9 Resumo das Situações de Inserção e Remoção numa Lista Simplesmente Encadeada	157
13.3 Listas Duplamente Encadeadas	158
13.4 Listas Genéricas.....	161
13.4.1 Manipulação de Listas Heterogêneas	164
Apêndice A: Bases Numéricas, Sinais Digitais e o PC	166
A.1 O Computador e a Informação	166
A.2 Sinais Digitais e Números Binários.....	166
A.2.1 Mais Sobre Números Binários.....	167
A.3 Base Hexadecimal	168
A.4 Conversões Entre Bases Numéricas.....	168
A.4.1 De Decimal para Binário	168
A.4.1.1 Números Inteiros.....	168
A.4.1.2 Parte Fracionária do Número	169
A.4.2 De Binário/Decimal para Hexadecimal	170
A.4.3 De Binário/Hexadecimal para Decimal	170
A.5 Conversão de Sinais: Analógico para Digital.....	170
A.6 Lógica Temporizada	171
A.6.1 Clock.....	171
A.6.2 Ciclo de Operação	171
A.7 Computador PC: Componentes Básicos	172
A.7.1 Processador.....	173
A.7.2 Memória.....	173
A.7.3 HD.....	174
A.7.4 Placa de Vídeo.....	175
A.7.5 Placa-Mãe	176
A.7.6 Hardware X Software	177
Referências Bibliográficas.....	179
Histórico de Versões	180

PARTE I

Nossos esforços nesta primeira parte serão voltados para o desenvolvimento de *Algoritmos*. Detalhes referentes à linguagem tenderão a ser abstraídos neste primeiro momento e expostos de maneira progressiva, na medida em que forem necessários.

O objetivo, ao final da disciplina, é que você tenha adquirido capacidade de transformar qualquer problema em um algoritmo de boa qualidade, ou seja, a intenção é que você aprenda a Lógica de Programação dando uma base teórica e prática suficientemente boa para que você domine os algoritmos e esteja habilitado a estender seus estudos em uma Linguagem de Programação posteriormente.

Cap. 1: INTRODUÇÃO

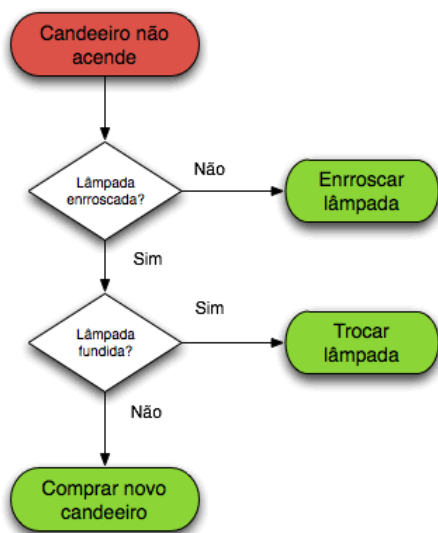
1.1 ALGORITMOS

Um algoritmo é uma sequência finita de instruções bem definidas e não ambíguas. Cada uma das quais pode ser executada mecanicamente num período de tempo finito e com uma quantidade de esforço finita.

O conceito de algoritmo é frequentemente ilustrado pelo exemplo de uma receita, embora muitos algoritmos sejam mais complexos. Eles podem repetir passos (fazer iterações) ou necessitar de decisões (tais como comparações ou lógica) até que a tarefa seja completada. Um algoritmo corretamente executado não irá resolver um problema se estiver implementado incorretamente ou se não for apropriado ao problema.

Um algoritmo não representa, necessariamente, um programa de computador, e sim os passos necessários para realizar uma tarefa. Sua implementação pode ser feita por um computador, por outro tipo de autômato ou mesmo por um ser humano. Diferentes algoritmos podem realizar a mesma tarefa usando um conjunto diferenciado de instruções em mais ou menos tempo, espaço ou esforço do que outros. Tal diferença pode ser reflexo da complexidade computacional aplicada, que depende de estruturas de dados adequadas ao algoritmo. Por exemplo, um algoritmo para se vestir pode especificar que você vista primeiro as meias e os sapatos antes de vestir a calça enquanto outro algoritmo especifica que você deve primeiro vestir a calça e depois as meias e os sapatos. Fica claro que o primeiro algoritmo é mais difícil de executar que o segundo apesar de ambos levarem ao mesmo resultado.

1.1.1 ALGORITMOS VS. PROGRAMAS DE COMPUTADOR



Um programa de computador é essencialmente um algoritmo que diz ao computador os passos específicos e em que ordem eles devem ser executados, como por exemplo, os passos a serem tomados para calcular as notas que serão impressas nos boletins dos alunos de uma escola. Logo, o algoritmo pode ser considerado uma sequência de operações que podem ser simuladas por uma máquina de Turing completa.

A maneira mais simples de se pensar um algoritmo é por uma lista de procedimentos bem definida, na qual as instruções são executadas passo a passo a partir do começo da lista, uma ideia que pode ser facilmente visualizada através de um fluxograma. Tal formalização adota as premissas da programação imperativa, que é uma forma mecânica para visualizar e desenvolver um algoritmo. Concepções alternativas para algoritmos variam em programação funcional e programação lógica.

1.1.2 APRENDENDO A CRIAR ALGORITMOS

A noção de algoritmo é central para toda a computação. A criação de algoritmos para resolver os problemas é uma das maiores dificuldades dos iniciantes em programação em computadores. Isto porque não existe um conjunto de regras, ou seja, um algoritmo, que nos permita criar algoritmos. Caso isto fosse possível, a função de criador de algoritmos desapareceria. Claro que existem linhas mestras e estruturas básicas, a partir das quais podemos criar algoritmos, mas a solução completa depende em grande parte do criador do

algoritmo. Geralmente existem diversos algoritmos para resolver o mesmo problema, cada um segundo o ponto de vista do seu criador.

Um algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria um único algoritmo. Um algoritmo é um “caminho” para a solução de um problema e, em geral, existem muitos caminhos que levam a uma solução satisfatória, ou seja, para resolver o mesmo problema podem-se obter vários algoritmos diferentes.

Para resolver um problema no computador é necessário que seja primeiramente encontrada uma maneira de descrever este problema de uma forma clara e precisa. É preciso que encontremos uma sequência de passos que permitam que o problema possa ser resolvido de maneira automática e repetitiva. Esta sequência de passos é chamada de **algoritmo**.

Sem dúvida alguma, uma das formas mais eficazes de aprender algoritmos é através de muitos exercícios. Poderíamos resumir com as seguintes dicas:

Algoritmos não se aprende...	Algoritmos se aprende...
Copiando algoritmos	Construindo algoritmos
Estudando algoritmos prontos	Testando algoritmos

Muitos alunos acreditam que, quanto mais exemplos resolvidos o professor passar em sala de aula, mais fácil será o seu aprendizado. Mas, qualquer professor de programação irá concordar que esse raciocínio é falso, por mais que o aluno tenha a “sensação de aprendizado”. A razão é simples: O exemplo resolvido reflete exclusivamente o raciocínio previamente desenvolvido pelo professor (ou autor do exemplo) e não por você. Analogamente seria como assistir alguém andar de bicicleta ou dirigir um carro e após isso se achar apto para fazer o mesmo. Sim, exemplos ajudam! Mas, acredite, se você não “bater cabeça” e tentar, começando com pequenas conquistas (sim, escrever uma mensagem “*Hello, World*” na tela é a primeira delas), você jamais será um programador.

O aprendizado da Lógica é essencial para a formação de um bom programador, servindo como base para o aprendizado de todas as Linguagens de Programação, estruturadas ou não. De um modo geral esses conhecimentos serão de suma importância, pois ajudarão no cotidiano, desenvolvendo um raciocínio rápido.

Ao longo de todo este curso, os conceitos básicos de Lógica necessários à programação de computadores estarão diluídos entre os capítulos.

1.1.3 FORMAS DE REPRESENTAÇÃO DE ALGORITMOS

Os algoritmos podem ser representados de várias formas, como por exemplo:

- Através de uma linguagem natural (português, inglês, etc.): forma utilizada nos manuais de instruções, nas receitas culinárias, bulas de medicamentos, etc.
- Através de representações gráficas: podem ser recomendáveis em alguns casos, já que um “desenho” (diagrama, fluxograma, etc.) muitas vezes substitui, com vantagem, várias palavras.
- Através de uma linguagem de programação (Pascal, C, etc.): o passo-a-passo do algoritmo é escrito de maneira formal diretamente na linguagem, que pode ser compilada e executada em um computador.
- Através de uma pseudo-linguagem (Portugol): esta seria uma forma intermediária, entre a linguagem natural e uma linguagem de programação real. As poucas regras existentes, apesar de simples, dão uma formalidade básica ao algoritmo, que pode posteriormente ser “traduzido” para uma linguagem de programação de verdade (como o C).

No ensino de programação, geralmente as duas últimas formas de representação são as mais usadas. Como engenheiro de computação e professor programação por diversas vezes, sou capaz de apontar prós e

contras de cada uma das abordagens. O uso de uma pseudo-linguagem (Portugol) torna a aprendizagem inicial significativamente mais fácil, já que não possui os detalhes (algumas vezes de difícil entendimento para um iniciante) inerentes a uma linguagem de programa real. No entanto, uma vez que é fato que a primeira linguagem aprendida tende a ficar de sobremaneira muito gravada em nosso cérebro, a posterior mudança ou adaptação para uma linguagem de programação, com todos os seus formalismos, pode ser complicada.

Dentre os prós e contras das duas abordagens, do ponto de vista de um profissional em que um sólido conhecimento de programação, frequentemente em mais de uma linguagem, é um requisito básico, iniciar seu aprendizado diretamente em uma linguagem de programação real pode ser o mais recomendado. Uma vez superadas as dificuldades iniciais, sua curva de aprendizagem não sofre interrupções. O aluno não terá assim que, de uma hora pra outra, se ver obrigado a aprender e forçosamente adotar todos os formalismos que uma linguagem de programação, como o C, exige para que os seus programas compilem e executem corretamente.

Com a metodologia adotada neste curso, espera-se minimizar ao máximo as dificuldades inerentes a contato direto com uma linguagem de programação. Apenas os detalhes básicos e essenciais serão inicialmente apresentados. Somente após uma sólida base em lógica e criação de algoritmos, é que os aspectos mais avançados da linguagem serão progressivamente introduzidos.

1.2 LINGUAGENS DE PROGRAMAÇÃO

Uma linguagem de programação é um método padronizado para expressar instruções para um computador. É um conjunto de regras sintáticas e semânticas usadas para definir um programa de computador. Uma linguagem permite que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas sob várias circunstâncias.

O conjunto de palavras (tokens), compostos de acordo com essas regras, constituem o código fonte de um software. Esse código fonte é depois traduzido para código de máquina, que é executado pelo processador.

Uma das principais metas das linguagens de programação é permitir que programadores tenham uma maior produtividade, permitindo expressar suas intenções mais facilmente do que quando comparado com a linguagem que um computador entende nativamente (código de máquina). Assim, linguagens de programação são projetadas para adotar uma sintaxe de nível mais alto, que pode ser mais facilmente entendida por programadores humanos. Linguagens de programação são ferramentas importantes para que programadores e engenheiros de software possam escrever programas mais organizados e com maior rapidez.

Linguagens de programação também tornam os programas menos dependentes de computadores ou ambientes computacionais específicos (propriedade chamada de portabilidade). Isto acontece porque programas escritos em linguagens de programação são traduzidos para o código de máquina do computador no qual será executado em vez de ser diretamente executado.

1.2.1 INTERPRETAÇÃO E COMPILAÇÃO

Uma linguagem de programação pode ser convertida, ou traduzida, em código de máquina por *compilação* ou *interpretação*, que juntas podem ser chamadas de *tradução*.

Se o método utilizado traduz todo o texto do programa (também chamado de código), para só depois executar (ou “rodar”, como se diz no jargão da computação) o programa, então diz-se que o programa foi compilado e que o mecanismo utilizado para a tradução é um compilador (que por sua vez nada mais é do que um programa). A versão compilada do programa tipicamente é armazenada, de forma que o programa pode ser executado um número indefinido de vezes sem que seja necessária nova compilação, o que compensa o tempo gasto na compilação. Isso acontece com linguagens como Pascal e C.

Se o texto do programa é traduzido à medida que vai sendo executado, como em Javascript, Python ou Perl, num processo de tradução de trechos seguidos de sua execução imediata, então diz-se que o programa foi interpretado e que o mecanismo utilizado para a tradução é um interpretador. Programas interpretados são geralmente mais lentos do que os compilados, mas são também geralmente mais flexíveis, já que podem interagir com o ambiente mais facilmente.

Embora haja essa distinção entre linguagens interpretadas e compiladas, as coisas nem sempre são tão simples. Há linguagens compiladas para um código de máquina de uma máquina virtual (sendo esta máquina virtual apenas mais um software, que emula a máquina virtual sendo executado em uma máquina real), como o Java e a plataforma MS.NET.

1.2.2 PARADIGMA DE PROGRAMAÇÃO

Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa. Por exemplo, em programação orientada a objetos, programadores podem abstrair um programa como uma coleção de objetos que interagem entre si, enquanto em programação funcional os programadores abstraem o programa como uma sequência de funções executadas de modo empilhado.

Diferentes linguagens de programação propõem diferentes paradigmas de programação. Algumas linguagens foram desenvolvidas para suportar um paradigma específico (Smalltalk e Java suportam o paradigma de orientação a objetos enquanto Haskell e Scheme suportam o paradigma funcional), enquanto outras linguagens suportam múltiplos paradigmas (como o Python e C++).

Os paradigmas de programação são muitas vezes diferenciados pelas técnicas de programação que proíbem ou permitem. Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais. Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de conceito de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.

1.3 A ESCOLHA DA LINGUAGEM C PARA ESTE CURSO

O capítulo que se segue introduzirá a linguagem C, que será adotada de maneira exclusiva ao longo de todo esse curso. Os motivos que motivam a escolha da linguagem C são vários, muitos dos quais serão vistos no capítulo seguinte. Porém, podemos citar:

- É uma linguagem muito usada no contexto de programação de microprocessadores e microcontroladores, especialmente de sistemas embarcados, comum a várias engenharias.
- É a linguagem que serviu de base para as principais linguagens usadas comercialmente, hoje, no mundo, como C++, Java e C#. Assim sendo, uma vez familiarizado com a linguagem C, o aprendizado das demais torna-se mais fácil.

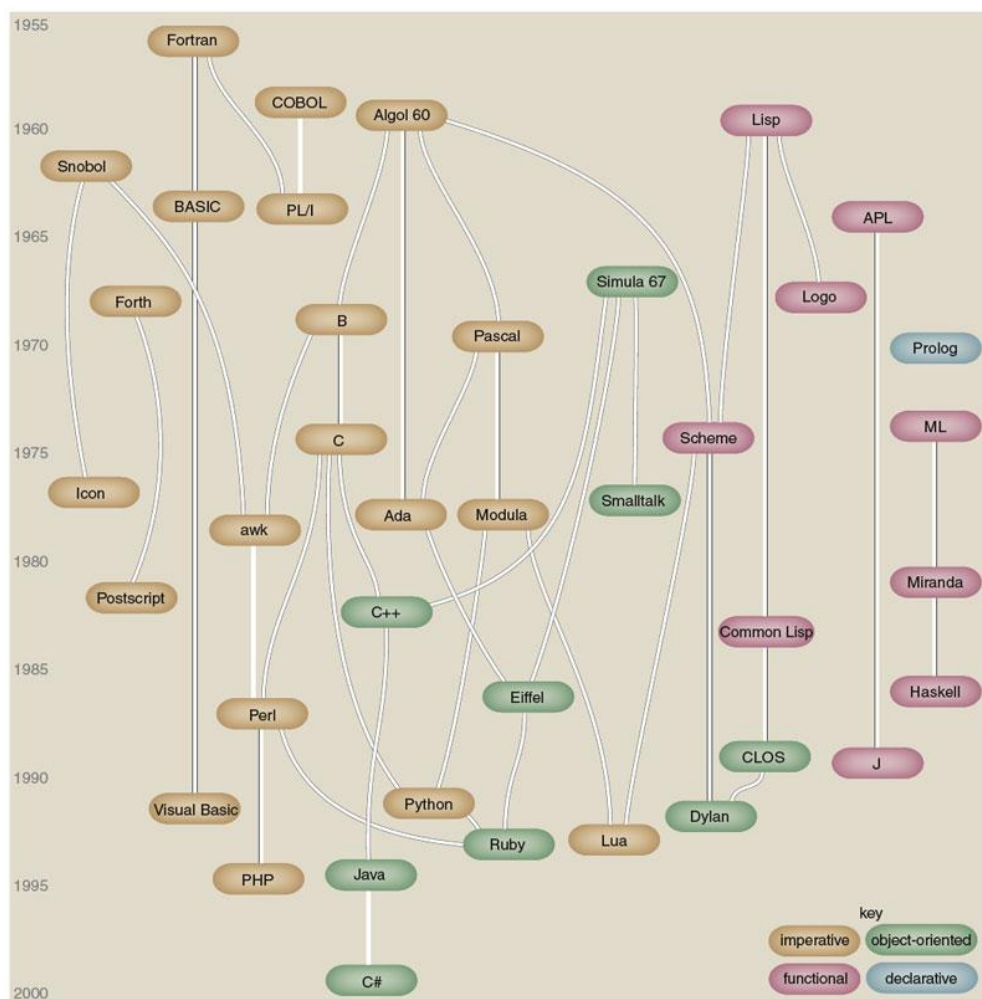
Cap. 2: A LINGUAGEM C

A finalidade deste capítulo é apresentar uma visão geral introdutória da linguagem de programação C, suas origens, seus usos e sua filosofia.

2.1 HISTÓRICO

A linguagem C foi inventada e implementada primeiramente por Dennis Ritchie em um DEC PDP-11 que utilizava o sistema operacional UNIX. C é o resultado de um processo de desenvolvimento que começou com uma linguagem mais antiga, chamada BCPL, que ainda esta em uso, em sua forma original, na Europa. BCPL foi desenvolvida por Martin Richards e influenciou uma linguagem chamada B, inventada por Ken Thompson. Na década de 1970, B levou ao desenvolvimento de C.

Por muitos anos, de fato, o padrão para C foi a versão fornecida com o sistema operacional UNIX versão 5. Ele é descrito em *The C Programming Language*, de Brian Kernighan e Dennis Ritchie. Com a popularidade dos microcomputadores, um grande número de implementações de C foi criado. Quase que por milagre, os códigos-fontes aceitos por essas implementações eram altamente compatíveis. (Isto é, um programa escrito com um deles podia normalmente ser compilado com sucesso usando-se um outro.) Porém, por não existir nenhum padrão, havia discrepâncias. Para remediar essa situação, o ANSI (American National Standards Institute) estabeleceu, no verão de 1983, um comitê para criar um padrão que definiria de uma vez por todas a linguagem C.



2.2 ALGUMAS CARACTERÍSTICAS DO C

Dentre as várias características da linguagem C, poderíamos citar:

C é uma linguagem de médio nível

C é frequentemente chamada de linguagem de *médio nível* para computadores. Isso não significa que C seja menos poderosa, difícil de usar ou menos desenvolvida que uma linguagem de alto nível como BASIC e Pascal, tampouco implica que C seja similar a linguagem Assembly e seus problemas correlatos aos usuários. C é tratada como uma linguagem de médio nível porque combina elementos de linguagens de alto nível com a funcionalidade da linguagem Assembly. Como uma linguagem de médio nível, C permite a manipulação de bits, bytes e endereços – os elementos básicos com os quais o computador funciona.

Um código escrito em C é muito *portável*. Portabilidade significa que é possível adaptar um software escrito para um tipo de computador (e/ou sistema operacional) para outro. Por exemplo, se você pode com relativa facilidade converter um programa escrito para Windows de tal forma a executar sob o Linux, então esse programa é portável.

Outro aspecto importante de C é que ele tem apenas 32 palavras-chaves (27 do padrão de fato estabelecido por Kernighan e Ritchie, mais 5 adicionadas pelo comitê ANSI de padronização), que são os comandos que compõem a linguagem C. As linguagens de alto nível tipicamente tem várias vezes esse número de palavras reservadas. Como comparação, considere que a maioria das versões de BASIC possuem bem mais de 100 palavras reservadas!

"Hello, World" em Assembly x86	"Hello, World" em C
<pre>variable: .message db "Hello, World!\$" code: mov ah, 9 mov dx, offset .message int 0x21 ret</pre>	<pre>main() { printf("Hello, World!"); }</pre>

C é uma linguagem estruturada

Embora o termo linguagem estruturada em blocos não seja rigorosamente aplicável a C, ela é normalmente referida simplesmente como linguagem estruturada. C tem muitas semelhanças com outras linguagens estruturadas, como o Pascal.

A característica especial de uma linguagem estruturada é a compartimentalização do código e dos dados. Trata-se da habilidade de uma linguagem seccionar e esconder do resto do programa todas as informações necessárias para se realizar uma tarefa específica. Uma das maneiras de conseguir essa compartimentalização é pelo uso de sub-rotinas que empregam variáveis locais (temporárias). Com o uso de variáveis locais é possível escrever sub-rotinas de forma que os eventos que ocorrem dentro delas não causem nenhum efeito inesperado nas outras partes do programa. Essa capacidade permite que seus programas em C compartilhem facilmente seções de código. Se você desenvolve funções compartimentalizadas, só precisa saber **o que** uma função faz, não **como** ela faz. O uso excessivo de variáveis globais (variáveis conhecidas por todo o programa) pode trazer muitos erros, por permitir efeitos colaterais indesejados.

O principal componente estrutural de C é a função — a sub-rotina isolada de C. Em C, funções são os blocos de construção em que toda a atividade do programa ocorre. Elas admitem que você defina e codifique separadamente as diferentes tarefas de um programa, permitindo, então, que seu programa seja modular. Após uma função ter sido criada, você pode esperar que ela trabalhe adequadamente em varias situações, sem criar efeitos inesperados em outras partes do programa. O fato de você poder criar funções isoladas é extremamente importante em projetos maiores nos quais um código de um programador não deve afetar acidentalmente o de outro.

Outra maneira de estruturar e compartimentalizar o código em C é pelo uso de blocos de código. Um bloco de código é um grupo de comandos de programa conectado logicamente que é tratado como uma unidade. Em C, um bloco de código é criado colocando-se uma sequência de comandos entre chaves.

C é uma linguagem para programadores

Surpreendentemente, nem todas as linguagens de computador são para programadores. Considere os exemplos clássicos de linguagens para não-programadores: COBOL e BASIC. COBOL não foi destinada para facilitar a vida do programador, aumentar a segurança do código produzido ou a velocidade em que o código pode ser escrito. Ao contrário, COBOL foi concebida, em parte, para permitir que não-programadores leiam e presumivelmente (embora isso seja improvável) entendam o programa. BASIC foi criada essencialmente para permitir que não-programadores programem um computador para resolver problemas relativamente simples.

Em contraposição, C foi criada, influenciada e testada em campo por programadores profissionais. O resultado final é que C dá ao programador o que ele quer: poucas restrições, poucas reclamações, estruturas de bloco, funções isoladas e um conjunto compacto de palavras-chave. Usando C, um programador pode conseguir aproximadamente a eficiência de código Assembly combinada com a estrutura de uma linguagem de alto nível. Não é de admirar que C seja tranquilamente a linguagem mais popular entre excelentes programadores profissionais.

Em virtude da sua portabilidade e eficiência, à medida que C cresceu em popularidade, muitos programadores começaram a usá-la para programar todas as tarefas. Por haver compiladores C para quase todos os computadores, é possível tomar um código escrito para uma máquina, compilá-lo e rodá-lo em outra com pouca ou nenhuma modificação. Esta portabilidade economiza tempo e dinheiro.

Além disso, os programadores usam C em vários tipos de trabalho de programação porque eles gostam de C! Ela oferece a velocidade da linguagem Assembly, mas poucas das restrições presentes em outras linguagens. Cada programador C pode, de acordo com sua própria personalidade, criar e manter uma biblioteca única de funções customizadas, para ser usada em muitos programas diferentes. Por admitir — na verdade encorajar — a compilação separada, C permite que os programadores gerenciem facilmente grandes projetos com mínima duplicação de esforço.

2.3 C vs. C++

Algumas vezes os novatos confundem o que é C++ e como difere de C. Para ser breve, C++ é uma versão estendida e melhorada de C que é projetada para suportar POO - Programação Orientada a Objetos (OOP, do inglês *Object Oriented Programming*). C++ contém e suporta toda a linguagem C e mais um conjunto de extensões orientadas a objetos. (Ou seja, C++ é um superconjunto de C.) Como C++ é construída sobre os fundamentos de C, você não pode programar em C++ se não entender C. Portanto, virtualmente tudo que é aprendido sobre C, aplica-se também a C++.

Hoje em dia, e por muitos anos ainda, muitos programadores ainda escreverão, manterão e utilizarão programas C, e não C++. Como mencionado, C suporta programação estruturada. A programação estruturada tem-se mostrado eficaz ao longo dos muitos anos em que tem sido usada largamente. C++ é projetada principalmente para suportar POO, que incorpora os princípios da programação estruturada, mas inclui objetos. Embora a POO seja muito eficaz para uma certa classe de tarefas de programação, muitos programas não se beneficiam da sua aplicação. Por isso, “código direto em C” estará em uso por muito tempo ainda, a depender da classe de problema em que é aplicado.

Durante a experiência com o ensino de linguagem C, pude, como professor, observar alguns alunos que tinham algum conhecimento prévio da linguagem. No entanto, apesar de haver uma clara distinção entre C e C++, alguns deles tinham um conhecimento “misto” das duas linguagens. Vamos deixar algo claro: Se você não usará os recursos de POO (classes, herança, polimorfismo, etc.) não faz sentido escrever um código que poderia ser muito mais simples, em C, nos moldes de C++. O exemplo mais claro disso talvez seja o uso dos

objetos `cin` e `cout` (que obrigam o uso dos operadores sobrecarregados `>>` e `<<`) sem nem mesmo saber o que são objetos. Não é muito mais simples e compreensível usar as funções `printf()` e `scanf()` da biblioteca padrão do C?

O ponto chave a ser enfatizado neste sentido é: Cada linguagem carrega consigo uma filosofia, que orienta a forma de pensar do programador. Cada uma dessas filosofias (também chamadas de paradigmas) tem seus próprios conceitos e definições, que devem ser plenamente entendidos pelo programador. A programação orientada a objetos torna uma categoria de projetos mais simples de ser desenvolvida e mantida. No entanto, incorporam diversos conceitos complexos para um programador iniciante que, se não compreendidos plenamente, podem muito mais complicar que ajudar. De longe o paradigma estruturado é mais simples e direto que o orientado a objetos. Portanto, o conselho é: Se não precisa dos recursos da POO e ainda por cima é um programador com pouca ou nenhuma experiência, não use C++. Seja simples. Adote primeiramente o C. Migre para o C++ apenas depois de dominar plenamente o C.

2.4 A FORMA DE UM PROGRAMA EM C

A tabela abaixo lista as 32 palavras-chave (ou palavras reservadas) que, combinadas com a sintaxe formal de C, formam a linguagem de programação C. Destas, 27 foram definidas pela versão original de C. As cinco restantes foram adicionadas pelo comitê ANSI: `enum`, `const`, `signed`, `void` e `volatile`.

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

O compilador pode também suportar outras extensões que ajudem a aproveitar melhor seu ambiente específico.

Todas as palavras-chave de C são minúsculas. Em C, maiúsculas e minúsculas são diferentes: `else` é uma palavra-chave, mas `ELSE` não. Uma palavra-chave não pode ser usada para nenhum outro propósito em um programa em C — ou seja, ela não pode servir como uma variável ou nome de uma função.

É preciso ressaltar esse ponto de suma importância: o C é “Case Sensitive”, isto é, *MAIÚSCULAS e minúsculas fazem diferença*. Se se declarar uma variável com o nome `soma` ela será diferente de `Soma`, `SOMA`, `SoMa` ou `sOmA`. Da mesma maneira, os comandos do C `if` e `for`, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis.

Todo programa em C consiste em uma ou mais funções. A única função que necessariamente precisa estar presente é a denominada `main()`, que é a primeira função a ser chamada quando a execução do programa começa. Em um código de C bem escrito, `main()` contém, em essência, um esboço do que o programa faz. O esboço é composto de chamadas de funções. Embora `main()` não seja tecnicamente parte da linguagem C, trate-a como se fosse. Não tente usar `main()` como nome de uma variável porque provavelmente confundirá o compilador.

A forma geral de um programa em C é ilustrada abaixo, onde $f1()$ até $fN()$ representam funções definidas pelo usuário.

```
declarações globais

tipo_devolvido main(lista_de_parâmetros)
{
    sequência de comandos
}

tipo_devolvido f1(lista_de_parâmetros)
{
    sequência de comandos
}

tipo_devolvido f2(lista_de_parâmetros)
{
    sequência de comandos
}
.
.
.
tipo_devolvido fN(lista_de_parâmetros)
{
    sequência de comandos
}
```

2.4.1 DOIS PRIMEIROS PROGRAMAS

Vejamos um primeiro programa em C:

```
#include <stdio.h>

/* Um Primeiro Programa */

int main()
{
    printf("Hello, World\n");
    return(0);
}
```

Compilando e executando este programa você verá que ele coloca a mensagem `Hello, World!` na tela.

Vamos analisar o programa por partes.

A linha `#include <stdio.h>` diz ao compilador que ele deve incluir o arquivo-cabeçalho `stdio.h`. Neste arquivo existem declarações de funções úteis para entrada e saída de dados (std = standard, padrão em inglês; io = Input/Output, entrada e saída ==> stdio = Entrada e saída padronizadas). Toda vez que você quiser usar uma destas funções deve-se incluir este comando. O C possui diversos arquivos-cabeçalhos.

Quando fazemos um programa, uma boa ideia é usar comentários que ajudem a elucidar o funcionamento do mesmo. No caso acima temos um comentário: `/* Um Primeiro Programa */`. O compilador C desconsidera qualquer coisa que esteja começando com `/*` e terminando com `*/`. Um comentário pode, inclusive, ter mais de uma linha.

A linha `int main()` indica que estamos definindo uma função de nome `main`. Todos os programas em C têm que ter uma função `main`, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves `{ }`. O código que estiver dentro das chaves será executado sequencialmente quando a função for chamada. A palavra `int` indica que esta função retorna um inteiro. O que significa este retorno será visto posteriormente, quando estudarmos um pouco mais detalhadamente as

funções do C. A última linha do programa, `return(0);`, indica o número inteiro que está sendo retornado pela função, no caso o número 0.

A única coisa que o programa *realmente* faz é chamar a função `printf()`, passando a string (uma string é uma sequência de caracteres, como veremos futuramente) `"Hello, World!\n"` como argumento. É por causa do uso da função `printf()` pelo programa que devemos incluir o arquivo-cabeçalho `stdio.h`. A função `printf()` neste caso irá apenas colocar a string na tela do computador. O `\n` é uma constante chamada de *constante barra invertida*. No caso, o `\n` é a constante barra invertida de “new line” e ele é interpretado como um comando de mudança de linha, isto é, após imprimir `Hello, World!` o cursor passará para a próxima linha. É importante observar também que os *comandos* do C terminam com `;`.

Podemos agora tentar um programa mais complicado:

```
#include <stdio.h>

int main()
{
    int dias;                /* Declaração de Variáveis */
    float anos;
    printf("Entre com o numero de dias: "); /* Entrada de Dados */
    scanf("%d", &dias);
    anos = dias/365.25;      /* Conversão dias->anos */
    printf("\n\n%d dias equivalem a %f anos.\n", dias, anos);
    return(0);
}
```

Vamos entender como o programa acima funciona. São declaradas duas variáveis chamadas `dias` e `anos`. A primeira é um `int` (inteiro) e a segunda um `float` (ponto flutuante). As variáveis declaradas como ponto flutuante existem para armazenar números que possuem casas decimais, como 5.1497. (OBS: Em C, assim como no inglês, a separação entre a parte inteira e a parte decimal é dada pelo ‘.’ (ponto))

É feita então uma chamada à função `printf()`, que coloca uma mensagem na tela.

Queremos agora ler um dado que será fornecido pelo usuário e colocá-lo na variável inteira `dias`. Para tanto usamos a função `scanf()`. A string `“%d”` diz à função que iremos ler um inteiro. O segundo parâmetro passado à função diz que o dado lido deverá ser armazenado na variável `dias`. É importante ressaltar a necessidade de se colocar um `&` antes do nome da variável a ser lida quando se usa a função `scanf()`. O motivo disto só ficará claro mais tarde. Observe que, no C, quando temos mais de um parâmetro para uma função, eles serão separados por vírgula.

Temos então uma expressão matemática simples que atribui a `anos` o valor de `dias` dividido por 365.25 (365.25 é uma constante ponto flutuante 365,25). Como `anos` é uma variável `float` o compilador fará uma conversão automática entre os tipos das variáveis (veremos isto com detalhes mais tarde).

A segunda chamada à função `printf()` tem três argumentos. A string `“\n\n%d dias equivalem a %f anos.\n”` diz à função para pular duas linhas, colocar um inteiro na tela, colocar a mensagem “dias equivalem a”, colocar um valor `float` na tela, colocar a mensagem “ anos.” e pular outra linha. Os outros parâmetros são as variáveis, `dias` e `anos`, das quais devem ser lidos os valores do `int` e do `float`, respectivamente.

Cap. 3: EXPRESSÕES E COMANDOS BÁSICOS EM C

Este capítulo examinará o elemento mais fundamental da linguagem C: a *expressão*. Como veremos, as expressões em C são substancialmente mais gerais e poderosas que na maioria das outras linguagens de programação. As expressões são formadas pelos elementos mais básicos de C: *dados* e *operadores*. Os dados podem ser representados por variáveis ou constantes. C, como a maioria das outras linguagens, suporta alguns tipos diferentes de dados. Também provê uma ampla variedade de operadores.

3.1 OS CINCO TIPOS BÁSICOS DE DADOS

O C tem cinco tipos básicos: `char`, `int`, `float`, `double` e `void`. Abaixo, comentamos cada um deles:

- `char`: Tipo caractere. Strings, por exemplo, são formadas por uma sequência de caracteres. Os caracteres são na verdade valores numéricos inteiros, cuja correspondência entre o valor numérico e o caractere é dada pela tabela ASCII.
- `int`: Tipo inteiro. Acomoda uma ampla e mais comum faixa de valores inteiros usados normalmente em programação.
- `float` e `double`: Tipos ponto flutuante. Ambos os tipos são usados ao especificarmos valores com parte inteira e parte decimal. A diferença entre os dois está na precisão e na faixa de valores que representam, sendo o tipo `double` o mais preciso e amplo, consumindo, porém, mais memória (o dobro do `float`).
- `void`: Tipo vazio, ou “tipo sem tipo”. A aplicação deste “tipo” é importante, porém só será vista posteriormente.

Para cada um dos tipos de variáveis existem os modificadores de tipo. Os modificadores de tipo do C são quatro: `signed`, `unsigned`, `long` e `short`. Ao `float` não se pode aplicar nenhum e ao `double` pode-se aplicar apenas o `long`. Os quatro modificadores podem ser aplicados a inteiros. A intenção é que `short` e `long` devam prover tamanhos diferentes de inteiros onde isto for prático. Inteiros menores (`short`) ou maiores (`long`). `int` normalmente terá o tamanho natural para uma determinada máquina. Assim, numa máquina de 16 bits, `int` provavelmente terá 16 bits. Numa máquina de 32, `int` deverá ter 32 bits. Na verdade, cada compilador é livre para escolher tamanhos adequados para o seu próprio hardware, com a única restrição de que `short int` e `int` devem ocupar pelo menos 16 bits, `long int` pelo menos 32 bits, e `short int` não pode ser maior que `int`, que não pode ser maior que `long int`. O modificador `unsigned` serve para especificar variáveis sem sinal. Um `unsigned int` será um inteiro que assumirá apenas valores positivos (dobrando, assim, a faixa máxima admissível ao tipo `int`). A seguir estão listados os tipos de dados permitidos e seus valores máximos e mínimos em um compilador típico para um hardware de 16 bits. Também nesta tabela está especificado o formato que deve ser utilizado para ler os tipos de dados com a função `scanf()`.

Tipo	Num de bits	Formato para leitura com scanf	Intervalo	
			Início	Fim
char	8	%c	-128	127
unsigned char	8	%c	0	255
signed char	8	%c	-128	127
int	16	%d ou %i	-32768	32767
unsigned int	16	%u	0	65535
signed int	16	%i	-32768	32767
short int	16	%hi	-32768	32767
unsigned short int	16	%hu	0	65535
signed short int	16	%hi	-32768	32767
long int	32	%li	-2 147 483 648	2 147 483 647
signed long int	32	%li	-2 147 483 648	2.147 483 647
unsigned long int	32	%lu	0	4 294 967 295
float	32	%f	±3.4E-38	±3.4E+38
double	64	%lf	±1.7E-308	±1.7E+308
long double	80	%Lf	±3.4E-4932	±3.4E+4932

Conforme comentado, a tabela acima não é uma “verdade absoluta”. O tamanho (consequentemente a faixa de valores associada) pode variar dependendo do compilador/hardware/sistema operacional adotado. Tome como referência, por exemplo, o código abaixo e sua respectiva saída, ao ser compilado pelo Microsoft Visual C++ 2008 e executado sobre o Windows XP 32bits.

```
#include <stdio.h>

main()
{
    printf("Tipos de Dados e seus Tamanhos\n\n");

    printf("char: %d bits\n", sizeof(char)*8);
    printf("unsigned char: %d bits\n", sizeof(unsigned char)*8);
    printf("signed char: %d bits\n", sizeof(signed char)*8);
    printf("int: %d bits\n", sizeof(int)*8);
    printf("unsigned int: %d bits\n", sizeof(unsigned int)*8);
    printf("signed int: %d bits\n", sizeof(signed int)*8);
    printf("short int: %d bits\n", sizeof(short int)*8);
    printf("unsigned short int: %d bits\n", sizeof(unsigned short int)*8);
    printf("signed short int: %d bits\n", sizeof(signed short int)*8);
    printf("long int: %d bits\n", sizeof(long int)*8);
    printf("signed long int: %d bits\n", sizeof(signed long int)*8);
    printf("unsigned long int: %d bits\n", sizeof(unsigned long int)*8);
    printf("float: %d bits\n", sizeof(float)*8);
    printf("double: %d bits\n", sizeof(double)*8);
    printf("long double: %d bits\n", sizeof(long double)*8);

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

Saída:

Tipos de Dados e seus Tamanhos

```
char: 8 bits
unsigned char: 8 bits
signed char: 8 bits
int: 32 bits
unsigned int: 32 bits
signed int: 32 bits
short int: 16 bits
unsigned short int: 16 bits
signed short int: 16 bits
long int: 32 bits
signed long int: 32 bits
unsigned long int: 32 bits
float: 32 bits
double: 64 bits
long double: 64 bits
```

3.1.1 UM POUCO MAIS A RESPEITO DO TIPO CHAR

Fato: Computadores lidam apenas com valores lógicos 0 e 1. Esses podem ser representados por tensão baixa/alta (sinais nas trilhas e processador), presença ou ausência de carga elétrica (memória RAM), polaridade de micro ímãs (Hard Disk) ou luz refletida ou absorvida (CD, DVD, Blu-ray). Em quaisquer desses casos, esses 0's e 1's podem, usando a base 2 (binária), representar valor numéricos (inteiros e fracionários). Para saber como isso é feito, consulte o Apêndice A.

E quanto às “letras” (ou melhor, os “caracteres” como são chamados em computação)? Como o computador representa os caracteres de a-z, A-Z e 0-9? A resposta é mais simples do que você pode imaginar. O tipo `char` nada mais é que um tipo numérico inteiro de 8 bits (256 valores diferentes). E existe uma tabela padronizada internacionalmente, denominada Tabela ASCII (*American Standard Code for Information Interchange*), que associa valores inteiros a caracteres, conforme você pode conferir abaixo.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

ASCII foi publicada como padrão em 1967 e atualizada em 1986. Ela define códigos para 33 comandos de controle (uma espécie de caractere não imprimível), a maior parte deles obsoletos, mais 95 caracteres imprimíveis (começando com o caractere número 32 que representa o espaço).

Assim sendo, o código abaixo:

```
#include <stdio.h>

int main()
{
    char caractere = 'F';
    printf("O caractere %c corresponde ao inteiro %d\n", caractere, caractere);
}
```

exibe, ao ser compilado e executado:

■ O caractere F corresponde ao inteiro 70 ■

Mais uma vez, **não há distinção** entre valores inteiros (0-127) e caracteres. Se, na inicialização da variável `caractere`, tivéssemos atribuído o valor 70 (inteiro), o resultado seria **exatamente** o mesmo. Isso precisa estar claro para você, OK?

Você deve ter reparado uma coisa: Não há caracteres acentuados nem caracteres especiais presentes em muitas línguas (como o português). O motivo é simples: Este padrão, que tornou-se dominante no mundo, é americano e para o inglês tais caracteres não fazem a menor falta. Esse é o motivo pelo qual sistemas muito antigos exibem e imprimem tudo sem acentos. Então, os computadores modernos conseguem lidar com acentos e demais caracteres especiais, facilmente encontrados em dezenas de outras línguas? A resposta é UNICODE. Mas isso ficará ao seu cargo pesquisar! ☺

3.2 VARIÁVEIS

Uma variável é uma posição nomeada de memória, que é usada para guardar um valor que pode ser modificado pelo programa. Todas as variáveis em C devem ser declaradas antes de serem usadas.

A forma geral de uma declaração é

```
tipo lista_de_variáveis;
```

Aqui, *tipo* deve ser um tipo de dado válido em C mais quaisquer modificadores; e *lista_de_variáveis* pode consistir em um ou mais nomes de identificadores (“nomes” das variáveis) separados por vírgulas. Aqui estão algumas declarações:

```
int i, j, l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Lembre-se de que, em C, o nome de uma variável não tem nenhuma relação com seu tipo.

Visto que, a princípio, todos os algoritmos que escreveremos em C estarão restritos a criação de uma única função (`main`), lembre-se de declarar todas as variáveis necessárias no começo da função `main`, antes de escrever qualquer outro comando. Há outros lugares possíveis de declarar variáveis, e isto está intimamente relacionado a um conceito importante que será visto no Capítulo 6, onde trataremos de modularização e escopo de variáveis.

Podemos inicializar variáveis no momento de sua declaração. Para fazer isto podemos usar a forma geral

```
tipo_da_variável nome_da_variável = constante;
```

Isto é importante, pois quando o C cria uma variável ele *não* a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor *indefinido* e que não pode ser utilizado para nada. *Nunca* presume que uma variável declarada vale zero ou qualquer outro valor. Exemplos de inicialização são dados abaixo:

```
char ch = 'D';
int count = 0;
float pi = 3.141;
```

Ressalte-se novamente que, em C, uma variável tem que ser declarada no início de um bloco de código¹. Assim, o programa a seguir não é válido em C (embora seja válido em C++).

```
int main()
{
    int i;
    int j;
    j = 10;
    int k = 20;    /* Esta declaracao de variável não é válida,
                   pois não está sendo feita no início do bloco */
    return(0);
}
```

3.3 CONSTANTES

Em C, constantes referem-se a valores fixos que o programa não pode alterar. Constantes em C podem ser de qualquer um dos cinco tipos de dados básicos. A maneira como cada constante é representada depende do seu tipo. Constantes de caractere são envolvidas por aspas simples ('). Por exemplo, 'a' e '%' são constantes tipo caractere.

Constantes inteiras são especificadas como números sem componentes fracionários. Por exemplo, 10 e -100 são constantes inteiras. Constantes em ponto flutuante requerem o ponto decimal seguido pela parte fracionária do número. Por exemplo, 11.123 é uma constante em ponto flutuante. C também permite que você use notação científica para números em ponto flutuante.

Existem dois tipos de ponto flutuante: `float` e `double`. Há também diversas variações dos tipos básicos que você pode gerar usando os modificadores de tipo. Por padrão, o compilador C encaixa uma constante numérica no menor tipo de dado compatível que pode contê-lo. Assim, 10 é um `int`, por padrão, mas 60000 é `unsigned int` e 100000 é `long int`. Muito embora o valor 10 possa caber em um tipo caractere, o compilador não atravessará os limites do tipo. A única exceção para a regra do menor tipo são constantes em ponto flutuante, assumidas como `double`.

Na maioria dos programas que você escreverá, os padrões do compilador são adequados. Porém, você pode especificar precisamente o tipo da constante numérica que deseja por meio da utilização de um sufixo. Para tipos em ponto flutuante, se você colocar um `F` após o número, ele será tratado como `float`. Se você colocar um `L`, ele se tornará um `long double`. Para tipos inteiros, o sufixo `U` representa `unsigned` e o `L` representa `long`. Aqui estão alguns exemplos:

¹ O C padrão ANSI exige que todas as variáveis sejam declaradas no começo de um bloco (por exemplo, o começo de uma função) e antes de qualquer outro comando. Portanto, compiladores que seguem rigorosamente o padrão ANSI reportarão erro de compilação caso essa exigência seja violada (ou seja, se você tentar declarar uma variável em outro lugar). Isso confunde alguns estudantes, pois, o conhecido compilador GCC não possui essa exigência, ao contrário do Microsoft Visual C++ (que segue o padrão C ANSI). O motivo é que essa restrição foi removida no C++. Logo, alguns compiladores (é o caso GCC) decidiram por fazer o mesmo com códigos escritos em C puro. Uma violação do padrão, mas, em prol de alguma flexibilidade para os programadores.

Tipo de dado	Exemplos de constantes
int	1 123 21000 -234
long int	35000L -34L
short int	10 -12 90
unsigned int	10000U 987U 40000
float	123.23F 4.34e-3F
double	123.23 12312333 -0.9876324
long double	1001.2L

3.3.1 CONSTANTES STRING

C suporta outro tipo de constante: a string. Uma string é um conjunto de caracteres colocado entre aspas duplas. Por exemplo, "isso é um teste" é uma string. Você viu exemplos de strings em alguns dos comandos `printf()` dos programas de exemplo. Embora C permita que você defina constantes string, ela não possui formalmente um tipo de dado string. Veremos futuramente (Seção 5.2) como são implementadas as variáveis do tipo string em C.

Você não deve confundir strings com caracteres. Uma constante de um único caractere é colocada entre aspas simples, como em `'a'`. Contudo, `"a"` é uma string contendo apenas uma letra.

3.3.2 CONSTANTES CARACTERE DE BARRA INVERTIDA

Colocar entre aspas simples todas as constantes tipo caractere funciona para a maioria dos caracteres imprimíveis. Uns poucos, porém, como o retorno de carro (CR), são impossíveis de inserir pelo teclado. Por essa razão, C criou as constantes especiais de caractere de barra invertida.

C suporta diversos códigos de barra invertida (listados na tabela abaixo) de forma que você pode facilmente entrar esses caracteres especiais como constantes. Você deve usar os códigos de barra invertida em lugar de seus ASCII equivalentes para aumentar a portabilidade.

Código	Significado
<code>\b</code>	Retrocesso (BS)
<code>\f</code>	Alimentação de formulário (FF)
<code>\n</code>	Nova linha (LF)
<code>\r</code>	Retorno de carro (CR)
<code>\t</code>	Tabulação horizontal (HT)
<code>\"</code>	Aspas duplas
<code>\'</code>	Aspas simples
<code>\0</code>	Nulo
<code>\\</code>	Barra invertida
<code>\v</code>	Tabulação vertical
<code>\a</code>	Alerta (beep)

Por exemplo, o programa seguinte envia a tela um caractere de nova linha e uma tabulação e, em seguida, escreve a string `Isso eh um teste.`

```
#include <stdio.h>

main()
{
    printf("\n\tIsso eh um teste");
}
```

3.4 OPERADORES

C é muito rica em operadores internos. (Na realidade, C dá mais ênfase aos operadores que a maioria das outras linguagens de computador.) C define quatro classes de operadores: aritméticos, relacionais, lógicos e bit a bit. Além disso, C tem alguns operadores especiais para tarefas particulares.

3.4.1 OPERADOR DE ATRIBUIÇÃO

Em C, você pode usar o operador de atribuição dentro de qualquer expressão válida de C. Isso não acontece na maioria das linguagens de computador (incluindo Pascal, BASIC e FORTRAN), que tratam os operadores de atribuição como um caso especial de comando. A forma geral do operador de atribuição é

```
nome_da_variavel = expressão;
```

onde expressão pode ser tão simples como uma única constante ou tão complexa quanto você necessite. Como BASIC e FORTRAN, C usa um único sinal de igual para indicar atribuição (ao contrario de Pascal e Modula-2, que usam a construção :=). O destino, ou a parte esquerda, da atribuição deve ser uma variável ou um ponteiro (a ser visto futuramente, no Capítulo 8), não uma função ou uma constante.

3.4.1.1 ATRIBUIÇÕES MÚLTIPLAS

C permite que você atribua o mesmo valor a muitas variáveis usando atribuições múltiplas em um único comando. Por exemplo, esse fragmento de programa atribui a x, y e z o valor 0:

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

Isto se dá porque o comando de atribuição é tratado como um operador, que atribui o valor da direita à variável à esquerda e retorna (ou seja, tem como “resultado”) o valor atribuído. Este valor retornado pode então ser atribuído a outra possível variável à esquerda. Em programas profissionais, valores comuns são atribuídos a variáveis usando esse método.

3.4.2 OPERADORES ARITMÉTICOS

Os operadores aritméticos são usados para desenvolver operações matemáticas. A seguir apresentamos a lista dos operadores aritméticos do C:

Operador	Ação
+	Soma (inteira e ponto flutuante)
-	Subtração ou Troca de sinal (inteira e ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Módulo (resto) de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
--	Decremento (inteiro e ponto flutuante)

O C possui operadores *unários* e *binários*. Os unários agem sobre uma variável apenas, modificando ou não o seu valor, e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores $-$ (subtração), $*$, $/$ e $\%$. O operador $-$ como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1 .

O operador $/$ (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão “real”. Por exemplo, $5/2$ será igual a 2 em uma divisão inteira. O operador $\%$ fornece o resto da divisão de dois inteiros. Assim seja o seguinte trecho de código:

```
int a = 17, b = 3;
int x, y;
float z = 17. , z1, z2;
x = a / b;
y = a % b;
z1 = z / b;
z2 = a / b;
```

ao final da execução destas linhas, os valores calculados seriam $x = 5$, $y = 2$, $z1 = 5.666666$ e $z2 = 5.0$. Note que, na linha correspondente a $z2$, primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente após efetuada a divisão é que o resultado é atribuído a uma variável `float`.

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

```
x = x + 1;
x = x - 1;
```

Estes operadores podem ser pré-fixados ou pós-fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Então, em

```
x = 23;
y = x++;
```

teremos, no final, $y=23$ e $x=24$. Em

```
x = 23;
y = ++x;
```

teremos, no final, $y=24$ e $x=24$. Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um “incremento” da linguagem C padrão. A linguagem C++ é igual a linguagem C só que com extensões que permitem a programação orientada a objeto, o que é um recurso extra.

O operador de atribuição do C é o $=$. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto, ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x = y = z = 1.5;    /* Expressão 1 */
if (k = w)...        /* Expressão 2 */
```

A expressão 1 é válida, pois quando fazemos $z = 1.5$ ela retorna 1.5 , que é passado adiante, fazendo $y = 1.5$ e posteriormente $x = 1.5$. A expressão 2 será verdadeira se w for diferente de zero², pois este será o valor retornado por $k=w$. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você *não* está comparando k e w . Você está atribuindo o valor de w a k e usando este valor para tomar a decisão. Os operadores de comparação são abordados na próxima seção.

3.4.3 OPERADORES RELACIONAIS E LÓGICOS

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

A ideia de verdadeiro e falso é a base dos conceitos dos operadores lógicos e relacionais. Em C, **verdadeiro** é qualquer valor diferente de zero. **Falso** é zero. As expressões que usam operadores relacionais ou lógicos devolvem zero para falso e 1 para verdadeiro.

Na tabela abaixo são listados os operadores lógicos e relacionais.

Operadores relacionais	
Operador	Ação
>	Maior que
>=	Maior que ou igual
<	Menor que
<=	Menor que ou igual
==	Igual
!=	Diferente
Operadores Lógicos	
Operador	Ação
&&	AND (E)
	OR (OU)
!	NOT (NEGAÇÃO)

A tabela verdade dos operadores lógicos é mostrada a seguir, usando 1's e 0's.

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

² Devido às convenções para valores "verdadeiro" e "falso" adotada pelo C, que serão vistas posteriormente.

Ambos os operadores são menores em precedência do que os operadores aritméticos. Isto é, uma expressão como `10 > 1 + 12` é avaliada como se fosse escrita `10 > (1+12)`. O resultado é, obviamente, falso.

É permitido combinar diversas operações em uma expressão como mostrado aqui:

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

Neste caso, o resultado é **verdadeiro**.

3.4.4 OUTROS OPERADORES

C possui ainda outros operadores que serão apenas citados aqui, cabendo explicações mais detalhadas em capítulos seguintes, uma vez que podem envolver conceitos de programação mais avançados.

3.4.4.1 OPERADORES BIT A BIT

Uma vez que C foi projetada para substituir a linguagem assembly na maioria das tarefas de programação, era importante que ela tivesse a habilidade de suportar muitas das operações que podem ser feitas em linguagem assembly. *Operação bit a bit* refere-se a testar, atribuir ou deslocar os bits efetivos em um byte ou uma palavra, que correspondem aos tipos de dados `char` e `int` e variantes do padrão C. Operações bit não podem ser usadas em `float`, `double`, `long double`, `void` ou outros tipos mais complexos. A tabela abaixo lista os operadores que se aplicam as operações bit a bit. Essas operações são aplicadas aos bits individuais dos operandos.

Operador	Ação
&	AND
	OR
^	OR exclusivo (XOR)
~	Complemento de um
>>	Deslocamento à direita
<<	Deslocamento à esquerda

3.4.4.2 O OPERADOR ?

C contém um operador muito poderoso e conveniente que substitui certas sentenças da forma if-then-else. O operador ternário ? tem a forma geral

```
Exp1 ? Exp2 : Exp3;
```

onde `Exp1`, `Exp2` e `Exp3` são expressões. Note o use e o posicionamento dos dois-pontos.

O operador ? funciona desta forma: `Exp1` é avaliada. Se ela for verdadeira, então `Exp2` é avaliada e se torna o valor da expressão. Se `Exp1` é falsa, então `Exp3` é avaliada e se torna o valor da expressão. Por exemplo, em

```
x = 10;
y = x > 9 ? 100 : 200;
```

a `y` é atribuído o valor 100. Se `x` fosse menor que 9, `y` teria recebido o valor 200. O mesmo código, usando o comando if-else, é

```
x = 10;
if (x > 9) y = 100;
else y = 200;
```

3.5 EXPRESSÕES

Operadores, constantes e variáveis são os elementos que constituem as expressões. Uma expressão em C é qualquer combinação válida desses elementos. Uma vez que a maioria das expressões tende a seguir as regras gerais da álgebra, elas são frequentemente tomadas como certas. Contudo, existem uns poucos aspectos de expressões que se referem especificamente a C.

Exemplos de expressões:

```
anos = dias/365.25;
i = i+3;
c= a*b + d/e;
c= a*(b+d)/e;
```

Quando montamos expressões temos que levar em consideração a ordem com que os operadores são executados, conforme a tabela de precedências da linguagem C, exibida abaixo.

Maior precedência	() [] ->
	! ~ ++ -- . - (unário) (cast) *(unário) &(unário) sizeof
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	&&
	? :
	= += -= *= /=
Menor precedência	,

Muitos dos operadores exibidos na tabela só serão abordados futuramente.

A fim de facilitar a memorização dos principais operadores encontrados nas expressões mais comuns, podemos resumir a tabela acima da seguinte forma:

Maior precedência	()
	!
	Op. Aritméticos: (1º) * / % (2º) + -
	Op. Relacionais: < <= > >= == !=
Menor precedência	Op. Lógicos: (1º) && (2º)

3.5.1 ORDEM DE AVALIAÇÃO

O padrão C ANSI não estipula que as subexpressões de uma expressão devam ser avaliadas em uma ordem especificada. Isso deixa o compilador livre para rearranjar uma expressão para produzir o melhor código. No entanto, isso também significa que seu código nunca deve contar com a ordem em que as subexpressões são avaliadas. Por exemplo, a expressão

```
x = f1( ) + f2( );
```

não garante que `f1()` será chamada antes de `f2()`.

3.5.2 CONVERSÃO DE TIPOS EM EXPRESSÕES

Quando o C avalia expressões onde temos variáveis de tipos diferentes, o compilador verifica se as conversões são possíveis. Se não são, ele não compilará o programa, dando uma mensagem de erro. Se as conversões forem possíveis ele as faz, seguindo as regras abaixo:

1. Todos os `char` e `short int` são convertidos para `int`. Todos os `float` são convertidos para `double`.
2. Para pares de operandos de tipos diferentes: se um deles é `long double` o outro é convertido para `long double`; se um deles é `double` o outro é convertido para `double`; se um é `long` o outro é convertido para `long`; se um é `unsigned` o outro é convertido para `unsigned`.

3.5.3 EXPRESSÕES QUE PODEM SER ABREVIADAS

O C admite as seguintes equivalências, que podem ser usadas para simplificar expressões ou para facilitar o entendimento de um programa:

Expressão Original	Expressão Equivalente
<code>x=x+k;</code>	<code>x+=k;</code>
<code>x=x-k;</code>	<code>x-=k;</code>
<code>x=x*k;</code>	<code>x*=k;</code>
<code>x=x/k;</code>	<code>x/=k;</code>
<code>x=x>>k;</code>	<code>x>>=k;</code>
<code>x=x<<k;</code>	<code>x<<=k;</code>
<code>x=x&k;</code>	<code>x&=k;</code>
etc...	

3.5.4 ENCADEANDO EXPRESSÕES: O OPERADOR ,

O operador `,` (vírgula) determina uma lista de expressões que devem ser executadas sequencialmente. Em síntese, a vírgula diz ao compilador: execute as duas expressões separadas pela vírgula, em sequência. O valor retornado por uma expressão com o operador `,` é sempre dado pela expressão mais à direita. No exemplo abaixo:

```
x = (y=2, y+3);
```

o valor 2 vai ser atribuído a `y`, se somará 3 a `y` e o retorno (5) será atribuído à variável `x`. Pode-se encadear quantos operadores `,` forem necessários.

O exemplo a seguir mostra outro uso para o operador `,` dentro de um `for`:

```
main()
{
    int x, y;
    /* Duas variáveis de controle: x e y .
       Foi atribuído o valor zero a cada uma delas
       na inicialização do for e ambas são incrementadas
       na parte de incremento do for */
    for(x=0 , y=0 ; x+y < 100 ; ++x , y++)
        printf("\n%d ", x+y); /* o programa imprimirá os números pares de 0 a 98 */
}
```

Dica: Você não precisa saber toda a tabela de precedências de cor. É útil que você conheça as principais relações, mas é aconselhável que ao escrever o seu código, você tente isolar as expressões com parênteses, para tornar o seu programa mais legível.

3.5.5 MODELADORES (CASTS)

Um modelador é aplicado a uma expressão. Ele *força* a mesma a ser de um tipo especificado. Sua forma geral é:

`(tipo) expressão`

Um exemplo:

```
#include <stdio.h>

main()
{
    int num;
    float f1, f2;
    num = 5;
    f1 = num/2;
    f2 = (float)num/2; /* Uso do cast. Força a transformação de num em float */
    printf("f1 = %f (Resultado sem cast)\n", f1);
    printf("f2 = %f (Resultado com cast)\n\n", f2);
}
```

Observe atentamente o resultado:

```
f1 = 2.000000 (Resultado sem cast)
f2 = 2.500000 (Resultado com cast)
```

Não utilizando o `cast`, o C considera a expressão `f1=num/2;` como uma divisão inteira, já que ambos os operandos são inteiros. Usando o `cast`, na expressão `f2=(float) num/2;` a variável `num` é convertida para o tipo `float`, de modo que o operador de divisão passa a tratar a expressão como uma divisão usando ponto flutuante (resultando no valor correto).

3.6 FUNÇÕES DE ENTRADA E SAÍDA

Nesta seção as duas principais funções de entrada/saída de texto serão brevemente discutidas, uma vez que são básicas a qualquer algoritmo que você escreverá. As discutiremos em detalhes posteriormente (Seção 9.12.3).

3.6.1 SAÍDA DE TEXTO

A função `printf()` é a mais conhecida e usada função de saída de texto formatado. Ela possui a seguinte forma geral:

```
printf(string_de_controle,lista_de_argumentos);
```

Teremos, na string de controle, uma descrição de tudo que a função vai colocar na tela. A string de controle mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação `%`. Na string de controle indicamos quais, de qual tipo e em que posição estão as variáveis a serem apresentadas. É muito importante que, para cada código de controle, tenhamos um argumento na lista de argumentos. Apresentamos agora alguns dos códigos `%`:

Código	Significado
<code>%d</code>	Inteiro
<code>%f</code>	Float
<code>%c</code>	Caractere
<code>%s</code>	String
<code>%%</code>	Coloca na tela um %

Vamos ver alguns exemplos de `printf()` e o que eles exibem:

```
#include <stdio.h>

main()
{
    printf("Teste %% %% \n");
    printf("%f \n", 40.345);
    printf("Um caractere %c eh um inteiro %d \n", 'D', 120);
    printf("%s eh um exemplo \n", "Este");
    printf("%s%d%% \n", "Juros de ", 10);

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

e sua saída:

```
Teste % %
40.345000
Um caractere D eh um inteiro 120
Este eh um exemplo
Juros de 10%
```

3.6.2 ENTRADA DE TEXTO

A função `scanf()` é praticamente “casada” com a função `printf()`, diferindo desta por ser uma função de entrada de texto. O formato geral da função `scanf()` é

```
scanf (string-de-controle,lista-de-argumentos);
```

Usando a função `scanf()` podemos pedir dados ao usuário. Por exemplo:

```
#include <stdio.h>

main()
{
    char ch;
    scanf("%c", &ch);
    printf("Voce pressionou a tecla %c \n\n", ch);
}
```

Mais uma vez, devemos ficar atentos a fim de colocar o mesmo número de argumentos que o de códigos de controle na string de controle. Outra coisa importante é lembrarmos-nos de colocar o `&` antes das variáveis da lista de argumentos. É impossível justificar isto agora, mas veremos depois a razão para este procedimento futuramente, quando estivermos estudando ponteiros (Capítulo 8).

Em algumas situações o iniciante em programação em C frustra-se com o comportamento da função `scanf()`. Quando o `scanf` é usado para capturar um caracter sendo que anteriormente a este tenha havido uma entrada de dados qualquer (e consequentemente um ENTER), este segundo `scanf` pode “pular”. Este, na verdade, foi executado. No entanto, como estava programado para capturar um caractere, acabou por capturar o caracter ‘`\n`’ do ENTER da entrada de dados anterior, que ficou no *buffer* de teclado (uma região de memória temporária). A seção 9.12.3.2 apresenta melhor o problema e uma possível solução. Quando usado compiladores Microsoft, é, no entanto, mais simples usar a instrução `fflush(stdin)` para limpar o buffer de teclado antes de novas entradas de dados.

Cap. 4: COMANDOS DE CONTROLE DO PROGRAMA

As estruturas de controle de fluxo são fundamentais para qualquer linguagem de programação. Sem elas só haveria uma maneira do programa ser executado: de cima para baixo comando por comando. Não haveria condições, repetições ou saltos. A linguagem C possui diversos comandos de controle de fluxo. É possível resolver todos os problemas sem utilizar todas elas, mas devemos nos lembrar de que a elegância e facilidade de entendimento de um programa dependem do uso correto das estruturas no local certo.

4.1 VERDADEIRO E FALSO EM C

Muitos comandos em C contam com um teste condicional que determina o curso da ação. Uma expressão condicional chega a um valor verdadeiro ou falso. Em C, ao contrário de muitas outras linguagens, um valor *verdadeiro* é *qualquer valor diferente de zero*, incluindo números negativos. Um valor *falso* é *0*. Esse método para verdadeiro e falso permite que uma ampla gama de rotinas sejam codificadas de forma extremamente eficiente.

Portanto, lembre-se sempre:

- *Falso* = **0** (zero).
- *Verdadeiro* = Qualquer valor diferente de zero. Frequentemente, por simplificação, adota-se o valor **1**.

4.2 COMANDOS DE SELEÇÃO

C suporta dois tipos de comandos de seleção: `if` e `switch`. Há situações em que ambos os comandos podem ser usados. Contudo, há diversas situações em que o uso do comando `switch` facilita o entendimento e a clareza do código. Cabe, portanto, perspicácia no uso de cada comando.

4.2.1 if

O comando condicional `if` é um dos elementos mais usados da linguagem. Sua forma geral é:

```
if (condição) declaração;
```

A expressão, na *condição*, será avaliada. Se o resultado dela for **zero**, a declaração não será executada. Se a condição for **diferente de zero** a declaração será executada. Aqui reapresentamos o exemplo de um uso do comando `if`:

```
#include <stdio.h>
main()
{
    int num;
    printf("Digite um numero: ");
    scanf("%d", &num);
    if (num > 10)
        printf("\n0 numero eh maior que 10.\n\n");
    if (num == 10)
    {
        printf("\nVoce acertou!\n");
        printf("0 numero eh igual a 10.\n\n");
    }
    if (num < 10)
        printf("\n0 numero eh menor que 10.\n\n");
}
```

O comando `if` pode ser complementado com uma cláusula `else`, a fim de tratar os casos onde a expressão condicional é falsa (ou seja, tem zero como resultado). Portanto, a forma geral completa do comando `if` é:

```
if (condição) declaração_1;
else declaração_2;
```

A expressão da condição será avaliada. Se ela for diferente de zero a `declaração_1` será executada. Se for zero a `declaração_2` será executada. É importante nunca esquecer que, quando usamos a estrutura `if-else`, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Abaixo está um exemplo do uso do `if-else`:

```
#include <stdio.h>

main()
{
    int num;

    printf("Digite um numero: ");

    scanf("%d", &num);

    if (num == 10)
    {
        printf("\n\nVoce acertou!\n");
        printf("O numero eh igual a 10.\n");
    }
    else
    {
        printf("\n\nVoce errou!\n");
        printf("O numero eh diferente de 10.\n");
    }
}
```

Tanto para o `if`, quanto para o `else` (e veremos, posteriormente, que o mesmo vale para outros comandos) é possível omitir o uso das chaves caso exista apenas um comando associado (como pode ser visto no exemplo do começo dessa seção). Caso contrário, as chaves são obrigatórias, já que delimitam um bloco de código associado.

É possível também encadear uma série de comandos `if`, dando origem a uma “escada” do tipo `if-else-if`. Sua forma geral pode ser escrita como sendo:

```
if (condição_1) declaração_1;
else if (condição_2) declaração_2;
else if (condição_3) declaração_3;
.
.
.
else if (condição_n) declaração_n;
else declaração_default;
```

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à primeira condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional. Um exemplo da estrutura acima:

```
#include <stdio.h>

main()
{
    int num;
    printf("Digite um numero: ");
    scanf("%d", &num);
    if (num > 10)
        printf("\n\nO numero eh maior que 10.");
    else if (num == 10) {
        printf("\n\nVoce acertou!\n");
        printf("O numero eh igual a 10.");
    }
    else if (num < 10)
        printf("\n\nO numero eh menor que 10.");
}
```

Quando uma sequência de comandos `if` encadeados (tal qual mostrada acima) está sendo usada simplesmente para executar um ou mais comandos caso o valor de uma variável seja igual a uma determinada constante, o uso de um comando `switch` pode ser altamente recomendado, conforme veremos na Seção 4.2.2.

4.2.1.1 A EXPRESSÃO CONDICIONAL

Quando o compilador avalia uma condição, ele quer um valor de retorno para poder tomar a decisão. Mas esta expressão não necessita ser uma expressão no sentido convencional. Uma variável sozinha pode ser uma “expressão” e esta retorna o seu próprio valor. Isto quer dizer que teremos as seguintes expressões:

```
int num;
if (num != 0) ...
if (num == 0) ...
```

equivalem a

```
int num;
if (num) ...
if (!num) ...
```

Isto quer dizer que podemos simplificar algumas expressões simples.

4.2.1.2 if's ANINHADOS

O `if` aninhado é simplesmente um `if` dentro da declaração de um outro `if` externo. O único cuidado que devemos ter é o de saber exatamente a qual `if` um determinado `else` está ligado. Vejamos um exemplo:

```
#include <stdio.h>

main()
{
    int num;
    printf("Digite um numero: ");
    scanf ("%d", &num);
    if (num == 10)
    {
        printf("\n\nVoce acertou!\n");
        printf("O numero eh igual a 10.\n");
    }
    else
    {
        if (num > 10)
            printf("O numero eh maior que 10.");
        else
            printf("O numero eh menor que 10.");
    }
}
```

4.2.1.3 O OPERADOR ?

Uma expressão como:

```
if (a > 0)
    b = -150;
else
    b = 150;
```

pode ser simplificada usando-se o operador ? da seguinte maneira:

```
b = (a > 0) ? -150 : 150;
```

De uma maneira geral expressões do tipo:

```
if (condição)
    expressão_1;
else
    expressão_2;
```

podem ser substituídas por:

```
condição ? expressão_1 : expressão_2;
```

4.2.2 switch

O comando `if` e o comando `switch` são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o `if`, por ser mais geral, mas o comando `switch` tem aplicações valiosas. Vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando `switch` é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
{
    case constante_1:
        declaração_1;
        break;
    case constante_2:
        declaração_2;
        break;
    .
    .
    .
    case constante_n:
        declaração_n;
        break;
    default:
        declaração_default;
}
```

Podemos fazer uma analogia entre o `switch` e a estrutura `if-else-if` apresentada anteriormente. A diferença fundamental é que o comando `switch` *não aceita expressões*. Aceita *apenas constantes*. O `switch` testa a variável e executa a declaração cujo `case` corresponda ao valor atual da variável. A declaração `default` é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.

O comando `break`, faz com que o `switch` seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando `switch`. Se após a execução da declaração não houver um

`break`, o programa continuará executando. Isto pode ser útil em algumas situações, mas é recomendado cuidado. Veremos agora um exemplo do comando `switch`:

```
#include <stdio.h>

main()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d", &num);
    switch (num)
    {
        case 9:
            printf("\n\n0 numero eh igual a 9.\n");
            break;
        case 10:
            printf("\n\n0 numero eh igual a 10.\n");
            break;
        case 11:
            printf("\n\n0 numero eh igual a 11.\n");
            break;
        default:
            printf("\n\n0 numero nao eh nem 9 nem 10 nem 11.\n");
    }
}
```

4.3 COMANDOS DE ITERAÇÃO

Em C, e em todas as outras linguagens modernas de programação, comandos de iteração, também chamados *loops* (laços), permitem que um conjunto de instruções seja executado até que ocorra uma certa condição. A linguagem define três comandos de loop: `for`, `while` e `do-while`. Inicialmente apresentaremos cada um deles e depois apresentaremos uma breve discussão sobre o uso apropriado de cada comando.

4.3.1 for

O formato geral do laço `for` de C é encontrado, de uma forma ou de outra, em todas as linguagens de programação baseadas em procedimentos. Contudo, em C, ele fornece flexibilidade e capacidade surpreendentes.

A forma geral do comando `for` é

```
for(inicialização; condição; incremento) comando;
```

O laço `for` permite muitas variações. Entretanto, a *inicialização* é, geralmente, um comando de atribuição que é usado para colocar um valor na variável de controle do laço. A *condição* uma expressão relacional que determina quando o laço acaba. O *incremento* define como a variável de controle do laço varia cada vez que o laço é repetido. Você deve separar essas três seções principais por pontos-e-vírgulas. Uma vez que a condição se torne falsa, a execução do programa continua no comando seguinte ao `for`.

Um modo de se entender o loop `for` é ver como ele funciona “por dentro”. O loop `for` é equivalente a se fazer o seguinte:

```
inicialização;
if (condição)
{
    comandos;
    incremento;
    "Volte para o comando if"
}
```

Podemos ver, então, que o `for` executa a *inicialização* incondicionalmente e testa a *condição*. Se a *condição* for falsa ele não faz mais nada. Se a *condição* for verdadeira ele executa o *comando*, faz o *incremento* e volta a testar a *condição*. Ele fica repetindo estas operações até que a *condição* seja falsa. Um ponto importante é que podemos omitir qualquer um dos elementos do `for`, isto é, se não quisermos uma *inicialização* poderemos omiti-la.

Por exemplo, o seguinte programa imprime os números, de 1 a 100 na tela:

```
#include <stdio.h>

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

No programa, `x` é inicialmente ajustado para 1. Uma vez que `x` é menor que 100, `printf()` é executado e `x` é incrementado em 1 e testado para ver se ainda é menor ou igual a 100. Esse processo se repete até que `x` fique maior que 100; nesse ponto, o laço termina. Nesse exemplo, `x` é a variável de controle do laço, que é alterada e testada toda vez que o laço se repete.

O seguinte exemplo é um laço `for` que contém múltiplos comandos:

```
for(x=100; x != 65; x-=5)
{
    z = x*x;
    printf("O quadrado de %d eh %f",x, z);
}
```

Tanto a multiplicação de `x` por si mesmo como a função `printf()` são, executadas até que `x` seja igual a 65. Note que o laço é executado de forma inversa: `x` é inicializado com 100 e será subtraído de 5 cada vez que o laço se repetir.

Nos laços `for`, o teste condicional sempre é executado no topo do laço. Isso significa que o código dentro do laço pode não ser executado se todas as condições forem falsas logo no início.

O `for` na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do `for` pode ser uma expressão qualquer do C, desde que ela seja válida. Isto nos permite fazer o que quisermos com o comando. As três formas do `for` abaixo são válidas:

```
/* O limite é dado por um valor literal, 100 no caso */
for (count = 1; count < 100 ; count++) { ... }

/* O limite é dado por uma "constante" NUMERO_DE_ELEMENTOS */
for (count = 1; count < NUMERO_DE_ELEMENTOS ; count++) { ... }

/* O limite é dado pelo resultado da chamada de uma função */
for (count = 1; count < BusqueNumeroDeElementos() ; count+=2) { ... }
```

Preste atenção ao último exemplo: o incremento está sendo feito de dois em dois. Além disto, no teste está sendo utilizada uma função (`BusqueNumeroDeElementos()`) que retorna um valor que está sendo comparado com `count`.

O Loop Infinito

O loop infinito tem a forma

```
for (inicialização; ;incremento) declaração;
```


Este loop chama-se loop infinito porque será executado para sempre (não existindo a condição, ela será sempre considerada verdadeira), a não ser que ele seja interrompido. Para interromper um loop como este usamos o comando `break`. O comando `break` vai quebrar o loop infinito e o programa continuará sua execução normalmente.

Como exemplo, vamos ver um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuário aperte uma tecla sinalizadora de final (uma FLAG). O nosso FLAG será a letra 'X'. Repare que tivemos que usar dois `scanf()` dentro do `for`. Um busca o caractere que foi digitado e o outro busca o outro caractere digitado na sequência, que é o caractere correspondente ao <ENTER>.

```
#include <stdio.h>

main()
{
    int count;
    char ch;
    printf(" Digite uma letra - <X para sair> ");
    for (count=1; ;count++)
    {
        scanf("%c", &ch);
        if (ch == 'X') break;
        printf("\nLetra: %c \n", ch);
        scanf("%c", &ch);
    }
}
```

O Loop Sem Conteúdo

Loop sem conteúdo é aquele no qual se omite a declaração. Sua forma geral é (atenção ao ponto e vírgula!):

```
for (inicialização;condição;incremento);
```

Uma das aplicações desta estrutura é gerar tempos de espera. O programa abaixo faz isso.

```
#include <stdio.h>

main()
{
    long int i;
    printf("\a");
    for (i=0; i<10000000; i++); /* Imprime o caracter de alerta (um beep) */
    printf("\a"); /* Espera 10.000.000 de iteracoes */
    printf("\a"); /* Imprime outro caracter de alerta */
}
```

É válido lembrar que gerar “tempo de espera” com loop sem conteúdo, como o programa exibido acima, não é uma forma precisa de executar essa tarefa. Embora útil e eficaz em microprocessadores de baixa frequência, um processador moderno é capaz de executar um loop com bilhões de iterações numa fração de tempo baixíssima. Ademais, basta mudar o modelo do processador que certamente mudará também a duração desse procedimento. Quando se deseja efetuar uma contagem de tempo real, é recomendado que se faça uso de funções específicas para isso, presentes em bibliotecas especializadas.

4.3.2 while

O segundo laço disponível em C é o laço `while`. A sua forma geral é

```
while (condição) comando;
```

onde *comando* é um comando vazio, um comando simples ou um bloco de comandos. A condição pode ser qualquer expressão, e verdadeiro é qualquer valor não-zero. O laço se repete quando a condição for verdadeira. Quando a condição for falsa, o controle do programa passa para a linha após o código do laço. Dizemos que o comando `while` é um **loop com teste no início**.

Por analogia, o comando `while` seria equivalente a:

```
if (condição)
{
    comando;
    "Volte para o comando if"
}
```

Podemos ver que o comando `while` testa uma *condição*. Se esta for verdadeira o comando é executado e faz-se o teste novamente, e assim por diante. Assim como no caso do `for`, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir os comandos e fazer um loop sem conteúdo.

Vamos ver um exemplo do uso do `while`. O programa abaixo é executado enquanto `i` for menor que 100.

```
#include <stdio.h>

main()
{
    int i = 0;
    while (i < 100)
    {
        printf(" %d", i);
        i++;
    }
}
```

O programa abaixo espera o usuário digitar a tecla 'q' e só depois finaliza:

```
#include <stdio.h>

main()
{
    char ch;
    ch = '\0';
    while (ch != 'q')
    {
        scanf("%c", &ch);
    }
}
```

4.3.3 do-while

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

```
do {
    comando;
} while(condicao);
```

Embora as chaves não sejam necessárias quando apenas um comando esta presente, elas são geralmente usadas para evitar confusão (para você, não para o compilador) com o `while`. O ponto-e-vírgula final é

obrigatório. O laço `do-while` repete até que a *condição* se torne falsa. Dizemos que o comando `do-while` é um **loop com teste no final**.

Vamos, como anteriormente, ver o funcionamento do comando `do-while` “por dentro”:

```
comando;
if (condição) "Volta para a declaração"
```

Vemos, pela análise do bloco acima, que o comando `do-while` executa o *comando*, testa a *condição* e, se esta for verdadeira, volta para o *comando*.

O seguinte laço `do-while` lerá números do teclado até que encontre um número menor ou igual a 100.

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

Um dos usos da estrutura `do-while` é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado abaixo:

```
#include <stdio.h>

main()
{
    int i;
    do {
        printf("\n\nEscolha a fruta pelo numero:\n\n");
        printf("\t(1)...Mamão\n");
        printf("\t(2)...Abacaxi\n");
        printf("\t(3)...Laranja\n");
        scanf("%d", &i);
    } while ( i<1 || i>3);

    switch (i)
    {
        case 1:
            printf("\t\tVoce escolheu Mamão.\n");
            break;
        case 2:
            printf("\t\tVoce escolheu Abacaxi.\n");
            break;
        case 3:
            printf("\t\tVoce escolheu Laranja.\n");
            break;
    }
}
```

4.3.4 UMA DISCUSSÃO SOBRE OS TRÊS COMANDOS DE ITERAÇÃO

Vamos ver um exemplo do uso do `while`. O programa abaixo é executado enquanto `i` for menor que 100.

```
#include <stdio.h>

main()
{
    int i = 0;
    while (i < 100)
    {
        printf(" %d", i);
        i++;
    }
}
```

Veja como ele seria implementado mais naturalmente com um `for`:

```
#include <stdio.h>

main()
{
    int i;
    for (i=0; i<100; i++) printf(" %d", i);
}
```

Pode-se perceber, com esses dois exemplos, porque o comando `for` mostrou-se mais apropriado (gerou menos código). Este é um clássico caso de *loop* controlado por uma variável que atua como um *contador*. Sempre, nesses casos, será necessário inicializar o contador e incrementá-lo a cada iteração. O comando `for` possui, naturalmente, a capacidade de fazer tudo isso. Ao contrário do `while`, que, neste caso, obriga o programador a incluir essas sentenças “manualmente”.

Com a experiência como programador, em pouco tempo você será capaz de enxergar facilmente qual é o comando de iteração mais adequado a cada situação. Porém, de maneira geral, poderíamos dizer o seguinte a respeito de cada comando:

- `for`: Usado principalmente em situações envolvendo contadores, ou índices (de matrizes, por exemplo, como veremos no Capítulo 5). Em todos esses casos, o código escrito com o `for` será mais simples (ou *elegante*, como se diz em programação).
- `while`: Aplicável em situações diversas, diferente das situações específicas onde o comando `for` se aplica. Visto que a condição de execução do loop é testada no início, quaisquer variáveis envolvidas precisam estar inicializadas adequadamente, antes do loop.
- `do-while`: A grande utilidade deste comando está presente nos casos em que a(s) variável(eis) envolva(s) na condição do loop ainda não tem seus valores definidos na primeira iteração. Um caso muito típico é quando você está criando um menu e solicita ao usuário que escolha uma opção. Caso a opção escolhida seja inválida, você repetiria o menu, forçando-o a digitar uma opção válida. Nestes casos, a variável que armazena a opção que o usuário escolheu só é conhecida após executada a primeira iteração. Portanto, o teste (validade da opção), deve ser feito ao *final* do loop. Visto que o comando `do-while` garante que pelo menos uma iteração será executada, neste caso, é um comando bastante apropriado.

4.4 COMANDOS DE DESVIO

C tem quatro comandos que realizam um desvio incondicional³: `return`, `goto`, `break` e `continue`. Os dois primeiros serão vistos em momento apropriado, no futuro. Nesta seção, abordaremos o `break` e `continue`. Estes dois comandos podem usados em conjunto com qualquer dos comandos de laço. Como discutido anteriormente neste capítulo, você também pode usar o `break` com `switch`.

4.4.1 break

O comando `break` tem dois usos. Você pode usá-lo para terminar um case em um comando `switch` (abordado anteriormente na seção sobre o `switch`, neste capítulo). Você também pode usá-lo para forçar uma terminação imediata de um laço (`for`, `while` ou `do-while`), evitando o teste condicional normal do laço.

³ Ou seja, o programa “salta” de um ponto a outro do código incondicionalmente, sem que qualquer condição seja verificada. É uma “ordem”, não estando sujeita a nenhum teste prévio.

O `break` faz com que a execução do programa continue na primeira linha seguinte ao loop ou bloco que está sendo interrompido.

Observe que um `break` causará uma saída somente do laço mais interno. Por exemplo:

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

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o `break` é encontrado, o controle é devolvido para o laço `for` externo.

Outra observação é o fato de que um `break` usado dentro de um comando `switch` somente afetará esse `switch`. Ele não afeta qualquer outro loop que por ventura contenha o `switch`.

4.4.2 continue

O comando `continue` trabalha de uma forma um pouco parecida com a do comando `break`. Em vez de forçar a terminação, porém, `continue` força que ocorra a próxima iteração do laço, pulando qualquer código intermediário. Para o laço `for`, `continue` faz com que o teste condicional e a porção de incremento do laço sejam executados. Para os laços `while` e `do-while`, o controle do programa passa para o teste condicional.

Observe o exemplo abaixo:

```
#include <stdio.h>

main()
{
    int i;
    for (i=-10; i<=10; i++)
    {
        if (i == 0) continue;
        printf("%f\n", 1.0/i);
    }
}
```

No exemplo acima, são exibidos na tela o inverso dos números inteiros, de -10 a 10. Contudo, como não existe o inverso de zero, ou seja, $1/0$, existe um teste para saber se `i` é igual a 0. Caso seja verdadeiro, o comando `continue` força o loop a ir para a próxima iteração.

O programa abaixo exemplifica o uso do `continue`:

```
#include <stdio.h>

main()
{
    int opcao;
    do
    {
        printf("\n\n Escolha uma opcao entre 1 e 5: ");
        scanf("%d", &opcao);
        if ((opcao > 5) || (opcao < 1)) continue;
        switch (opcao)
        {
            case 1:
                printf("\n --> Primeira opcao..");
                break;
        }
    }
}
```

```

    case 2:
        printf("\n --> Segunda opcao..");
        break;
    case 3:
        printf("\n --> Terceira opcao..");
        break;
    case 4:
        printf("\n --> Quarta opcao..");
        break;
    case 5:
        printf("\n --> Abandonando..");
        break;
    }
} while (opcao != 5);
}

```

O programa acima ilustra uma aplicação simples para o `continue`. Ele recebe uma opção do usuário. Se esta opção for inválida, o `continue` faz com que o fluxo seja desviado para o final do `do-while`, que testará a condição e voltará ao início do loop. Caso a opção escolhida seja válida o programa segue normalmente. Observe também o uso do loop `do-while` é mais adequado que o `while`, neste caso, uma vez que a variável usada no teste condicional (`opcao`), ainda não seu valor definido ao adentrar ao loop pela primeira vez.

Cap. 5: VETORES, MATRIZES E STRINGS

Uma matriz é uma coleção de variáveis do mesmo tipo (homogênea) que é referenciada por um nome comum. Um elemento específico em uma matriz é acessado por meio de um índice. Em C, todas as matrizes consistem em posições contíguas (lado a lado) na memória. O endereço mais baixo corresponde ao primeiro elemento e o mais alto, ao último elemento. Matrizes podem ter de uma a várias dimensões. A matriz mais comum em C é a de char (chamada de string), que é simplesmente uma matriz de caracteres terminada por um caractere nulo (`'\0'`). Essa abordagem a strings dá a C maior poder e eficiência que as outras linguagens (Apesar de um pouco menos de praticidade).

Em C, matrizes e ponteiros estão intimamente relacionados; uma discussão sobre um deles normalmente refere-se ao outro. Este capítulo focaliza matrizes, enquanto o Capítulo 8 examina mais profundamente os ponteiros. Ambos os capítulos, juntos, completam o entendimento dessas construções importantes de C.

Um vetor é um caso especial de matrizes. Um vetor nada mais é que uma matriz unidimensional. Normalmente usa-se o termo *vetor*, para o caso de matrizes com apenas uma dimensão, e reserva-se o termo *matriz* para indicar duas ou mais dimensões.

5.1 VETORES

Conforme mencionado, vetores nada mais são que matrizes unidimensionais. Vetores são estruturas de dados muito utilizadas. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizadas por terem todos os elementos pertencentes ao mesmo tipo de dado. Em outras palavras, uma vez declarado que um vetor/matriz é do tipo `int`, ele não admitirá a inserção de valores do tipo `float`, por exemplo. Dizemos, portanto, que vetores/matrizes são estruturas de dados *homogêneas*.

Para se declarar um vetor podemos utilizar a seguinte forma geral:

```
tipo_da_variável nome_da_variável[tamanho];
```

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo[20];
```

o C irá reservar $4 \times 20 = 80$ bytes. Estes bytes são reservados de maneira contígua. Na linguagem C a numeração começa *sempre em zero*. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```
exemplo[0]
exemplo[1]
.
.
.
exemplo[19]
```

Mas ninguém o impede de escrever:

```
exemplo[30]
exemplo[103]
```

Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que *você* deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobre-escritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

```
#include <stdio.h>

main()
{
    int num[100]; /* Declara um vetor de inteiros de 100 posicoes */
    int count = 0;
    int totalnums;

    do
    {
        printf("\nEntre com um numero (-999 p/ terminar): ");
        scanf("%d", &num[count]);
        count++;
    } while (num[count-1] != -999);

    totalnums = count-1;

    printf("\n\n\t Os números que você digitou foram:\n\n");
    for (count=0; count < totalnums; count++)
        printf (" %d", num[count]);
}
```

No exemplo acima, o inteiro `count` é inicializado com 0. O programa pede pela entrada de números até que o usuário entre com a *flag* -999. Os números são armazenados no vetor `num`. A cada número armazenado, o contador do vetor é incrementado para na próxima iteração escrever na próxima posição do vetor. Quando o usuário digita a *flag*, o programa abandona o primeiro loop e armazena o total de números gravados. Por fim, todos os números são impressos. É bom lembrar aqui que nenhuma restrição é feita quanto a quantidade de números digitados. Se o usuário digitar mais de 100 números, o programa tentará ler normalmente, mas o programa os escreverá em uma parte não alocada de memória, pois o espaço alocado foi para somente 100 inteiros. Isto pode resultar nos mais variados erros no instante da execução do programa.

5.2 STRINGS

Strings são vetores de `char`. Nada mais e nada menos. As strings são o uso mais comum para os vetores. Devemos apenas ficar atentos para o fato de que as strings têm o seu último elemento como um `'\0'`. A declaração geral para uma string é:

```
char nome_da_string[tamanho];
```

Devemos lembrar que o tamanho da string deve incluir o `'\0'` final. A biblioteca padrão do C possui diversas funções que manipulam strings adequadamente. Estas funções são úteis pois não se pode, por exemplo, igualar duas strings:

```
string1 = string2; /* NÃO faça isto */
```

Fazer isto é um desastre. Somente com um pleno entendimento de ponteiros (Capítulo 8) é que é possível compreender o porquê disto. As strings devem ser igualadas elemento a elemento.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com `'\0'` (isto é, o número inteiro 0). Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):


```
#include <stdio.h>

main()
{
    int count;
    char str1[100], str2[100];
    .... /* Aqui o programa le str1 que será copiada para str2 */
    for (count=0; str1[count]; count++)
        str2[count] = str1[count];
    str2[count]='\0';
    .... /* Aqui o programa continua */
}
```

A condição no loop `for` acima é baseada no fato de que a string que está sendo copiada termina em `'\0'`. Quando o elemento encontrado em `str1[count]` é o `'\0'`, o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

Vamos ver agora algumas funções básicas para manipulação de strings.

5.2.1 gets()

A função `gets()` lê uma string do teclado. Sua forma geral é:

```
gets(nome_da_string);
```

O programa abaixo demonstra o funcionamento da função `gets()`:

```
#include <stdio.h>

main()
{
    char string[100];
    printf("Digite o seu nome: ");
    gets(string);
    printf("\n\n Ola %s \n\n", string);
}
```

Repare que é válido passar para a função `printf()` o nome da string. Você verá mais adiante porque isto é válido. Como o primeiro argumento da função `printf()` é uma string também é válido fazer:

```
printf(string);
```

isto simplesmente imprimirá a string.

5.2.2 strcpy()

A função `strcpy()` copia a string-origem para a string-destino. Seu funcionamento é semelhante ao da rotina apresentada na seção anterior. As funções apresentadas nestas seções estão no arquivo cabeçalho `string.h`. Sua forma geral é:

```
strcpy(string_destino, string_origem);
```

A seguir apresentamos um exemplo de uso da função `strcpy()`:

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

main ()
{
    char str1[100], str2[100], str3[100];
```

```

printf("Entre com uma string: ");
gets(str1);
strcpy(str2, str1);           /* Copia str1 em str2 */
strcpy(str3, "Voce digitou a string "); /* Copia "Voce digitou a string" em str3 */
printf("\n\n%s%s\n\n", str3, str2);
}

```

5.2.3 strcat()

A função `strcat()` concatena (ou seja, une/junta) duas strings. Ela tem a seguinte forma geral:

```
strcat(string_destino, string_origem);
```

A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Um exemplo:

```

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

main()
{
    char str1[100], str2[100];
    printf("Entre com uma string: ");
    gets(str1);
    strcpy(str2, "Voce digitou a string ");
    strcat(str2, str1); /* str2 armazenará "Voce digitou a string " + conteúdo de str1 */
    printf("\n\n%s \n\n", str2);
}

```

5.2.4 strlen()

A função `strlen()` retorna o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser um a mais que o inteiro retornado por `strlen()`. Sua forma geral é:

```
strlen(string);
```

Um exemplo do seu uso:

```

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

main()
{
    int tamanho;
    char str[100];
    printf("Entre com uma string: ");
    gets(str);
    tamanho = strlen(str);
    printf("\n\nA string que voce digitou tem tamanho %d\n\n", tamanho);
}

```

5.2.5 strcmp()

A função `strcmp()` compara duas strings. Sua forma geral é:

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

Há três possíveis valores de retorno desta função:

- -1: Caso a *string1* anteceda, alfabeticamente, a *string2*.
- 0: Caso a *string1* seja idêntica a *string2*
- 1: Caso a *string1* suceda, alfabeticamente, a *string2*.

Em resumo, se as duas strings forem idênticas a função retorna zero. Se elas forem diferentes a função retorna um valor diferente de zero.

Um exemplo da sua utilização:

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

main()
{
    char str1[100], str2[100];
    int resultado;
    printf ("Entre com uma string: ");
    gets(str1);
    printf("\nEntre com outra string: ");
    gets(str2);

    resultado = strcmp(str1,str2);

    switch (resultado)
    {
        case -1:
            printf("\'%s\' vem antes de \''s\' \n\n", str1, str2);
            break;
        case 0:
            printf("\'%s\' eh identica a \''s\' \n\n", str1, str2);
            break;
        case 1:
            printf("\'%s\' vem depois de \''s\' \n\n", str1, str2);
    }
}
```

5.3 MATRIZES

5.3.1 MATRIZES BIDIMENSIONAIS

Já vimos como declarar matrizes unidimensionais (vetores). Contudo, C suporta matrizes multidimensionais. A forma mais simples de matriz multidimensional é a matriz bidimensional — uma matriz de matrizes unidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

```
tipo_da_variável nome_da_variável[altura][largura];
```

É muito importante ressaltar que, nesta declaração, o índice da esquerda indexa as *linhas* e o da direita indexa as *colunas*. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador.

Para declarar uma matriz bidimensional de inteiros `mat` de tamanho 10 x 20, ou seja, 10 linhas e 20 colunas, você escreveria

```
int mat[10][20];
```

Similarmente, caso deseje posteriormente armazenar o inteiro 7 na posição 1,2 da matriz `mat`, bastaria escrever

```
mat[1][2] = 7;
```

A convenção para linha x coluna no acesso a uma matriz bidimensional em C é similar ao adotado em Álgebra Linear, exceto pelo fato de que ambos os índices começam em 0 (zero), não em 1.

O exemplo seguinte carrega uma matriz bidimensional com os números de 1 a 12 e escreve-os linha por linha.

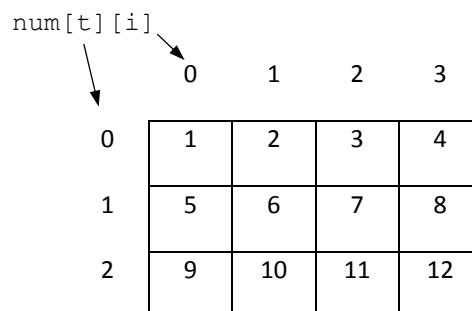
```
#include <stdio.h>

main()
{
    int t, i, num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; i<4; ++i)
            num[t][i] = (t*4)+i+1;

    /* agora imprima-os */
    for(t=0; t<3; ++t) {
        for(i=0; i<4; ++i)
            printf("%3d ", num[t][i]);
        printf("\n");
    }
}
```

Neste exemplo, `num[0][0]` tem o valor 1, `num[0][1]`, o valor 2, `num[0][2]`, o valor 3 e assim por diante. O valor de `num[2][3]` será 12. Você pode visualizar a matriz `num` como mostrada aqui:



	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

5.3.2 MATRIZES DE STRINGS

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se fizermos um vetor de strings estaremos fazendo uma lista de vetores. Esta estrutura de dados é uma matriz bidimensional de `char`. Podemos ver a forma geral de uma matriz de strings como sendo:

```
char nome_da_variável [num_de_strings][compr_das_strings];
```

Aí surge a pergunta: como acessar uma string individual? Fácil. É só usar apenas o primeiro índice. Então, para acessar uma determinada string faça:

```
nome_da_variável[índice]
```

Aqui está um exemplo de um programa que lê 5 strings e as exibe na tela:

```
#include <stdio.h>

main()
{
    char strings[5][100];
    int count;
```

```

for (count=0; count<5; count++)
{
    printf("\n\nDigite uma string: ");
    gets(strings[count]);
}
printf("\n\nAs strings que voce digitou foram:\n\n");
for (count=0; count<5; count++)
    printf("%s\n", strings[count]);
}

```

5.3.3 MATRIZES MULTIDIMENSIONAIS

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
```

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

5.3.4 INICIALIZAÇÃO

Podemos inicializar matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

```
tipo_da_variável nome_da_variável[tam1][tam2] ... [tamN] = {lista_de_valores};
```

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```

float vect[6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matr[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str[10] = { 'J', 'o', 'a', 'o', '\0' };
char str[10] = "Joao";
char str_vect[3][10] = { "Joao", "Maria", "Jose" };

```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde `matr` está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

5.3.4.1 INICIALIZAÇÃO SEM ESPECIFICAÇÃO DE TAMANHO

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho *a priori*. O compilador C vai, neste caso, verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```

char mess[] = "Linguagem C: flexibilidade e poder.";
int matr[][2] = {1, 2, 2, 4, 3, 6, 4, 8, 5, 10};

```

No primeiro exemplo, a string `mess` terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

5.4 INTRODUÇÃO À PESQUISA E ORDENAÇÃO

Algoritmos de busca e ordenação são ferramentas muito úteis à computação. São inúmeras as aplicações práticas em que, não basta apenas a existência do conjunto de dados (um vetor, por exemplo), a ordem dos elementos também é importante.

Um algoritmo de ordenação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem – em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica (segundo nosso alfabeto, por exemplo). Por exemplo, quando você clica no título de uma coluna, no Windows Explorer, os nomes dos arquivos e pastas ali presentes são imediatamente apresentados usando algum tipo de ordenação.

Há diversos algoritmos (métodos) conhecidos e muito estudados para ordenação de dados. Tanto que graduações em Computação contam com uma disciplina inteira voltada para seu estudo e aplicação. Tais métodos variam muitíssimo no nível de sofisticação (mais ou menos “elaborados” ou “inteligentes”) e complexidade computacional (custo, em unidade de tempo, necessário a sua execução completa). Em geral, métodos simples são os mais lentos e os mais sofisticados, mais rápidos (a depender de diversos fatores, como a disposição inicial dos dados, tamanho do conjunto de dados, etc.).

5.4.1 ALGORITMOS SIMPLES DE ORDENAÇÃO

A título de aplicação prática de vetores, três algoritmos considerados simples serão introduzidos nesta seção.

5.4.1.1 INSERTION SORT

Insertion sort, ou ordenação por inserção, é um simples algoritmo de ordenação, eficiente quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados. O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer.

Abaixo, uma implementação em C:

```
void insertionSort(int v[], int n)
{
    int i, j, chave;

    for(j=1; j<n; j++)
    {
        chave = v[j];
        i = j-1;
        while(i >= 0 && v[i] > chave)
        {
            v[i+1] = v[i];
            i--;
        }
        v[i+1] = chave;
    }
}
```

O algoritmo acima é dado de maneira “isolada” na forma de uma *função*, em C. Este assunto será introduzido no próximo capítulo. No entanto, a fim de entender o funcionamento básico do algoritmo (o qual você deverá tentar fazer sozinho, como exercício), assumo que o vetor v , bem como o seu tamanho n são dados (estes são, na verdade, os parâmetros da função, definidos em sua primeira linha). O código entre chaves contém apenas elementos que você já é capaz de entender (com algum esforço, claro).

5.4.1.2 SELECTION SORT

O *Selection Sort* (do inglês, “ordenação por seleção”) é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos.

Abaixo, uma implementação em C, apresentada sob o mesmo formato da Insertion Sort.

```
void selectionSort(int v[], int n)
{
    int i, j, min;
    for (i=0; i<(n-1); i++)
    {
        min = i;
        for (j=i+1; j<n; j++)
        {
            if(v[j] < v[min])
            {
                min = j;
            }
        }
        if (i != min)
        {
            int swap = v[i];
            v[i] = v[min];
            v[min] = swap;
        }
    }
}
```

5.4.1.3 BUBBLE SORT

O *Bubble Sort*, ou ordenação por flutuação (literalmente “por bolha”), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

Abaixo, uma implementação em C, apresentada sob o mesmo formato da Insertion Sort.

```
void bubbleSort(int v[], int n)
{
    int i;
    int j;
    int aux;
    int k = n-1 ;

    for(i=0; i<n; i++)
    {
        for(j=0; j<k; j++)
        {
            if(v[j] > v[j+1])
            {
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
        }
        k--;
    }
}
```

5.4.2 PESQUISA BINÁRIA

Existem várias razões para se ordenar uma sequência e uma delas é a possibilidade de acessar seus dados de modo mais eficiente, através de técnicas como a chamada *Busca Binária*. Ela parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca (divisão e conquista) comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor.

Abaixo, este interessante algoritmo é apresentado, em uma versão implementada em C.

```
int pesquisaBinaria(int v[], int chave , int n)
{
    int inf = 0;
    int sup = n-1;
    int meio;

    while (inf <= sup)
    {
        meio = (inf+sup)/2;
        if (chave == v[meio])
            return meio; // Retorna o índice do elemento encontrado.
        else if (chave < v[meio])
            sup = meio-1;
        else
            inf = meio+1;
    }
    return -1; // Indica elemento não encontrado
}
```

Cap. 6: INTRODUÇÃO À MODULARIZAÇÃO: SUBPROGRAMAS

A medida que você progressivamente for desenvolvendo algoritmos para problemas mais complexos, você perceberá que eles atingirão facilmente milhares de linhas, mesmo em projetos relativamente pequenos. O fato é que, este crescimento, se não for feito de maneira estruturada, com planejamento e diretrizes adequadas, poderá gerar um enorme caos. Isto quer dizer que, a manutenção ou inserção de novas funções se tornará tão impraticável que poderá forçar o projeto todo a recomençar, da estaca zero.

O que faremos neste capítulo é estudar algumas diretrizes que se mostrarão muito úteis daqui para frente em seus projetos. Introduziremos também alguns recursos fornecidos pela linguagem C para este fim.

6.1 PLANEJAMENTO

Há um velho ditado que diz que “uma imagem vale por mil palavras”. Por analogia, podemos dizer que “um minuto de planejamento vale por mil minutos de dor de cabeça na hora da manutenção (do software)”.

O que significa *planejar* na área de programação? Em primeiro lugar, significa romper com uma tradição terrível mantida por um grande número de programadores, que é fazer tudo diante do computador, na frente do teclado. Esse método pode ser um bom exercício para os dedos, mas não é nada eficiente em termos de produto final. Construir um programa de computador é como levantar um edifício. Não se começa juntando tijolos e argamassa, mas sim fazendo plantas e planos. Nesta seção, serão apresentadas algumas dicas que podem ajudar no seu planejamento.

6.1.1 SAIBA ONDE QUER CHEGAR

Antes de iniciar um programa qualquer certifique-se de que sabe o que se espera desse programa.

Pode parecer absurdo dizer isso, mas existe muita gente que escreve códigos sem ter uma ideia clara de onde quer chegar. E sem saber aonde se quer chegar, fica muito difícil chegar lá, não é mesmo?

6.1.2 FAÇA UMA ESPECIFICAÇÃO DO SEU PROGRAMA

Faça uma especificação completa e detalhada do seu programa antes de começar a codificar. Isso pode parecer tedioso quando comparado à emoção de ficar horas na frente do computador digitando como um lunático, mas vai ajudar em vários aspectos.

Há várias razões para especificar um sistema detalhadamente:

- a) Com uma boa especificação você não fica perdido, não fica em dúvida e nem tem de consultar o cliente (o solicitante do sistema) repetidas vezes durante o processo de desenvolvimentos e teste;
- b) Com uma boa especificação você sempre sabe aonde chegar e como chegar lá;
- c) Com uma boa especificação você tem um mecanismo para avaliar prazos e custos. Algumas métricas como a dos Pontos de Função baseiam-se na especificação do sistema;
- d) Com uma boa especificação você terá uma excelente ferramenta para a manutenção futura, como veremos adiante;
- e) Com uma boa especificação você tem um mecanismo para se proteger daquele cliente que, depois de combinar com você o preço do programa e o prazo de entrega, fica pedindo para acrescentar uma coisinha aqui e outra ali no sistema. Esse tipo de cliente quer sempre mais, mas quando você fala em extensão do prazo ou em aumento dos custos ele reclama, dizendo que só pediu umas “besteirinhas”... Prepare a especificação e mostre ao cliente, fazendo com que ele concorde em ater-

se a ela e deixando claro que o software será entregue daquele jeito e que se alguma necessidade nova surgir será implementada na versão seguinte. Sugere-se que isto seja posto no papel, mediante um contrato formal. Os clientes sempre dão um jeito de bagunçar tudo, se deixarmos.

6.1.3 PLANEJE SEU PROGRAMA PENSANDO EM VÁRIAS COISAS

Muita gente pensa que projetar um bom programa consiste apenas em planejar como ele vai se comportar, como ele vai funcionar. A dura realidade, porém, é que se devem levar em conta muitos outros fatores:

- a) Seu programa será fácil de manter? O trabalho de desenvolvimento é só um pedaço da tarefa. Para que um programa tenha um longo ciclo de vida e se torne economicamente interessante para os seus clientes, é necessário que ele seja de fácil manutenção. Para isso é necessário que haja uma boa especificação, uma boa documentação e uma boa organização. Finalmente, a organização abrange coisas como a nomenclatura dos fontes, a estrutura de diretórios onde eles são colocados e até a velha indentação do código, que é o (bom) hábito de colocar as linhas dentro de uma estrutura hierárquica de blocos dentro do programa.
- b) Seu programa será flexível? A boa estruturação de um programa deve permitir que ele seja alterado sem que haja necessidade de um esforço sobre-humano para isso. Um dos segredos para a isso é a modularização, ou seja, quebrar o programa em pequenas partes bem específicas (módulos), de modo que esses módulos possam ser substituídos por outros similares sem impacto no funcionamento ou na performance do programa como um todo. Esse assunto será abordado na Seção 6.2.
- c) Como seu programa vai se comportar quando não funcionar? Será que seu programa vai ter mensagens inteligentes e úteis nas situações de erro? Ou irá apenas travar ou finalizar? Um bom programa deve funcionar bem até quando não funcionar, ou seja, até quando encontrar situações de erro. Nesses casos ele deve ser comunicativo, explicando o erro ao usuário e oferecendo alternativas.

Essas são algumas (não todas) as coisas nas quais você deve pensar para fazer bons programas.

6.1.4 PESQUISE SEMPRE A MELHOR FORMA DE FAZER

Não basta fazer um programa funcionar. Tem de fazer ele funcionar bem, ter uma boa performance e ser estável. Isso exige de analistas e programadores uma constante pesquisa, buscando metodologias alternativas, novas tecnologias e novos conhecimentos.

A fim de exemplificar como pequenos detalhes podem impactar grandemente no funcionamento dos seus programas, observe o exemplo abaixo. Nele são apresentados três formas muito parecidas de encontrar todos os números primos menores que um certo valor, configurável a partir da variável `nMaximo`. As três “versões” estão apresentadas num único programa a fim de facilitar a comparação. Como não existe fórmula para se determinar um número primo, deve-se partir da definição: “Um número é primo quando ele é divisível somente por 1 e por ele mesmo”. O princípio das três versões é exatamente o mesmo, testar cada número, a partir de 2, a fim de descobrir se ele atende ao requisito, ou seja, se é primo ou não.

```
#include <stdio.h>
#include <time.h>
#include <math.h>

main()
{
    int i, n, nPrimos, ehPrimo, nMaximo = 100000;
    double tempo1, tempo2, tempo3;
    clock_t tInicio, tFim;

    // ----- Versão 1 -----
    printf("Numeros primos menores que %d - Versao 1\n", nMaximo);
```

```

nPrimos = 0;
tInicio = clock();
for (n=2; n<nMaximo; n++)
{
    ehPrimo = 1;
    for (i=2; i<=(n-1); i++) // Testando até n-1
        if ((n % i) == 0)
        {
            ehPrimo = 0;
            break;
        }
    if (ehPrimo)
    {
        nPrimos++;
        //printf("%d, ", n);
    }
}
tFim = clock();
printf("%d primos encontrados\n", nPrimos);
tempo1 = (double)(tFim-tInicio)/CLOCKS_PER_SEC;
printf("Tempo total: %f segundos (%.2f%% do tempo)\n\n", tempo1, (tempo1/tempo1)*100);

// ----- Versão 2 -----
printf("Numeros primos menores que %d - Versao 2\n", nMaximo);
nPrimos = 0;
tInicio = clock();
for (n=2; n<nMaximo; n++)
{
    ehPrimo = 1;
    for (i=2; i<=(n/2); i++) // Testando até n/2
        if ((n % i) == 0)
        {
            ehPrimo = 0;
            break;
        }
    if (ehPrimo)
    {
        nPrimos++;
        //printf("%d, ", n);
    }
}
tFim = clock();
printf("%d primos encontrados\n", nPrimos);
tempo2 = (double)(tFim-tInicio)/CLOCKS_PER_SEC;
printf("Tempo total: %f segundos (%.2f%% do tempo)\n\n", tempo2, (tempo2/tempo1)*100);

// ----- Versão 3 -----
printf("Numeros primos menores que %d - Versao 3\n", nMaximo);
nPrimos = 0;
tInicio = clock();
for (n=2; n<nMaximo; n++)
{
    ehPrimo = 1;
    for (i=2; i<=sqrt(n); i++) // Testando até sqrt(n)
        if ((n % i) == 0)
        {
            ehPrimo = 0;
            break;
        }
    if (ehPrimo)
    {
        nPrimos++;
        //printf("%d, ", n);
    }
}
tFim = clock();
printf("%d primos encontrados\n", nPrimos);
tempo3 = (double)(tFim-tInicio)/CLOCKS_PER_SEC;
printf("Tempo total: %f segundos (%.2f%% do tempo)\n\n", tempo3, (tempo3/tempo1)*100);
}

```

Ao ser executado num determinado sistema, obteve-se a seguinte saída:

```

Numeros primos menores que 100000 - Versao 1
9592 primos encontrados
Tempo total: 3.093000 segundos (100.00% do tempo)

Numeros primos menores que 100000 - Versao 2
9592 primos encontrados
Tempo total: 1.610000 segundos (52.05% do tempo)

Numeros primos menores que 100000 - Versao 3
9592 primos encontrados
Tempo total: 0.093000 segundos (3.01% do tempo)

```

Na questão da *eficácia*, as três versões são igualmente eficazes. Isto porque todas elas chegam ao mesmo resultado, encontram todos os 9592 números primos menores que 100.000. No entanto, no quesito *eficiência*, medido através do tempo de processamento necessário à tarefa, os resultados mudam significativamente. Avalie cuidadosamente as três versões e encontre o que mudou e principalmente por que isto pode ser mudado a fim de melhorar a performance do algoritmo, sem alterar em nada no seu resultado.

Como você pode ver, uma pequena alteração pode impactar drasticamente a performance de um programa! É por isso que vale a pena planejar e otimizar. Programadores precisam ter um conhecimento amplo. Devem procurar estudar bastante os assuntos que estão envolvidos nos programas que estão fazendo. Só assim poderão descobrir a melhor forma de fazer as coisas acontecerem. No entanto, vale lembrar que, em geral, a “melhor forma” não é a primeira a aparecer na sua mente. O importante é não desistir da busca por algoritmos cada vez melhores, cada vez mais “inteligentes”.

6.2 MODULARIZAÇÃO

À medida que vamos resolvendo problemas mais complexos, o tamanho dos nossos programas vai crescendo, assim, fica difícil acompanhar as funcionalidades dos trechos do programa.

A modularização é um processo que aborda os aspectos da decomposição de algoritmos em módulos. Módulo é um grupo de comandos, constituindo um trecho do algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo.

A maneira mais intuitiva de proceder a modularização de problemas é feita definindo-se um módulo principal de controle e módulos específicos para as funções do algoritmo.

6.2.1 BENEFÍCIOS DA MODULARIZAÇÃO

A divisão de um algoritmo em módulos traz benefícios como:

1. Facilita a detecção de erros, pois é em princípio simples verificar qual é o módulo responsável pelo erro.
2. É mais fácil testar os módulos individualmente do que o programa completo.
3. É mais fácil fazer a manutenção (correção de erros, melhoramentos, etc.) módulo por módulo do que no programa total. Além disso, a modularização diminui a probabilidade dessa manutenção provocar consequências desastrosas nos outros módulos do programa.
4. Permite o desenvolvimento independente dos módulos. Isto simplifica o trabalho em equipe, pois cada elemento, ou cada subequipe, tem a seu cargo apenas alguns módulos do programa.
5. Porventura a mais evidente vantagem da modularização em programação é a possibilidade de reutilização do código desenvolvido. Ou seja, um módulo independente pode vir a ser utilizado em outros algoritmos que requeiram o mesmo processamento por ele executado.

6.2.2 UNIDADES BÁSICAS

As unidades básicas dos programas estruturados são os procedimentos (procedures) e/ou funções (functions). Um procedimento (ou função) executa um certo trecho de código, recebendo ou não valores do programa que a chamou e retornando ou não valores para esse programa.

Em algumas linguagens, como o PASCAL, existem procedimentos e funções como entidades diferentes. Em PASCAL um procedimento nunca retorna valores, enquanto uma função sempre o faz. Já em C tal distinção não existe explicitamente. Essa é, aliás, a tendência das linguagens mais novas. Conforme veremos, em C podemos criar funções com ou sem algum valor de retorno (ou seja, um “resultado” do processamento da função).

Seja de que forma for, o que temos de entender inicialmente é que o nível de complexidade dos problemas que temos de resolver em programação tende a ser muito alto. Assim, é quase impossível escrever um trecho de código único que resolva um grande problema. A lógica envolvida seria muito complexa e intratável pela mente humana. Temos então de aprender a “quebrar” grandes problemas em problemas menores, em partes cuja complexidade seja controlável.

Em C, por ser uma linguagem que segue o paradigma estruturado, o conceito de módulo está intimamente ligado ao trecho de código delimitado por uma função. Um módulo, contudo, pode ser entendido de uma forma mais abrangente, englobando uma ou mais funções correlatas, bem como possíveis estruturas de dados definidas⁴.

A modularização, porém, não pode ser feita de forma aleatória. Não basta sair quebrando. Devemos aproveitar para fazer essa quebra de forma útil e interessante, tendo em vista a performance do programa como um todo e alguns outros critérios.

6.2.3 O PRINCÍPIO DA CAIXA PRETA

Em programação, o *Princípio da Caixa-Preta* nos diz que ao planejar bons módulos devemos conhecer apenas o input (entrada) e o output (saída) de cada um, ignorando completamente o funcionamento interno dos módulos. Em outras palavras, *abstraindo* o funcionamento interno do mesmo.

Assim, quando especificamos um módulo, não devemos estar preocupados, por exemplo, com o algoritmo que será usado na implementação, mas apenas com o que temos de fornecer ao módulos (input), ou seja, os parâmetros de entrada, e com o que podemos obter dele (output). O princípio da caixa-preta é extremamente valioso, pois ele torna nossos módulos independentes de implementação.

Você pode estar se perguntando qual a vantagem em termos módulos assim? Pois bem, tomemos um exemplo. Imagine que queremos escrever um sistema contábil que usa uma função X para calcular o imposto a pagar de uma lista de contribuintes da Receita Federal. É bem sabido que a Receita Federal muda as regras do Imposto de Renda todos os anos, de forma que a função X tem de ser alterada anualmente. Imagine a confusão que haveria em todo o sistema se outros módulos se baseassem no algoritmo interno da função X para realizarem suas tarefas. Cada vez que a função X fosse alterada, todos os demais módulos do sistema passariam a funcionar de modo diferente, com novos erros, etc.

Porém, se os arquitetos do sistema contábil tivessem usado o princípio da caixa-preta, não haveria problema. A função X receberia o CPF do contribuinte (input), pesquisaria os dados necessários para o cálculo e faria o cálculo desejado de acordo com as regras atuais (processing) e retornaria o valor do imposto a pagar (output).

Os módulos do sistema que usassem a função X saberiam apenas que têm de mandar o CPF quando a chamam e que ela retorna o valor do imposto a pagar. Conheceriam o input e o output, mas o processamento

⁴ A linguagem C permite que o programador defina novos tipos de dados por meio da criação de estruturas de dados mais complexas, formadas a partir dos tipos básicos ou mesmo de outras estruturas. Pormenores deste assunto serão abordados no futuro (Cap. 10).

(os detalhes de como os cálculos são feitos) seria transparente para eles. Assim, quando as regras da Receita Federal mudassem o funcionamento desses módulos não seria afetado pelas mudanças da função X.

Portanto, o uso do Princípio da Caixa-Preta é essencial para a facilidade de manutenção.

6.2.4 CRITÉRIOS ÚTEIS NA MODULARIZAÇÃO

6.2.4.1 REUSABILIDADE

Um módulo deve, idealmente, ser aproveitado em muitos lugares do programa. Isso significará um melhor aproveitamento do código, evitando a repetição de trechos de código, o que costuma dificultar a manutenção.

Tomemos um exemplo. Imagine que em diversos momentos da operação de um sistema, deseja-se validar os dígitos verificadores de um número de CPF qualquer que é fornecido. Caso o código de verificação seja escrito repetidamente em todos os lugares em que ele for necessário, vários problemas poderiam ocorrer:

- a) O esforço de escrever tudo novamente, diversas vezes;
- b) A possibilidade de, ao escrever tudo novamente, introduzir acidentalmente erros;
- c) Dificuldade de manutenção, pois se um dia a Receita Federal resolver mudar a fórmula de cálculo dos dígitos verificadores do CPF, seria necessário procurar em cada trecho do código os lugares em que aquele cálculo aparece e reescrevê-lo. Basta que um desses lugares seja acidentalmente esquecido para que um problema grande se instale.

Por isso é boa prática escrever essa rotina de verificação na forma de uma função e invocar essa função sempre que necessário, fornecendo a ela o CPF que se quer verificar e recebendo dela um informe sobre a validade ou não daquele CPF.

Em geral, quanto mais genérico é um código, maiores são as possibilidades de reuso do mesmo. No entanto, códigos muito genéricos (capazes de lidar com diversas situações) podem se tornar muito complexos, violando o princípio da baixa complexidade, mencionado a seguir. Estamos diante, portanto, de um *trade off* muito comum em tecnologia, que é a adoção de uma postura “intermediária” que seja satisfatória.

6.2.4.2 BAIXA COMPLEXIDADE

Cada módulo de um sistema deve ter um nível de complexidade aceitável. Imaginemos agora um módulo que, para poder executar suas funções, tem de invocar os serviços de uma centena de outros módulos. Controlar esse nível de complexidade é bastante difícil. Para reduzir a complexidade de um módulo devemos quebrá-lo em módulos menores e um bom critério para observar quando isso é necessário é ver o número de subordinados que ele tem.

É desejável que um módulo não tenha um número exagerado de submódulos a ele subordinados.

6.2.4.3 ACOPLAMENTO

Imagine que você acabou de comprar um maravilhoso Home Theater para seu apartamento. É um equipamento de alta tecnologia, com excelente qualidade de som e imagem, e você está imensamente satisfeito com a aquisição. Só que quando você vai instalar o aparelho, percebe que há apenas uma entrada de eletricidade para atender todos os módulos do aparelho, e que essa entrada está situada numa das caixas de som, sendo que a eletricidade é distribuída de lá para os outros módulos. Alguns meses depois de adquirido o aparelho, a referida caixa de som que distribui eletricidade para o Home Theater queima. E aí você começa a ver que o esquema de passar toda a eletricidade por um único lugar é problemático, pois agora você simplesmente não pode ligar o seu equipamento apenas porque uma das caixas queimou. Assim, vemos que a ligação desnecessária (elétrica) entre a caixa e os demais módulos é prejudicial. Uma caixa de som devia receber do equipamento apenas o sinal de som, e não devia enviar nada em retorno. Qualquer ligação adicional é desnecessária e prejudicial, comprometendo o funcionamento do sistema como um todo. Esse é um exemplo de mau acoplamento.

A ligação ideal entre dois módulos de um sistema é a menor possível, ou seja, os módulos devem trocar apenas e tão somente as informações necessárias, nada mais. Isso concorda com o princípio da caixa-preta, que nós já vimos.

Tomemos um exemplo mais específico. Imagine que um módulo necessita de outro que calcule o valor da soma dos itens de uma conta de restaurante. Então ele deve enviar para o outro apenas os números a serem somados, recebendo deste apenas o valor total. Isso seria o acoplamento ideal, conhecido como “acoplamento de dados”.

Agora imagine que além dos números a serem somados, o módulo chamador passa também o número do CPF do cliente que vai pagar a conta. Há aí vários riscos potenciais:

1. Perda de performance, já que o fluxo de informações entre os módulos implica em movimento dos registradores e manipulação de memória, coisas que consomem tempo;
2. Possibilidade de alteração da informação desnecessária, o que causaria erros mais adiante, muito difíceis de serem localizados;
3. Confusão de raciocínio, pois o módulo teria um desnecessário acréscimo da lógica, sem qualquer benefício adicional; e,
4. Dificuldade de manutenção, pois um programador de manutenção iria perder horas tentando entender a função do CPF dentro daquele contexto.

Assim, o acoplamento entre dois módulos deve ser apenas o mínimo necessário para o cumprimento da função de ambos. O que passar disso pode vir a ser bastante prejudicial.

6.2.4.4 COESÃO

Enquanto o acoplamento é relacionado com dois ou mais módulos, a coesão é um critério de um só módulo. Um módulo é dito coeso quando ele realiza apenas uma tarefa. Uma das melhores maneiras de avaliar a coesão de um módulo é observar o nome dele.

Vejamos um exemplo. Suponha um módulo chamado `CalculaSomaEImprime`. Como se pode ver, esse módulo realiza duas tarefas. Para melhorar a coesão, seria bom dividi-lo em dois, sendo um para calcular a soma e outro para imprimir o resultado. Com isso tornaríamos as duas operações independentes e reduziríamos a complexidade. Um módulo como esse é pouco reutilizável, já que só pode ser usado por outro módulo que precise da soma seguida da impressão. Já os módulos `CalculaSoma` e `ImprimeValorSoma`, que resultariam da quebra dele, seriam melhor aproveitados.

6.2.4.5 RESUMINDO OS CRITÉRIOS...

Um bom módulo, portanto, tem de ser projetado tendo em mente o *Princípio da Caixa-Preta*, deve ser o mais reusável possível, apresentar baixa complexidade (chamar poucos “auxiliares”), acoplar-se com outros módulos apenas através dos dados estritamente necessários e ser coeso, executando apenas uma tarefa.

Por mais que existam critérios para a divisão em módulos, programar é tido por muitos como “arte”. Assim sendo, da mesma forma que a arte tem seus princípios, mas é o estilo/personalidade do artista que acaba se destacando, em programação cada programador acaba criando seu próprio “estilo”. E, com o tempo, estudos e experiência acumulados, pode-se esperar que a qualidade do seu código melhore muito.

6.3 FUNÇÕES EM C: ASPECTOS BÁSICOS

Funções, conforme mencionado previamente, são os principais “blocos de construção” de qualquer módulo escrito em C. Há muito que dizer a respeito da definição e uso de funções em C. Essa complexidade, no entanto, será dividida em duas partes. Inicialmente, nesta seção, serão vistos apenas os aspectos básicos, suficientes à aplicação da teoria de modularização estudada neste capítulo. Somente depois de consolidados

os conceitos básicos da divisão de um programa/sistema em módulos é que os aspectos avançados serão vistos, no Capítulo 9.

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O uso de funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fundo, de várias funções colocadas juntas.

Apresentamos aqui a forma geral de uma função:

```
tipo_de_retorno nome_da_função (lista_de_argumentos)
{
    código_da_função
}
```

Cabem aqui algumas observações.

- Quanto ao *tipo_de_retorno*: Quando uma função não retorna nada, é recomendável dizer isso explicitamente, adotando o tipo de retorno como `void`. Caso o tipo de retorno seja simplesmente omitido, o compilador adotará, por padrão, o retorno como sendo do tipo `int`. Caso isso aconteça e o comando `return` não é encontrado retornando algum valor inteiro, então o valor de retorno da função será tecnicamente indefinido (cada compilador pode adotar o valor que quiser). Isto não vem a ser um problema caso você não especifique um tipo de retorno (`int` é então adotado por padrão) e esta função não é usada em nenhuma expressão (ou seja, esse valor de retorno indefinido não é usado em lugar algum). Quando o tipo de retorno é `void`, o compilador impede que essa função seja usada em qualquer expressão.
- Quanto a *lista_de_argumentos*: Caso a função não receba qualquer argumento e o tipo `void` seja colocado na *lista_de_argumentos*, o compilador impedirá que a função seja chamada com algum parâmetro. Caso nada seja especificado, ou seja, os parênteses estejam vazios, o comportamento será “quase” o mesmo. A diferença é que se a função for chamada acidentalmente com algum argumento, o compilador não reclamará disso.

Abaixo, um exemplo muito simples de uma função:

```
#include <stdio.h>

void Mensagem()
{
    printf("Hello, World!");
}

void main()
{
    printf("Exemplo de Funcao\n");
    Mensagem();
}
```

Este programa simplesmente imprimirá duas mensagens na tela. O que ele faz é definir uma função `Mensagem()` que coloca string `Hello, World!` na tela e não retorna nada (`void`). Esta função é chamada a partir de `main()`, que, como já vimos, também é uma função. A diferença fundamental entre `main` e as demais funções do problema é que `main` é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

6.3.1 ARGUMENTOS

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos *parâmetros* para a função. Já vimos funções com argumentos. As funções `printf()` e `scanf()` são funções que recebem argumentos. Vamos ver outro exemplo simples de função com argumentos:

```
#include <stdio.h>

void square(int x) /* Calcula e imprime o quadrado de x */
{
    printf("O quadrado eh %d\n\n",x*x);
}

void main()
{
    int num;
    printf("Entre com um numero: ");
    scanf("%d", &num);
    printf("\n\n");
    square(num);
}
```

Na definição de `square()` dizemos que a função receberá um argumento inteiro `x`. Quando fazemos a chamada à função, o inteiro `num` é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável `num`, ao ser passada como argumento para `square()` é copiada para a variável `x`. Dentro de `square()` trabalha-se apenas com `x`. Se mudarmos o valor de `x` dentro de `square()` o valor de `num` na função `main()` permanece inalterado.

Vamos dar um exemplo de função de mais de uma variável. Repare que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Note, também, que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
#include <stdio.h>

void Multiplica(float a, float b, float c) /* Multiplica 3 números */
{
    printf("%f", a*b*c);
}

void main()
{
    float x, y;
    x = 23.5;
    y = 12.9;
    Multiplica(x, y, 3.87);
}
```

6.3.2 RETORNANDO VALORES

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui não retornavam nada. Podemos especificar um tipo de retorno indicando-o antes do nome da função. Mas para dizer ao C o que vamos retornar precisamos da palavra reservada `return`. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação. Veja:

```
#include <stdio.h>

int produto(int x, int y)
{
    return (x*y);
}

void main ()
{
    int resultado;
    resultado = produto(12, 7);
    printf("O resultado eh: %d\n", resultado);
}
```

Veja que, como `produto()` retorna o valor de 12 multiplicado por 7, este valor pode ser usado em uma expressão qualquer. No programa fizemos a atribuição deste resultado à variável `resultado`, que posteriormente foi impressa usando o `printf()`.

Mais um exemplo de função, que agora recebe dois `float` e também retorna um `float`:

```
#include <stdio.h>

float produto(float x, float y)
{
    return (x*y);
}

void main()
{
    float resultado;
    resultado = produto(45.2, 0.0067);
    printf("O resultado eh: %f\n", resultado);
}
```

6.4 ESCOPO DE VARIÁVEIS

As variáveis em C podem ser declaradas em três lugares básicos: dentro de funções, na definição dos parâmetros das funções e fora de todas as funções. Estas são variáveis locais, parâmetros formais e variáveis globais, respectivamente. Há diversas particularidades importantes a serem observados em cada um dos casos, os quais serão feitos nesta seção.

6.4.1 VARIÁVEIS LOCAIS

Variáveis que são declaradas dentro de uma função são chamadas de variáveis locais. Variáveis locais só podem ser referenciadas por comandos que estão dentro do bloco no qual as variáveis foram declaradas. Em outras palavras, variáveis locais não são reconhecidas fora de seu próprio bloco de código. Lembre-se, um bloco de código inicia-se em abre-chaves `{}` e termina em fecha-chaves `}`.

Variáveis locais existem apenas enquanto o bloco de código em que foram declaradas está sendo executado. Ou seja, uma variável local é criada na entrada de seu bloco e destruída na saída.

O bloco de código mais comum em que as variáveis locais são declaradas é a função. Por exemplo, considere as seguintes funções:

```
void func1(void)
{
    int x;
    x = 10;
}
```

```
void func2(void)
{
    int x;
    x = -199;
}
```

A variável inteira `x` é declarada duas vezes, uma vez em `func1()` e outra em `func2()`. O `x` em `func1()` não tem nenhuma relação ou correspondência com o `x` em `func2()`. A razão para isso é que cada `x` é reconhecido apenas pelo código que está dentro do mesmo bloco da declaração de variável.

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

```
void f(void)
{
    int t;
    scanf("%d", &t);
    if(t == 1)
    {
        char s[80];      /* isto é criado apenas na entrada deste bloco */
        printf("Entre com o nome: ");
        gets(s);

        /* faz alguma coisa */
    }
}
```

Aqui, a variável local `s` é criada na entrada do bloco de código `if` e destruída na saída. Além disso, `s` é reconhecida apenas dentro do bloco `if` e não pode ser referenciada em qualquer outro lugar — mesmo nas outras partes da função que a contém.

A principal vantagem em declarar uma variável local dentro de um bloco condicional é que a memória para ela só será alocada se necessário. Isso acontece porque variáveis locais não existirão até que o bloco em que elas são declaradas seja iniciado. Você deve preocupar-se com isso quando estiver produzindo código para controladores dedicados (como um controlador de porta de garagem, que responde a um código de segurança digital) em que a memória RAM é escassa, por exemplo.

Declarar variáveis dentro do bloco de código que as utiliza também ajuda a evitar efeitos colaterais indesejados. Como a variável não existe fora do bloco em que é declarada, ela não pode ser acidentalmente alterada. Porém, quando cada função realiza uma tarefa lógica bem definida, você não precisa “proteger” variáveis dentro de uma função do código que constitui a função. Isso explica por que as variáveis usadas por uma função são geralmente declaradas no início da função.

```
/* Esta função está errada. */

void f(void)
{
    int i;
    i = 10;
    int j;    /* esta linha irá provocar um erro */
    j = 20;
}
```

Porém, se você tivesse declarado `j` dentro de seu próprio bloco de código ou antes do comando `i = 10`, a função teria sido aceita. Por exemplo, as duas versões mostradas aqui estão sintaticamente corretas:

```

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

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

```

Como ponto interessante, a restrição que exige que todas as variáveis sejam declaradas no início do bloco foi removida em C++. Em C++, as variáveis podem ser declaradas em qualquer ponto dentro de um bloco.

Como todas as variáveis locais são criadas e destruídas a cada entrada e saída do bloco em que elas são declaradas, seu conteúdo é perdido quando o bloco deixa de ser executado. É especialmente importante lembrar disso ao chamar uma função. Quando uma função é chamada, suas variáveis locais são criadas e, ao retornar, elas são destruídas. Isso significa que as variáveis locais não podem reter seus valores entre chamadas. (No entanto, você pode ordenar ao compilador que retenha seus valores usando o modificador `static`.)

A menos que especificado de outra forma, variáveis locais são armazenadas na pilha. O fato de a pilha ser uma região de memória dinâmica e mutável explica por que variáveis locais não podem, em geral, reter seus valores entre chamadas de funções.

6.4.2 PARÂMETROS FORMAIS

Se uma função usa argumentos, ela deve declarar variáveis que receberão os valores dos argumentos. Essas variáveis são denominadas *parâmetros formais* da função. Elas se comportam como qualquer outra variável local dentro da função. Suas declarações ocorrem depois do nome da função e dentro dos parênteses.

Você deve informar a C que tipo de variáveis são os parâmetros formais, declarando-as conforme a regra de declaração de funções. Uma vez feito isso, elas podem ser usadas dentro da função como variáveis locais normais. Tenha sempre em mente que, como variáveis locais, elas também são dinâmicas e são destruídas na saída da função.

Você deve ter certeza de que os parâmetros formais que estão declarados são do mesmo tipo dos argumentos que você utiliza para chamar a função. Se há uma discordância de tipos, resultados inesperados podem ocorrer. Ao contrário de muitas outras linguagens, C geralmente fará alguma coisa, inclusive em circunstâncias não usuais, mesmo que não seja o que você quer. Há poucos erros em tempo de execução e nenhuma verificação de limites. Como programador, você deve ter certeza de que erros de incongruência de tipo não ocorrerão.

Analogamente às variáveis locais, você pode fazer atribuições a parâmetros formais de uma função ou usá-los em qualquer expressão permitida em C. Embora essas variáveis recebam o valor dos argumentos passados para a função, elas podem ser usadas como qualquer outra variável local.

6.4.3 VARIÁVEIS GLOBAIS

Ao contrario das variáveis locais, as variáveis globais são reconhecidas pelo programa inteiro e podem ser usadas por qualquer pedaço de código. Além disso, elas guardam seus valores durante toda a execução do programa. Você cria variáveis globais declarando-as fora de qualquer função. Elas podem ser acessadas por qualquer expressão independentemente de qual bloco de código contém a expressão.

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

```
#include <stdio.h>

int count; /* count é global */

void func2(void)
{
    int count;
    for (count=1; count<10; count++)
        printf(".");
}

void func1(void)
{
    int temp;
    temp = count;
    func2();
    printf("count eh %d", count); /* imprimirá 100 */
}

void main(void)
{
    count = 100;
    func1();
}
```

Olhe atentamente para esse programa. Observe que, apesar de nem `main()` nem `func1()` terem declarado a variável `count`, ambas podem usá-la. A função `func2()`, porém, declarou uma variável local chamada `count`. Quando `func2()` referencia `count`, ela referencia apenas sua variável local, não a variável global. Se uma variável global e uma variável local possuem o mesmo nome, todas as referências ao nome da variável dentro do bloco onde a variável local foi declarada dizem respeito a variável local e não tem efeito algum sobre a variável global. Pode ser conveniente, mas esquecer-se disso poderá fazer com que seu programa seja executado estranhamente, embora pareça correto.

O armazenamento de variáveis globais encontra-se em uma região fixa da memória, separada para esse propósito pelo compilador C. Variáveis globais são úteis quando o mesmo dado é usado em muitas funções em seu programa. No entanto, você deve evitar usar variáveis globais desnecessárias. Elas ocupam memória durante todo o tempo em que seu programa está executando, não apenas quando são necessárias. Além disso, usar uma variável global onde uma variável local poderia ser usada torna uma função menos geral, porque ela conta com alguma coisa que deve ser definida fora dela. Finalmente, usar um grande número de variáveis globais pode levar a erros no programa por causa de desconhecidos — e indesejáveis — efeitos colaterais. Isso pode ser evidenciado em linguagens em que todas as variáveis são globais. Um problema maior no desenvolvimento de grandes projetos é a mudança acidental do valor de uma variável porque ela é usada em algum outro lugar do programa. Isso pode acontecer em C se você usar variáveis globais demais em seus programas.

Uma das principais razões para uma linguagem estruturada é a compartimentalização ou separação de código e dados. Em C, esse isolamento é conseguido pelo uso de variáveis locais e funções. Por exemplo, o código abaixo mostra duas maneiras de escrever `multiplica()` — uma função simples que calcula o produto de dois inteiros.

Geral

```
int multiplica(int x, int y)
{
    return (x*y);
}
```

Específica

```
int x, y;
int multiplica(void)
{
    return (x*y);
}
```

Ambas as funções retornam o produto das variáveis `x` e `y`. Contudo, a versão generalizada, ou parametrizada, pode ser usada para retornar o produto de quaisquer dois inteiros, enquanto a versão específica só pode ser usada para encontrar o produto das variáveis globais `x` e `y`.

PARTE II

Uma vez construída uma base sólida na criação de Algoritmos, é chegada a hora de estender os estudos em programação para a principal ferramenta do programador: *A Linguagem*.

Compreender melhor os recursos proporcionados pela linguagem C o tornará apto a usar mais sabiamente os limitados recursos de hardware e software em seus projetos.

Sem dúvida alguma, sua habilidade ao trabalhar com Algoritmos continuará progredindo. No entanto, a partir de agora, nosso foco se voltará para o estudo de técnicas avançadas de programação, proporcionadas pela Linguagem C.

Os pontos principais dessa parte são: ponteiros, uso avançado de funções, alocação dinâmica, structs e E/S em arquivos.

Cap. 7: CRIAÇÃO, COMPILAÇÃO E EXECUÇÃO DE PROGRAMAS EM C

Neste capítulo serão abordados aspectos técnicos envolvendo a criação, compilação e execução de programas escritos em linguagem C.

7.1 A BIBLIOTECA E A LINKEDIÇÃO

Tecnicamente falando, é possível criar um programa útil e funcional que consista apenas nos comandos realmente criados pelo programador. Porém, isso é muito raro porque C, dentro da atual definição da linguagem, não oferece nenhum método de executar operações de entrada/saída (E/S). Como resultado, a maioria dos programas inclui chamadas a várias funções contidas na *biblioteca C padrão*.

Todo compilador C vem com uma biblioteca C padrão de funções que realizam as tarefas necessárias mais comuns. O padrão C ANSI especifica o conjunto mínimo de funções que estará contido na biblioteca. No entanto, seu compilador provavelmente conterá muitas outras funções. Por exemplo, o padrão C ANSI não define nenhuma função gráfica, mas seu compilador provavelmente inclui alguma.

Em algumas implementações de C, a biblioteca aparece em um grande arquivo; em outras, ela está contida em muitos arquivos menores, uma organização que aumenta a eficiência e a praticidade. Porém, para simplificar, usaremos a forma singular em referência a biblioteca.

Os implementadores do seu compilador C já escreveram a maioria das funções de propósito geral que você usará. Quando chama uma função que não faz parte do programa que você escreveu, o compilador C “memoriza” seu nome. Mais tarde, o *linkeditor* (linker) combina o código que você escreveu com código-objeto já encontrado na biblioteca padrão. Esse processo é chamado de *linkedição*. Alguns compiladores C tem seu próprio linkeditor, enquanto outros usam o linkeditor padrão fornecido pelo seu sistema operacional.

As funções guardadas na biblioteca estão em formato *relocável*. Isso significa que os endereços de memória das várias instruções em código de máquina não estão absolutamente definidos — apenas informações relativas são guardadas. Quando seu programa é linkeditado com as funções da biblioteca padrão, esses endereços relativos são utilizados para criar os endereços realmente usados. Há diversos manuais e livros técnicos que explicam esse processo com mais detalhes. Tais conhecimentos frequentemente são incluídos nas ementas das disciplinas de Sistemas Operacionais nas graduações em Computação. Contudo, você não precisa de nenhuma informação adicional sobre o processo real de relocação para programar em C.

Muitas das funções de que você precisará ao escrever seus programas estão na biblioteca padrão. Elas agem como blocos básicos que você combina. Se escreve uma função que usará muitas vezes, você também pode colocá-la em uma biblioteca. Alguns compiladores permitem que você coloque sua função na biblioteca padrão; outros exigem a criação de uma biblioteca adicional. De qualquer forma, o código estará lá para ser usado repetidamente.

Lembre-se de que o padrão ANSI apenas especifica uma biblioteca padrão mínima. A maioria dos compiladores fornece bibliotecas que contém muito mais funções que aquelas definidas pelo ANSI. Além disso, algumas funções encontradas na versão original de C para UNIX não são definidas pelo padrão ANSI por serem redundantes.

7.2 COMPILAÇÃO SEPARADA

Muitos programas curtos de C estão completamente contidos em um arquivo-fonte. Contudo, quando o tamanho de um programa cresce, também aumenta seu tempo de compilação (e tempos de compilação longos contribuem para paciências curtas!). Logo, C permite que um programa seja contido em muitos arquivos e que cada arquivo seja compilado separadamente. Uma vez que todos os arquivos estejam compilados, eles são linkeditados com qualquer rotina de biblioteca, para formar um código-objeto completo. A vantagem da compilação separada é que, se houver uma mudança no código de um arquivo, não será necessária a recompilação do programa todo. Em tudo, menos nos projetos mais simples, isso economiza um tempo considerável.

7.3 COMPILANDO UM PROGRAMA EM C

Compilar um programa em C consiste nestes três passos:

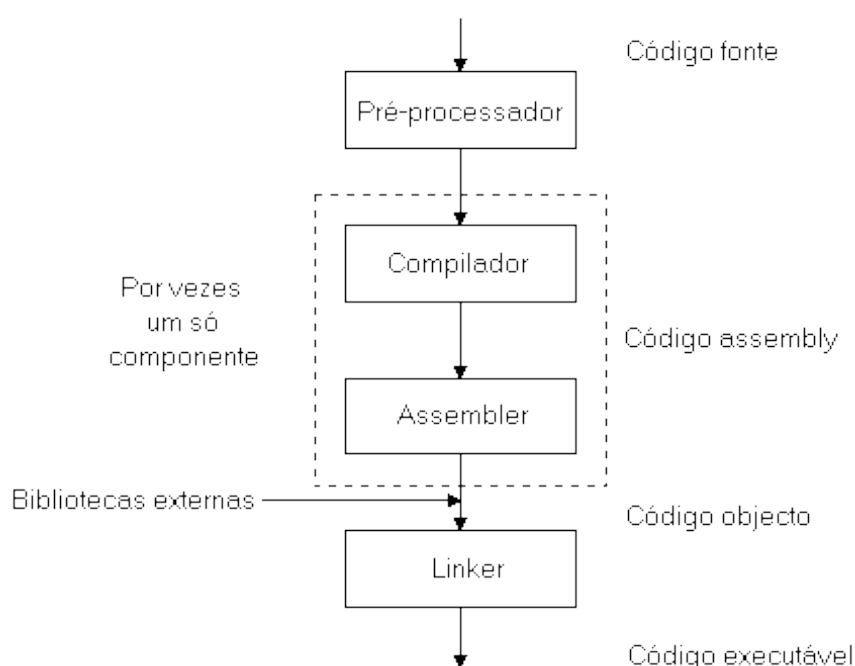
1. Criar o programa
2. Compilar o programa
3. Linkeditar o programa com as funções necessárias da biblioteca

Alguns compiladores fornecem ambientes de programação integrados que incluem um editor. Com outros, é necessário usar um editor separado para criar seu programa. Os compiladores só aceitam a entrada de arquivos de texto padrão. Por exemplo, seu compilador não aceitará arquivos criados por certos processadores de textos porque eles tem códigos de controle e caracteres não-imprimíveis.

O método exato que você utiliza para compilar um programa depende do compilador que esta em uso. Além disso, a linkedição varia muito entre os compiladores e os ambientes.

7.3.1 O MODELO DE COMPILAÇÃO DA LINGUAGEM C

Apesar do processo de compilação, conforme já mencionado, depender do compilador em questão, salientaremos aqui os pontos principais do modelo de compilação da linguagem C. Esse modelo pode ser ilustrado através da figura seguinte.



7.3.1.1 O PRÉ-PROCESSADOR

O pré-processador (que será visto em mais detalhes na Seção 7.5) atua apenas ao nível do código fonte, modificando-o. Trabalha apenas com texto. Algumas das suas funções são:

- Remover os comentários de um programa;
- Interpretar diretivas especiais a ele dirigidas, que começam pelo caractere #.

Por exemplo:

- `#include`: insere o conteúdo de um arquivo de texto no arquivo corrente. Esses arquivos são usualmente designados por cabeçalhos (header files) e têm a extensão `.h`. Por exemplo:
 - `#include <math.h>` - Insere o conteúdo do arquivo `math.h` com a declaração das funções matemáticas da biblioteca padrão.
 - `#include <stdio.h>` - Idem para as funções padrão de entrada/saída.
- `#define`: define um nome simbólico cujas ocorrências no arquivo serão substituídas por outro nome ou constante:
 - `#define MAX_ARRAY_SIZE 100` - substitui todas as ocorrências de `MAX_ARRAY_SIZE` por `100`.

7.3.1.2 O COMPILADOR

Alguns compiladores traduzem o código fonte (texto) recebido do pré-processador para linguagem assembly (também texto). No entanto são também comuns os compiladores capazes de gerarem diretamente *código objeto* (instruções do processador já em código binário).

7.3.1.3 O ASSEMBLER

O *assembler* traduz código em linguagem assembly (texto) para código objeto. Pode estar integrado no compilador. O código objeto é geralmente armazenado em arquivos com a extensão `.o` (unix) ou `.obj` (MS-DOS).

7.3.1.4 O LINKER

Se o programa referencia funções da biblioteca padrão ou outras funções contidas em arquivos com código fonte diferentes do principal (que contém a função `main()`), o *linker* combina todos os objetos com o resultado compilado dessas funções num único arquivo com código executável. As referências a variáveis globais externas também são resolvidas pelo linker.

7.3.1.5 A UTILIZAÇÃO DE BIBLIOTECAS

A linguagem C é muito compacta. Muitas das funções que fazem parte de outras linguagens não estão diretamente incluídas na linguagem C. Temos como exemplo as operações de entrada/saída, a manipulação de strings e certas operações matemáticas.

A funcionalidade correspondente a estas e outras operações não faz parte integrante da linguagem, mas está incluída numa biblioteca externa, bastante rica e padrão. Todas essas operações são executadas invocando funções externas definidas nessa biblioteca padrão. Ao longo de toda esta apostila, podem ser encontrados vários exemplos usando funções da biblioteca padrão do C.

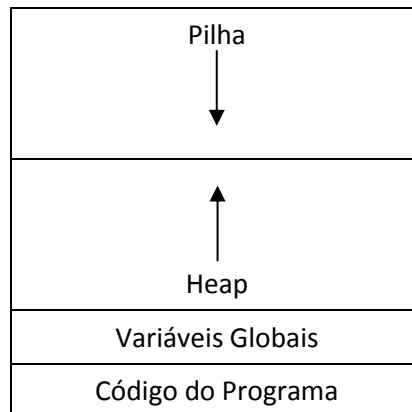
7.4 O MAPA DE MEMÓRIA DE C

Um programa C compilado cria e usa quatro regiões, logicamente distintas na memória, que possuem funções específicas. A primeira região é a memória que contém o código do seu programa. A segunda é aquela onde as variáveis globais são armazenadas. As duas regiões restantes são a pilha e o “heap”. A *pilha* tem diversos usos durante a execução de seu programa. Ela possui o endereço de retorno das chamadas de função, argumentos para funções e variáveis locais. Ela também guarda o estado atual da CPU. O *heap* é uma região

de memória livre que seu programa pode usar, via funções de alocação dinâmica de C, em aplicações como listas encadeadas e árvores.

A disposição exata de seu programa pode variar de compilador para compilador e de ambiente para ambiente. Por exemplo, a maioria dos compiladores para a família de processadores 8086 tem seis maneiras diferentes de organizar a memória em razão da arquitetura segmentada de memória do 8086.

Embora a disposição física exata de cada uma das quatro regiões possa diferir entre tipos de CPU e implementações de C, o diagrama abaixo mostra conceitualmente como seu programa aparece na memória.



7.5 O PRÉ-PROCESSADOR DO C

Você pode incluir diversas instruções do compilador no código-fonte de um programa em C. Elas são chamadas de diretivas do pré-processador e, embora não sejam realmente parte da linguagem de programação C, expandem o escopo do ambiente de programação em C.

Como definido pelo padrão C ANSI, o pré-processador de C contém as seguintes diretivas:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
#include
#define
#undef
#line
#error
#pragma
```

Como você pode observar, todas as diretivas do pré-processador começam com um símbolo #. Além disso, cada diretiva do pré-processador deve estar na sua própria linha. Por exemplo,

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

não funcionará.

Nesta seção, examinaremos as diretivas `#define` e `#include`. As demais podem ser encontradas em detalhes nas literaturas especializadas na linguagem C.

7.5.1 #define

A diretiva `#define` define um identificador e uma string que o substituirá toda vez que for encontrado no arquivo-fonte. O padrão C ANSI refere-se ao identificador como um *nome de macro* e ao processo de substituição como *substituição de macro*. A forma geral da diretiva é

```
#define nome_macro string
```

Observe que não há nenhum ponto-e-vírgula nesse comando. Pode haver qualquer número de espaços entre o identificador e a string, mas, assim que a string começar, será terminada apenas por uma nova linha.

Por exemplo, se deseja usar a palavra `VERDADEIRO` para o valor 1 e a palavra `FALSO` para o valor 0, você declarará duas macros `#define`

```
#define VERDADEIRO 1
#define FALSO 0
```

Isso faz com que o compilador substitua por 1 ou 0 toda vez que encontrar `VERDADEIRO` ou `FALSO` no seu arquivo-fonte. Por exemplo, o fragmento seguinte escreve 0 1 2 na tela:

```
printf("%d %d %d", FALSO, FALSO, VERDADEIRO+1);
```

Uma vez que um nome de macro tenha sido definido, ele pode ser usado como parte da definição de outros nomes de macro. Por exemplo, este código define os valores `UM`, `DOIS` e `TRES`:

```
#define UM 1
#define DOIS UM+UM
#define TRES UM+DOIS
```

Substituição de macro é simplesmente a transposição de sua string associada. Portanto, se você quisesse definir uma mensagem de erro padrão, poderia escrever algo como isto:

```
#define E_MS "erro padrão na entrada\n"
printf(E_MS);
```

O compilador substituirá a string `"erro padrão na entrada\n"` quando o identificador `E_MS` for encontrado. Para o compilador, o comando com o `printf()` aparecerá, na realidade, como

```
printf("erro padrão na entrada\n");
```

Nenhuma substituição de texto ocorre se o identificador esta dentro de uma string entre aspas. Por exemplo,

```
#define XYZ isso é um teste
printf("XYZ");
```

não escreve `isso é um teste`, mas `XYZ`.

Os programadores C geralmente usam letras maiúsculas para identificadores definidos. Essa convenção ajuda qualquer um que esteja lendo o programa a saber de relance que uma substituição de macro irá ocorrer. Além disso, é melhor colocar todos os `#define` no início do arquivo ou em um arquivo de cabeçalho separado em lugar de espalhá-los pelo programa.

Substituição de macro é usada mais frequentemente para definir nomes para “números mágicos” que aparecem em um programa. Por exemplo, você pode ter um programa que defina uma matriz e tenha diversas

rotinas que acessam essa matriz. Em lugar de fixar o tamanho da matriz com uma constante, você poderia definir um tamanho e usar esse nome sempre que o tamanho da matriz for necessário. Dessa forma, você só precisa fazer uma alteração e recompilar para alterar o tamanho da matriz. Por exemplo,

```
#define TAMANHO_MAX 100

/* ... */

float balance[TAMANHO_MAX];

/* ... */

for (i=0; i<TAMANHO_MAX; i++) printf("%f", balance[i]);
```

Como `TAMANHO_MAX` define o tamanho da matriz `balance`, se o tamanho desta precisar ser modificado no futuro, você só precisa modificar a definição de `TAMANHO_MAX`. Todas as referências subsequentes a ela serão automaticamente atualizadas quando você recompilar seu programa.

7.5.1.1 DEFININDO MACROS SEMELHANTES A FUNÇÕES

A diretiva `#define` possui outro recurso poderoso: o nome da macro pode ter argumentos. Cada vez que o nome da macro é encontrado, os argumentos usados na sua definição são substituídos pelos argumentos reais encontrados no programa. Esta forma de macro é chamada de *macro semelhante a função*. Por exemplo,

```
#include <stdio.h>

#define ABS(a) (a)<0 ? -(a) : (a)

void main(void)
{
    printf("abs de -1 e 1: %d %d", ABS(-1), ABS(1));
}
```

Quando este programa é compilado, o `a` na definição da macro será substituído pelos valores `-1` e `1`. Os parênteses ao redor de `a` garantem a substituição correta em todos os casos. Por exemplo, se os parênteses ao redor de `a` fossem removidos, esta expressão

```
ABS(10-20)
```

seria convertida em

```
10-20<0 ? -10-20 : 10-20
```

e geraria o valor errado.

O uso de macros semelhantes a funções no lugar de funções reais possui uma grande vantagem: ele incrementa a velocidade de execução do código porque não há a necessidade de executar código para gerenciar a chamada de uma função. No entanto, se o tamanho da macro semelhante a função for muito grande, este aumento de velocidade pode ser pago com um aumento no tamanho do programa em função do código duplicado.

7.5.2 #include

A diretiva `#include` instrui o compilador a ler outro arquivo-fonte adicionado àquele que contém a diretiva `#include`. O nome do arquivo adicional deve estar entre aspas ou símbolos de maior e menor. Por exemplo,

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

Ambas instruem o compilador a ler e compilar o arquivo de cabeçalho para as rotinas de arquivos em disco da biblioteca.

Arquivos de inclusão podem ter diretivas `#include` neles. Isso é denominado *includes aninhados*. O número de níveis de aninhamento varia entre compiladores. Porém, o padrão C ANSI estipula que pelo menos oito níveis de inclusões aninhadas estão disponíveis.

Se o nome do arquivo está envolvido por chaves angulares (sinais de maior e menor), o arquivo será procurado de forma definida pelo criador do compilador. Frequentemente, isso significa procurar em algum diretório especialmente criado para arquivos de inclusão. Se o nome do arquivo está entre aspas, o arquivo é procurado de uma maneira definida pela implementação. Para muitas implementações, isso significa uma busca no diretório de trabalho atual. Se o arquivo não for encontrado, a busca será repetida como se o nome do arquivo estivesse envolvido por chaves angulares.

A maioria dos programadores tipicamente usa chaves angulares para incluir os arquivos de cabeçalho padrão. O uso de aspas é reservado geralmente para a inclusão de arquivos do projeto. No entanto, não existe nenhuma regra que exija este uso.

7.6 UMA REVISÃO DE TERMOS

Os termos a seguir serão usados frequentemente durante todo o contexto desta disciplina. Você deve estar completamente familiarizado com eles.

- **Código-Fonte:** O texto de um programa que um usuário pode ler, normalmente interpretado como o programa. O código-fonte é a entrada para o compilador C.
- **Código-Objeto:** Tradução do código-fonte de um programa em código de máquina que o computador pode ler e executar diretamente. O código-objeto é a entrada para o linkeditor.
- **Linkeditor:** Um programa que une funções compiladas separadamente em um programa. Ele combina as funções da biblioteca C padrões com o código que você escreveu. A saída do linkeditor é um programa executável.
- **Biblioteca:** O arquivo contendo as funções padrão que seu programa pode usar. Essas funções incluem todas as operações de E/S como também outras rotinas úteis.
- **Tempo de compilação:** Os eventos que ocorrem enquanto o seu programa está sendo compilado. Uma ocorrência comum em tempo de compilação é um erro de sintaxe.
- **Tempo de execução:** Os eventos que ocorrem enquanto o seu programa é executado.

Cap. 8: PONTEIROS

O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C. Há três razões para isso: primeiro, ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos; segundo, eles são usados para suportar as rotinas de alocação dinâmica de C, e terceiro, o uso de ponteiros pode aumentar a eficiência de certas rotinas.

Ponteiros são um dos aspectos mais fortes e mais perigosos de C. Por exemplo, ponteiros não-inicializados, ou *ponteiros selvagens*, podem provocar uma quebra do sistema. Talvez pior, é fácil usar ponteiros incorretamente, ocasionando erros que são muito difíceis de encontrar.

8.1 O QUE SÃO PONTEIROS?

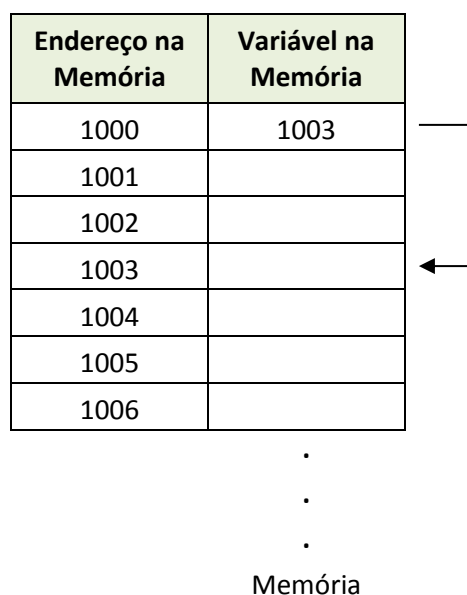
Os `int` guardam inteiros. Os `float` guardam números de ponto flutuante. Os `char` guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. O C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais terem o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro `int` aponta para um inteiro, isto é, guarda o endereço de um inteiro.

Portanto, um ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de outra variável na memória. Se uma variável contém o endereço de outra, então a primeira variável é dita apontar para a segunda. A figura abaixo ilustra essa situação.



Do ponto de vista lógico, a memória principal do computador (RAM) é um enorme vetor de bytes. Milhões (ou mesmo bilhões) de bytes, cada um com um endereço, ou seja, com um valor inteiro associado. Assim sendo, independente do tipo de ponteiro, na realidade todos são variáveis que guardam um valor inteiro, um endereço de um byte na memória. Qual o tamanho dessa variável? A resposta é: depende da arquitetura do processador. Por exemplo, um processador de 32 bits pode endereçar até 2^{32} bytes (4 GB). Logo, um ponteiro (de qualquer tipo), compilado para uma máquina de 32 bits tem, exatamente, 32 bits (4 bytes).

Agora, você pode estar se perguntando por que existem ponteiros de diferentes tipos, já que todos eles endereçam uma posição de memória (um byte). Entender o porquê disso depende de sabermos que, através de um ponteiro para uma região de memória (uma variável, por exemplo), o programador pode alterar o seu conteúdo (ou seja, o valor armazenado naquela posição de memória). No entanto, diferentes tipos de dados têm diferentes formatos e tamanhos. Logo, visto ponteiros estão associados ao tipo de dado para o qual eles apontam (a exceção do ponteiro para `void`, a ser visto no futuro), o compilador pode gerar código para manipular adequadamente o valor apontado (sendo capaz de reconhecer, por exemplo, que um `int` numa determinada arquitetura tem 32 bits, logo ele terá que manipular os 4 bytes que se iniciam no endereço atribuído ao ponteiro). Isso ficará mais claro quando estudarmos como dereferenciar (acessar o valor apontado) ponteiros, nas seções seguintes.

8.2 DECLARANDO E UTILIZANDO PONTEIROS

Para declarar um ponteiro temos a seguinte forma geral:

```
tipo_do_ponteiro *nome_da_variável;
```

É o asterisco (*) que faz o compilador saber que aquela variável não vai guardar um valor, mas sim um endereço para aquele tipo especificado. Vamos ver exemplos de declarações:

```
int *pt;
char *temp, *pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador `&` (“endereço de...”). Veja o exemplo:

```
int count = 10;
int *pt;
pt = &count;
```

Criamos um inteiro `count` com o valor 10 e um apontador para um inteiro `pt`. A expressão `&count` nos dá o endereço de `count`, o qual armazenamos em `pt`. Simples, não é? Repare que *não* alteramos o valor de `count`, que continua valendo 10.

Como nós colocamos um endereço em `pt`, ele está agora “liberado” para ser usado. Podemos, por exemplo, alterar o valor de `count` usando `pt`. Para tanto vamos usar o operador “inverso” do operador `&`. É o

operador * ("valor de..."). No exemplo acima, uma vez que fizemos `pt=&count` a expressão `*pt` é equivalente ao próprio `count`. Isto significa que, se quisermos mudar o valor de `count` para 12, basta fazer `*pt=12`.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador `&`. Ou seja, o operador `&` aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador `*` ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador `*` (multiplicação) não é o mesmo operador que o `*` (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
void main()
{
    int num, valor;
    int *p;
    num = 55;
    p = &num;          /* Pega o endereço de num */
    valor = *p;         /* Valor é igualado a num de uma maneira indireta */
    printf("\n\n %d \n", valor);
    printf("Endereço para onde o ponteiro aponta: %p\n", p);
    printf("Valor da variavel apontada: %d\n", *p);
}

#include <stdio.h>
void main()
{
    int num, *p;
    num = 55;
    p = &num;          /* Pega o endereço de num */
    printf("\nValor inicial: %d\n", num);
    *p = 100;          /* Muda o valor de num de uma maneira indireta */
    printf("\nValor final: %d\n", num);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código `%p` usado na função `printf()` indica à função que ela deve imprimir um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se tivermos dois ponteiros `p1` e `p2` podemos igualá-los fazendo `p1=p2`. Repare que estamos fazendo com que `p1` aponte para o mesmo lugar que `p2`. Se quisermos que a variável apontada por `p1` tenha o mesmo conteúdo da variável apontada por `p2` devemos fazer `*p1=*p2`. Basicamente, depois que se aprende a usar os dois operadores (`&` e `*`) fica fácil entender operações com ponteiros.

As próximas operações, também muito usadas, são o incremento e o decremento. Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta. Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro. Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro `char*` ele anda 1 byte na memória e se você incrementa um ponteiro `double*` ele anda 8 bytes na memória. O decremento funciona semelhantemente. Supondo que `p` é um ponteiro, as operações são escritas como:

```
p++;
p--;
```

Mais uma vez é importante frisar. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro *p*, faz-se:

```
(*p)++;
```

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;
```

ou

```
p+=15;
```

E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);
```

A subtração funciona da mesma maneira.

Outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros? Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais (==) ou diferentes (!=). No caso de operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*. Então uma comparação entre ponteiros pode nos dizer qual dos dois está “mais adiante” na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1 > p2
```

Há, entretanto, operações que você *não* pode efetuar num ponteiro. Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair `float` ou `double` de ponteiros.

8.3 PONTEIROS E VETORES

Veremos nesta seção que ponteiros e vetores têm uma ligação muito forte.

8.3.1 VETORES COMO PONTEIROS

Vamos dar agora uma ideia de como o C trata vetores.

Quando você declara uma matriz da seguinte forma:

```
tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];
```

o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:

```
tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo
```

O compilador então aloca este número de bytes em um espaço livre de memória. O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*. Este conceito é fundamental. Eis o

porquê: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.

Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

```
nome_da_variável[indice]
```

Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:

```
*(nome_da_variável+índice)
```

Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato. Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, **nome_da_variável* e então devemos ter um índice igual a zero. Então sabemos que:

```
*nome_da_variável
```

é equivalente a

```
nome_da_variável[0]
```

Outra coisa. Apesar de, na maioria dos casos não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices. Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura sequencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
#include <stdio.h>

void main()
{
    float matrx[50][50];
    int i, j;
    for (i=0; i<50; i++)
        for (j=0; j<50; j++)
            matrx[i][j] = 0.0;
}
```

Podemos reescrevê-lo usando ponteiros:

```
#include <stdio.h>

void main()
{
    float matrx[50][50];
    float *p;
    int count;
    p = matrx[0];
    for (count=0; count<2500; count++)
    {
        *p = 0.0;
        p++;
    }
}
```

No primeiro programa, *cada* vez que se faz `matrx[i][j]` o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos. No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos⁵.

Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa que não se consegue alterar o endereço que é apontado pelo “nome do vetor”. Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

/* as operações a seguir são inválidas */
vetor = vetor + 2;      /* ERRADO: vetor não é variável */

vetor++;                /* ERRADO: vetor não é variável */

vetor = ponteiro;       /* ERRADO: vetor não é variável */
```

Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que `vetor` não é um Lvalue. Lvalue, significa “Left value”, um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros compiladores dirão que tem-se “incompatible types in assignment”, tipos incompatíveis em uma atribuição.

```
/* as operações abaixo são válidas */

ponteiro = vetor;      /* CERTO: ponteiro é variável */

ponteiro = vetor+2;    /* CERTO: ponteiro é variável */
```

O que você aprendeu nesta seção é de suma importância. Não siga adiante antes de entendê-la bem.

8.3.2 PONTEIROS COMO VETORES

Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor. Como consequência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>

void main ()
{
    int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *p;

    p = matrx;

    printf("O terceiro elemento do vetor eh: %d", p[2]);
}
```

Podemos ver que `p[2]` equivale a `*(p+2)`.

8.3.3 STRINGS

Seguindo o raciocínio acima, nomes de strings, são do tipo `char*`. Isto nos permite escrever a nossa função `StrCpy()`, que funcionará de forma semelhante à função `strcpy()` da biblioteca de strings do C:

⁵ O significado da expressão `p = matrx[0]`; ficará mais claro posteriormente, na seção 8.5.

```

#include <stdio.h>

void StrCpy(char *destino, char *origem)
{
    while (*origem)
    {
        *destino = *origem;
        origem++;
        destino++;
    }

    *destino = '\0';
}

void main()
{
    char str1[100], str2[100], str3[100];

    printf("Entre com uma string: ");
    gets(str1);

    StrCpy(str2, str1);
    StrCpy(str3, "Voce digitou a string ");

    printf ("\n\n%s%s", str3, str2);
}

```

Há vários pontos a destacar no programa acima. Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como `gets()` e `strcpy()` funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia. No comando `while (*origem)` estamos usando o fato de que a string termina com `'\0'` como critério de parada. Quando fazemos `origem++` e `destino++` o leitor poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que foi recomendado que se deveria fazer, anteriormente. O que o leitor talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro `origem` na função `StrCpy()` o ponteiro `str2` permanece inalterado na função `main()`.

8.3.4 ENDEREÇOS DE ELEMENTOS DE VETORES

Nesta seção vamos apenas ressaltar que a notação

`&nome_da_variável[índice]`

é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a `nome_da_variável + índice`. É interessante notar que, como consequência, o ponteiro `nome_da_variável` tem o endereço `&nome_da_variável[0]`, que indica onde na memória está guardado o valor do primeiro elemento do vetor.

8.3.5 VETORES DE PONTEIROS

Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx[10];
```

No caso acima, `pmatrx` é um vetor que armazena 10 ponteiros para inteiros.

8.4 INICIALIZANDO PONTEIROS

Podemos inicializar ponteiros. Vamos ver um caso interessante dessa inicialização de ponteiros com strings.

Precisamos, para isto, entender como o C trata as strings constantes. Toda string que o programador insere no programa é colocada num banco de strings que o compilador cria. No local onde está uma string no programa, o compilador coloca o endereço do início daquela string (que está no banco de strings). É por isto que podemos usar `strcpy()` do seguinte modo:

```
strcpy(string, "String constante.");
```

A função `strcpy()` pede dois parâmetros do tipo `char*`. Como o compilador substitui a string `"String constante."` pelo seu endereço no banco de strings, tudo está bem para a função `strcpy()`.

O que isto tem a ver com a inicialização de ponteiros? É que, para uma string que vamos usar várias vezes, podemos fazer:

```
char *str1 = "String constante.";
```

Aí poderíamos, em todo lugar que precisarmos da string, usar a variável `str1`. Devemos apenas tomar cuidado ao usar este ponteiro. Se o alterarmos vamos perder a string. Se o usarmos para alterar a string podemos facilmente corromper o banco de strings que o compilador criou.

Mais uma vez fica o aviso: ponteiros são poderosos, mas, se usados com descuido, podem ser uma ótima fonte de dores de cabeça.

8.5 INDIREÇÃO MÚLTIPLA (PONTEIROS PARA PONTEIROS)

Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo. Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

```
tipo_da_variável **nome_da_variável;
```

Na **declaração**, o duplo asterisco `**` indica quantos níveis de indireção o ponteiro possui. Apenas um, indica um ponteiro simples. Dois, um ponteiro para ponteiro. E assim sucessivamente.

No **uso** da variável ponteiro, `**nome_da_variável` é o conteúdo final da variável apontada; `*nome_da_variável` é o conteúdo do ponteiro intermediário e `nome_da_variável` é o ponteiro inicial, que aponta para a outra variável ponteiro.

Tome, por exemplo, a declaração abaixo:

```
int i, *pi, **ppi;
pi = &i;
ppi = &pi;
```

Na primeira linha, uma variável inteira, um ponteiro para inteiro, e um ponteiro para ponteiro para inteiro estão sendo declarados, respectivamente. Após isso, o ponteiro `pi` passa a apontar para a região de memória em que `i` encontra-se, em outras palavras, para o endereço de `i`. A seguir, `ppi` aponta para o ponteiro `pi`, ou seja, seu conteúdo é o endereço de `pi`.

Esquemáticamente, seria algo como:

Variável	Endereço	Conteúdo
i	1000	?
	⋮	
pi	1012	1000
	⋮	
ppi	1040	1012
	⋮	

No C podemos declarar ponteiros para ponteiros para ponteiros, ou então, ponteiros para ponteiros para ponteiros para ponteiros e assim por diante. Para fazer isto (não pergunte a utilidade disto!) basta aumentar o número de asteriscos na declaração. A lógica é a mesma.

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

```
#include <stdio.h>

void main()
{
    float fpi = 3.1415, *pf, **ppf;
    pf = &fpi;           /* pf armazena o endereço de fpi */
    ppf = &pf;           /* ppf armazena o endereço de pf */
    printf("%f", **ppf); /* Imprime o valor de fpi */
    printf("%f", *pf);   /* Também imprime o valor de fpi */
}
```

Entender indireção múltipla, especialmente em dois níveis, é essencial para compreender a relação que existe entre ponteiros e matrizes 2D em C. Esse entendimento é exigido quando se faz alocação dinâmica de matrizes, assunto a ser abordado no Capítulo 10.

A fim de clarear a sua mente neste assunto, considere a matriz `mat[4][4]` abaixo definida:

```
int mat[4][4] = { 2,  4,  6,  8,
                  10, 12, 14, 16,
                  18, 20, 22, 24,
                  26, 28, 30, 32 };
```

A memória do computador nada mais é que um imenso vetor de bits, ou seja, uma sequência de dados linearmente organizados. A ideia de bi-dimensionidade, neste caso, é “traduzida” pelo compilador a fim de serem dispostos de maneira linear na memória. Assim sendo, quando dados uma coordenada (linha/coluna) o compilador se encarrega de encontrar a posição correta do dado na memória. Para o caso 2D, se tivermos uma matriz `mat[M][N]`, o acesso a um elemento particular `mat[k][y]` é convertido pelo compilador no seguinte elemento:

$$\text{mat}[k][y] = \text{endereço_primeiro_elemento} + (k*N) + y$$

Ou seja, a matriz é disposta linearmente simplesmente alocando as linhas lado a lado na memória. Na matriz do exemplo acima, podemos dizer que a mesma é armazenada na memória da seguinte forma:

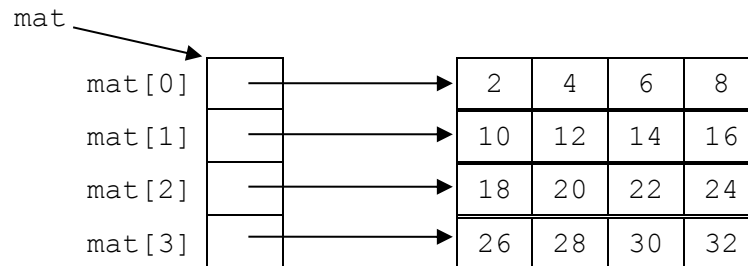
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

e, por exemplo, o elemento `mat[2][1]` (cujo valor é 20) está $2*4+1 = 9$ posições após o primeiro elemento. Ou seja, ele é o elemento 1 (segundo elemento) da linha 2 (terceiro agrupamento). Confira!

Calma que a coisa é ainda mais complexa! Lembra-se de que dissemos, na seção 8.3.1 que o nome de um vetor na verdade é um ponteiro para o seu primeiro elemento? E quanto as matrizes (2D)? O que representa o nome da matriz? O texto abaixo tenta esclarecer:

*“O nome de uma variável do tipo matriz (2D) é na verdade um ponteiro para um vetor de ponteiros (cujo tamanho corresponde ao número de **linhas** da matriz), ou seja, um ponteiro para um ponteiro (indireção dupla). Cada um dos elementos desse vetor é um endereço de um vetor de elementos (cujo tamanho corresponde ao número de **colunas** da matriz).”*

Esquemáticamente falando...



Assim sendo:

```
mat = endereço do vetor de ponteiros (ponteiro para ponteiro)
mat[k] = endereço de um vetor de elementos
mat[k][y] = um valor
```

Ainda confuso? Normal... Abaixo é dado um exemplo completo que vai ainda mais a fundo e poderá exigir do aluno muito esforço a fim de ser compreendido. No entanto, acredito eu, é 100% elucidativo. Compile e execute-o. Faça alterações a fim de testar sua compreensão, se julgar necessário. **Não avance sem compreendê-lo plenamente!**

```
#include <stdio.h>

main()
{
    unsigned char mat[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
    unsigned char *pmat;
    int i, j;

    pmat = mat[0];

    printf("\t\t\t ::: MATRIZES X PONTEIROS :::\n\n");

    printf("Declaracao da matriz:  unsigned char mat[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9};\n");
    printf("Declaracao do ponteiro: unsigned char *pmat;\n");
    printf("\n-----\n\n");

    printf("Impressao da matriz usando a notacao mat[i][j]\n\n");
    printf("      j=0  j=1  j=2\n");
    for (i=0; i<3; i++)
    {
        printf("i=%d", i);
        for (j=0; j<3; j++)
            printf("  %d ", mat[i][j]);
        printf("\n");
    }
    printf("\n-----\n\n");

    printf("METODO 1: Acesso usando a matriz como ponteiro...\n\n");
    printf("      Valor      Endereco\n");
    for (i=0; i<9; i++)
        printf("*((*mat)+%d) -> %d      (*mat)+%d -> %p\n", i, *((*mat)+i), i, (*mat)+i);
    printf("\n-----\n\n");

    printf("METODO 2: Fazendo pmat = mat[0] e usando pmat...\n\n");
```



```

printf("          Valor          Endereco\n");
for (i=0; i<9; i++)
    printf("(*(pmat+%d) -> %d      pmat+%d -> %p\n", i, *(pmat+i), i, pmat+i);
printf("\n-----\n\n");

printf("Observe! Dois diferentes metodos de acesso obtendo o mesmo resultado.\n");
printf("\n-----\n\n");

printf("Agora observe a indirecao multipla quando mat eh vista como ponteiro...\n\n");
printf("          Valor          Endereco\n");
for (i=0; i<3; i++)
    printf("(*(mat+%d) -> %d      *(mat+%d) -> %p      valor e end. do elem 0 da linha %d\n", i,
***(mat+i), i, *(mat+i), i);
printf("\nObserve que mat eh um ponteiro para ponteiro...\n");
printf("Fazer (mat+1) eh o mesmo que passar para a proxima linha da matriz...\n");
printf("\n-----\n\n");

printf("Ou no caso da matriz toda...\n\n");
printf("          Valor          Endereco\n");
for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        printf("(*(*(mat+%d)+%d) -> %d      *(mat+%d)+%d -> %p      valor e end. do elem %d da linha
%d\n", i, j, (*(mat+i)+j), i, j, *(mat+i)+j, j, i);
printf("\n-----\n\n");

printf("Portanto...\n\n");
printf("  mat          -> ponteiro para ponteiro (indirecao multipla)\n");
printf("  *mat e mat[0]   -> sao um referencias (ponteiros)\n");
printf("  **mat e mat[0][0] -> sao valores");

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

```

8.6 CUIDADOS A SEREM TOMADOS AO SE USAR PONTEIROS

O principal cuidado ao se usar um ponteiro deve ser: saiba sempre *para onde* o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como **não** usar um ponteiro:

```

#include <stdio.h>

void main () /* Errado - Nao Execute */
{
    int x,*p;
    x = 13;
    *p = x;
}

```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro `p` pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

Cap. 9: FUNÇÕES: ASPECTOS AVANÇADOS

Funções são os blocos de construção de C e o local onde toda a atividade do programa ocorre. Elas são uma das características mais importantes de C. Assim sendo, apesar de termos visto o assunto de maneira geral na Seção 6.3, há diversos outros aspectos importantes a serem vistos. E este é um momento adequado para este estudo, já que diversos outros assuntos necessários a um pleno entendimento das funções em C já foram vistos (ponteiros, em especial).

9.1 A FUNÇÃO

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.

Uma função no C tem a seguinte forma geral:

```
tipo_de_retorno nome_da_função(declaração_de_parâmetros)
{
    corpo_da_função
}
```

O *tipo_de_retorno* é o tipo de variável que a função vai retornar. O default é o tipo `int`, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A *declaração_de_parâmetros* é uma lista com a seguinte forma geral:

```
tipo nome1, tipo nome2, ... , tipo nomeN
```

Repare que o tipo deve ser especificado para cada uma das *N* variáveis de entrada. É na *declaração_de_parâmetros* que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no *tipo_de_retorno*). Uma função pode não ter parâmetros, neste caso a lista de parâmetros é vazia. No entanto, mesmo que não existam parâmetros, os parênteses ainda são necessários.

O *corpo_da_função* é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

9.2 O COMANDO return

O comando `return` tem dois importantes usos. Primeiro, ele provoca uma saída imediata da função que o contém. Isto é, faz com que a execução do programa retorne ao código chamador. Segundo, ele pode ser usado para devolver um valor.

O comando `return` tem a seguinte forma geral:

```
return valor_de_retorno;
```

ou

```
return;
```

Digamos que uma função está sendo executada. Quando se chega a uma declaração `return` a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante

lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração `return`. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração `return`. Abaixo estão dois exemplos de uso do `return`:

```
#include <stdio.h>

int Square(int a)
{
    return (a*a);
}

void main()
{
    int num;
    printf("Entre com um numero: ");
    scanf("%d", &num);
    num = Square(num);
    printf("\n\n0 seu quadrado vale: %d\n", num);
}

#include <stdio.h>

int EhPar(int a)
{
    if (a % 2)      /* Verifica se a é divisível por dois */
        return 0;  /* Retorna 0 se não for divisível */
    else
        return 1;  /* Retorna 1 se for divisível */
}

void main()
{
    int num;
    printf("Entre com numero: ");
    scanf("%d", &num);
    if (EhPar(num))
        printf("\n\n0 numero eh par.\n");
    else
        printf("\n\n0 numero eh impar.\n");
}
```

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

```
func(a,b) = x; /* Errado! */
```

No segundo exemplo vemos o uso de mais de um `return` em uma função.

Algumas vezes, funções que, na realidade, não produzem um resultado relevante de qualquer forma devolvem um valor. Por exemplo, `printf()` devolve o número de caracteres escritos. Entretanto, é muito incomum encontrar um programa que realmente verifique isso. Em outras palavras, embora todas as funções, exceto aquelas do tipo `void`, devolvam valores, você não tem necessariamente de usar o valor de retorno. Uma questão envolvendo valores de retorno de funções é: “Eu não tenho de atribuir esse valor a alguma variável já que um valor está sendo devolvido?”. A resposta é *não*. Se não há nenhuma atribuição especificada, o valor de retorno é simplesmente descartado.

9.3 TRÊS TIPOS DE FUNÇÃO

Quando você escreve programas, suas funções geralmente serão de três tipos. O primeiro tipo é simplesmente computacional. Essas funções são projetadas especificamente para executar operações em seus

argumentos e devolver um valor. Uma função computacional e uma função “pura”. Exemplos são as funções da biblioteca padrão `sqrt()` e `sin()`, que calculam a raiz quadrada e o seno de seus argumentos.

O segundo tipo de função manipula informações e devolve um valor que simplesmente indica o sucesso ou a falha dessa manipulação. Um exemplo é a função da biblioteca padrão `fclose()`, que é usada para fechar um arquivo. Se a operação de fechamento for bem-sucedida, a função devolverá 0; se a operação não for bem-sucedida, ela devolverá um código de erro.

O último tipo não tem nenhum valor de retorno explícito. Em essência, a função é estritamente de procedimento e não produz nenhum valor. Um exemplo é `exit()`, que termina um programa. Todas as funções que não devolvem valores devem ser declaradas como retornando o tipo `void`. Ao declarar uma função como `void`, você a protege de ser usada em uma expressão, evitando uma utilização errada acidental.

9.4 PROTÓTIPOS DE FUNÇÕES

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função `main()`. Isto é, as funções estão fisicamente antes da função `main()`. Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função `main()`, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto as funções foram colocadas antes da função `main()`: quando o compilador chegasse à função `main()` ele já teria compilado as funções e já saberia seus formatos.

Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?

A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

```
tipo_de_retorno nome_da_função(declaração_de_parâmetros);
```

onde o *tipo_de_retorno*, o *nome_da_função* e a *declaração_de_parâmetros* são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

```
#include <stdio.h>

float Square(float a);

void main()
{
    float num;

    printf("Entre com um numero: ");
    scanf("%f", &num);

    num = Square(num);

    printf("\n\n0 seu quadrado vale: %f\n", num);
}

float Square(float a)
{
    return (a*a);
}
```

Observe que a função `Square()` está colocada depois de `main()`, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

E como fica o protótipo quando a função não tem nenhum parâmetro? A resposta é: quando uma função não tem parâmetros, seu protótipo usa `void` dentro dos parênteses. Por exemplo, se uma função chamada `f()` devolve um `float` e não tem parâmetros, seu protótipo será:

```
float f(void);
```

Isso informa ao compilador que a função não tem parâmetros e qualquer chamada a função com parâmetros é um erro.

9.5 O TIPO void

Agora vamos ver o único tipo da linguagem C que não detalhamos ainda: o `void`. Em inglês, `void` quer dizer vazio e é isto mesmo que o `void` é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros! Podemos agora escrever o protótipo de uma função que não retorna nada:

```
void nome_da_função(declaração_de_parâmetros);
```

Numa função, como a acima, não temos valor de retorno na declaração `return`. Podemos, também, fazer funções que não têm parâmetros:

```
tipo_de_retorno nome_da_função(void);
```

ou, ainda, que não tem parâmetros e não retornam nada:

```
void nome_da_função(void);
```

Um exemplo de funções que usam o tipo `void`:

```
#include <stdio.h>

void Mensagem(void);

void main()
{
    Mensagem();
    printf("\tDiga de novo:\n");
    Mensagem();
}

void Mensagem(void)
{
    printf("Ola! Eu estou vivo.\n");
}
```

Se quisermos que a função retorne algo, devemos usar a declaração `return`. Se não quisermos, basta declarar a função como tendo `tipo_de_retorno void`. Devemos lembrar agora que a função `main()` é uma função e como tal devemos tratá-la. O compilador acha que a função `main()` deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função

`main()`. Se assim o quisermos, devemos nos lembrar da seguinte convenção: se o programa retornar zero, significa que ele terminou normalmente, e, se o programa retornar um valor diferente de zero, significa que o programa teve um término anormal. Se não estivermos interessados neste tipo de coisa, basta declarar a função `main` como retornando `void` (como muitos exemplos dados nesta apostila).

As duas funções `main()` abaixo são válidas:

```
main(void)
{
    ....
    return 0;
}
```

```
void main(void)
{
    ....
}
```

A primeira forma é válida porque, como já vimos, as funções em C têm, por padrão, retorno inteiro. Alguns compiladores reclamarão da segunda forma de `main()`, dizendo que `main()` sempre deve retornar um inteiro. Se isto acontecer com o compilador que você está utilizando, basta fazer `main()` retornar um inteiro.

9.6 ARQUIVOS-CABEÇALHOS

Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em `.h`. A extensão `.h` vem de header (cabeçalho, em inglês). Já vimos exemplos como `stdio.h`, `string.h`. Estes arquivos, na verdade, não possuem os códigos completos das funções. Eles só contêm *protótipos* de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto. O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da “linkagem”. Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas.

Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também.

Suponha que a função `int EhPar(int a)`, da Seção 9.2 seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado, por exemplo, de `funcao.h` teremos a seguinte declaração:

```
int EhPar(int a);
```

O código da função será escrito num arquivo a parte. Vamos chamá-lo de `funcao.c`. Neste arquivo teremos a definição da função:

```
int EhPar(int a)
{
    if (a%2)           /* Verifica se a eh divisível por dois */
        return 0;
    else
        return 1;
}
```

Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de `principal.c`.

```
#include <stdio.h>
#include "funcao.h"

void main()
{
    int num;
    printf("Entre com numero: ");
```

```

scanf("%d",&num);
if (EhPar(num))
    printf("\n\n0 numero eh par.\n");
else
    printf("\n\n0 numero eh impar.\n");
}

```

É vital que o compilador tome conhecimento do arquivo `funcao.c`, ao qual o `funcao.h` está “atrelado”. Isso pode ser feito explicitamente na linha de comando que efetuará a compilação, por exemplo, quando usado o compilador `gcc`, este programa poderia ser compilado da seguinte forma:

```
gcc principal.c funcao.c -o saida
```

onde “saída” seria o arquivo executável gerado.

Quando usado um ambiente de desenvolvimento, como o Microsoft Visual Studio, uma linha de comando equivalente a essa seria gerada automaticamente, bastando que você crie todos os arquivos (`.c` e `.h`) a partir do comando de adicionar novo item, aos arquivos fonte do seu projeto. Com isso, o ambiente tomará conhecimento de todos os arquivos envolvidos no seu programa e cuidará para que os mesmos sejam corretamente compilados e “linkados”.

9.7 PASSAGEM DE PARÂMETROS POR VALOR E PASSAGEM POR REFERÊNCIA

Já vimos que, na linguagem C, quando chamamos uma função, os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado *chamada por valor*. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros. Veja o exemplo abaixo:

```

#include <stdio.h>

float sqr(float num);

void main()
{
    float num, sq;
    printf("Entre com um numero: ");
    scanf("%f", &num);
    sq = sqr(num);
    printf("\n\n0 numero original eh: %f\n", num);
    printf("0 seu quadrado vale: %f\n", sq);
}

float sqr(float num)
{
    num = num*num;
    return num;
}

```

No exemplo acima o parâmetro formal `num` da função `sqr()` sofre alterações dentro da função, mas a variável `num` da função `main()` permanece inalterada: é uma chamada por valor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de *chamada por referência*. Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há, entretanto, no C, um recurso de programação que podemos usar para *simular* uma chamada por referência.

Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a “referência” que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar-nos de colocar um & na frente das variáveis que estivermos passando para a função. Veja um exemplo:

```
#include <stdio.h>

void troca(int *a, int *b);

void main(void)
{
    int num1, num2;
    num1 = 100;
    num2 = 200;
    troca(&num1, &num2);
    printf("\n\nEles agora valem %d %d\n", num1, num2);
}

void troca(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função `troca()` o endereço das variáveis `num1` e `num2`. Estes endereços são copiados nos ponteiros `a` e `b`. Através do operador `*` estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em `num1` e `num2`, que, portanto, estão sendo modificados!

Espere um momento... será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de &? Já! É assim que nós chamamos a função `scanf()`. Mas por quê? Vamos pensar um pouco. A função `scanf()` usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e, portanto, precisa alterar seus valores. Por isto passamos para a função o endereço da variável a ser modificada!

9.8 MATRIZES COMO ARGUMENTOS DE FUNÇÕES

A operação de passagem de matrizes, como argumentos, para funções, é uma exceção à convenção de passagem de parâmetros com chamada por valor.

Quando uma matriz é usada como um argumento para uma função, apenas o endereço da matriz é passado, não uma cópia da matriz inteira. Quando você chama uma função com um nome de matriz, um ponteiro para o primeiro elemento na matriz é passado para a função. (Não se esqueça: em C, um nome de matriz sem qualquer índice é um ponteiro para o primeiro elemento na matriz.) Isso significa que a declaração de parâmetros deve ser de um tipo de ponteiro compatível. Existem três maneiras de declarar um parâmetro que receberá um ponteiro para matriz. Primeiro, ele pode ser declarado como uma matriz, conforme mostrado aqui (para o caso de matrizes 1D, ou seja, vetores):


```
#include <stdio.h>

void display(int num[10], int n);

void main()
{
    int t[10], i;
    for(i=0; i<10; i++) t[i] = i;
    display(t, 10);
}

void display(int num[10], int n)
{
    int i;
    for(i=0; i<n; i++) printf("%d ", num[i]);
}
```

Muito embora o parâmetro `num` seja declarado como uma matriz de inteiros com 10 elementos, o compilador C converte-o automaticamente para um ponteiro de inteiros. Isso é necessário porque nenhum parâmetro pode realmente receber uma matriz inteira. Assim, como apenas um ponteiro para matriz é passado, um parâmetro de ponteiro deve estar lá para recebê-lo.

A segunda forma de declarar um parâmetro de matriz é especificá-lo como uma matriz sem dimensão, conforme mostrado aqui:

```
void display(int num[], int n)
{
    int i;
    for(i=0; i<n; i++) printf("%d ", num[i]);
}
```

Neste caso, `num` é declarado como uma matriz de inteiros de tamanho desconhecido. Como C não fornece nenhuma verificação de limites em matrizes, o tamanho real da matriz é irrelevante para o parâmetro (mas não para o programa, é claro). Esse método de declaração realmente define `num` como um ponteiro de inteiros.

A última forma em que `num` pode ser declarado — a mais comum em programas escritos profissionalmente em C — é como um ponteiro, conforme mostrado a seguir:

```
void display(int *num, int n)
{
    int i;
    for(i=0; i<n; i++) printf("%d ", num[i]);
}
```

Isso é permitido porque qualquer ponteiro pode ser indexado usando `[]`, como se fosse uma matriz. (Na verdade, como vimos na Seção 8.3, matrizes e ponteiros estão intimamente ligados.)

Observe que, em quaisquer das três formas de passar a matriz como parâmetro, um valor inteiro (`n`) também foi especificado. O motivo disto é que, como passamos apenas o endereço do primeiro elemento da matriz, a função não tem como adivinhar onde a região delimitada pela matriz termina. Em todos os exemplos citados existe um loop que varre a matriz toda. Sem um parâmetro que indique o fim do loop, a função pode gerar uma falha grave de acesso a memória. A exceção é quando estamos lidando com strings (vetores de `char`) já que, neste caso, há o caractere especial `'\0'` que indica o final da string.

É importante entender que, quando uma matriz é usada como um argumento para uma função, seu endereço é passado para a função. Isso é uma exceção à convenção de C no que diz respeito a passar parâmetros. Nesse caso, o código dentro da função está operando com, e potencialmente alterando, o conteúdo real da matriz usada para chamar a função.

A passagem de uma matriz 2D para uma função é similar à passagem de um vetor. O método de passagem do endereço da matriz para a função é idêntico, não importando quantas dimensões ela possua, já que sempre é passado o endereço da matriz.

Entretanto, na declaração da função, a matriz é um pouco diferente. A função deve receber o tamanho das dimensões a partir da segunda dimensão. Por exemplo:

```
void Determinante(double A[][5]);
```

Note que é fornecido a segunda dimensão da matriz. Isto é necessário para que, ao chamarmos o elemento `A[i][j]`, a função saiba a partir de que elemento ela deve mudar de linha. Com o número de elementos de cada linha, a função pode obter qualquer elemento multiplicando o número da linha pelo tamanho de cada linha e somando ao número da coluna. Por exemplo, o elemento `A[2][3]`, é o elemento de índice 13, já que $2 * 5 + 3 = 13$.

Outra forma de declarar uma função que receba uma matriz 2D é usando indireção múltipla de ponteiros, baseada na relação entre matrizes e ponteiros e ponteiros para ponteiros, assunto abordado no Capítulo 8. No entanto, **esse método é válido apenas quando a matriz é alocada dinamicamente**, conforme será abordado na seção 10.3.2.2. Assim sendo, poderia ser:

```
void Determinante(double **A, int M, int N);
```

Observe que, adicionalmente, foram passadas as dimensões da matriz, a fim de tornar a função geral para qualquer tamanho da matriz A.

9.9 OS ARGUMENTOS `argc` E `argv`

A função `main()` pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função `main()` é:

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

Os parâmetros `argc` e `argv` dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O `argc` (argument count) é um inteiro e possui o número de argumentos com os quais a função `main()` foi chamada na linha de comando. Ele é no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

O `argv` (argument values) é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O `argv[0]` sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em `argv` que temos `argc`.

Exemplo: Escreva um programa que faça uso dos parâmetros `argv` e `argc`. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame `data`:

```
data 19 04 99
```

O programa deverá imprimir:

```
19 de abril de 1999
```

```

#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    int mes;
    char *nomemes[] = {"Janeiro", "Fevereiro", "Março", "Abril", "Maio", "Junho", "Julho",
                       "Agosto", "Setembro", "Outubro", "Novembro", "Dezembro"};

    /* Testa se o numero de parametros fornecidos está correto o primeiro parametro é o
       nome do programa, o segundo o dia o terceiro o mes e o quarto os dois ultimos
       algarismos do ano */

    if(argc == 4)
    {
        mes = atoi(argv[2]); /* argv contém strings. A string referente ao mes deve ser
                               transformada em um número inteiro. A função atoi esta
                               sendo usada para isto: recebe a string e transforma
                               no inteiro equivalente */

        if (mes<1 || mes>12) /* Testa se o mes é válido */
            printf("Erro!\nUso: data dia mes ano, todos inteiros");
        else
            printf("\n%s de %s de 19%s", argv[1], nomemes[mes-1], argv[3]);
    }
    else printf("Erro!\nUso: data dia mes ano, todos inteiros");
}

```

9.10 RECURSÃO

Em C, funções podem chamar a si mesmas. A função é recursiva se um comando no corpo da função a chama. Recursão é o processo de definir algo em termos de si mesmo e é, algumas vezes, chamado de *definição circular*.

Um exemplo simples de função recursiva é `fatorialRec()`, que calcula o fatorial de um inteiro. O fatorial de um número n é o produto de todos os números inteiros entre 1 e n . Por exemplo, fatorial de 3 é $1 \times 2 \times 3$, ou seja, 6. Observe uma definição muitas vezes usada em matemática para fatorial (em simplificações, muitas vezes):

$$\begin{aligned}
 N! &= N * (N-1)! \\
 0! &= 1
 \end{aligned}$$

Esta é uma clara definição recursiva, já que o fatorial de um número está sendo definido como este número multiplicado pelo fatorial do inteiro anterior. O processo se repete até a base da recursão (onde a recursão “para”), que no caso do fatorial, é na definição de que o fatorial de 0 (zero) é igual a 1.

Tanto `fatorialRec()` como sua equivalente iterativa são mostradas aqui:

```

/* Versão Recursiva */
int fatorialRec(int n)
{
    if (n == 1) return 1; /* Base da Recursão */
    else return n * fatorialRec(n-1); /* chamada recursiva */
}

/* Versão Não-Recursiva (Iterativa) */
int fatorial(int n)
{
    int t, fn = 1;
    for (t = 1; t <= n; t++)
        fn = fn * t;
    return fn;
}

```

A versão não-recursiva para o cálculo do fatorial (`fatorial()`) deve ser clara. Ela usa um laço que é executado de 1 a `n` e multiplica progressivamente cada número pelo produto móvel.

A operação de `fatorialRec()` é um pouco mais complexa. Quando `fatorialRec()` é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto de `n*fatorialRec(n-1)`. Para avaliar essa expressão, `fatorialRec()` é chamada com `n-1`. Isso acontece até que `n` se iguale a 1 e as chamadas a função comecem a retornar.

Calculando o fatorial de 2, a primeira chamada a `fatorialRec()` provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original de `n`). A resposta, então, é 2. (Você pode achar interessante inserir comandos `printf()` em `fatorialRec()` para ver o nível de cada chamada e quais são as respostas intermediárias.) Uma observação interessante neste exemplo é que ele só será executado sem erros se `n` for relativamente pequeno. Isto por que o fatorial de um número cresce astronomicamente à medida que tal número cresce. Para que este exemplo funcione com números maiores, é preciso mudar o retorno da função fatorial recursiva para um tipo numérico mais amplo, como um `double`, por exemplo. A mesma observação vale para a variável `fn` na versão não-recursiva.

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada a função dentro da função.

A maioria das funções recursivas não minimiza significativamente o tamanho do código nem melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas a função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro da pilha. Como o armazenamento para os parâmetros da função e variáveis locais está na pilha e cada nova chamada cria uma nova cópia dessas variáveis, a pilha pode provavelmente escrever sobre outra memória de dados ou de programa. Contudo, você possivelmente nunca terá de se preocupar com isso, a menos que uma função recursiva seja executada de forma desenfreada.

A principal vantagem das funções recursivas é que você pode usá-las para criar versões mais claras e simples de vários algoritmos. Por exemplo, o QuickSort (um algoritmo de ordenação) é muito difícil de implementar numa forma iterativa. Além disso, alguns problemas, especialmente aqueles relacionados com inteligência artificial, resultaram em soluções recursivas. Finalmente, algumas pessoas parecem pensar recursivamente com mais facilidade que iterativamente.

Ao escrever funções recursivas, você deve ter um comando `if` em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se você não o fizer, a função nunca retornará quando chamada. Omitir o `if` é um erro comum quando se escrevem funções recursivas. Use `printf()` e `getchar()` deliberadamente durante o desenvolvimento do programa de forma que você possa ver o que está acontecendo e encerrar a execução se localizar um erro.

9.11 PONTEIROS PARA FUNÇÕES

Um recurso confuso, mas poderoso de C, é o ponteiro para função. Muito embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. O endereço de uma função é o ponto de entrada da função. Portanto, um ponteiro de função pode ser usado para chamar uma função.

Para entender como funcionam os ponteiros de funções, você deve conhecer um pouco como uma função é compilada e chamada em C. Primeiro, quando cada função é compilada, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido. Quando é feita uma chamada a função, enquanto seu

programa está sendo executado, é efetuada uma chamada em linguagem de máquina para esse ponto de entrada. Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, ele pode ser usado para chamar essa função.

O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos. (Isso é semelhante a maneira como o endereço de uma matriz é obtido quando apenas o nome da matriz, sem índices, é usado.) Um ponteiro para uma função tem a seguinte declaração:

```
tipo_de_retorno (*nome_do_ponteiro)();
```

ou

```
tipo_de_retorno (*nome_do_ponteiro)(declaração_de_parâmetros);
```

Repare nos parênteses que devem ser colocados obrigatoriamente. Se declaramos:

```
tipo_de_retorno * nome(declaração_de_parâmetros);
```

estariamos, na realidade, declarando uma função que retornaria um ponteiro para o tipo especificado.

Porém, não é obrigatório se declarar os parâmetros da função. Veja um exemplo do uso de ponteiros para funções:

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

void PrintString(char *str, int (*func)(const char *));

void main (void)
{
    char String [20]= "Curso de C.";
    int (*p)(const char *); /* Declaracao do ponteiro para função
                             Funcao apontada é inteira e recebe como parametro
                             uma string constante */
    p=puts;                 /* O ponteiro p passa a apontar para a função puts
                             que tem o seguinte prototipo: int puts(const char *) */
    PrintString (String, p); /* O ponteiro é passado como parametro para PrintString */
}

void PrintString(char *str, int (*func)(const char *))
{
    (*func)(str); /* chamada a função através do ponteiro para função */
    func(str);    /* maneira também válida de se fazer a chamada a função puts
                  através do ponteiro para função func */
}
```

Veja que fizemos a atribuição de `puts` a `p` simplesmente usando:

```
p = puts;
```

Disto, concluímos que o nome de uma função (sem os parênteses) é, na realidade, o endereço daquela função! Note, também, as duas formas alternativas de se chamar uma função através de um ponteiro. No programa acima, fizemos esta chamada por:

```
(*func) (str);
```

e

```
func(str);
```

Estas formas são equivalentes entre si.

Além disto, no programa, a função `PrintString()` usa uma função qualquer `func` para imprimir a string na tela. O programador pode então fornecer não só a string mas também a função que será usada para imprimi-la. No `main()` vemos como podemos atribuir, ao ponteiro para funções `p`, o endereço da função `puts()` do C.

Em síntese, ao declarar um ponteiro para função, podemos atribuir a este ponteiro o endereço de uma função e podemos também chamar a função apontada através dele. **Não** podemos fazer algumas coisas que fazíamos com ponteiros “normais”, como, por exemplo, incrementar ou decrementar um ponteiro para função.

9.12 ALGUMAS FUNÇÕES DE ENTRADA/SAÍDA PADRONIZADAS

O sistema de entrada e saída da linguagem C está estruturado na forma de uma biblioteca de funções. Já vimos algumas destas funções, e agora elas serão reestudadas. Novas funções também serão apresentadas.

Não é objetivo deste curso explicar, em detalhes, todas as possíveis funções da biblioteca de entrada e saída do C. A sintaxe completa destas funções pode ser encontrada no manual do seu compilador. Alguns sistemas trazem uma descrição das funções na ajuda do compilador, que pode ser acessada “on-line”. O Microsoft Visual C++, por exemplo, dispõe de um help bastante completo.

Um ponto importante é que agora, quando apresentarmos uma função, vamos, em primeiro lugar, apresentar o seu protótipo. Você já deve ser capaz de interpretar as informações que um protótipo nos passa. Se não, deve voltar a estudar a seção a respeito de protótipos de funções na Seção 9.4.

9.12.1 LENDO E ESCRIVENDO CARACTERES

Uma das funções mais básicas de um sistema é a entrada e saída de informações em dispositivos. Estes podem ser um monitor, uma impressora ou um arquivo em disco. Vamos ver os principais comandos que o C nos fornece para isto.

9.12.1.1 `getche()` e `getch()`

As funções `getch()` e `getche()` não são definidas pelo padrão ANSI. Porém, elas geralmente são incluídas em compiladores baseados no DOS, e se encontram no header file `conio.h`. Vale a pena repetir: são funções comuns apenas para compiladores baseados em DOS e, se você estiver no UNIX normalmente não terá estas funções disponíveis.

Protótipos:

```
int getch(void);
int getche(void);
```

`getch()` espera que o usuário digite uma tecla e retorna este caractere. Você pode estar estranhando o fato de `getch()` retornar um inteiro, mas não há problema pois este inteiro é tal que quando igualado a um `char` a conversão é feita corretamente. A função `getche()` funciona exatamente como `getch()`. A diferença é que `getche()` gera um “echo” na tela antes de retornar a tecla.

Se a tecla pressionada for um caractere especial estas funções retornam zero. Neste caso você deve usar as funções novamente para pegar o código da tecla estendida pressionada.

A função equivalente a `getche()` no mundo ANSI é o `getchar()`. O problema com `getchar()` é que o caractere lido é colocado em uma área intermediária até que o usuário digite um <ENTER>, o que pode ser extremamente inconveniente em ambientes interativos.

9.12.1.2 putchar()

Protótipo:

```
int putchar(int c);
```

`putchar()` coloca o caractere `c` na tela. Este caractere é colocado na posição atual do cursor. Mais uma vez os tipos são inteiros, mas você não precisa se preocupar com este fato. O header file é `stdio.h`.

9.12.2 LENDO E ESCRIVENDO STRINGS

9.12.2.1 gets()

Protótipo:

```
char *gets(char *s);
```

Pede ao usuário que entre uma string, que será armazenada na string `s`. O ponteiro que a função retorna é o próprio `s`. `gets()` não é uma função segura. Por quê? Simplesmente porque com `gets()` pode ocorrer um estouro da quantidade de posições que foi especificada na string. Veja o exemplo abaixo:

```
#include <stdio.h>
void main()
{
    char buffer[20];
    printf("Digite o nome da instituicao: ");
    gets(buffer);
    printf("O nome eh: %s", buffer);
}
```

Se o usuário digitar como entrada:

```
Instituto Federal de Educacao Tecnologica
```

ou seja, digitar um total de 41 caracteres: 42 posições (incluindo o `'\0'`) serão utilizadas para armazenar a string. Como a string `buffer[]` só tem 20 caracteres, os 22 caracteres adicionais serão colocados na área de memória subsequente à ocupada por ela, escrevendo uma região de memória que não está reservada à string. Este efeito é conhecido como “*estouro de buffer*” e pode causar problemas imprevisíveis. Uma forma de se evitar este problema é usar a função `fgets()`, conforme veremos posteriormente (Seção 11.5.2).

9.12.2.2 puts()

Protótipo:

```
int puts(char *s);
```

`puts()` coloca a string `s` na tela.

9.12.3 ENTRADA E SAÍDA FORMATADA

As funções que resumem todas as funções de entrada e saída formatada no C são as funções `printf()` e `scanf()`. O domínio destas funções é fundamental ao programador. Apesar de terem sido amplamente usadas até então, tais funções são muito mais poderosas e customizáveis que você pode imaginar. Por isto, elas serão mais detalhadamente abordadas aqui.

9.12.3.1 printf()

Protótipo:

```
int printf(char *str,...);
```

As reticências no protótipo da função indicam que esta função tem um número de argumentos variável. Este número está diretamente relacionado com a string de controle *str*, que deve ser fornecida como primeiro argumento. A string de controle tem dois componentes. O primeiro são caracteres a serem impressos na tela. O segundo são os comandos de formato (também chamados de *format tags*). Como já vimos, os últimos determinam uma exibição de variáveis na saída. Os comandos de formato são precedidos de %. A cada comando de formato deve corresponder um argumento na função `printf()`. Se isto não ocorrer podem acontecer erros imprevisíveis no programa.

Abaixo apresentamos a tabela de códigos de formato, onde as partes entre `[]` são consideradas opcionais:

```
%[flags][width][.precision][length]specifier
```

O *specifier* é o mais significativo e define o tipo e a interpretação do valor do correspondente argumento.

specifier	Saída	Exemplo
c	Caractere	a
d ou i	Decimal inteiro com sinal	392
e	Notação científica (mantissa/expoente) usando o caractere e	3.9265e+2
E	Notação científica (mantissa/expoente) usando o caractere E	3.9265E+2
f	Decimal de Ponto Flutuante	392.65
g	Usa o mais curto de %e ou %f	392.65
G	Usa o mais curto de %E ou %f	392.65
o	Octal com sinal	610
s	String de caracteres	sample
u	Decimal inteiro sem sinal	7235
x	Hexadecimal inteiro sem sinal	7fa
X	Hexadecimal inteiro sem sinal (letras maiúsculas)	7FA
p	Endereço ponteiro	B800:0000
n	Nada é impresso. O argument deve ser um ponteiro para um <code>signed int</code> , onde o número de caracteres escritos até então é armazenado.	
%	Um % seguido de outro caractere % vai escrever % na saída.	

O comando pode conter *flags*, *width*, *.precision* e *modifiers*, os quais são opcionais e especificados abaixo.

flags	Descrição
-	Alinhamento à esquerda dentro dos limites especificados pelo campo width; Alinhamento a direita é o default.
+	Força o resultado a mostrar o sinal de mais ou menos (+ or -) mesmo para números positivos. Por default, somente números negativos são precedidos por um sinal de menos.
(space)	Se nenhum sinal será escrito, um espaço em branco é inserido antes do valor.

#	<p>Usado com os specifiers <code>o</code>, <code>x</code> ou <code>X</code> o valor é precedido com <code>0</code>, <code>0x</code> ou <code>0X</code> respectivamente para valores diferentes de zero.</p> <p>Usado com <code>e</code>, <code>E</code> ou <code>f</code>, ele força a saída escrita a conter um ponto decimal mesmo se nenhum dígito deveria seguir (ou seja, a parte decimal é zero). Por padrão, se a parte decimal é zero, o ponto de decimal não é escrito.</p> <p>Usado com <code>g</code> ou <code>G</code>, o resultado é o mesmo que com <code>e</code> ou <code>E</code>, mas zeros à direita não são removidos.</p>
0	Preenche a esquerda o número com zeros no lugar de espaços, onde o <code>width</code> é especificado.

width	Descrição
(number)	Número mínimo de caracteres a serem impressos. Se o valor a ser impresso é menor que este número, o resultado é preenchido com espaços em branco. O valor não é truncado mesmo se o resultado é maior.
*	O <code>width</code> não é especificado na <code>string format</code> , mas como um valor inteiro adicional precedendo o argumento que deve ser formatado. Em outras palavras, isto permite que o <code>width</code> seja uma variável, da mesma forma que o valor a ser impresso.

.precision	Descrição
.number	<p>Para specifiers inteiros (<code>d</code>, <code>i</code>, <code>o</code>, <code>u</code>, <code>x</code>, <code>X</code>): <code>precision</code> especifica o número mínimo de dígitos a serem escritos. Se o valor a ser escrito é mais curto que este número, o resultado é “preenchido” com zeros. O valor não é truncado mesmo se o resultado é mais longo. Uma <code>precision</code> de <code>0</code> significa que nenhum caractere é escrito para o valor <code>0</code>.</p> <p>Para specifiers <code>e</code>, <code>E</code> e <code>f</code>: este é o número de dígitos a serem impressos depois do ponto decimal.</p> <p>Para specifiers <code>g</code> e <code>G</code>: Este é o número máximo de dígitos significantes a serem impressos.</p> <p>Para <code>s</code>: este é o número máximo de caracteres a serem impressos. Por padrão, todos os caracteres são impressos até que o caractere que indica o final da <code>string</code> é encontrado.</p> <p>Para <code>c</code>: não tem qualquer efeito.</p> <p>Quando nenhum <code>precision</code> é especificado, o valor padrão é <code>1</code>. Se o período é especificado sem um valor explícito para <code>precision</code>, <code>0</code> é assumido.</p>
.*	O <code>precision</code> não é especificado na <code>string de formatação</code> , mas como um argumento inteiro adicional precedendo o argumento que deve ser formatado.

length	Descrição
h	O argumento é interpretado como um <code>short int</code> ou um <code>unsigned short int</code> (somente se aplica para specifiers inteiros: <code>i</code> , <code>d</code> , <code>o</code> , <code>u</code> , <code>x</code> e <code>X</code>).
l	O argumento é interpretado como um <code>long int</code> ou <code>unsigned long int</code> para specifiers inteiros (<code>i</code> , <code>d</code> , <code>o</code> , <code>u</code> , <code>x</code> e <code>X</code>), e como um caractere estendido (<code>wide char</code>) ou cadeia de caracteres estendidos para os specifiers <code>c</code> e <code>s</code> .
L	O argumento é interpretado como um <code>long double</code> (aplica-se somente a especificadores de ponto flutuante: <code>e</code> , <code>E</code> , <code>f</code> , <code>g</code> e <code>G</code>).

Exemplos:

```
#include <stdio.h>
void main()
{
    printf("Characters: %c %c \n", 'a', 65);
    printf("Decimals: %d %ld\n", 1977, 650000L);
    printf("Preceding with blanks: %10d \n", 1977);
    printf("Preceding with zeros: %010d \n", 1977);
    printf("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
    printf("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
    printf("Width trick: %*d \n", 5, 10);
    printf("%s \n", "A string");
}
```

E a saída:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:      1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
width trick:    10
A string
```

9.12.3.2 scanf()

Protótipo:

```
int scanf(char *str,...);
```

A string de controle *str* determina, assim como com a função `printf()`, quantos parâmetros a função vai necessitar. Devemos sempre nos lembrar que a função `scanf()` deve receber ponteiros como parâmetros. Isto significa que as variáveis que não sejam por natureza ponteiros devem ser passadas precedidas do operador `&`. Os especificadores de formato de entrada são muito parecidos com os de `printf()`.

Código	Formato
%c	Um único caracter (char)
%d	Um número decimal (int)
%i	Um número inteiro
%hi	Um short int
%li	Um long int
%e	Um ponto flutuante
%f	Um ponto flutuante
%lf	Um double
%h	Inteiro curto
%o	Número octal
%s	String
%x	Número hexadecimal
%p	Ponteiro

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

```
/* exemplo de scanf */
#include <stdio.h>

int main ()
{
    char str[80];
    int i;

    printf("Digite seu Primeiro Nome: ");
    scanf("%s", str);
    printf("Digite sua idade: ");
    scanf("%d", &i);
    printf("Sr(a). %s , %d anos de idade.\n", str, i);
    printf("Digite um numero hexadecimal: ");
    scanf("%x", &i);
    printf("Voce digitou %#x (%d).\n",i,i);

    return 0;
}
```

Iniciantes muitas vezes queixam-se do `scanf()` ao tentar ler uma string contendo espaços. O fato é que o caracter ‘espaço’ naturalmente é usado como delimitador, fazendo com que a função pare de ler ao encontrar o primeiro espaço em branco. No entanto, existe uma sequência de controle mais flexível que o `%s`: o `%[]`. Graças a esta sequência de controle, podemos escolher o que queremos ler. Dentro dos parênteses escrevemos os caracteres permitidos (ex: `%[aeiou]`) ou negados (ex: `%[^aeiou]`). Para lermos uma string podemos usar, por exemplo, `%[^\n]`, que vai ler todos os caracteres até encontrar o `'\n'` neste caso (o `'\n'` não é incluído na string). A sintaxe deste especificador funciona de modo semelhante às expressões regulares POSIX. Por exemplo:

```
#include <stdio.h>

main()
{
    char str2[100];

    printf("Digite seu nome completo: ");
    scanf("%[^\n]", str2);

    printf("Ola, %s\n\n", str2);
}
```

Neste exemplo, o `scanf()` lerá até que encontre o `'\n'`, ou seja, uma mudança de linha (que é o que acontece quando o usuário digita ENTER no teclado).

Se quisermos limitar o tamanho da string, podemos limitá-la antes do conjunto de caracteres a ler. Por exemplo, se quiséssemos ler uma string com 60 caracteres no máximo, poderíamos usar a string de controle: `%60[^\n]`.

Segue outro exemplo:

```
/* Ler os dados */
scanf("%[^\n]", str);

/* Limpar o buffer */
scanf("%*[^\n]"); scanf("%*c");
```

No exemplo acima, após ler a string para `str`, o buffer de entrada é limpo usando as duas chamadas a função `scanf()`, até aí nenhuma novidade. O “truque” novo (e útil) encontra-se na limpeza do buffer, produzido pelas duas chamadas consecutivas à função `scanf()`. A primeira chamada lê e descarta, ou seja, retira todos os caracteres que eventualmente existam no buffer de entrada, até ao aparecimento do caractere `'\n'` (Um ```*` iniciando a string de formatação indica que o dado deve ser lido a partir de `stdin` mas ignorado, isto é, ele não é armazenado no argumento correspondente). A segunda chamada lê e descarta um caractere apenas que, devido à instrução anterior, é o caractere `'\n'`. Logo, recorrendo a estas duas instruções, todos os caracteres da linha que estava sendo processada foram descartados. Consequentemente, foi lida uma linha completa do buffer de entrada e a posição de leitura ficou colocada no início da linha seguinte.

9.12.3.3 `sprintf()` e `sscanf()`

`sprintf()` e `sscanf()` são semelhantes a `printf()` e `scanf()`. Porém, ao invés de escreverem na saída padrão ou lerem da entrada padrão, escrevem ou leem em uma string. Os protótipos são:

```
int sprintf(char *destino, char *controle, ...);
int sscanf(char *destino, char *controle, ...);
```

Estas funções são muito utilizadas para fazer a conversão entre dados na forma numérica e sua representação na forma de strings. No programa abaixo, por exemplo, a variável `i` é “impressa” em `string1`. Além da representação de `i` como uma string, `string1` também conterá “Valor de i=”.

```
#include <stdio.h>

void main()
{
    int i;
    char string1[20];
    printf("Entre um valor inteiro: ");
    scanf("%d", &i);
    sprintf(string1, "Valor de i = %d", i);
    puts(string1);
}
```

Já no programa abaixo, foi utilizada a função `sscanf()` para converter a informação armazenada em `string1` em seu valor numérico:

```
#include <stdio.h>

void main()
{
    int i, j, k;
    char string1[] = "10 20 30";
    sscanf(string1, "%d %d %d", &i, &j, &k);
    printf("Valores lidos: %d, %d, %d", i, j, k);
}
```

Cap. 10: TIPOS DE DADOS AVANÇADOS E DEFINIDOS PELO USUÁRIO

Neste capítulo abordaremos conceitos avançados a respeito da definição, criação e manipulação de dados permitidos pela linguagem C, em destaque estão as *estruturas* e os mecanismos de *alocação dinâmica* de memória.

10.1 MODIFICADORES DE ACESSO

Já vimos que uma variável é declarada como

```
tipo_da_variável lista_de_variáveis;
```

Vimos também que existem modificadores de tipos. Estes modificam o tipo da variável declarada. Destes, já vimos os modificadores `signed`, `unsigned`, `long`, e `short`. Estes modificadores são incluídos na declaração da variável da seguinte maneira:

```
modificador_de_tipo tipo_da_variável lista_de_variáveis;
```

Vamos discutir agora outros modificadores de tipo, usados em algumas situações especiais. Estes modificadores, denominados *modificadores de acesso*, como o próprio nome indica, mudam a maneira com a qual a variável é acessada e modificada.

10.1.1 const

O modificador **const** faz com que a variável não possa ser modificada no programa. Como o nome já sugere é útil para se declarar constantes. Poderíamos ter, por exemplo:

```
const float PI = 3.1416;
```

Podemos ver pelo exemplo que as variáveis com o modificador `const` podem ser inicializadas. Mas `PI` não poderia ser alterado em qualquer outra parte do programa. Se o programador tentar modificar `PI` o compilador gerará um erro de compilação.

O uso mais importante de `const` não é declarar variáveis constantes no programa. Seu uso mais comum é evitar que um parâmetro de uma função seja alterado pela função. Isto é muito útil no caso de um ponteiro, pois o conteúdo de um ponteiro pode ser alterado por uma função. Para tanto, basta declarar o parâmetro como `const`. Veja o exemplo:

```
#include <stdio.h>

int sqr(const int *num);

void main() {
    int a=10;
    int b;
    b = sqr(&a);
}

int sqr(const int *num) {
    return ((*num)*(*num));
}
```

No exemplo, `num` está protegido contra alterações. Isto quer dizer que, se tentássemos fazer

```
*num = 10;
```

dentro da função `sqr()` o compilador daria uma mensagem de erro.

10.1.2 volatile

O modificador `volatile` diz ao compilador que a variável em questão pode ser alterada sem que este seja avisado. Isto evita “bugs” seríssimos. Digamos que, por exemplo, tenhamos uma variável que o BIOS do computador altera de minuto em minuto (um relógio, por exemplo). Seria muito bom que declarássemos esta variável como sendo `volatile`.

10.2 CONVERSÃO DE TIPOS

Em atribuições no C temos o seguinte formato:

```
destino = origem;
```

Se o *destino* e a *origem* são de tipos diferentes o compilador faz uma conversão entre os tipos. Nem todas as conversões são possíveis. O primeiro ponto a ser ressaltado é que o valor de origem é convertido para o valor de destino antes de ser atribuído e não o contrário.

É importante lembrar que quando convertemos um tipo numérico para outro nós nunca *ganhamos* precisão. Nós podemos perder precisão ou no máximo manter a precisão anterior. Isto pode ser entendido de outra forma. Quando convertemos um número não estamos introduzindo no sistema nenhuma informação adicional. Isto implica que nunca vamos *ganhar* precisão.

Abaixo vemos uma tabela de conversões numéricas com *perda* de precisão, para um compilador com palavra de 16 bits:

De	Para	Informação Perdida
<code>unsigned char</code>	<code>char</code>	Valores maiores que 127 são alterados
<code>short int</code>	<code>char</code>	Os 8 bits de mais alta ordem
<code>int</code>	<code>char</code>	Os 8 bits de mais alta ordem
<code>long int</code>	<code>char</code>	Os 24 bits de mais alta ordem
<code>long int</code>	<code>short int</code>	Os 16 bits de mais alta ordem
<code>long int</code>	<code>int</code>	Os 16 bits de mais alta ordem
<code>float</code>	<code>int</code>	Precisão - resultado arredondado
<code>double</code>	<code>float</code>	Precisão - resultado arredondado
<code>long double</code>	<code>double</code>	Precisão - resultado arredondado

10.3 ALOCAÇÃO DINÂMICA DE MEMÓRIA

No Capítulo 8, os aspectos fundamentais dos ponteiros em C foram estudados. No entanto, uma das vantagens mais importantes dos ponteiros é fornecer o suporte necessário para o poderoso sistema de alocação dinâmica de C, assunto importantíssimo e que será estudado nesta seção.

Alocação dinâmica é o meio pelo qual um programa pode obter memória enquanto está em execução. Como você sabe, variáveis globais tem o armazenamento alocado em tempo de compilação. Variáveis locais

usam a pilha. No entanto, nem variáveis globais nem locais podem ser acrescentadas durante o tempo de execução. Porém, haverá momentos em que um programa precisará usar quantidades de armazenamento variáveis. Por exemplo, um processador de texto ou um banco de dados aproveita toda a RAM de um sistema. Porém, como a quantidade de RAM varia entre computadores esses programas não poderão usar variáveis normais. Em vez disso, esses e outros programas alocam memória conforme necessário, usando as funções do sistema de alocação dinâmica de C.

A memória alocada pelas funções de alocação dinâmica de C é obtida do *heap* — a região de memória livre que está entre seu programa e a área de armazenamento permanente e a pilha. Embora o tamanho do *heap* seja desconhecido, ele geralmente contém uma quantidade razoavelmente grande de memória livre.

O coração do sistema de alocação dinâmica de C consiste nas funções `malloc()` e `free()`. (Na verdade, C tem diversas outras funções de alocação dinâmica, mas essas duas são as mais importantes.) Essas funções operam em conjunto, usando a região de memória livre para estabelecer e manter uma lista de armazenamento disponível. A função `malloc()` aloca memória e `free()` a libera. Isto é, cada vez que é feita uma solicitação de memória por `malloc()`, uma porção da memória livre restante é alocada. Cada vez que é efetuada uma chamada a `free()` para liberação de memória, a memória é devolvida ao sistema. Qualquer programa que use essas funções deve incluir o cabeçalho `stdlib.h`.

10.3.1 FUNÇÕES DE ALOCAÇÃO DINÂMICA DO C

Examinaremos agora as funções de alocação dinâmica do C, definidas em `stdlib.h`.

10.3.1.1 malloc()

A função `malloc()` serve para alocar memória e tem o seguinte protótipo:

```
void *malloc(unsigned int num);
```

A função toma o número de bytes que queremos alocar (*num*), aloca na memória e retorna um ponteiro `void*` para o primeiro byte alocado. O ponteiro `void*` pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função `malloc()` retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com `malloc()`:

```
#include <stdio.h>
#include <stdlib.h> /* Para usar malloc() */

main()
{
    int *p;
    int a;
    int i;

    ... /* Determina o valor de a em algum lugar */

    p = (int*)malloc(a*sizeof(int)); /* Aloca a números inteiros; p pode agora ser tratado como
                                     um vetor com a posicoes */

    if (!p)
    {
        printf("*** Erro: Memoria Insuficiente ***");
        exit(1);
    }

    for (i=0; i<a ; i++) /* p pode ser tratado como um vetor com a posicoes */
        p[i] = i*i;

    ...
}
```

No exemplo acima, é alocada memória suficiente para se armazenar *a* números inteiros. O operador `sizeof()` retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro `void*` que `malloc()` retorna é convertido para um `int*` pelo cast e é atribuído a `p`. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, `p` terá um valor nulo, o que fará com que `!p` retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de `p[0]` a `p[(a-1)]`.

10.3.1.2 calloc()

A função `calloc()` também serve para alocar memória, mas possui um protótipo um pouco diferente:

```
void *calloc(unsigned int num, unsigned int size);
```

A função aloca uma quantidade de memória igual a `num * size`, isto é, aloca memória suficiente para um vetor de `num` objetos de tamanho `size`. Retorna um ponteiro `void*` para o primeiro byte alocado. O ponteiro `void*` pode ser atribuído a qualquer tipo de ponteiro. Se não houver memória suficiente para alocar a memória requisitada a função `calloc()` retorna um ponteiro nulo. Em relação a `malloc`, `calloc` tem uma diferença (além do fato de ter protótipo diferente): `calloc` inicializa o espaço alocado com 0. Veja um exemplo de alocação dinâmica com `calloc()`:

```
#include <stdio.h>
#include <stdlib.h> /* Para usar calloc() */

main()
{
    int *p;
    int a;
    int i;

    ... /* Determina o valor de a em algum lugar */

    p = (int*)calloc(a, sizeof(int)); /* Aloca a números inteiros
                                     p pode agora ser tratado como um vetor com
                                     a posicoes */

    if (!p)
    {
        printf("*** Erro: Memoria Insuficiente ***");
        exit(1);
    }

    for (i=0; i<a ; i++) /* p pode ser tratado como um vetor com a posicoes */
        p[i] = i*i;

    ...
}
```

No exemplo acima, é alocada memória suficiente para se colocar *a* números inteiros. O operador `sizeof()` retorna o número de bytes de um inteiro. Ele é útil para se saber o tamanho de tipos. O ponteiro `void*` que `calloc()` retorna é convertido para um `int*` pelo cast e é atribuído a `p`. A declaração seguinte testa se a operação foi bem sucedida. Se não tiver sido, `p` terá um valor nulo, o que fará com que `!p` retorne verdadeiro. Se a operação tiver sido bem sucedida, podemos usar o vetor de inteiros alocados normalmente, por exemplo, indexando-o de `p[0]` a `p[(a-1)]`.

10.3.1.3 realloc()

A função `realloc()` serve para realocar memória e tem o seguinte protótipo:

```
void *realloc(void *ptr, unsigned int num);
```


A função modifica o tamanho da memória previamente alocada apontada por `*ptr` para aquele especificado por `num`. O valor de `num` pode ser maior ou menor que o original. Um ponteiro para o bloco é devolvido porque `realloc()` pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida. Se `ptr` for nulo, aloca `num` bytes e devolve um ponteiro; se `num` é zero, a memória apontada por `ptr` é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.

```
#include <stdio.h>
#include <stdlib.h> /* Para usar malloc() e realloc() */

main()
{
    int *p;
    int a;
    int i;

    a = 30;

    p = (int*)malloc(a*sizeof(int)); /* Aloca a números inteiros. p pode agora ser
                                     tratado como um vetor com a posicoes */

    if (!p) {
        printf("*** Erro: Memoria Insuficiente ***");
        exit(1);
    }

    for (i=0; i<a; i++) /* p pode ser tratado como um vetor com a posicoes */
        p[i] = i*i;

    /* O tamanho de p deve ser modificado, por algum motivo ... */
    a = 100;
    p = (int*)realloc(p, a*sizeof(int));

    for (i=0; i<a ; i++) /* p pode ser tratado como um vetor com a posicoes */
        p[i] = a*i*(i-6);
}
```

10.3.1.4 free()

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função `free()` cujo protótipo é:

```
void free(void *p);
```

Basta então passar para `free()` o ponteiro que aponta para o início da memória alocada. Mas você pode se perguntar: como é que o programa vai saber quantos bytes devem ser liberados? Ele sabe, pois, quando você alocou a memória, ele guardou o número de bytes alocados numa “tabela de alocação” interna. Vamos reescrever o exemplo usado para a função `malloc()` usando o `free()` também agora:

```
#include <stdio.h>
#include <stdlib.h> /* Para usar malloc() e free() */

main()
{
    int *p;
    int a;

    ...

    p = (int*)malloc(a*sizeof(int));
    if (!p) {
        printf("*** Erro: Memoria Insuficiente ***");
        exit(1);
    }
}
```

```

...
free(p);
...
}

```

10.3.2 ALOCAÇÃO DINÂMICA DE VETORES E MATRIZES

10.3.2.1 ALOCAÇÃO DINÂMICA DE VETORES

A alocação dinâmica de vetores utiliza os conceitos aprendidos sobre ponteiros e as funções de alocação dinâmica apresentadas. Um exemplo de implementação para vetor real é fornecido a seguir:

```

#include <stdio.h>
#include <stdlib.h>

float *Alocar_vetor_real(int n)
{
    float *v;          /* ponteiro para o vetor */
    if (n < 1) {        /* verifica parâmetros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (NULL);
    }
    /* aloca o vetor */
    v = calloc(n, sizeof(float));
    if (v == NULL) {
        printf ("** Erro: Memoria Insuficiente **");
        return (NULL);
    }
    return (v);        /* retorna o ponteiro para o vetor */
}

float *Liberar_vetor_real(float *v)
{
    if (v == NULL) return (NULL);
    free(v);           /* libera o vetor */
    return (NULL);     /* retorna o ponteiro */
}

void main(void)
{
    float *p;
    int a;
    ... /* outros comandos, inclusive a inicializacao de a */
    p = Alocar_vetor_real(a);
    ... /* outros comandos, utilizando p[] normalmente */
    p = Liberar_vetor_real(p);
}

```

10.3.2.2 ALOCAÇÃO DINÂMICA DE MATRIZES

A alocação dinâmica de memória para matrizes é realizada da mesma forma que para vetores, com a diferença que teremos um ponteiro apontando para outro ponteiro que aponta para o valor final, ou seja, é um ponteiro para ponteiro, o que é denominado *indireção múltipla*. A indireção múltipla pode ser levada a qualquer dimensão desejada, mas raramente é necessário mais de um ponteiro para um ponteiro. Um exemplo de implementação para matriz real bidimensional é fornecido a seguir. A estrutura de dados utilizada neste exemplo é composta por um vetor de ponteiros (correspondendo ao primeiro índice da matriz), sendo que cada ponteiro aponta para o início de uma linha da matriz. Em cada linha existe um vetor alocado dinamicamente, como descrito anteriormente (compondo o segundo índice da matriz).

```

#include <stdio.h>
#include <stdlib.h>

float **Alocar_matriz_real(int m, int n)
{
    float **v; /* ponteiro para a matriz */
    int i; /* variável auxiliar */

    if (m < 1 || n < 1) { /* verifica parametros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (NULL);
    }

    /* aloca as linhas da matriz */
    v = calloc(m, sizeof(float*)); /* Um vetor de m ponteiros para float */
    if (v == NULL) {
        printf("** Erro: Memoria Insuficiente **");
        return (NULL);
    }

    /* aloca as colunas da matriz */
    for ( i = 0; i < m; i++ ) {
        v[i] = calloc(n, sizeof(float)); /* m vetores de n floats */
        if (v[i] == NULL) {
            printf("** Erro: Memoria Insuficiente **");
            return (NULL);
        }
    }

    return (v); /* retorna o ponteiro para a matriz */
}

float **Liberar_matriz_real(int m, int n, float **v)
{
    int i; /* variável auxiliar */

    if (v == NULL) return (NULL);

    if (m < 1 || n < 1) { /* verifica parâmetros recebidos */
        printf ("** Erro: Parametro invalido **\n");
        return (v);
    }

    for (i=0; i<m; i++) free (v[i]); /* libera as linhas da matriz */

    free (v); /* libera a matriz (vetor de ponteiros) */

    return (NULL); /* retorna um ponteiro nulo */
}

void main (void)
{
    float **mat; /* matriz a ser alocada */
    int l, c; /* número de linhas e colunas da matriz */
    int i, j;
    ... /* outros comandos, inclusive inicializacao para l e c */

    mat = Alocar_matriz_real(l, c);

    for (i = 0; i < l; i++)
        for ( j = 0; j < c; j++)
            mat[i][j] = i+j;

    ... /* outros comandos utilizando mat[][] normalmente */

    mat = Liberar_matriz_real(l, c, mat);

    ...
}

```

10.4 ESTRUTURAS

Em C, uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas. Uma *definição de estrutura* forma um modelo que pode ser usado para criar variáveis de estruturas. As variáveis que compreendem a estrutura são chamadas membros da estrutura. (Os membros da estrutura são comumente chamados *elementos* ou *campos*.)

Geralmente, todos os elementos na estrutura são logicamente relacionados. Pense, por exemplo, no documento de identidade de um cidadão. Ele contém, principalmente: nome completo, data de nascimento, data de emissão, filiação (pai e mãe), CPF e naturalidade. O fragmento de código seguinte mostra uma possível descrição para a estrutura que contenha todos os campos acima. A palavra-chave `struct` informa ao compilador que um modelo de estrutura está sendo definido.

```
struct RegGeral
{
    char nome[50];
    char dataNascimento[11];
    char dataEmissao[11];
    char nomePai[50];
    char nomeMae[50];
    char CPF[13];
    char naturalidade[20];
};
```

Observe que a definição termina com um ponto-e-virgula. Isso ocorre porque uma definição de estrutura é um comando. Além disso, o identificador (tag) da estrutura `RegGeral` indica essa estrutura de dados particular e é o seu especificador de tipo.

Nesse ponto do código, nenhuma variável foi de fato declarada. Apenas a forma dos dados foi definida. Para declarar uma variável de tipo `RegGeral`, escreva

```
struct RegGeral registro;
```

Isso declara uma variável do tipo `struct RegGeral` chamada `registro`. Quando você define uma estrutura, está essencialmente definindo um tipo complexo de variável, não uma variável. Não existe uma variável desse tipo até que ela seja realmente declarada.

Quando uma variável do tipo estrutura é declarada, o compilador C aloca automaticamente memória suficiente para acomodar todos os seus membros.

Você também pode declarar uma ou mais variáveis ao definir a estrutura. Por exemplo,

```
struct RegGeral
{
    char nome[50];
    char dataNascimento[11];
    char dataEmissao[11];
    char nomePai[50];
    char nomeMae[50];
    char CPF[13];
    char naturalidade[20];
} reg1, reg2, reg3;
```

define uma estrutura chamada `RegGeral` e declara as variáveis `reg1`, `reg2` e `reg3` desse tipo.

Se você precisa de apenas uma variável estrutura, o nome da estrutura não é necessário. Isso significa que a estrutura abaixo definida declara uma variável chamada `registro` como definido pela estrutura que a precede:

```
struct
{
    char nome[50];
    char dataNascimento[11];
    char dataEmissao[11];
    char nomePai[50];
    char nomeMae[50];
    char CPF[13];
    char naturalidade[20];
} registro;
```

A forma geral de uma definição de estrutura é

```
struct identificador {
    tipo nome_da_variável;
    tipo nome_da_variável;
    tipo nome_da_variável;
    .
    .
    .
} variáveis_estrutura;
```

onde *identificador* ou *variáveis_estrutura* podem ser omitidos, mas não ambos.

10.4.1 REFERENCIANDO ELEMENTOS DE ESTRUTURAS

Elementos individuais de estruturas são referenciados por meio do operador '.' (algumas vezes chamado de *operador ponto*). Por exemplo, o código seguinte atribui um valor ao campo `nomePai`, da variável estrutura `registro` declarada anteriormente:

```
strcpy(registro.nomePai, "Jose da Silva");
```

O nome da variável estrutura seguido por um ponto e pelo nome do elemento referencia esse elemento individual da estrutura, de modo que a combinação `variavelEstrutura.campo` comporta-se **exatamente** como uma variável do mesmo tipo que `campo` (ou seja, `registro.nomePai` é uma string).

A forma geral para acessar um elemento de estrutura é

```
nome_da_estrutura.nome_do_elemento
```

Assim, para imprimir o nome do pai na tela, escreva

```
printf("%s", registro.nomePai);
```

Para acessar os elementos individuais de `registro.nomePai`, você pode indexar `nomePai`. Por exemplo, você pode imprimir o conteúdo de `registro.nomePai`, um caractere por vez, usando o seguinte código:

```
int t;
for (t=0; registro.namePai[t]; ++t)
    putchar(registro.namePai[t]);
```

10.4.2 ESTRUTURAS CONTENDO ESTRUTURAS

Os campos internos de um `struct` podem ser um tipo primitivo, matrizes, ou mesmo outras estruturas, desde que previamente declaradas. Assim sendo, poderíamos projetar um `struct` específico para lidar com datas

```
struct data
{
    unsigned int dia;
    unsigned int mes;
    unsigned int ano;
};
```

e reescrevermos a estrutura `RegGeral` da seguinte forma:

```
struct RegGeral
{
    char nome[50];
    struct data dataNascimento;
    struct data dataEmissao;
    char nomePai[50];
    char nomeMae[50];
    char CPF[13];
    char naturalidade[20];
};
```

O que ganhamos com isso? A abstração cria uma forma mais elaborada de enxergar uma data de nascimento como sendo uma composição de três valores (dia, mês e ano), além de podermos acessar cada um desses campos de maneira direta. Por exemplo, o trecho abaixo atribui o dia 8 a data de nascimento:

```
registro.dataNascimento.dia = 8;
```

10.4.3 ATRIBUIÇÃO DE ESTRUTURAS

Se seu compilador C é compatível com o padrão C ANSI, a informação contida em uma estrutura pode ser atribuída a outra estrutura do mesmo tipo. Isto é, em lugar de ter de atribuir os valores de todos os elementos separadamente, você pode empregar um único comando de atribuição. O programa seguinte ilustra atribuições de estruturas:

```
#include <stdio.h>

void main(void)
{
    struct {
        int a;
        int b;
    } x, y;

    x.a = 10;

    y = x; /* atribui uma estrutura a outra */

    printf("%d", y.a);
}
```

Após a atribuição, `y.a` conterá o valor 10.

10.4.4 MATRIZES DE ESTRUTURAS

Talvez o uso mais comum de estruturas seja em matriz de estruturas. Para declarar uma matriz de estruturas, você deve primeiro definir uma estrutura e, então, declarar uma variável matriz desse tipo. Por exemplo, para declarar uma matriz de estruturas com 100 elementos do tipo `RegGeral`, que foi definido anteriormente, deve-se escrever

```
struct RegGeral registros[100];
```

Isso cria 100 conjuntos de variáveis que estão organizados como definido na estrutura `RegGeral`.

Para acessar uma estrutura específica, deve-se indexar o nome da estrutura. Por exemplo, para imprimir o CPF da estrutura 3, escreva

```
printf("%s", registros[2].CPF);
```

Como todas as outras matrizes, matrizes de estruturas começam a indexação em 0.

10.4.5 PASSANDO ESTRUTURAS PARA FUNÇÕES

Esta seção discute a passagem de estruturas e seus elementos para funções.

10.4.5.1 PASSANDO ELEMENTOS DE ESTRUTURA PARA FUNÇÕES

Quando você passa um elemento de uma variável estrutura para uma função, está, de fato, passando o valor desse elemento para a função. Assim, você está passando uma variável simples (a menos, e claro, que o elemento seja complexo, como uma matriz de caracteres). Por exemplo, considere esta estrutura:

```
struct fred
{
    char x;
    int y;
    float z;
    char s[10];
} mike;
```

A seguir são mostrados exemplos de cada elemento sendo passado para uma função:

```
func(mike.x);      /* passa o valor do caractere de x */
func2(mike.y);     /* passa o valor inteiro de y */
func3(mike.z);     /* passa o valor float de z */
func4(mike.s);     /* passa o endereço da string s */
func(mike.s[2]);   /* passa o valor do caractere de s[2] */
```

Porém, se você quiser passar o endereço de um elemento individual da estrutura, ponha o operador `&` antes do nome da estrutura. Por exemplo, para passar o endereço dos elementos da estrutura `mike`, escreva

```
func(&mike.x);     /* passa o endereço do caractere x */
func2(&mike.y);    /* passa o endereço do inteiro y */
func3(&mike.z);    /* passa o endereço do float z */
func4(mike.s);     /* passa o endereço da string s */
func(&mike.s[2]);  /* passa o endereço do caractere s[2] */
```

Lembre-se de que o operador `&` precede o nome da estrutura, não o nome do elemento individual. Note também que o elemento string `s` já significa um endereço, de forma que o `&` não é necessário.

10.4.5.2 PASSANDO ESTRUTURAS INTEIRAS PARA FUNÇÕES

Quando uma estrutura é usada como um argumento para uma função, a estrutura inteira é passada usando o método padrão de chamada por valor. Obviamente, isso significa que quaisquer alterações podem ser feitas no conteúdo da estrutura dentro da função para a qual ela é passada sem afetar a estrutura usada como argumento.

Quando usar uma estrutura como um parâmetro, lembre-se de que o tipo de argumento deve coincidir com o tipo de parâmetro. Por exemplo, neste programa, tanto o argumento `arg` como o parâmetro `parm` são declarados como o mesmo tipo de estrutura.

```

#include <stdio.h>

/* Define um tipo de estrutura. */
struct struct_type {
    int a, b;
    char ch;
};

void f1(struct struct_type parm);

void main(void)
{
    struct struct_type arg;
    arg.a = 1000;
    f1(arg);
}

void f1(struct struct_type parm)
{
    printf("%d", parm.a);
}

```

Como este programa ilustra, se você declarar parâmetros que são estruturas, deverá tornar a declaração do tipo de estrutura global, para que todas as partes do seu programa possam usá-la. Por exemplo, se `struct_type` tivesse sido declarada dentro de `main()` (por exemplo), então não seria visível a `f1()`.

Como acabamos de enunciar, ao passar estruturas, o tipo do argumento deve coincidir com o tipo do parâmetro. Não é suficiente que eles sejam fisicamente semelhantes; os nomes dos seus tipos devem coincidir.

10.4.6 PONTEIROS PARA ESTRUTURAS

C permite ponteiros para estruturas exatamente como permite ponteiros para outros tipos de variáveis. No entanto, há alguns aspectos especiais de ponteiros de estruturas que você deve conhecer.

10.4.6.1 DECLARANDO UM PONTEIRO PARA ESTRUTURA

Como outros ponteiros, você declara ponteiros para estrutura colocando `*` na frente do nome da estrutura. Por exemplo, assumindo a estrutura previamente definida `RegGera1`, o código seguinte declara `pReg` como um ponteiro para dados daquele tipo.

```

struct RegGera1 *pReg;

```

10.4.6.2 USANDO PONTEIROS PARA ESTRUTURAS

Há dois usos primários para ponteiros de estrutura: gerar uma chamada por referência para uma função e criar listas encadeadas e outras estruturas de dados dinâmicas usando o sistema de alocação de C. Nesta Parte II, abordaremos apenas o primeiro uso. O segundo uso é abordado na Parte III desta apostila.

Há um prejuízo maior em passar todas as estruturas, exceto as mais simples, para funções: o tempo extra necessário para colocar (e tirar) todos os elementos da estrutura na pilha. Em estruturas simples, com poucos elementos, esse tempo extra não é tão grande. Se vários elementos são usados, porém, ou se alguns dos elementos são matrizes, a performance pode ser reduzida a níveis inaceitáveis. A solução para esse problema é passar apenas um ponteiro para uma função.

Quando um ponteiro para uma estrutura é passado para uma função, apenas o endereço da estrutura é colocado (e tirado) da pilha. Isso contribui para chamadas muito rápidas a funções. Uma segunda vantagem, em alguns casos, é quando a função precisa referenciar o argumento real em lugar de uma cópia. Passando um ponteiro, é possível alterar o conteúdo dos elementos reais da estrutura usada na chamada.

Para encontrar o endereço da variável estrutura, deve-se colocar o operador `&` antes do nome da estrutura. Por exemplo, dado o seguinte fragmento:

```
struct ponto
{
    float x;
    float y;
} pt;

struct ponto *ppt; /* declara um ponteiro para estrutura */
então
```

```
ppt = &pt;
```

põe o endereço da estrutura `pt` no ponteiro `ppt`.

Para acessar os elementos de uma estrutura usando um ponteiro para a estrutura, você deve usar o operador `->`. Por exemplo, isso referencia o campo `x`:

```
ppt->x
```

O `->` é normalmente chamado de *operador seta*, e consiste no sinal de “subtração” seguido pelo sinal de “maior”. A seta é usada no lugar do operador ponto quando se esta acessando um elemento de estrutura por meio de um ponteiro para a estrutura.

O operador `->` é equivalente, porém de forma mais prática, a seguinte expressão

```
(*ppt).x
```

na qual há dois operadores envolvidos. Primeiro é tomado o valor (ou seja, os dados propriamente ditos) da estrutura para o qual o ponteiro `ppt` aponta. A seguir, o operador `.` (ponto) seleciona o campo `x` desta estrutura.

10.4.6.3 ALOCAÇÃO DINÂMICA DE ESTRUTURAS

Ponteiros para estruturas, vistos na seção anterior, fornecem a base necessária para se trabalhar com estruturas dinamicamente alocadas. Isto não apenas é útil, mas requisito básico quando se trabalha com Tipos Abstratos de Dados (Listas, Filas, Pilhas, Árvores,...), assunto este abordado na Parte III deste material.

O exemplo abaixo ilustra seu uso:

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

struct ponto {
    float x;
    float y;
};

void main()
{
    struct ponto *p;

    p = (struct ponto *)malloc(sizeof(struct ponto));

    p->x = 2.0;
    p->y = 2.7;

    printf("X = %f\nY = %f\n\n", p->x, p->y);
}
```

10.5 ENUMERAÇÕES

Numa enumeração podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

```
enum nome_do_tipo_da_enumeração {lista_de_valores} lista_de_variáveis;
```

Vamos considerar o seguinte exemplo:

```
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};
```

O programador diz ao compilador que qualquer variável do tipo `dias_da_semana` só pode ter os valores enumerados. Isto quer dizer que poderíamos fazer o seguinte programa:

```
#include <stdio.h>
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};

main() {
    enum dias_da_semana d1, d2;
    d1 = segunda;
    d2 = sexta;
    if (d1 == d2)
        printf("O dia eh o mesmo.");
    else
        printf("Sao dias diferentes.");
}
```

Você deve estar se perguntando como é que a enumeração funciona. Simples. O compilador pega a lista que você fez de valores e associa, a cada um, um número inteiro. Então, ao primeiro da lista, é associado o número zero, o segundo ao número 1 e assim por diante. As variáveis declaradas são então variáveis `int`.

10.6 O COMANDO sizeof

O operador `sizeof` é usado para se saber o tamanho de variáveis ou de tipos. Ele retorna o tamanho do tipo ou variável em bytes. Devemos usá-lo para garantir portabilidade. Por exemplo, o tamanho de um inteiro pode depender do sistema para o qual se está compilando. O `sizeof` é um operador porque ele é substituído pelo tamanho do tipo ou variável *no momento da compilação*. Ele não é uma função. O `sizeof` admite duas formas:

```
sizeof nome_da_variável
sizeof (nome_do_tipo)
```

Se quisermos então saber o tamanho de um `float` fazemos `sizeof(float)`. Se declararmos a variável `f` como `float` e quisermos saber o seu tamanho faremos `sizeof f`. O operador `sizeof` também funciona com estruturas, uniões e enumerações.

Outra aplicação importante do operador `sizeof` é para se saber o tamanho de tipos definidos pelo usuário. Seria, por exemplo, uma tarefa um tanto complicada a de alocar a memória para um ponteiro para a estrutura `ficha_pessoal` (exemplo abaixo), se não fosse o uso de `sizeof`. Veja o exemplo:

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

struct tipo_endereco
{
    char rua[50];
    int numero;
    char bairro[20];
```

```

    char cidade[30];
    char sigla_estado[3];
    long int CEP;
};

struct ficha_pessoal
{
    char nome[50];
    long int telefone;
    struct tipo_endereco endereco;
};

void main(void)
{
    struct ficha_pessoal *ex;
    ex = (struct ficha_pessoal *)malloc(sizeof(struct ficha_pessoal));
    ...
    free(ex);
}

```

10.7 O COMANDO typedef

O comando `typedef` permite ao programador definir um novo nome para um determinado tipo. Sua forma geral é:

```
typedef antigo_nome novo_nome;
```

Como exemplo vamos dar o nome de `inteiro` para o tipo `int`:

```
typedef int inteiro;
```

Agora podemos declarar o tipo `inteiro`.

O comando `typedef` também pode ser utilizado para dar nome a tipos complexos, como as estruturas. As estruturas criadas no exemplo da página anterior poderiam ser definidas como tipos através do comando `typedef`. O exemplo ficaria:

```

#include <stdio.h>

typedef struct tipo_endereco {
    char rua[50];
    int numero;
    char bairro[20];
    char cidade[30];
    char sigla_estado[3];
    long int CEP;
} TEndereco;

typedef struct ficha_pessoal {
    char nome[50];
    long int telefone;
    TEndereco endereco;
} TFicha;

void main(void)
{
    TFicha *ex;
    ...
}

```

Veja que não é mais necessário usar a palavra chave `struct` para declarar variáveis do tipo `ficha_pessoal`. Basta agora usar o novo tipo definido `TFicha`.

Cap. 11: ENTRADA/SAÍDA COM ARQUIVOS

Neste capítulo, apresentaremos alguns conceitos básicos sobre arquivos, e alguns detalhes da forma de tratamento de arquivos em disco na linguagem C. A finalidade desta apresentação é discutir variadas formas para salvar (e recuperar) informações em arquivos. Com isto, será possível implementar funções para salvar (e recuperar) as informações armazenadas nas estruturas de dados que temos discutido.

Um arquivo em disco representa um elemento de informação do dispositivo de memória secundária. A memória secundária (disco) difere da memória principal em diversos aspectos. As duas diferenças mais relevantes são: eficiência e persistência. Enquanto o acesso a dados armazenados na memória principal é muito eficiente do ponto de vista de desempenho computacional, o acesso a informações armazenadas em disco é, em geral, extremamente ineficiente. Para contornar essa situação, os sistemas operacionais trabalham com buffers, que representam áreas da memória principal usadas como meio de transferência das informações de/para o disco. Normalmente, trechos maiores (alguns kbytes) são lidos e armazenados no buffer a cada acesso ao dispositivo. Desta forma, uma subsequente leitura de dados do arquivo, por exemplo, possivelmente não precisará acessar o disco, pois o dado requisitado pode já se encontrar no buffer. Os detalhes de como estes acessos se realizam dependem das características do dispositivo e do sistema operacional empregado.

A outra grande diferença entre memória principal e secundária (disco) consiste no fato das informações em disco serem persistentes, e em geral são lidas por programas e pessoas diferentes dos que as escreveram, o que faz com que seja mais prático atribuir nomes aos elementos de informação armazenados do disco (em vez de endereços), falando assim em arquivos e diretórios (pastas). Cada arquivo é identificado por seu nome e pelo diretório onde encontra-se armazenado numa determinada unidade de disco. Os nomes dos arquivos são, em geral, compostos pelo nome em si, seguido de uma extensão. A extensão pode ser usada para identificar a natureza da informação armazenada no arquivo ou para identificar o programa que gerou (e é capaz de interpretar) o arquivo. Assim, a extensão “.c” é usada para identificar arquivos que têm códigos fontes da linguagem C e a extensão “.doc” é, no Windows, usada para identificar arquivos gerados pelo editor Word da Microsoft.

Um arquivo pode ser visto de duas maneiras, na maioria dos sistemas operacionais: em “modo texto”, como um texto composto de uma sequência de caracteres, ou em “modo binário”, como uma sequência de bytes (números binários). Podemos optar por salvar (e recuperar) informações em disco usando um dos dois modos, texto ou binário. Uma vantagem do arquivo texto é que pode ser lido por uma pessoa e editado com editores de textos convencionais. Em contrapartida, com o uso de um arquivo binário é possível salvar (e recuperar) grandes quantidades de informação de forma bastante eficiente. O sistema operacional pode tratar arquivos “texto” de maneira diferente da utilizada para tratar arquivos “binários”. Em casos especiais, pode ser interessante tratar arquivos de um tipo como se fossem do outro, tomando os cuidados apropriados.

Para minimizar a dificuldade com que arquivos são manipulados, os sistemas operacionais oferecem um conjunto de serviços para ler e escrever informações do disco. A linguagem C disponibiliza esses serviços para o programador através de um conjunto de funções. Os principais serviços que nos interessam são:

- **Abertura de arquivos:** o sistema operacional encontra o arquivo com o nome dado e prepara o buffer na memória.
- **Leitura do arquivo:** o sistema operacional recupera o trecho solicitado do arquivo. Como o buffer contém parte da informação do arquivo, parte ou toda a informação solicitada pode vir do buffer.

- **Escrita no arquivo:** o sistema operacional acrescenta ou altera o conteúdo do arquivo. A alteração no conteúdo do arquivo é feita inicialmente no buffer para depois ser transferida para o disco.
- **Fechamento de arquivo:** toda a informação constante do buffer é atualizada no disco e a área do buffer utilizada na memória é liberada.

Uma das informações que é mantida pelo sistema operacional é um ponteiro de arquivo (file pointer), que indica a posição de trabalho no arquivo. Para ler um arquivo, este apontador percorre o arquivo, do início até o fim, conforme os dados vão sendo recuperados (lidos) para a memória. Para escrever, normalmente, os dados são acrescentados quando o apontador se encontra no fim do arquivo.

Nas seções subsequentes, vamos apresentar as funções mais utilizadas em C para acessar arquivos e vamos discutir diferentes estratégias para tratar arquivos. Todas as funções da biblioteca padrão de C que manipulam arquivos encontram-se na biblioteca de entrada e saída, com interface em `stdio.h`.

11.1 ABRINDO E FECHANDO UM ARQUIVO

O sistema de entrada e saída do ANSI C é composto por uma série de funções, cujos protótipos estão reunidos em `stdio.h`. Todas estas funções trabalham com o conceito de “ponteiro de arquivo”. Este não é um tipo propriamente dito, mas uma definição usando o comando `typedef`. Esta definição também está no arquivo `stdio.h`. Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

`p` será então um ponteiro para um arquivo. É usando este tipo de ponteiro que vamos poder manipular arquivos no C.

11.1.1 `fopen()`

Esta é a função de abertura de arquivos. Seu protótipo é:

```
FILE *fopen(char *nome_do_arquivo, char *modo);
```

O `nome_do_arquivo` determina qual arquivo deverá ser aberto. Este nome deve ser válido no sistema operacional que estiver sendo utilizado. O `modo` de abertura diz à função `fopen()` que tipo de uso você vai fazer do arquivo. A tabela abaixo mostra os valores de modo válidos:

Modo	Significado
"r"	Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
"w"	Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído.
"a"	Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"rb"	Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário.
"wb"	Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário.
"ab"	Acrésceta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário.
"r+"	Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado.
"w+"	Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado.

"a+"	Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente.
"r+b"	Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário.
"w+b"	Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário.
"a+b"	Acrescenta dados ou cria um arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário

Poderíamos então, para abrir um arquivo binário para escrita, escrever:

```
FILE *fp;          /* Declaração da estrutura */
fp = fopen("exemplo.bin", "wb"); /* o arquivo se chama exemplo.bin e está localizado no
                                diretório corrente */
if (!fp) printf("Erro na abertura do arquivo.");
```

A condição `!fp` testa se o arquivo foi aberto com sucesso porque no caso de um erro a função `fopen()` retorna um ponteiro nulo (`NULL`).

Uma vez aberto um arquivo, vamos poder ler ou escrever nele utilizando as funções que serão apresentadas nas próximas páginas.

Toda vez que estamos trabalhando com arquivos, há uma espécie de posição atual no arquivo. Esta é a posição de onde será lido ou escrito o próximo caractere. Normalmente, num acesso sequencial a um arquivo, não temos que mexer nesta posição, pois, quando lemos um caractere, a posição no arquivo é automaticamente atualizada. Num acesso randômico teremos que mexer nesta posição (ver `fseek()`).

11.1.2 exit()

Aqui abrimos um parênteses para explicar a função `exit()` cujo protótipo é:

```
void exit(int codigo_de_retorno);
```

Para utilizá-la deve-se colocar um `include` para o arquivo de cabeçalho `stdlib.h`. Esta função aborta a execução do programa. Pode ser chamada de qualquer ponto no programa e faz com que o programa termine e retorne, para o sistema operacional, o *código_de_retorno*. A convenção mais usada é que um programa retorne zero no caso de um término normal e retorne um número não nulo no caso de ter ocorrido um problema. A função `exit()` se torna importante em casos como alocação dinâmica e abertura de arquivos pois nestes casos, se o programa não conseguir a memória necessária ou abrir o arquivo, a melhor saída pode ser terminar a execução do programa. Poderíamos reescrever o exemplo da seção anterior usando agora o `exit()` para garantir que o programa não deixará de abrir o arquivo:

```
#include <stdio.h>
#include <stdlib.h> /* Para a função exit() */

int main(void)
{
    FILE *fp;
    ...
    fp = fopen("exemplo.bin", "wb");
    if (!fp)
    {
        printf("Erro na abertura do arquivo. Fim de programa.");
        exit(1);
    }
}
```

```

    ...
    return 0;
}

```

11.1.3 fclose()

Quando acabamos de usar um arquivo que abrimos, devemos fechá-lo. Para tanto usa-se a função `fclose()`:

```
int fclose(FILE *fp);
```

O ponteiro `fp` passado à função `fclose()` determina o arquivo a ser fechado. A função retorna zero no caso de sucesso.

Fechar um arquivo faz com que qualquer caractere que tenha permanecido no “buffer” associado ao fluxo de saída seja gravado. Mas, o que é este “buffer”? Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o “buffer”) em vez de serem escritos em disco imediatamente. Quando o “buffer” estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caractere que fossemos gravar, tivéssemos que posicionar a cabeça de gravação em um ponto específico do disco, apenas para gravar aquele caractere, as gravações seriam muito lentas. Assim estas gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.

A função `exit()` fecha todos os arquivos que um programa tiver aberto.

11.2 PRINCIPAIS FUNÇÕES DE LEITURA/ESCRITA EM MODO TEXTO

Os fluxos padrão em arquivos permitem ao programador ler e escrever em arquivos da maneira padrão com a qual o já líamos e escrevíamos na tela.

11.2.1 fprintf()

A função `fprintf()` funciona como a função `printf()`. A diferença é que a saída de `fprintf()` é um arquivo e não a tela do computador. Protótipo:

```
int fprintf(FILE *fp, char *str, ...);
```

Como já poderíamos esperar, a única diferença do protótipo de `fprintf()` para o de `printf()` é a especificação do arquivo destino através do ponteiro de arquivo.

11.2.2 fscanf()

A função `fscanf()` funciona como a função `scanf()`. A diferença é que `fscanf()` lê de um arquivo e não do teclado do computador. Protótipo:

```
int fscanf(FILE *fp, char *str, ...);
```

Como já poderíamos esperar, a única diferença do protótipo de `fscanf()` para o de `scanf()` é a especificação do arquivo destino através do ponteiro de arquivo.

Talvez a forma mais simples de escrever o programa da página anterior seja usando `fprintf()` e `fscanf()`. Fica assim:

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    FILE *p;
    char str[80],c;

    /* Le um nome para o arquivo a ser aberto: */
    printf("\n\n Entre com um nome para o arquivo:\n");
    gets(str);

    if (!(p = fopen(str,"w"))) /* Caso ocorra algum erro na abertura do arquivo..*/
    {                          /* o programa aborta automaticamente */
        printf("Erro! Impossivel abrir o arquivo!\n");
        exit(1);
    }

    /* Se nao houve erro, imprime no arquivo, fecha ...*/
    fprintf(p,"Este e um arquivo chamado:\n%s\n", str);
    fclose(p);

    /* abre novamente para a leitura */
    p = fopen(str,"r");
    while (!feof(p)) {
        fscanf(p,"%c",&c);
        printf("%c",c);
    }

    fclose(p);
}

```

11.3 LENDO E ESCRREVENDO CARACTERES EM ARQUIVOS

11.3.1 putc()

A função `putc()` é a primeira função de escrita de arquivo que veremos. Seu protótipo é:

```
int putc(int ch,FILE *fp);
```

Escreve um caractere no arquivo.

O programa a seguir lê uma string do teclado e escreve-a, caractere por caractere em um arquivo em disco (o arquivo `arquivo.txt`, que será aberto no diretório corrente).

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    FILE *fp;
    char string[100];
    int i;
    fp = fopen("arquivo.txt","w"); /* Arquivo ASCII, para escrita */
    if(!fp) {
        printf("Erro na abertura do arquivo");
        exit(0);
    }
    printf("Entre com a string a ser gravada no arquivo:");
    gets(string);
    for (i=0; string[i]; i++)
        putc(string[i], fp); /* Grava a string, caractere a caractere */
    fclose(fp);
}

```


Depois de executar este programa, verifique o conteúdo do arquivo `arquivo.txt` (você pode usar qualquer editor de textos). Você verá que a string que você digitou está armazenada nele.

11.3.2 `getc()`

Retorna um caractere lido do arquivo. Protótipo:

```
int getc(FILE *fp);
```

11.3.3 `feof()`

EOF (“End of file”) indica o fim de um arquivo. Às vezes, é necessário verificar se um arquivo chegou ao fim. Para isto podemos usar a função `feof()`. Ela retorna não-zero se o arquivo chegou ao EOF, caso contrário retorna zero. Seu protótipo é:

```
int feof(FILE *fp);
```

Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por `getc()` com EOF. O programa a seguir abre um arquivo já existente e o lê, caractere por caractere, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela:

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

int main()
{
    FILE *fp;
    int c;
    fp = fopen("arquivo.txt", "r");    /* Arquivo ASCII, para leitura */

    if(!fp)
    {
        printf("Erro na abertura do arquivo");
        exit(0);
    }

    while ((c = getc(fp)) != EOF)    /* Enquanto não chegar ao final do arquivo */
        printf("%c", c);            /* Imprime o caracter lido */

    fclose(fp);

    return 0;
}
```

11.3.4 EXEMPLO

A seguir é apresentado um programa onde várias operações com arquivos são realizadas, usando as funções vistas nesta página. Primeiro o arquivo é aberto para a escrita, e imprime-se algo nele. Em seguida, o arquivo é fechado e novamente aberto para a leitura. Verifique o exemplo abaixo.

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

void main()
{
    FILE *p;
    char c, str[30], frase[80] = "Este e um arquivo chamado: ";
    int i;
    /* Le um nome para o arquivo a ser aberto: */
    printf("\n\n Entre com um nome para o arquivo:\n");
    gets(str);
}
```

```

if (!(p = fopen(str, "w"))) /* Caso ocorra algum erro na abertura do arquivo */
{
    /* o programa aborta automaticamente */
    printf("Erro! Impossível abrir o arquivo!\n");
    exit(1);
}

/* Se não houve erro, imprime no arquivo e o fecha ...*/
strcat(frase, str);
for (i=0; frase[i]; i++)
    putc(frase[i], p);
fclose(p);

/* Abre novamente para leitura */
p = fopen(str, "r");
c = getc(p);          /* Le o primeiro caracter */
while (!feof(p))      /* Enquanto não se chegar no final do arquivo */
{
    printf("%c", c);   /* Imprime o caracter na tela */
    c = getc(p);       /* Le um novo caracter no arquivo */
}

fclose(p);           /* Fecha o arquivo */
}

```

11.4 ARQUIVOS EM MODO BINÁRIO

Arquivos em modo binário servem para salvarmos (e depois recuperarmos) o conteúdo da memória principal diretamente no disco. A memória é escrita copiando-se o conteúdo de cada byte da memória para o arquivo. Uma das grandes vantagens de se usar arquivos binários é que podemos salvar (e recuperar) uma grande quantidade de dados de forma bastante eficiente. Neste curso, vamos apenas apresentar as duas funções básicas para manipulação de arquivos binários.

11.4.1 FUNÇÕES PARA SALVAR E RECUPERAR

Para escrever (salvar) dados em arquivos binários, usamos a função `fwrite`. O protótipo dessa função pode ser simplificado por:

```
int fwrite (void* p, int tam, int nelem, FILE* fp);
```

O primeiro parâmetro dessa função representa o endereço de memória cujo conteúdo deseja-se salvar em arquivo. O parâmetro `tam` indica o tamanho, em bytes, de cada elemento e o parâmetro `nelem` indica o número de elementos. Por fim, passa-se o ponteiro do arquivo binário para o qual o conteúdo da memória será copiado.

A função para ler (recuperar) dados de arquivos binários é análoga, sendo que agora o conteúdo do disco é copiado para o endereço de memória passado como parâmetro. O protótipo da função pode ser dado por:

```
int fread (void* p, int tam, int nelem, FILE* fp);
```

Para exemplificar a utilização dessas funções, vamos considerar que uma aplicação tem um conjunto de pontos armazenados num vetor. O tipo que define o ponto pode ser:

```

typedef struct ponto {
    float x, y, z;
} Ponto;

```

Uma função para salvar o conteúdo de um vetor de pontos pode receber como parâmetros o nome do arquivo, o número de pontos no vetor, e o ponteiro para o vetor. Uma possível implementação dessa função é ilustrada abaixo:

```
void salva (char* arquivo, int n, Ponto* vet)
{
    FILE* fp = fopen(arquivo, "wb");
    if (fp==NULL) {
        printf("Erro na abertura do arquivo.\n");
        exit(1);
    }
    fwrite(vet, sizeof(Ponto), n, fp);
    fclose(fp);
}
```

A função para recuperar os dados salvos pode ser:

```
void carrega (char* arquivo, int n, Ponto* vet)
{
    FILE* fp = fopen(arquivo, "rb");
    if (fp==NULL) {
        printf("Erro na abertura do arquivo.\n");
        exit(1);
    }
    fread(vet, sizeof(Ponto), n, fp);
    fclose(fp);
}
```

11.5 OUTROS COMANDOS DE ACESSO A ARQUIVOS

11.5.1 ARQUIVOS PRÉ-DEFINIDOS

Quando se começa a execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos:

- `stdin`: dispositivo de entrada padrão (geralmente o teclado)
- `stdout`: dispositivo de saída padrão (geralmente o vídeo)
- `stderr`: dispositivo de saída de erro padrão (geralmente o vídeo)
- `stdaux`: dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
- `stdprn`: dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

Cada uma destas constantes pode ser utilizada como um ponteiro para `FILE`, para acessar os periféricos associados a eles. Desta maneira, pode-se, por exemplo, usar:

```
ch = getc(stdin);
```

para efetuar a leitura de um caractere a partir do teclado, ou :

```
putc(ch, stdout);
```

para imprimi-lo na tela.

11.5.2 `fgets()`

Para se ler uma string num arquivo podemos usar `fgets()` cujo protótipo é:

```
char *fgets(char *str, int tamanho, FILE *fp);
```

A função recebe 3 argumentos: a string a ser lida, o limite máximo de caracteres a serem lidos e o ponteiro para `FILE`, que está associado ao arquivo de onde a string será lida. A função lê a string até que um caractere de nova linha seja lido ou `tamanho-1` caracteres tenham sido lidos. Se o caractere de nova linha (`'\n'`) for lido, ele fará parte da string, o que não acontecia com `gets()`. A string resultante sempre terminará com `'\0'` (por isto somente `tamanho-1` caracteres, no máximo, serão lidos).

A função `fgets()` é semelhante à função `gets()`, porém, além dela poder fazer a leitura a partir de um arquivo de dados e incluir o caractere de nova linha na string, ela ainda especifica o tamanho máximo da string de entrada. Como vimos, a função `gets()` não tinha este controle, o que poderia acarretar erros de “estouro de buffer”. Portanto, levando em conta que o ponteiro `fp` pode ser substituído por `stdin`, como vimos acima, uma alternativa ao uso de `gets()` é usar a seguinte construção:

```
fgets(str, tamanho, stdin);
```

onde `str` é a string que se está lendo e `tamanho` deve ser igual ao tamanho alocado para a string subtraído de 1, por causa do `'\0'`.

11.5.3 fputs()

Protótipo:

```
char *fputs(char *str, FILE *fp);
```

Escreve uma string num arquivo.

11.5.4 ferror() e perror()

Protótipo de `ferror()`:

```
int ferror(FILE *fp);
```

A função retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo.

`ferror()` se torna muito útil quando queremos verificar se cada acesso a um arquivo teve sucesso, de modo que consigamos garantir a integridade dos nossos dados. Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc.

Uma função que pode ser usada em conjunto com `ferror()` é a função `perror()` (print error), cujo argumento é uma string que normalmente indica em que parte do programa o problema ocorreu.

No exemplo a seguir, fazemos uso de `ferror()`, `perror()` e `fputs()`:

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

main()
{
    FILE *pf;
    char string[100];
    if((pf = fopen("arquivo.txt", "w")) == NULL)
    {
        printf("\nNao consigo abrir o arquivo !");
        exit(1);
    }
}
```

```

do
{
    printf("\nDigite uma nova string. Para terminar, digite <enter>: ");
    gets(string);
    fputs(string, pf);
    putc('\n', pf);

    if(ferror(pf))
    {
        perror("Erro na gravacao");
        fclose(pf);
        exit(1);
    }

} while (strlen(string) > 0);

fclose(pf);
}

```

11.5.5 fseek()

Para se fazer procuras e acessos randômicos em arquivos usa-se a função `fseek()`. Esta move a posição corrente de leitura ou escrita no arquivo de um valor especificado, a partir de um ponto especificado. Seu protótipo é:

```
int fseek(FILE *fp, long numbytes, int origem);
```

O parâmetro *origem* determina a partir de onde os *numbytes* de movimentação serão contados. Os valores possíveis são definidos por macros em `stdio.h` e são:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Ponto corrente no arquivo
SEEK_END	2	Fim do arquivo

Tendo-se definido a partir de onde irá se contar, *numbytes* determina quantos bytes de deslocamento serão dados na posição atual.

11.5.6 rewind()

A função `rewind()` de protótipo

```
void rewind(FILE *fp);
```

retorna a posição corrente do arquivo para o início.

11.5.7 remove()

Protótipo:

```
int remove(char *nome_do_arquivo);
```

Apaga um arquivo especificado.

11.5.8 EXEMPLO

O exemplo da anterior poderia ser reescrito usando-se, por exemplo, `fgets()` e `fputs()`, ou `fwrite()` e `fread()`. A seguir apresentamos uma segunda versão que se usa das funções `fgets()` e `fputs()`, e que acrescenta algumas inovações.

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

main()
{
    FILE *p;
    char str[30], frase[] = "Este e um arquivo chamado: ", resposta[80];
    int i;

    /* Lê um nome para o arquivo a ser aberto: */
    printf("\n\n Entre com um nome para o arquivo:\n");
    fgets(str, 29, stdin);          /* Usa fgets como se fosse gets */

    for(i=0; str[i]; i++) if(str[i]=='\n') str[i]=0; /* Elimina o \n da string lida */

    if (!(p = fopen(str, "w")))      /* Caso ocorra algum erro na abertura do arquivo..*/
    {                                /* o programa aborta automaticamente */
        printf("Erro! Impossivel abrir o arquivo!\n");
        exit(1);
    }

    /* Se nao houve erro, imprime no arquivo, e o fecha ...*/
    fputs(frase, p);
    fputs(str, p);
    fclose(p);

    /* abre novamente e le */
    p = fopen(str, "r");
    fgets(resposta, 79, p);
    printf("\n\n%s\n", resposta);
    fclose(p);          /* Fecha o arquivo */
    remove(str);        /* Apaga o arquivo */
}
```

11.6 ESTRUTURAÇÃO DE DADOS EM ARQUIVO TEXTO

Existem diferentes formas para estruturarmos os dados em arquivos em modo texto, e diferentes formas de capturarmos as informações contidas neles. A forma de estruturar e a forma de tratar as informações dependem da aplicação. A seguir, apresentaremos três formas de representarmos e acessarmos dados armazenados em arquivos: caractere a caractere, linha a linha, e usando palavras chaves.

11.6.1 ACESSO CARACTERE A CARACTERE

Para exemplificar o acesso caractere a caractere, vamos discutir duas aplicações simples. Inicialmente, vamos considerar o desenvolvimento de um programa que conta as linhas de um determinado arquivo (para simplificar, vamos supor um arquivo fixo, com o nome “entrada.txt”). Para calcular o número de linhas do arquivo, podemos ler, caractere a caractere, todo o conteúdo do arquivo, contando o número de ocorrências do caractere que indica mudança de linha, isto é, o número de ocorrências do caractere ‘\n’.

```
/* Conta número de linhas de um arquivo */

#include <stdio.h>

int main (void)
{
```

```

int c;
int nlinhas = 0;    /* contador do número de linhas */
FILE *fp;
/* abre arquivo para leitura */
fp = fopen("entrada.txt", "r");
if (fp==NULL) {
    printf("Não foi possível abrir arquivo.\n");
    return 1;
}
/* lê caractere a caractere */
while ((c = fgetc(fp)) != EOF) {
    if (c == '\n')
        nlinhas++;
}
/* fecha arquivo */
fclose(fp);
/* exibe resultado na tela */
printf("Numero de linhas = %d\n", nlinhas);
return 0;
}

```

Como segundo exemplo, vamos considerar o desenvolvimento de um programa que lê o conteúdo do arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. Uma possível implementação desse programa é mostrada a seguir:

```

/* Converte arquivo para maiúsculas */
#include <stdio.h>
#include <ctype.h>    /* função toupper */
int main (void)
{
    int c;
    char entrada[121];    /* armazena nome do arquivo de entrada */
    char saida[121];      /* armazena nome do arquivo de saída */
    FILE* e;              /* ponteiro do arquivo de entrada */
    FILE* s;              /* ponteiro do arquivo de saída */

    /* pede ao usuário os nomes dos arquivos */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s", entrada);
    printf("Digite o nome do arquivo de saída: ");
    scanf("%120s", saida);

    /* abre arquivos para leitura e para escrita */
    e = fopen(entrada, "r");
    if (e == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }
    s = fopen(saida, "w");
    if (s == NULL) {
        printf("Não foi possível abrir arquivo de saída.\n");
        fclose(e);
        return 1;
    }

    /* lê da entrada e escreve na saída */
    while ((c = fgetc(e)) != EOF)
        fputc(toupper(c), s);

    /* fecha arquivos */
    fclose(e);
    fclose(s);
    return 0;
}

```

11.6.2 ACESSO LINHA A LINHA

Em diversas aplicações, é mais adequado tratar o conteúdo do arquivo linha a linha. Um caso simples que podemos mostrar consiste em procurar a ocorrência de uma sub-cadeia de caracteres dentro de um arquivo (análogo a o que é feito pelo utilitário `grep` dos sistemas Unix). Se a sub-cadeia for encontrada, apresentamos como saída o número da linha da primeira ocorrência. Para implementar esse programa, vamos utilizar a função `strstr()`, que procura a ocorrência de uma sub-cadeia numa cadeia de caracteres maior. A função retorna o endereço da primeira ocorrência ou `NULL`, se a sub-cadeia não for encontrada. O protótipo dessa função é:

```
char* strstr (char* s, char* sub);
```

A nossa implementação consistirá em ler, linha a linha, o conteúdo do arquivo, contando o número da linha. Para cada linha, verificamos se a ocorrência da sub-cadeia, interrompendo a leitura em caso afirmativo.

```
/* Procura ocorrência de sub-cadeia no arquivo */
#include <stdio.h>
#include <string.h> /* função strstr */
int main (void)
{
    int n = 0; /* número da linha corrente */
    int achou = 0; /* indica se achou sub-cadeia */
    char entrada[121]; /* armazena nome do arquivo de entrada */
    char subcadeia[121]; /* armazena sub-cadeia */
    char linha[121]; /* armazena cada linha do arquivo */
    FILE* fp; /* ponteiro do arquivo de entrada */
    /* pede ao usuário o nome do arquivo e a sub-cadeia */
    printf("Digite o nome do arquivo de entrada: ");
    scanf("%120s", entrada);
    printf("Digite a sub-cadeia: ");
    scanf("%120s", subcadeia);
    /* abre arquivos para leitura */
    fp = fopen(entrada, "r");
    if (fp == NULL) {
        printf("Não foi possível abrir arquivo de entrada.\n");
        return 1;
    }
    /* lê linha a linha */
    while (fgets(linha, 121, fp) != NULL) {
        n++;
        if (strstr(linha, subcadeia) != NULL) {
            achou = 1;
            break;
        }
    }
    /* fecha arquivo */
    fclose(fp);
    /* exibe saída */
    if (achou)
        printf("Achou na linha %d.\n", n);
    else
        printf("Nao achou.");
    return 0;
}
```

Como segundo exemplo de arquivos manipulados linha a linha, podemos citar o caso em que salvamos os dados com formatação por linha. Para exemplificar, vamos considerar que queremos salvar informações de uma lista de figuras geométricas contendo retângulos, triângulos e círculos.

Para salvar essas informações num arquivo, temos que escolher um formato apropriado, que nos permita posteriormente recuperar a informação salva. Para exemplificar um formato válido, vamos adotar uma formatação por linha: em cada linha salvamos um caractere que indica o tipo da figura (`r`, `t` ou `c`), seguido dos

parâmetros que definem a figura, `base` e `altura` para os retângulos e triângulos ou `raio` para os círculos. Para enriquecer o formato, podemos considerar que as linhas iniciadas com o caractere `#` representam comentários e devem ser desconsideradas na leitura. Por fim, linhas em branco são permitidas e desprezadas. Um exemplo do conteúdo de um arquivo com esse formato é apresentado abaixo (note a presença de linhas em branco e linhas que são comentários):

```
# Lista de figuras geométricas

r 2.0 1.2
c 5.8
# t 1.23 12
t 4 1.02

c 5.1
```

Para recuperarmos as informações contidas num arquivo com esse formato, podemos ler do arquivo cada uma das linhas e depois ler os dados contidos na linha. Para tanto, precisamos introduzir uma função adicional muito útil. Trata-se da função que permite ler dados de uma cadeia de caracteres. A função `sscanf()` é similar às funções `scanf()` e `fscanf()`, mas captura os valores armazenados numa string. O protótipo dessa função é:

```
int sscanf (char* s, char* formato, ...);
```

A primeira cadeia de caracteres passada como parâmetro representa a string da qual os dados serão lidos. Com essa função, é possível ler uma linha de um arquivo e depois ler as informações contidas na linha. (Analogamente, existe a função `sprintf()` que permite escrever dados formatados numa string.)

Faremos a interpretação do arquivo da seguinte forma: para cada linha lida do arquivo, tentaremos ler do conteúdo da linha um caractere (desprezando eventuais caracteres brancos iniciais) seguido de dois números reais. Se nenhum dado for lido com sucesso, significa que temos uma linha vazia e devemos desprezá-la. Se pelo menos um dado (no caso, o caractere) for lido com sucesso, podemos interpretar o tipo da figura geométrica armazenada na linha, ou detectar a ocorrência de um comentário. Se for um retângulo ou um triângulo, os dois valores reais também deverão ter sido lidos com sucesso. Se for um círculo, apenas um valor real deverá ter sido lido com sucesso. O fragmento de código abaixo ilustra essa implementação. Supõe-se que `fp` representa um ponteiro para um arquivo com formato válido aberto para leitura, em modo texto.

```
char c;
float v1, v2;
FILE* fp;
char linha[121];

...

while (fgets(linha,121,fp)) {
    int n = sscanf(linha, " %c %f %f",&c,&v1,&v2);
    if (n>0) {
        switch(c) {
            case '#':
                /* desprezar linha de comentário */
                break;
            case 'r':
                if (n!=3)
                {
                    /* tratar erro de formato do arquivo */
                }
            }
        }
    }
}
```

```

        else
        {
            /* interpretar retângulo: base = v1, altura = v2 */
            ...
        }
        break;
    case 't':
        if (n!=3)
        {
            /* tratar erro de formato do arquivo */
            ...
        }
        else
        {
            /* interpretar triângulo: base = v1, altura = v2 */
            ...
        }
        break;
    case 'c':
        if (n!=2)
        {
            /* tratar erro de formato do arquivo */
            ...
        }
        else
        {
            /* interpretar círculo: raio = v1 */
            ...
        }
        break;
    default:
        /* tratar erro de formato do arquivo */
        ...
    }
}
...

```

A rigor, para o formato descrito, não precisávamos fazer a interpretação do arquivo linha a linha. O arquivo poderia ter sido interpretado capturando-se inicialmente um caractere que então indicaria qual a próxima informação a ser lida. No entanto, em algumas situações a interpretação linha a linha ilustrada acima é a única forma possível. Para exemplificar, vamos considerar um arquivo que representa um conjunto de pontos no espaço 3D. Esses pontos podem ser dados pelas suas três coordenadas x , y e z . Um formato bastante flexível para esse arquivo considera que cada ponto é dado em uma linha e permite a omissão da terceira coordenada, se essa for igual a zero. Dessa forma, o formato atende também a descrição de pontos no espaço 2D. Um exemplo desse formato é ilustrado abaixo

```

2.3    4.5    6.0
1.2    10.4
7.4    1.3    9.6
...

```

Para interpretar esse formato, devemos ler cada uma das linhas e tentar ler três valores reais de cada linha (aceitando o caso de apenas dois valores serem lidos com sucesso).

11.6.3 ACESSO VIA PALAVRAS CHAVES

Quando os objetos num arquivo têm descrições de tamanhos variados, é comum adotarmos uma formatação com o uso de palavras chaves. Cada objeto é precedido por uma palavra chave que o identifica. A interpretação desse tipo de arquivo pode ser feita lendo-se as palavras chaves e interpretando a descrição do objeto correspondente. Para ilustrar, vamos considerar que, além de retângulos, triângulos e círculos, também temos polígonos quaisquer no nosso conjunto de figuras geométricas. Cada polígono pode ser descrito pelo número de vértices que o compõe, seguido das respectivas coordenadas desses vértices. A figura ao lado ilustra esse formato.

```
RETANGULO
  b    h
TRIANGULO
  b    h
CIRCULO
  r
POLIGONO
  n
  x1   y1
  x2   y2
  ...
  xn   yn
```

O fragmento de código a seguir ilustra a interpretação desse formato, onde `fp` representa o ponteiro para o arquivo aberto para leitura.

```
...
FILE* fp;
char palavra[121];
...
while (fscanf(fp,"%120s",palavra) == 1)
{
    if (strcmp(palavra,"RETANGULO")==0) {
        /* interpreta retângulo */
    }
    else if (strcmp(palavra,"TRIANGULO")==0) {
        /* interpreta triângulo */
    }
    else if (strcmp(palavra,"CIRCULO")==0) {
        /* interpreta círculo */
    }
    else if (strcmp(palavra,"POLIGONO")==0) {
        /* interpreta polígono */
    }
    else {
        /* trata erro de formato */
    }
}
```

PARTE III

Uma vez dominados os principais recursos avançados da linguagem C (ponteiros, alocação dinâmica, estruturas, E/S em arquivos, ...), a sequência natural é partir para um nível onde o foco agora é o desenvolvimento do software. Em outras palavras, apenas conhecer e saber usar tais ferramentas não assegura que você seja capaz de partir do zero e codificar um software de maior complexidade. Para tanto, é preciso conhecer mais profundamente algumas ***diretrizes*** que nortearão o desenvolvimento.

O primeiro passo nesse novo campo é a plena compreensão e uso de um dos principais conceitos da computação: a **abstração**. Assim sendo, após uma introdução conceitual (mas com exemplos) neste campo, abordaremos os Tipos Abstratos de Dados (TAD).

Cap. 12: ABSTRAÇÃO DE DADOS

12.1 INTRODUÇÃO

Um programa de computador desenvolvido para atender alguma finalidade prática é, normalmente, um artefato complexo. Por esse motivo, a atividade de desenvolvimento desses artefatos, a programação de computadores, está entre as atividades mais complexas desempenhadas pelo homem. Se você cursou disciplinas introdutórias de programação, pode estar se questionando: – *Ora, desenvolver um programa não é tão complexo assim! Basta compreender o problema, suas entradas e suas saídas e construir a solução usando uma linguagem de programação.* Simples não? Não! A história da programação tem dado provas que programar não é tão simples assim.

Programar é uma atividade complexa por diversos aspectos, tanto de cunho teórico quanto prático. Dentre estes aspectos destacamos os seguintes:

- Programar um computador significa desenvolver programas de computadores (**formais e precisos**) para atender a finalidades práticas definidas em termos de conceitos do mundo real (**informais e imprecisos**). Existe um abismo entre o mundo dos problemas reais e o mundo das soluções.
- Muitas vezes, em disciplinas iniciais de programação, deparamo-nos com o desenvolvimento de programas mais simples, de cunho didático. Por exemplo, programas para calcular médias ou somatórios. Em outros momentos fazemos programas para aprender a usar um certo mecanismo, por exemplo, apontadores. Aqui, estamos nos referindo a programas de computadores para ajudar pessoas a resolverem problemas do mundo real, problemas grandes e complexos! Exemplos desses problemas incluem:
 - a. Gerenciar as operações financeiras de uma empresa;
 - b. Controlar uma aeronave;
 - c. Controlar os trens de uma malha ferroviária;
 - d. Produzir edições diárias de um jornal;
 - e. Gerenciar o processo de produção de um filme.
- Problemas como esses não são simples de se resolver. Consequentemente, programas voltados para essas finalidades são complexos, levam muito tempo para ficar prontos, têm de ser desenvolvidos por uma equipe de pessoas, utilizando diversas tecnologias e seguindo um processo de desenvolvimento sistemático.
- Para atender às funcionalidades esperadas, um programa deve apresentar um conjunto de características que juntas vão tornar o programa efetivamente útil e determinarão a qualidade do mesmo. Essas características são as seguintes:
 - a. Um programa deve estar **correto**, livre de erros;
 - b. Um programa deve ser **robusto**. Um programa robusto ou sistema robusto é aquele que consegue funcionar, mesmo que precariamente, diante de uma adversidade. Por exemplo, suponha que um programa precise dos dados x , y e z para realizar uma tarefa. Se este for robusto, na falta de um dos dados, o programa pode tentar realizar o processamento possível com a ausência dado;
 - c. Um programa deve ser **eficiente**. A eficiência de um programa está relacionada ao seu tempo de execução (eficiência na execução) ou ao espaço em memória (eficiência na ocupação) de que necessita para executar (área de dados). Para um problema há infinitas soluções

- (programas). Quanto menores esses valores mais eficiente o programa. Em computação pode-se verificar se uma solução é ótima (mais eficiente possível) para um problema;
- d. Um programa deve ser **compatível** com outros programas;
 - e. Um programa deve ser **fácil de usar**;
 - f. Um programa deve ser **portável**, podendo funcionar em diferentes plataformas ou sistemas operacionais;
 - g. Um programa deve ser **íntegro**, ou seja, deve evitar que os dados sejam corrompidos ou violados;
 - h. O **processamento** realizado pelo programa deve ser **verificável**;
 - i. O programa ou partes dele devem poder ser **utilizados em outros cenários** diferentes daquele para o qual o programa foi originalmente desenvolvido.
- Por último, devemos considerar a atividade de programação como atividade econômica. Assim como outras atividades econômicas, o desenvolvimento de software é regido por leis de mercado que impõem severas exigências aos desenvolvedores. De acordo com essas leis, não basta apenas desenvolver um programa que atenda à finalidade esperada. Esses programas devem ser desenvolvidos dentro dos prazos e custos estabelecidos. Além disso, o programa precisa ter outras características importantes que permitam a sua evolução. Essas características são chamadas de fatores internos. São eles:
 - a. Facilidade de manutenção;
 - b. Facilidade de evolução;
 - c. Facilidade de entendimento;
 - d. Modularidade.

Pelos motivos discutidos acima, **o desenvolvimento de programas requer a aplicação de princípios, métodos e técnicas que diminuam a complexidade desse desenvolvimento**. A Abstração de Dados envolve uma série de conceitos e princípios para esse fim. A seguir discutiremos esses conceitos.

12.2 ABSTRAÇÃO EM COMPUTAÇÃO

Abstração... A essa altura você certamente já entende esse conceito aplicado à programação, mesmo que não saiba definir exatamente. Ok, é exatamente o que tentaremos fazer aqui nesta introdução.

A **abstração** é um dos conceitos mais importantes da computação. Sem o uso deste conceito podemos afirmar que a evolução apresentada pela computação teria sido mais lenta.

Segundo o dicionário Michaelis, temos a seguinte definição para a palavra Abstração:

1. O ato ou efeito de abstrair. Consideração das qualidades independentemente dos objetos a que pertencem. Abstrair: Considerar um dos caracteres de um objeto separadamente; 2. Excluir, prescindir de.

A finalidade principal de uso de abstração em qualquer área do conhecimento é **colocarmos foco em um subconjunto dos aspectos de um sistema ou processo complexo**. Isso implica diretamente em focar-se em determinados aspectos e “esquecer” dos demais. Não é por acaso que *esquecer* está entre aspas. No contexto de programação de computadores, abstrair está relacionado com focar-se em **“O quê?”** um sistema faz, sem se importar, inicialmente, em **“Como?”** este o faz.

Por exemplo, considere a agenda eletrônica de um simples celular. Para que você possa fazer pleno uso dela é suficiente que você conheça suas funcionalidades e saiba **o que** elas fazem. Por exemplo, as funcionalidades adicionar contato, ordenar contatos, exibir contatos, excluir contato, etc, são comuns a

qualquer agenda. Não é necessário que você saiba **como** essas funcionalidades são implementadas. Por exemplo, os questionamentos: Qual o tamanho e tipo de cada campo de dados de um contato? Como são armazenados? Como é feita a ordenação? Não são (e nem devem ser) conhecimentos necessários a quem apenas deseja usar esse sistema.

Outro exemplo mais próximo da programação é a **variável**. Sim, uma variável é um conceito abstrato que esconde diversos aspectos técnicos que não interessam ao programador (de alto nível, pelo menos). Quando declaramos uma variável em um programa, essa declaração implica em “coisas” que não irão interferir no seu desenvolvimento. Por exemplo, por trás de uma variável do tipo inteiro em C, (ex. `int x;`), estão “escondidas” as características físicas do armazenamento da variável em memória, a saber:

- A forma de representação de números inteiros usando base 2 (por exemplo, complemento de 2);
- O padrão usado pelo hardware (ex. *little endian*, *big endian*);
- O número de bytes que uma variável do tipo inteiro ocupa;
- O endereço da variável na memória principal;
- O conjunto de bits responsável por armazenar o sinal do número inteiro.

Para o programador em C, geralmente, nenhuma dessas informações é importante. O que o programador deseja é utilizar a variável realizando as operações permitidas aos números inteiros (operações aritméticas e comparações), atribuir, recuperar e modificar o valor contido na variável. Assim podemos dizer que uma variável permite ao programador abstrair-se de detalhes que não interessam e não irão influenciar no comportamento do programa.

12.3 TAD: TIPO ABSTRATO DE DADOS

Um **tipo abstrato de dados** pode ser definido matematicamente pelo par (V, O) onde V é um **conjunto de valores (dados)** e O um **conjunto de operações** sobre estes valores.

Um tipo abstrato de dados está desvinculado de sua implementação, ou seja, quando definimos um tipo abstrato de dados estamos preocupados com o que ele faz e não como ele faz.

Em geral, quando usamos o conceito de tipo abstrato de dados, dividimos a programação em duas etapas: **aplicação** e **implementação**

Na aplicação, apenas as operações definidas abstratamente são utilizadas, não sendo permitido o acesso direto aos dados, ou seja, o usuário só tem acesso às operações.

Suponha que estamos trabalhando com listas de números inteiros e apenas as operações de inclusão e busca estão presentes. Neste tipo abstrato de dados o usuário nunca poderia trocar a posição de dois elementos na lista, pois a ele só é permitido inserir ou procurar por um elemento.

As características da implementação permanecem inacessíveis, ou seja, o usuário não consegue saber, por exemplo, se a lista é dinâmica ou estática.

Nesse contexto, **abstrato** significa “*esquecida a forma de implementação*”, ou seja, um TAD é descrito pela finalidade do tipo e de suas operações, e não pela forma como está implementado.

Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado. Isto facilita a manutenção e o re-uso de códigos.

12.3.1 ESTRUTURA DE DADOS VERSUS TIPO ABSTRATO DE DADOS

É muito importante que você perceba que um tipo abstrato de dados não se resume a uma nova estrutura de dados.

A definição de uma estrutura de dados se preocupa em demonstrar como o objeto é representado na memória de um computador (representação). Nessa definição são considerados aspectos do tipo: quais as informações que serão armazenadas ali e qual a quantidade destas informações. A definição de um Tipo Abstrato de Dados segue uma abordagem diferente. Essa definição é feita em termos das operações que podem ser realizadas sobre o tipo.

A definição de um Tipo Abstrato de Dado (TAD) é chamada de especificação e consiste, basicamente, em definir as operações relacionadas ao tipo. Dizemos que essas operações definem o comportamento do TAD.

12.3.2 EXEMPLOS

12.3.2.1 TAD PONTO

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no R^2 . Para isso, devemos definir um tipo abstrato, que denominaremos de **Ponto**, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- *cria*: operação que cria um ponto com coordenadas x e y ;
- *libera*: operação que libera a memória alocada por um ponto;
- *acessa*: operação que devolve as coordenadas de um ponto;
- *atribui*: operação que atribui novos valores às coordenadas de um ponto;
- *distancia*: operação que calcula a distância entre dois pontos.

Acontece às vezes que um módulo inclui (`#include`) um arquivo `.h`, e um segundo arquivo `.h` também inclui o primeiro. O compilador gera erros sobre definições que ocorrem mais de uma vez. A maneira de evitar este problema é a utilização de diretivas de pré-compilação condicionais para garantir que as definições apareçam apenas uma vez para o compilador.

A interface desse módulo pode ser dada pelo código a seguir:

```
/* Arquivo ponto.h */

#ifndef _PONTO_H
#define _PONTO_H

typedef struct ponto Ponto;

Ponto* cria (float x, float y);

void libera (Ponto* p);

void acessa (Ponto* p, float* x, float* y);

void atribui (Ponto* p, float x, float y);

float distancia (Ponto* p1, Ponto* p2);

#endif
```

Note que a composição da estrutura `ponto` (*struct ponto*) não é definida pelo módulo. Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura. Os clientes desse TAD só terão acesso às informações que possam ser obtidas através das funções definidas pelo arquivo `ponto.h`.

Agora, mostraremos uma implementação para esse tipo abstrato de dados. O arquivo de implementação do módulo (arquivo `ponto.c`) deve sempre incluir o arquivo de interface do módulo. Isto é necessário por duas razões. Primeiro, podem existir definições na interface que são necessárias na implementação. No nosso

caso, por exemplo, precisamos da definição do tipo Ponto. A segunda razão é garantirmos que as funções implementadas correspondem às funções da interface. Como o protótipo das funções é incluído, o compilador verifica, por exemplo, se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos. Além da própria interface, precisamos naturalmente incluir as interfaces das funções que usamos da biblioteca padrão.

```
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include <math.h> /* sqrt */
#include "ponto.h"
```

Como só precisamos guardar as coordenadas de um ponto, podemos definir a estrutura ponto da seguinte forma:

```
struct ponto {
    float x;
    float y;
};
```

A função que cria um ponto dinamicamente deve alocar a estrutura que representa o ponto e inicializar os seus campos:

```
Ponto* cria (float x, float y) {
    Ponto* p;
    p = (Ponto *)malloc(sizeof(Ponto));
    if (p == NULL) exit(1); /* memória insuficiente */
    p->x = x;
    p->y = y;
    return p;
}
```

Para esse TAD, a função que libera um ponto deve apenas liberar a estrutura que foi criada dinamicamente através da função cria:

```
void libera (Ponto* p) {
    free(p);
}
```

As funções para acessar e atribuir valores às coordenadas de um ponto são de fácil implementação, como pode ser visto a seguir.

```
void acessa (Ponto* p, float* x, float* y) {
    *x = p->x;
    *y = p->y;
}

void atribui (Ponto* p, float x, float y) {
    p->x = x;
    p->y = y;
}
```

Já a operação para calcular a distância entre dois pontos pode ser implementada da seguinte forma:

```
float distancia (Ponto* p1, Ponto* p2) {
    float dx, dy;
    dx = p2->x - p1->x;
    dy = p2->y - p1->y;
    return sqrt(dx*dx + dy*dy);
}
```

12.3.2.2 TAD MATRIZ

Como foi discutido anteriormente, a implementação de um TAD fica “*escondida*” dentro de seu módulo. Assim, podemos experimentar diferentes maneiras de implementar um mesmo TAD, sem que isso afete os seus clientes. Para ilustrar essa independência de implementação, vamos considerar a criação de um tipo abstrato de dados para representar matrizes de valores reais alocadas dinamicamente, com dimensões m por n fornecidas em tempo de execução. Para tanto, devemos definir um tipo abstrato, que denominaremos de **Matriz**, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- *cria*: operação que cria uma matriz de dimensão m por n ;
- *libera*: operação que libera a memória alocada para a matriz;
- *acessa*: operação que acessa o elemento da linha i e da coluna j da matriz;
- *atribui*: operação que atribui o elemento da linha i e da coluna j da matriz;
- *linhas*: operação que devolve o número de linhas da matriz;
- *colunas*: operação que devolve o número de colunas da matriz.

A interface do módulo pode ser dada pelo código abaixo:

```
/* Arquivo matriz.h */
#ifndef _MATRIZ_H
#define _MATRIZ_H

typedef struct matriz Matriz;

Matriz* cria (int m, int n);
void libera (Matriz* mat);
float acessa (Matriz* mat, int i, int j);
void atribui (Matriz* mat, int i, int j, float v);
int linhas (Matriz* mat);
int colunas (Matriz* mat);

#endif
```

A seguir, mostraremos a implementação deste tipo abstrato usando as duas estratégias apresentadas: matrizes dinâmicas representadas por vetores simples e matrizes dinâmicas representadas por vetores de ponteiros.

A interface do módulo *independe da estratégia de implementação adotada*, o que é altamente desejável, pois podemos mudar a implementação sem afetar as aplicações que fazem uso do tipo abstrato.

O arquivo `matriz1.c` apresenta a implementação através de vetor simples e o arquivo `matriz2.c` apresenta a implementação através de vetor de ponteiros.

```
/* Arquivo matriz1.c */

#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include "matriz.h"

struct matriz
{
    int lin;
    int col;
    float* v;
};
```

```

Matriz* cria (int m, int n) {
    Matriz* mat;
    mat = (Matriz *)malloc(sizeof(Matriz));
    if (mat == NULL) exit(1); /* memória insuficiente */
    mat->lin = m;
    mat->col = n;
    mat->v = (float *)malloc(m*n*sizeof(float));
    return mat;
}

void libera (Matriz* mat) {
    free(mat->v);
    free(mat);
}

float acessa (Matriz* mat, int i, int j) {
    int k; /* índice do elemento no vetor */
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col)
        exit(1); /* acesso inválido */
    k = i*mat->col + j;
    return mat->v[k];
}

void atribui (Matriz* mat, int i, int j, float v) {
    int k; /* índice do elemento no vetor */
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col)
        exit(1); /* atribuição inválida */
    k = i*mat->col + j;
    mat->v[k] = v;
}

int linhas (Matriz* mat) {
    return mat->lin;
}

int colunas (Matriz* mat) {
    return mat->col;
}

```

```

/* Arquivo matriz2.c */

#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include "matriz.h"

struct matriz {
    int lin;
    int col;
    float** v;
};

Matriz* cria (int m, int n) {
    int i;
    Matriz* mat;
    mat = (Matriz *)malloc(sizeof(Matriz));
    if (mat == NULL)
        exit(1); /* memória insuficiente */
    mat->lin = m;
    mat->col = n;
    mat->v = (float **)malloc(m*sizeof(float *));
    if (mat->v == NULL)
        exit(1); /* memória insuficiente */
    for (i=0; i<m; i++) {
        mat->v[i] = (float *)malloc(n*sizeof(float));
        if (mat->v[i] == NULL)
            exit(1); /* memória insuficiente */
    }
}

```

```

    return mat;
}

void libera (Matriz* mat) {
    int i;
    for (i=0; i<mat->lin; i++)
        free(mat->v[i]);
    free(mat->v);
    free(mat);
}

float acessa (Matriz* mat, int i, int j) {
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col)
        exit(1); /* acesso inválido */
    return mat->v[i][j];
}

void atribui (Matriz* mat, int i, int j, float v) {
    if (i<0 || i>=mat->lin || j<0 || j>=mat->col)
        exit(1); /* atribuição inválida */
    mat->v[i][j] = v;
}

int linhas (Matriz* mat) {
    return mat->lin;
}

int colunas (Matriz* mat) {
    return mat->col;
}

```

Cap. 13: TAD LISTA

Em muitas aplicações, é importante a preservação de algumas relações existentes entre os dados. Também, às vezes, não é possível prever-se a demanda por memória durante a execução de um programa.

As Listas são TADs, formados por estruturas que permitem representar um conjunto de dados de forma a preservar a relação de ordem existente entre eles e também prever o seu crescimento e decrescimento.

Existem várias formas de se representar uma Lista. A escolha de uma destas formas dependerá da frequência com que determinadas operações serão executadas sobre ela, uma vez que algumas representações são favoráveis a algumas operações, enquanto que outras não o são, no sentido de se exigir um maior esforço computacional para a sua execução.

Exemplos de Lista:

- Lista telefônica;
- Lista de clientes de uma agência bancária;
- Lista de setores de disco a serem acessados por um sistema operacional;
- Lista de pacotes a serem transmitidos em um nó de uma rede de computação de pacotes.

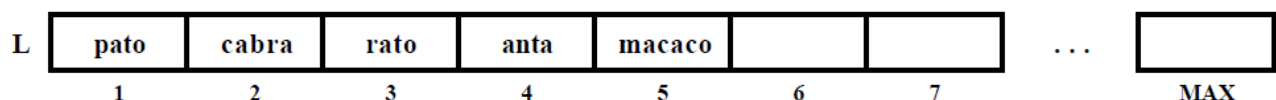
Operações Realizadas com Listas:

- Criar uma lista vazia;
- Verificar se uma lista está vazia;
- Obter o tamanho da uma lista;
- Obter/modificar o valor do elemento de uma determinada posição na lista;
- Obter a posição de elemento cujo valor é dado;
- Inserir um novo elemento após (ou antes) de uma determinada posição na lista;
- Remover um elemento de uma determinada posição na lista;
- Exibir os elementos de uma lista;
- Concatenar duas listas.

Neste capítulo abordaremos inicialmente as listas sequenciais, simplesmente encadeadas, duplamente encadeadas e por fim o conceito e implementação de uma lista genérica (podendo, assim, lidar com dados heterogêneos).

13.1 LISTAS SEQUENCIAIS

Explora a sequencialidade da memória do computador, de tal forma que os nós de uma lista sejam armazenados em endereços contíguos, ou igualmente distanciados um do outro. Esta forma sugere a utilização do tipo vetor na memória principal ou um arquivo sequencial em disco rígido. Veja a figura abaixo.



No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que

precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

Uma interface desse tipo de lista pode ser dada pelo código a seguir:

```
/* Arquivo lista.h */

#ifndef _LISTA_H
#define _LISTA_H

typedef int tipo;
typedef struct lista *Lista;

Lista lista_criar();

void lista_liberar(Lista l);

/* insere um elemento na posicao desejada
   caso a posicao seja invalida, insere no fim */
void lista_inserir(Lista l, tipo v, unsigned int posicao);

/* retira um elemento de uma posicao */
tipo lista_retirar(Lista l, unsigned int posicao);

/* retorna o tamanho da lista */
int lista_tamanho(Lista l);

/* retorna o valor de uma posicao da lista */
tipo lista_valor(Lista l, unsigned int posicao);

/* retorna o indice de um elemento na lista
   caso o elemento nao esteja na lista, retorna -1 */
int lista_indice(Lista l, tipo v);

void lista_imprimir(Lista l);

#endif
```

Uma implementação para estas funções pode ser dada pelo código a seguir:

```
/* Arquivo lista_vetor.c */

#include "lista.h"
#include <stdlib.h>
#include <stdio.h>

#define TAM 1000

struct lista
{
    tipo *vet;
    int tamvet, tamlista;
};

Lista lista_criar()
{
    Lista l;
    l = (Lista) malloc(sizeof(struct lista));
    if (l == NULL) exit(1);
    l->tamvet = TAM;
    l->vet = (tipo *) malloc(sizeof(tipo)*l->tamvet);
    if (l->vet == NULL) exit(1);
    l->tamlista = 0;
    return l;
}
```

```

void lista_liberar(Lista l)
{
    free(l->vet);
    free(l);
}

/* aumenta o tamanho do vetor caso ele esteja no maximo */
void aumenta_vetor(Lista l)
{
    int i;
    tipo *vet;
    l->tamvet += TAM;
    vet = (tipo *) malloc(sizeof(tipo) * l->tamvet);
    if (vet == NULL) exit(1);
    for(i=0; i<l->tamlista; i++)
        vet[i] = l->vet[i];
    free(l->vet);
    l->vet = vet;
}

/* insere um elemento na posicao desejada; caso a posicao seja invalida, insere no fim */
void lista_inserir(Lista l, tipo v, unsigned int posicao)
{
    int i;
    if(l->tamvet == l->tamlista)
        aumenta_vetor(l);
    if(posicao >= l->tamlista)
        l->vet[l->tamlista] = v;
    else
    {
        for(i = l->tamlista; i > posicao ; i--)
        {
            l->vet[i] = l->vet[i-1];
        }
        l->vet[posicao] = v;
    }
    l->tamlista++;
}

/* retira um elemento de uma posicao */
tipo lista_retirar(Lista l, unsigned int posicao)
{
    tipo v;
    int i;
    if(posicao >= l->tamlista)
        exit(-1);
    v = l->vet[posicao];
    for(i = posicao; i < l->tamlista-1; i++)
        l->vet[i] = l->vet[i+1];
    l->tamlista--;
    return v;
}

/* retorna o tamanho da lista */
int lista_tamanho(Lista l)
{
    return l->tamlista;
}

/* retorna o valor de uma posicao da lista */
tipo lista_valor(Lista l, unsigned int posicao)
{
    if(posicao >= l->tamlista)
        exit(-1);
    return l->vet[posicao];
}

```

```

/* retorna o indice de um elemento na lista; caso o elemento nao esteja na lista, retorna -1 */
int lista_indice(Lista l, tipo v)
{
    int i;
    for(i=0; i < l->tamlista; i++)
        if(l->vet[i] == v) return i;
    return -1;
}

void lista_imprimir(Lista l)
{
    int i;
    for(i=0; i < l->tamlista; i++)
        printf("%d ", l->vet[i]);
    printf("\n");
}

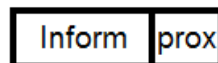
```

13.2 LISTAS SIMPLEMENTE ENCADEADAS

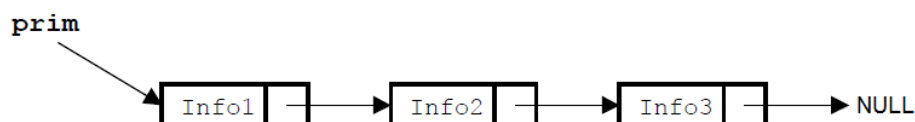
A alocação encadeada nos oferece a necessária flexibilidade para mantermos, sem um grande número de movimentações de nós na lista, a estrutura devidamente ordenada a cada inserção e/ou retirada efetuadas.

Observações:

- A sequência de dados definida pelo encadeamento dos elementos na lista;
- Cada elemento é chamado de nó (nodo) da lista e contém além das informações um apontador para o próximo nó da lista (figura abaixo);



- O início da lista é estabelecido por um apontador para o primeiro nó da lista. Caso a lista esteja vazia, primeiro aponta para NULL;
- Esquemáticamente é mostrada na figura abaixo:



É possível criar uma estrutura tal, capaz de armazenar um item de dado, e ainda, conter um endereço que indica o próximo elemento na sequência.

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de **Nós** (nodos) da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó). Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

Para exemplificar a implementação de listas encadeadas em C, vamos considerar, inicialmente, um exemplo simples em que queremos armazenar **valores inteiros** numa lista encadeada. O nó da lista pode ser representado pela estrutura abaixo:

```

typedef struct lista {
    int info;
    struct lista* prox;
} Lista;

```


Devemos notar que trata-se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definirmos o tipo `Lista` como sinônimo de `struct lista`, conforme ilustrado acima. O tipo `Lista` representa um nó da lista e a estrutura de lista encadeada é representada pelo ponteiro para seu primeiro elemento (tipo `Lista*`).

Considerando a definição de `Lista`, podemos definir as principais funções necessárias para implementarmos uma lista encadeada.

13.2.1 FUNÇÃO DE INICIALIZAÇÃO

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro `NULL`, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é `NULL`. Uma possível implementação da função de inicialização é mostrada a seguir:

```
/* função de inicialização: retorna uma lista vazia */
Lista* inicializa ()
{
    return NULL;
}
```

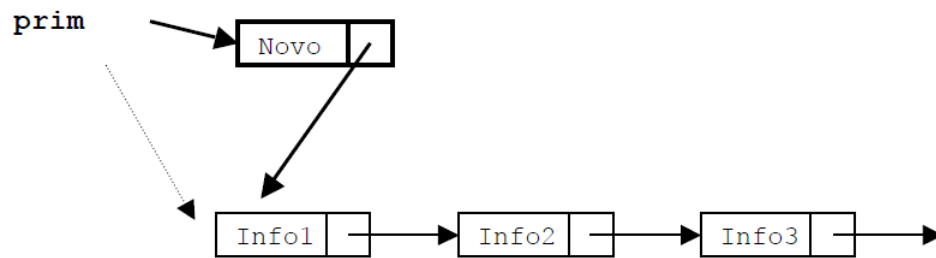
13.2.2 FUNÇÃO DE INSERÇÃO

Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no início: retorna a lista atualizada */
Lista* insere (Lista* l, int i)
{
    Lista* novo;
    novo = (Lista*) malloc(sizeof(Lista));
    if (novo == NULL) exit(1);
    novo->info = i;
    novo->prox = l;
    return novo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A figura abaixo ilustra a operação de inserção de um novo elemento no início da lista.



13.2.3 FUNÇÃO QUE PERCORRE OS ELEMENTOS DA LISTA

Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```

/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
    Lista* p; /* variável auxiliar para percorrer a lista */
    for (p = l; p != NULL; p = p->prox)
        printf("info = %d\n", p->info);
}

```

13.2.4 FUNÇÃO QUE VERIFICA SE LISTA ESTÁ VAZIA

Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é NULL. Uma implementação dessa função é mostrada a seguir:

```

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
    if (l == NULL)
        return 1;
    else
        return 0;
}

```

13.2.5 FUNÇÃO DE BUSCA

Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```

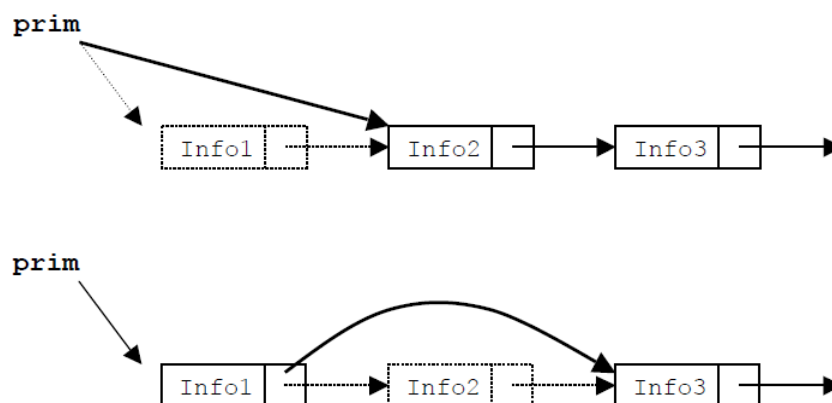
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{
    Lista* p;
    for (p=l; p!=NULL; p=p->prox)
        if (p->info == v)
            return p;
    return NULL; /* não achou o elemento */
}

```

13.2.6 FUNÇÃO QUE RETIRA UM ELEMENTO DA LISTA

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As figuras abaixo ilustram as operações de remoção.



Uma possível implementação da função para retirar um elemento da lista é mostrada a seguir. Inicialmente, busca-se o elemento que se deseja retirar, guardando uma referência para o elemento anterior.

```
/* função retira: retira elemento da lista */
Lista* retira (Lista* l, int v) {
    Lista* p; /* ponteiro para percorrer a lista*/
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    p = l;
    /* procura elemento na lista, guardando anterior */
    while (p != NULL && p->info != v) {
        ant = p;
        p = p->prox;
    }
    /* verifica se achou elemento */
    if (p == NULL)
        return l; /* não achou: retorna lista original */
    /* retira elemento */
    if (ant == NULL) {
        /* retira elemento do inicio */
        l = p->prox;
    }
    else {
        /* retira elemento do meio da lista */
        ant->prox = p->prox;
    }
    free(p);
    return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação anterior.

13.2.7 FUNÇÃO PARA LIBERAR A LISTA

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

```
void libera (Lista* l)
{
    Lista* p;
    Lista* t; /* variavel auxiliar */
    p = l;
    while (p != NULL) {
        t = p->prox; /* guarda referência */
        free(p); /* libera a memória apontada por p */
        p = t; /* faz p apontar para o próximo */
    }
}
```

Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>
#include "lista_enc.h"

int main () {
    Lista* l; /* declara uma lista não iniciada */
    l = inicializa(); /* inicia lista vazia */
    l = insere(l, 23); /* insere na lista o elemento 23 */
    l = insere(l, 45); /* insere na lista o elemento 45 */
    l = insere(l, 56); /* insere na lista o elemento 56 */
    l = insere(l, 78); /* insere na lista o elemento 78 */
    imprime(l); /* imprimirá: 78 56 45 23 */
    l = retira(l, 78);
    imprime(l); /* imprimirá: 56 45 23 */
    l = retira(l, 45);
    imprime(l); /* imprimirá: 56 23 */
    libera(l);
    return 0;
}
```

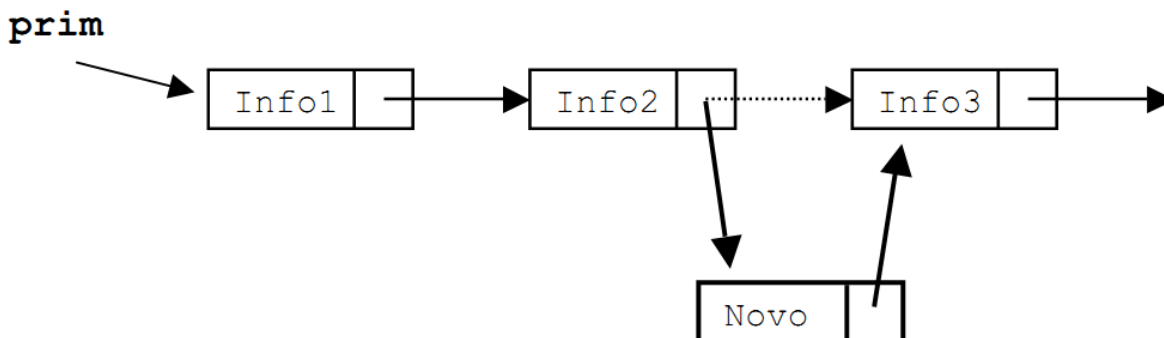
Observe que não podemos deixar de atualizar a variável que representa a lista a cada inserção e a cada remoção de um elemento. Esquecer de atribuir o valor de retorno à variável que representa a lista pode gerar erros graves. Se, por exemplo, a função retirar o primeiro elemento da lista, a variável que representa a lista, se não fosse atualizada, estaria apontando para um nó já liberado. Como alternativa, poderíamos fazer com que as funções insere e retira recebessem o endereço da variável que representa a lista. Nesse caso, os parâmetros das funções seriam do tipo ponteiro para lista (`Lista** l`) e seu conteúdo poderia ser acessado/atualizado de dentro da função usando o operador conteúdo (`*l`).

13.2.8 MANUTENÇÃO DA LISTA ORDENADA

A função de inserção vista acima armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se quisermos manter os elementos na lista numa determinada ordem, temos que encontrar a posição correta para inserir o novo elemento. Essa função não é eficiente, pois temos que percorrer a lista, elemento por elemento, para acharmos a posição de inserção. Se a

ordem de armazenamento dos elementos dentro da lista não for relevante, optamos por fazer inserções no início, pois o custo computacional disso independe do número de elementos na lista.

No entanto, se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente. A função de inserção, neste caso, tem a mesma assinatura da função de inserção mostrada, mas percorre os elementos da lista a fim de encontrar a posição correta para a inserção do novo. Com isto, temos que saber inserir um elemento no meio da lista. A figura abaixo ilustra a inserção de um elemento no meio da lista.



Conforme ilustrado na figura, devemos localizar o elemento da lista que irá preceder o elemento novo a ser inserido. De posse do ponteiro para esse elemento, podemos encadear o novo elemento na lista. O novo apontará para o próximo elemento na lista e o elemento precedente apontará para o novo. O código abaixo ilustra a implementação dessa função. Neste caso, utilizamos uma função auxiliar responsável por alocar memória para o novo nó e atribuir o campo da informação.

```
/* função auxiliar: cria e inicializa um nó */
Lista* cria (int v)
{
    Lista* p = (Lista*) malloc(sizeof(Lista));
    p->info = v;
    return p;
}

/* função insere_ordenado: insere elemento em ordem */
Lista* insere_ordenado (Lista* l, int v)
{
    Lista* novo = cria(v); /* cria novo nó */
    Lista* ant = NULL; /* ponteiro para elemento anterior */
    Lista* p = l; /* ponteiro para percorrer a lista */
    /* procura posição de inserção */
    while (p != NULL && p->info < v) {
        ant = p;
        p = p->prox;
    }
    /* insere elemento */
    if (ant == NULL) { /* insere elemento no início */
        novo->prox = l;
        l = novo;
    }
    else { /* insere elemento no meio da lista */
        novo->prox = ant->prox;
        ant->prox = novo;
    }
    return l;
}
```

Devemos notar que essa função, analogamente ao observado para a função de remoção, também funciona se o elemento tiver que ser inserido no final da lista.

13.2.9 RESUMO DAS SITUAÇÕES DE INSERÇÃO E REMOÇÃO NUMA LISTA SIMPLEMENTE ENCADEADA

Três situações possíveis podem ocorrer quando você **insere** um item em uma lista singularmente encadeada. Primeiro, ele pode tornar-se o novo primeiro item; segundo, ele pode ser inserido entre dois outros itens; terceiro, ele pode tornar-se o ultimo elemento. Lembre-se de que, se você alterar o primeiro item da lista, precisará atualizar o ponto de entrada para a lista em outro ponto do seu programa. A figura abaixo mostra como os elos são alterados em cada caso.

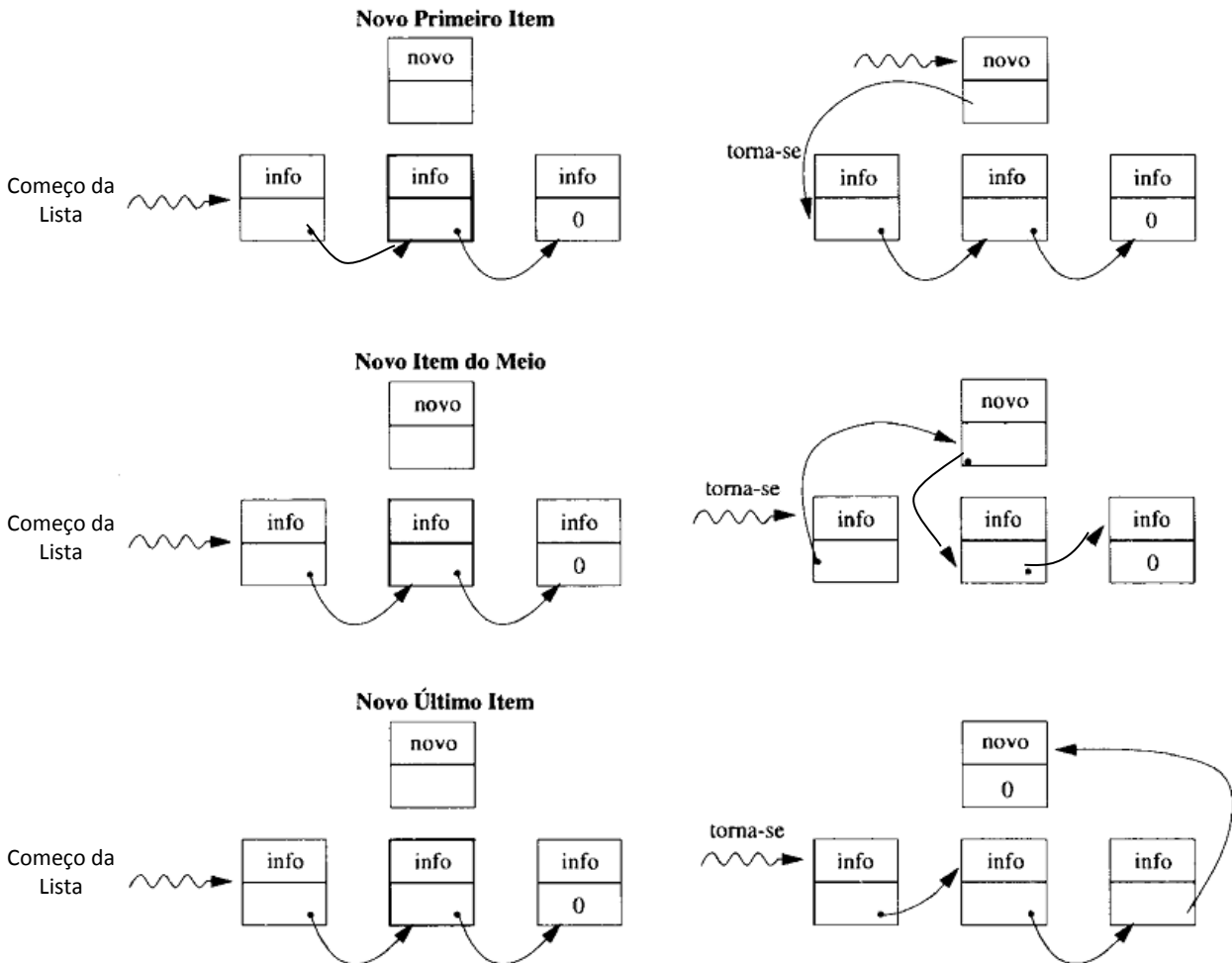


DIAGRAMA DAS OPERAÇÕES DE INSERÇÃO NUMA LISTA SINGULARMENTE ENCADEADA.

De maneira análoga a inserção, a **exclusão** de um item de uma lista encadeada pode acontecer em três situações diferentes, as quais podem ser conferidas no esquema abaixo.

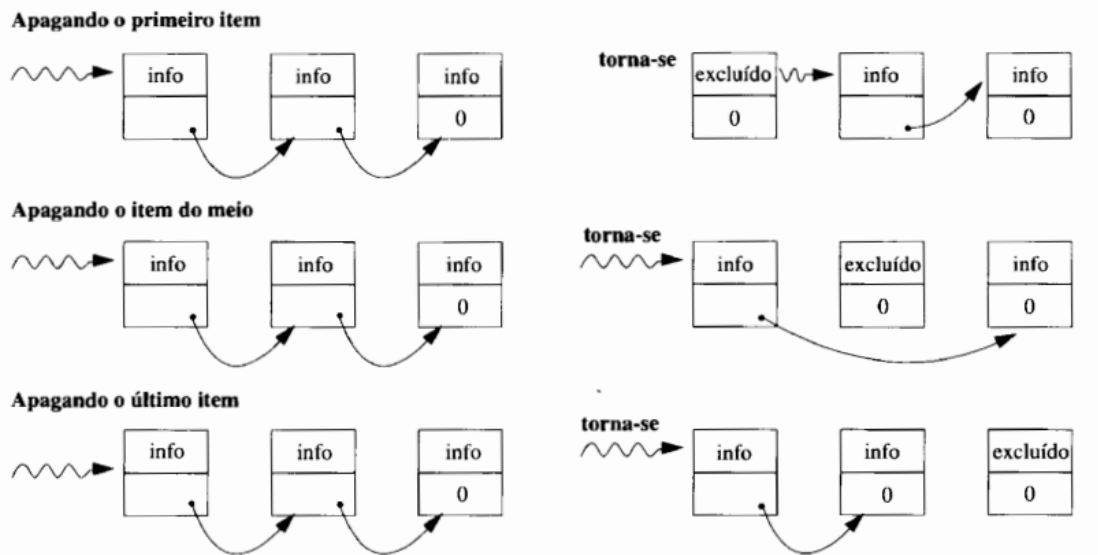


DIAGRAMA DAS OPERAÇÕES DE REMOÇÃO NUMA LISTA SINGULARMENTE ENCADEADA.

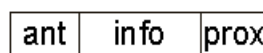
[Fonte: C Completo e Total]

13.3 LISTAS DUPLAMENTE ENCADEADAS

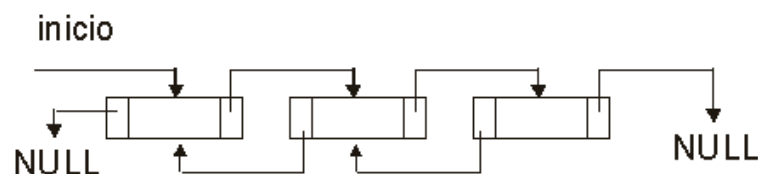
Algumas aplicações requerem características que não podem ser fornecidas pelas listas simplesmente encadeadas como, por exemplo:

- Eliminar um elemento conhecida a sua posição;
- Eliminar o elemento à esquerda de uma posição;
- O percurso da lista em ordem decrescente;

Nesses casos justifica-se o uso de listas duplamente encadeadas, onde cada nó da lista conhece o seu sucessor e o seu antecessor (figura abaixo).



Esquemáticamente é mostrada na figura abaixo:



Com o mesmo arquivo cabeçalho das listas lineares sequenciais (`lista.h`), uma implementação para as mesmas funções, contudo usando listas lineares duplamente encadeadas pode ser dada pelo código a seguir.

```
/* Arquivo lista_encadeada.c */
#include "lista.h"
#include <stdlib.h>
#include <stdio.h>
```

```

typedef struct nodo *Nodo;

struct nodo
{
    Nodo ant, prox;
    tipo v;
};

struct lista
{
    Nodo inicio;
    int tam;
};

Lista lista_criar()
{
    Lista l;
    l = (Lista)malloc(sizeof(struct lista));
    if (l == NULL) exit(1);
    l->inicio = NULL;
    l->tam = 0;
    return l;
}

void lista_liberar(Lista l)
{
    while(lista_tamanho(l)>0)
        lista_retirar(l, 0);
    free(l);
}

/* retorna o tamanho da lista */
int lista_tamanho(Lista l)
{
    return l->tam;
}

/* insere um elemento na posicao desejada caso a posicao seja invalida, insere no fim */
void lista_inserir(Lista l, tipo v, unsigned int posicao)
{
    int i;
    Nodo aux, novo;
    novo = (Nodo) malloc(sizeof(struct nodo));
    if (novo == NULL) exit(1);
    novo->v = v;
    aux = l->inicio;
    if(l->inicio == NULL)
    {
        l->inicio = novo;
        novo->prox = novo->ant = novo;
    }
    else
    {
        if(posicao < l->tam)
        {
            for(i=0; i<posicao; i++)
                aux=aux->prox;
            novo->prox = aux;
            novo->ant = aux->ant;
            novo->ant->prox = novo;
            novo->prox->ant = novo;
            if(posicao == 0)
                l->inicio = l->inicio->ant;
        }
        l->tam++;
    }
}

```



```

/* retira um elemento de uma posicao */
tipo lista_retirar(Lista l, unsigned int posicao)
{
    int i;
    tipo v;
    Nodo aux;
    aux = l->inicio;
    if(posicao >= lista_tamanho(l))
        exit(-1);
    for(i=0; i < posicao; i++)
        aux = aux->prox;
    v = aux->v;
    if(posicao == 0)
    {
        if(l->tam == 1)
            l->inicio = NULL;
        else
            l->inicio = l->inicio->prox;
    }
    aux->prox->ant = aux->ant;
    aux->ant->prox = aux->prox;
    free(aux);
    l->tam--;
    return v;
}

/* retorna o valor de uma posicao da lista */
tipo lista_valor(Lista l, unsigned int posicao)
{
    int i;
    Nodo aux;
    aux = l->inicio;
    if(posicao >= lista_tamanho(l))
        exit(-1);
    for(i=0; i < posicao; i++)
        aux = aux->prox;
    return aux->v;
}

/* retorna o indice de um elemento na lista caso o elemento nao esteja na lista, retorna -1 */
int lista_indice(Lista l, tipo v)
{
    int i;
    Nodo aux;
    aux = l->inicio;
    for(i=0; aux->v != v && i < l->tam; i++)
        aux = aux->prox;
    if(i == l->tam)
        return -1;
    return i;
}

void lista_imprimir(Lista l)
{
    int i;
    Nodo aux;
    aux = l->inicio;
    for(i=0; i<l->tam; i++)
    {
        printf("%d ", aux->v);
        aux = aux->prox;
    }
    printf("\n");
}

```

13.4 LISTAS GENÉRICAS

Um nó de uma lista encadeada contém basicamente duas informações: o encadeamento e a informação armazenada. Assim, a estrutura de um nó para representar uma lista de números inteiros é dada por:

```
struct lista {
    int info;
    struct lista *prox;
};

typedef struct lista Lista;
```

Analogamente, se quisermos representar uma lista de números reais, podemos definir a estrutura do nó como sendo:

```
struct lista {
    float info;
    struct lista *prox;
};

typedef struct lista Lista;
```

A informação armazenada na lista não precisa ser necessariamente um dado simples. Podemos, por exemplo, considerar a construção de uma lista para armazenar um conjunto de retângulos. Cada retângulo é definido pela base *b* e pela altura *h*. Assim, a estrutura do nó pode ser dada por:

```
struct lista {
    float b;
    float h;
    struct lista *prox;
};

typedef struct lista Lista;
```

Esta mesma composição pode ser escrita de forma mais clara se definirmos um tipo adicional que represente a informação. Podemos definir um tipo *Retangulo* e usá-lo para representar a informação armazenada na lista (Observe que podemos “mesclar” o comando `typedef` com a definição do `struct`).

```
typedef struct retangulo {
    float b;
    float h;
} Retangulo;

typedef struct lista {
    Retangulo info;
    struct lista *prox;
} Lista;
```

Aqui, a informação volta a ser representada por um único campo (*info*), que é uma estrutura. Se *p* fosse um ponteiro para um nó da lista, o valor da base do retângulo armazenado nesse nó seria acessado por: *p->info.b*. Ainda mais interessante é termos o campo da informação representado por um ponteiro para a estrutura, em vez da estrutura em si.

```
struct retangulo {
    float b;
    float h;
};

typedef struct retangulo Retangulo;
```

```

struct lista {
    Retangulo *info;
    struct lista *prox;
};

typedef struct lista Lista;

```

Neste caso, para criarmos um nó, temos que fazer duas alocações dinâmicas: uma para criar a estrutura do retângulo e outra para criar a estrutura do nó. O código abaixo ilustra uma função para a criação de um nó.

```

Lista* cria (void)
{
    Retangulo* r = (Retangulo*)malloc(sizeof(Retangulo));
    Lista* p = (Lista*)malloc(sizeof(Lista));
    p->info = r;
    p->prox = NULL;
    return p;
}

```

Naturalmente, o valor da base associado a um nó p seria agora acessado por: `p->info->b`. A vantagem dessa representação (utilizando ponteiros) é que, independente da informação armazenada na lista, a estrutura do nó é sempre composta por um ponteiro para a informação e um ponteiro para o próximo nó da lista.

A representação da informação por um ponteiro nos permite construir listas heterogêneas, isto é, listas em que as informações armazenadas diferem de nó para nó. Diversas aplicações precisam construir listas heterogêneas, pois necessitam agrupar elementos afins, mas não necessariamente iguais. Como exemplo, vamos considerar uma aplicação que necessite manipular listas de objetos geométricos planos para cálculos de áreas. Para simplificar, vamos considerar que os objetos podem ser apenas retângulos, triângulos ou círculos. Sabemos que as áreas desses objetos são dadas por:

$$r = b * h \quad , \quad t = \frac{b * h}{2} \quad , \quad c = \pi * r^2$$

Devemos definir um tipo para cada objeto a ser representado:

```

typedef struct retangulo {
    float b;
    float h;
} Retangulo;

typedef struct triangulo {
    float b;
    float h;
} Triangulo;

typedef struct circulo {
    float r;
} Circulo;

```

O nó da lista deve ser composto por três campos:

- um identificador de qual objeto está armazenado no nó
- um ponteiro para a estrutura que contém a informação
- um ponteiro para o próximo nó da lista

É importante salientar que, a rigor, a lista é homogênea, no sentido de que todos os nós contêm as mesmas informações. O ponteiro para a informação deve ser do tipo genérico, pois não sabemos a princípio para que estrutura ele irá apontar: pode apontar para um retângulo, um triângulo ou um círculo. Um ponteiro

genérico em C é representado pelo tipo `void*`. A função do tipo “ponteiro genérico” pode representar qualquer endereço de memória, independente da informação de fato armazenada nesse espaço. No entanto, de posse de um ponteiro genérico, não podemos acessar a memória por ele apontada, já que não sabemos a informação armazenada. Por esta razão, o nó de uma lista genérica deve guardar explicitamente um identificador do tipo de objeto de fato armazenado. Consultando esse identificador, podemos converter o ponteiro genérico no ponteiro específico para o objeto em questão e, então, acessarmos os campos do objeto.

Como identificador de tipo, podemos usar valores inteiros definidos como constantes simbólicas:

```
#define RET 0
#define TRI 1
#define CIR 2
```

Assim, na criação do nó, armazenamos o identificador de tipo correspondente ao objeto sendo representado. A estrutura que representa o nó pode ser dada por:

```
/* Define o nó da estrutura */
struct listagen {
    int tipo;
    void *info;
    struct listagen *prox;
};

typedef struct listagen ListaGen;
```

A função para a criação de um nó da lista pode ser definida por três variações, uma para cada tipo de objeto que pode ser armazenado.

```
/* Cria um nó com um retângulo, inicializando os campos base e altura */
ListaGen* cria_ret (float b, float h)
{
    Retangulo* r;
    ListaGen* p;
    /* aloca retângulo */
    r = (Retangulo*) malloc(sizeof(Retangulo));
    r->b = b;
    r->h = h;
    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = RET;
    p->info = r;
    p->prox = NULL;
    return p;
}

/* Cria um nó com um triângulo, inicializando os campos base e altura */
ListaGen* cria_tri (float b, float h)
{
    Triangulo* t;
    ListaGen* p;
    /* aloca triângulo */
    t = (Triangulo*) malloc(sizeof(Triangulo));
    t->b = b;
    t->h = h;
    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = TRI;
    p->info = t;
    p->prox = NULL;
    return p;
}
```

```

/* Cria um nó com um círculo, inicializando o campo raio */
ListaGen* cria_cir (float r)
{
    Circulo* c;
    ListaGen* p;

    /* aloca círculo */
    c = (Circulo*) malloc(sizeof(Circulo));
    c->r = r;

    /* aloca nó */
    p = (ListaGen*) malloc(sizeof(ListaGen));
    p->tipo = CIR;
    p->info = c;
    p->prox = NULL;
    return p;
}

```

Uma vez criado o nó, podemos inseri-lo na lista como já vínhamos fazendo com nós de listas homogêneas. As constantes simbólicas que representam os tipos dos objetos podem ser agrupadas numa enumeração (seção 10.5)

```
enum { RET, TRI, CIR};
```

13.4.1 MANIPULAÇÃO DE LISTAS HETEROGÊNEAS

Para exemplificar a manipulação de listas heterogêneas, considerando a existência de uma lista com os objetos geométricos apresentados acima, vamos implementar uma função que forneça como valor de retorno a maior área entre os elementos da lista. Uma implementação dessa função é mostrada abaixo, onde criamos uma função auxiliar que calcula a área do objeto armazenado num determinado nó da lista:

```

#define PI 3.14159

/* função auxiliar: calcula área correspondente ao nó */
float area (ListaGen *p)
{
    float a;      /* área do elemento */
    switch (p->tipo) {
        case RET:
            {
                /* converte para retângulo e calcula área */
                Retangulo *r = (Retangulo*) p->info;
                a = r->b * r->h;
            }
            break;
        case TRI:
            {
                /* converte para triângulo e calcula área */
                Triangulo *t = (Triangulo*) p->info;
                a = (t->b * t->h) / 2;
            }
            break;
        case CIR:
            {
                /* converte para círculo e calcula área */
                Circulo *c = (Circulo)p->info;
                a = PI * c->r * c->r;
            }
            break;
    }
    return a;
}

```

```

/* Função para cálculo da maior área */
float max_area (ListaGen* l)
{
    float amax = 0.0; /* maior área */
    ListaGen* p;
    for (p=l; p!=NULL; p=p->prox) {
        float a = area(p); /* área do nó */
        if (a > amax)
            amax = a;
    }
    return amax;
}

```

A função para o cálculo da área mostrada acima pode ser subdividida em funções específicas para o cálculo das áreas de cada objeto geométrico, resultando em um código mais estruturado.

```

/* função para cálculo da área de um retângulo */
float ret_area (Retangulo* r)
{
    return r->b * r->h;
}

/* função para cálculo da área de um triângulo */
float tri_area (Triangulo* t)
{
    return (t->b * t->h) / 2;
}

/* função para cálculo da área de um círculo */
float cir_area (Circulo* c)
{
    return PI * c->r * c->r;
}

/* função para cálculo da área do nó (versão 2) */
float area (ListaGen* p)
{
    float a;
    switch (p->tipo) {
        case RET:
            a = ret_area(p->info);
            break;
        case TRI:
            a = tri_area(p->info);
            break;
        case CIR:
            a = cir_area(p->info);
            break;
    }
    return a;
}

```

Neste caso, a conversão de ponteiro genérico para ponteiro específico é feita quando chamamos uma das funções de cálculo da área: passa-se um ponteiro genérico que é atribuído, através da conversão implícita de tipo, para um ponteiro específico⁶.

Devemos salientar que, quando trabalhamos com conversão de ponteiros genéricos, temos que garantir que o ponteiro armazene o endereço onde de fato existe o tipo específico correspondente. O compilador não tem como checar se a conversão é válida; a verificação do tipo passa a ser responsabilidade do programador.

⁶ Este código não é válido em C++. A linguagem C++ não tem conversão implícita de um ponteiro genérico para um ponteiro específico. Para compilar em C++, devemos fazer a conversão explicitamente.

Por exemplo:

```
a = ret_area((Retangulo*)p->info);
```

Apêndice A: BASES NUMÉRICAS, SINAIS DIGITAIS E O PC

A.1 O COMPUTADOR E A INFORMAÇÃO

A troca de informações entre os seres humanos e o universo a sua volta diferem bastante da troca de informações entre os dispositivos eletrônicos (câmeras e computadores) a qual estamos acostumados. Esta diferença é referente à forma de representação da informação. Todas as informações que chegam até nossos olhos provem de ondas eletromagnéticas analógicas. Uma informação analógica pode teoricamente assumir qualquer valor numérico real de $-\infty$ a $+\infty$. Isto significa que a faixa de valores que determinada informação analógica pode assumir é infinita. Mesmo que você restrinja os intervalos, como a tensão a que um dispositivo ficará submetido é de 0 a 5V. Pergunta: Quantos valores são possíveis entre 0 e 5? A resposta é: Infinitos. A medida que costumamos fazer é na verdade uma aproximação, limitada obviamente pelo dispositivo responsável pela medição do valor.

Você já deve estar começando a perceber como seria difícil ou mesmo impossível para um computador manipular informações analógicas. Além da complexidade, nunca saberíamos ao certo se o nosso resultado final é de fato rigorosamente válido. O passo chave foi, portanto, a criação de uma nova forma de representação da informação, o que implicou diretamente em um novo sistema numérico.

A.2 SINAIS DIGITAIS E NÚMEROS BINÁRIOS

Um sistema digital é um sistema no qual os sinais tem um número finito de valores discretos (bem definidos, enumeráveis). Como um exemplo elementar, uma balança digital mede o peso através de sinais discretos que indicam a massa (gramas ou kilogramas); por outro lado, uma balança analógica mede o peso através de um sinal contínuo correspondente a posição de um ponteiro sobre uma escala.

Os benefícios dos sistemas digitais são muitos. Por exemplo:

1. A representação digital é bem adequada tanto para processamento numérico como não-numérico de informação. Um exemplo de informação não-numérica é a linguagem escrita, na qual as letras tem valores do alfabeto finito A, B, C... etc..
2. O processamento da informação pode usar um sistema para propósitos gerais (um computador) que seja programado para uma tarefa de processamento particular (como o de imagens), eliminando a necessidade de haver um sistema diferente para cada tarefa.
3. O número finito de valores num sinal digital pode ser representado por um vetor (conjunto de valores) de sinais com apenas dois valores (sinais binários). Por exemplo, os dez valores de um dígito decimal podem ser representados por um vetor de quatro sinais binários (ou bits), da seguinte maneira:

dígito	0	1	2	3	4	5	6	7	8	9
vetor	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

4. Esta representação permite implementações nas quais todos os sinais são binários; em consequência, os dispositivos que processam esses sinais são muito simples (fundamentalmente, apenas chaves com dois estados: aberto e fechado). Falaremos mais sobre números binários à frente.
5. Os sinais digitais são bastante insensíveis a variações nos valores dos parâmetros dos componentes (por exemplo, temperatura de operação, ruído), de modo que pequenas variações na representação física não mudam o valor efetivo.

6. Os sistemas digitais numéricos podem se tornar mais exatos simplesmente aumentando-se o número de dígitos usados na sua representação.

A.2.1 MAIS SOBRE NÚMEROS BINÁRIOS

Antes de compreender melhor como funciona a representação em base binária (base 2), observe como um número qualquer é formado na base em que estamos acostumados a lidar, a base 10. Tome por exemplo o número decimal 123. Veja só como o mesmo é formado:

$$123_{10} = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

Perceba que cada algarismo, em particular, dependendo da posição em que o mesmo se encontra, ele determina um valor relativo. O algarismo 1 na casa das centenas, determina de fato uma centena. O 2 determina duas dezenas e o 3, três unidades. Perceba que a base é dita “dez” uma vez que qualquer número escrito nessa base pode ser desmembrado numa soma de potências de 10, multiplicadas por uma faixa de 10 algarismos definidos (0 – 9). Observe que não é coincidência termos n diferentes algarismos para uma base n.

Pense agora que ao invés de dispor de 10 algarismos diferentes, dispomos de apenas dois. Vamos chamar esses algarismos de 0 (zero) e 1 (um). De forma absolutamente análoga, pense no número 1111011 expresso na base 2. Desmembrando o mesmo temos:

$$1111011_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

O número representado acima na base binária é exatamente o mesmo número 123 representado na base 10 acima. É vital que você observe que quanto menos algarismos eu dispuser para a minha representação, ou seja, quanto menor for a minha base, uma sequência cada vez maior de algarismos é necessária para representar um valor qualquer. Na base binária em especial, cada algarismo, seja ele 0 ou 1, é chamado de bit (**binary digit**).

E como ficam os números não inteiros (ou seja, aqueles com “vírgula”)? A ideia é acima é estendida para expoentes negativos na base 10. Observe:

$$456,78_{10} = 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$$

De maneira análoga, o raciocínio acima pode ser usado em qualquer base, por exemplo, a base 2:

$$101,101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 5,625_{10}$$

Além do mais, sabe-se que, na base dez, para se multiplicar um número pela base, isto é, por dez, basta deslocar a vírgula uma casa para a direita. Na divisão por dez, basta deslocar a vírgula para a esquerda. O mesmo ocorre com qualquer base, em particular com a base dois. Para multiplicar um número por dois, basta deslocar a vírgula uma casa para a direita. Ao deslocar para a esquerda, estamos dividindo por dois.

Não é muito comum descrevermos quantidades binárias em bits. Costumamos expressar os valores em bytes, que são agrupamentos formados por 8 bits. Outros múltiplos também existem e são expressos em relação a byte, alguns deles são:

1 Kilobyte (KB)	2^{10} bytes	1.024 bytes
1 Megabyte (MB)	2^{20} bytes	1.048.576 bytes
1 Gigabyte (GB)	2^{30} bytes	1.073.741.824 bytes
1 Terabyte (TB)	2^{40} bytes	1.099.511.627.776 bytes

Outro importante fato a ser observado no caso dos números binários é que a faixa de valores possível de ser expressa numa dada quantidade de bits é relativamente pequena comparada aos números decimais. Por exemplo, na base dez, dispondo de uma sequência formada por oito algarismos, conseguimos representar 10^8 valores diferentes, ou seja, de 0 a 99.999.999. Já na base binária, com uma sequência de oito dígitos (8 bits), conseguimos representar 2^8 valores diferentes, ou seja, 256 valores distintos (0 a 255).

Imagine um número binário qualquer, de 6 dígitos. Vamos chamá-lo de $b_5b_4b_3b_2b_1b_0$ onde cada b é um dígito, que pode ser 0 ou 1, naturalmente. O valor desse número, na base decimal é:

$$b_5b_4b_3b_2b_1b_0 = b_5 \times 2^5 + b_4 \times 2^4 + b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 = b_5 \times 32 + b_4 \times 16 + b_3 \times 8 + b_2 \times 4 + b_1 \times 2 + b_0 \times 1$$

Repare que, quando b vale 1, a “parcela” entra na soma que formará o número em decimal. Caso seja b seja 0, aquela “parcela” não conta. Voltemos ao caso do 53. Pergunte-se: Qual é o “fator de multiplicação” (1, 2, 4, 8, 16, 32, 64, 128 ...) mais próximo de 53, sem ultrapassar 53? É 32, certo? Então, já temos 32. Assim, $53_{10} = 1b_4b_3b_2b_1b_0$. Falta-nos descobrir os demais dígitos. Repare que o fator de multiplicação do próximo dígito é 16. Pergunte-se: Posso somar 32 + 16 sem ultrapassar 53? A resposta é sim, pois 32 + 16 = 48 que é menor que 53. Portanto, o próximo dígito é também 1. Já temos: $53_{10} = 11b_3b_2b_1b_0$. O próximo fator de multiplicação é 8. Pergunte-se 32 + 16 + 8 é menor que 53? Não, já que 32 + 16 + 8 = 56 que é maior que 53. Logo, a parcela do 8 não entra na conta e o dígito é, portanto, 0. Já temos: $53_{10} = 110b_2b_1b_0$. O raciocínio continua da mesma forma: 32 + 16 + 4 = 52, que é menor que 53. Logo $53_{10} = 1101b_1b_0$. Já que 52 + 2 ultrapassa 53, o penúltimo dígito é 0. E, finalmente, o último dígito é 1. Assim: $53_{10} = 110101_2$ que é, naturalmente, o mesmo resultado que encontramos da primeira forma, com divisões sucessivas. Com um pouco de prática, você verá que a segunda forma é muito mais rápida e prática.

A.4.1.2 PARTE FRACIONÁRIA DO NÚMERO

A conversão da parte fracionária do número será feita, algarismo a algarismo, da esquerda para a direita, baseada no fato de que se o número é maior ou igual a 0,5, em binário aparece 0,1, isto é, o correspondente a 0,5 decimal.

Assim, 0,6 será 0,1__ ..., ao passo que 0,4 será 0,0__ ...

Por que isso? Pense... Você já sabe que:

$2^0 = 1 \mid 2^1 = 2 \mid 2^2 = 4 \mid 2^3 = 8 \mid 2^4 = 16 \mid 2^5 = 32 \dots$ Logo, para formar um número inteiro qualquer, combinamos (ou seja, somamos) essas porções a fim de gerar o número. Ex: $(35)_{10} = (100011)_2$

Agora vamos pensar para o lado das potências negativas, na base 2:

$2^{-1} = 1/2 = 0,5 \mid 2^{-2} = 1/4 = 0,25 \mid 2^{-3} = 1/8 = 0,125 \mid 2^{-4} = 1/16 = 0,0625 \mid 2^{-5} = 1/32 = 0,03125 \dots$ Logo, para formar uma parte fracionária (após a vírgula) qualquer, basta igualmente combinar (somar) os fatores de forma análoga ao que é feito na parte inteira. Observe que, quanto mais negativa se torna o expoente, menor se torna o fator.

Tendo isso como base, basta multiplicar o número por dois e verificar se o resultado é maior ou igual a 1. Se for, coloca-se 1 na correspondente casa fracionária, se 0 coloca-se 0 na posição. Em qualquer dos dois casos, o processo continua, lembrando-se, ao se multiplicar o número por dois, a vírgula move-se para a direita e, a partir desse ponto, estamos representando, na casa à direita, a parte decimal do número multiplicado por dois.

Vamos ao exemplo, representando, em binário, o número 0,625.

$0,625 \times 2 = 1,25$, logo a primeira casa fracionária é 1.

Resta representar o 0,25 que restou ao se retirar o 1 já representado.

$0,25 \times 2 = 0,5$, logo a segunda casa é 0.

Falta representar o 0,5.

$0,5 \times 2 = 1$, logo a terceira casa é 1.

$$0,625_{10} = 0,101_2$$

Quando o número tiver parte inteira e parte fracionária, podemos calcular, cada uma, separadamente.

Tentando representar 0,8, verifica-se que é uma dízima.

$$0,8 = 0,110011001100\dots$$

Da mesma forma, vê-se que $5,8 = 101,11001100\dots$, também uma dízima.

$11,6 = 1011,10011001100\dots$ o que era óbvio, bastaria deslocar a vírgula uma casa para a direita, pois $11,6 = 2 \times 5,8$.

A.4.2 DE BINÁRIO/DECIMAL PARA HEXADECIMAL

A conversão de binário para hexadecimal é bastante simples. Basta agrupar os dígitos em grupos de 4, a partir da direita e completar o último grupo com zeros. O ponto, neste caso, é apenas para melhor visualização. Observe:

$$53_{10} = 0011.0101_2$$

Agora, basta converter cada quarteto para o algarismo hexadecimal correspondente. Assim:

$$53_{10} = 0011.0101_2 = 35_{16}$$

Outro exemplo:

$$12972_{10} = 0011.0010.1010.1100_2 = 32AC_{16}$$

E por último, para rir um pouco:

$$12237514_{10} = 1011.1010.1011.1010.1100.1010_2 = BABACA_{16}$$

A.4.3 DE BINÁRIO/HEXADECIMAL PARA DECIMAL

As conversões de hexadecimal/binário para decimal podem ser facilmente executadas a partir da própria expressão do número na base correspondente, efetuando a soma de produtos, conforme extensamente mostrado nesta seção.

Por exemplo:

$$110101_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53_{10}$$

$$32AC_{16} = 3 \times 16^3 + 2 \times 16^2 + 10 \times 2^1 + 12 \times 2^0 = 12972_{10}$$

A.5 CONVERSÃO DE SINAIS: ANALÓGICO PARA DIGITAL

Uma vez que os sinais do mundo físico são analógicos, é necessário convertê-los para sinais digitais e vice-versa sempre que sistemas digitais tenham de interagir com estes sinais físicos.

Considere como exemplo uma máquina fotográfica digital. Numa visão simplificada, podemos entender o seu funcionamento da seguinte forma:

1. A luz entra através das lentes e atinge um sensor que é formado por uma matriz (linhas e colunas) de microcircuitos baseados em elementos semicondutores (transistores) que são sensíveis à radiação luminosa.
2. Cada um desses sensores é capaz de detectar um comprimento de onda em específico (ou seja, uma cor), bem como a sua intensidade, e converter isso para valores de tensão. No caso de uma imagem colorida, tem-se sensores sensíveis as 3 luzes primárias: *Red* (Vermelho), *Green* (Verde) e *Blue* (Azul). No modelo de cores RGB (usado por monitores, por exemplo), “qualquer” cor pode ser expressa pelas suas 3 componentes (RGB).
3. O próximo passo é de fato a conversão dos valores analógicos de tensão para valores digitais. Esse processo é chamado de quantização ou digitalização.
4. Uma vez digitais, os sinais passam a ser representados por um vetor de bits. Cada ponto de uma imagem é representado por um valor de cor limitado ao número de bits da representação. A imagem

completa é, portanto a enorme sequência formada pela junção de todos os vetores de bits individuais de cada ponto. No fundo, uma imagem digital é uma sequência de 0s e 1s que descrevem a cor de todos os pontos que formam a imagem, um por um.

5. Uma vez tido a representação completa da imagem por uma extensa cadeia de bits, a mesma pode sofrer inúmeros processamentos (manipulações) até chegar ao resultado final, que é efetivamente o arquivo guardado no dispositivo de memória e posteriormente transferido para o computador.

A.6 LÓGICA TEMPORIZADA

Na comunicação entre o processador e memória, as instruções, os dados e os endereços “trafegam” no computador através dos barramentos (de dados, de endereços e de controle), sob a forma de bits representados por sinais elétricos: uma tensão positiva alta (“high” - geralmente no em torno de 5 volts) significando **1** e uma tensão baixa (“low” - próxima de zero) significando **0**. Mas os dados no computador não ficam estáticos; pelo contrário, a cada ciclo (cada “estado”) dos circuitos, os sinais variam, de forma a representar novas instruções, dados e endereços. Ou seja, os sinais ficam estáticos apenas por um curto espaço de tempo, necessário e suficiente para os circuitos poderem detectar os sinais presentes no barramento naquele instante e reagir de forma apropriada. Assim, periodicamente, uma nova configuração de bits é colocada nos circuitos, e tudo isso só faz sentido se pudermos de alguma forma organizar e sincronizar essas variações, de forma a que, num dado instante, os diversos circuitos do computador possam “congelar” uma configuração de bits e processá-las. Para isso, é preciso que exista um outro elemento que forneça uma base de tempo para que os circuitos e os sinais se sincronizem. Este circuito é chamado *clock* - o relógio interno do computador. Cada um dos estados diferentes que os circuitos assumem, limitados pelo sinal do *clock*, é chamado um ciclo de operação.

A.6.1 CLOCK

A Unidade de Controle do processador envia a todos os componentes do computador um sinal elétrico regular - o pulso de “*clock*” - que fornece uma referência de tempo para todas as atividades e permite o sincronismo das operações internas. O *clock* é um pulso alternado de sinais de tensão, gerado pelos circuitos de relógio (composto de um cristal oscilador e circuitos auxiliares).

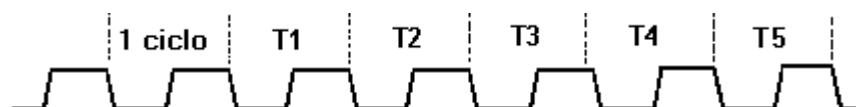
A.6.2 CICLO DE OPERAÇÃO

Cada um destes intervalos regulares de tempo é delimitado pelo início da descida do sinal, e um ciclo é equivalente à excursão do sinal por um “low” e um “high” do pulso.

O tempo do ciclo equivale ao período da oscilação. A física diz que período é o inverso da frequência. Ou seja, $P = 1 / f$.

A frequência f do *clock* é medida em Hertz. Inversamente, a duração de cada ciclo é chamada de período, definido por $P=1/f$ (o período é o inverso da frequência).

Por exemplo, se $f = 10 \text{ Hz}$ logo $P = 1/10 = 0,1 \text{ s}$.



1 MHz (1 megahertz) equivale a um milhão de ciclos por segundo. Sendo a frequência de um processador medida em megahertz, o período será então medido em nanossegundos, como vemos no exemplo abaixo:

$$f = 10 \text{ MHz} = 10 \times 10^6 \text{ Hz}$$

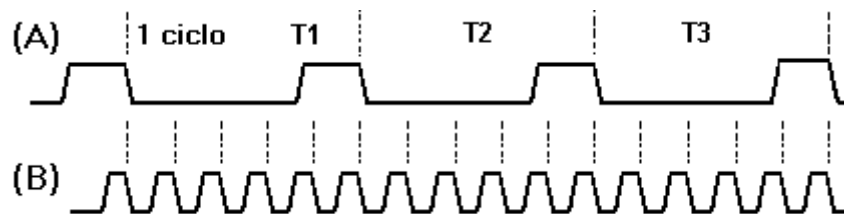
$P = 10 / 10^6 = 0,0000001 \text{ s (segundo)} = 0,0001 \text{ ms (milissegundo)} = 0,1 \text{ } \mu\text{s (microssegundo)} = 100 \text{ ns (nanosegundo)}$

Sempre que se fala sobre máquinas velozes, citamos números em megahertz. Para um melhor entendimento sobre o que ocorre na máquina, em vez de falar sobre a frequência do *clock* seria mais ilustrativo discutirmos outra grandeza: o período (isto é, o tempo de duração de cada ciclo ou simplesmente tempo de ciclo).

Quando se diz que um processador é de 200 MHz, está-se definindo a frequência de operação de seu processador (seu *clock*), significando que o processador pode alternar seus estados internos 200 milhões de vezes por segundo. Isto acarreta que cada ciclo (equivalente a um estado lógico) de operação dura

$$1 / 200.000.000 \text{ s} = 5 \times 10^{-9} \text{ s, ou seja, } 5 \text{ ns.}$$

Como podemos ver pelo exemplo a seguir, o processador com o *clock* ilustrado em (B) teria um tempo de ciclo cinco vezes menor que o (A) e, portanto teria (teoricamente) condições de fazer cinco vezes mais operações no mesmo tempo.



Quando analisamos os números de *clock* de um processador ou barramento, pode ficar uma impressão que esses números não fazem sentido: 133 MHz, 166 MHz... Vejamos como ficam seus períodos, e como esses números apresentam um padrão regular:

Frequência (MHz)	Período (ns)
25	40
33	30
40	25
50	20
66	15
100	10
133	7.5
166	6
200	5
266	3.75

A.7 COMPUTADOR PC: COMPONENTES BÁSICOS

Qualquer PC é composto pelos mesmos componentes básicos: processador, memória, HD, placa-mãe, placa de vídeo e monitor. Essa mesma divisão básica se aplica também a outros aparelhos eletrônicos, como palmtops e celulares. A principal diferença é que neles os componentes são integrados numa única placa de circuito (muitas vezes no mesmo chip) e são utilizados chips de memória flash no lugar do HD.

Antigamente, a placa-mãe funcionava apenas como um ponto central, contendo os slots e barramentos usados pelos demais componentes. Além do processador e pentes de memória, era necessário comprar a placa de vídeo, placa de som, modem, rede, etc. Cada componente era uma placa separada.

Com a integração dos componentes, a placa-mãe passou a incluir cada vez mais componentes, dando origem às placas "tudo onboard" que utilizamos atualmente (existem placas que já vêm até com o processador e chips de memória!). Isso permitiu que os preços dos PCs caíssem assustadoramente, já que, com menos componentes, o custo de fabricação é bem menor. Para quem quer mais desempenho ou recursos, é sempre possível instalar placas adicionais, substituindo os componentes onboard.

Com o micro montado, o próximo passo é instalar o sistema operacional e programas, que finalmente vão permitir que ele faça algo de útil. Vamos começar com um overview da função de cada um destes componentes.

A.7.1 PROCESSADOR

O processador é o cérebro do micro, encarregado de processar a maior parte das informações. Ele é também o componente onde são usadas as tecnologias de fabricação mais recentes.

Existem no mundo apenas quatro grandes empresas com tecnologia para fabricar processadores competitivos para micros PC: a Intel (que domina mais de 60% do mercado), a AMD (que disputa diretamente com a Intel), a VIA (que fabrica alguns chips em pequenas quantidades) e a IBM, que esporadicamente fabrica processadores para outras empresas.

O processador é o componente mais complexo e frequentemente o mais caro, mas ele não pode fazer nada sozinho. Como todo cérebro, ele precisa de um corpo, que é formado pelos outros componentes do micro, incluindo memória, HD, placa de vídeo e de rede, monitor, teclado e mouse.

O transístor é a unidade básica do processador, capaz de processar um bit de cada vez. Mais transistores permitem que o processador processe mais instruções de cada vez enquanto a frequência de operação determina quantos ciclos de processamento são executados por segundo.

A.7.2 MEMÓRIA

Depois do processador, temos a memória RAM, usada por ele para armazenar os arquivos e programas que estão sendo executados, como uma espécie de mesa de trabalho. A quantidade de memória RAM disponível tem um grande efeito sobre o desempenho, já que sem memória RAM suficiente o sistema passa a usar memória swap, que é muito mais lenta.

A principal característica da memória RAM é que ela é volátil, ou seja, os dados se perdem ao reiniciar o micro. É por isso que ao ligar é necessário sempre refazer todo o processo de carregamento, em que o sistema operacional e aplicativos usados são transferidos do HD para a memória, onde podem ser executados pelo processador.



Os chips de memória são vendidos na forma de pentes de memória. Existem pentes de várias capacidades, e normalmente as placas possuem dois ou três encaixes disponíveis. Hoje em dia, 1 GB por módulo é normalmente o mínimo que se encontra disponível no mercado.

Ao contrário do processador, que é extremamente complexo, os chips de memória são formados pela repetição de uma estrutura bem simples, formada por um par de um transístor e um capacitor. Um transístor

solitário é capaz de processar um único bit de cada vez, e o capacitor permite armazenar a informação por um certo tempo. Essa simplicidade faz com que os pentes de memória sejam muito mais baratos que os processadores, principalmente se levarmos em conta o número de transistores.

Todos os micros modernos são equipados com algum tipo de memória DDR, seja ela DDR, DDR2 ou DDR3. Antigamente, na época dos Pentium II e III e os primeiros Athlons e Durons, eram as SDR (tipo mais antigo e mais lento) que dominavam o mercado.

Apesar de fisicamente muito parecidas, é fácil diferenciar os pentes SDR, DDR, DDR2 e DDR3, pois cada uma delas possui um ou mais chanfros que impedem que sejam usadas inadequadamente (uma vez que todas são incompatíveis entre si).

De qualquer forma, apesar de toda a evolução a memória RAM continua sendo muito mais lenta que o processador. Para atenuar a diferença, são usados dois níveis de cache, incluídos no próprio processador: o cache L1, L2 e em alguns modelos de processador, o cache L3.

O cache L1 é extremamente rápido, trabalhando próximo à frequência nativa do processador. Na verdade, os dois trabalham na mesma frequência, mas são necessários alguns ciclos de clock para que a informação armazenada no L1 chegue até as unidades de processamento. Possui alguns poucos KB de capacidade (geralmente menos de 128KB) e é dividido em cache de dados e cache de instruções.

Em seguida vem o cache L2, que é mais lento tanto em termos de tempo de acesso (o tempo necessário para iniciar a transferência) quanto em largura de banda, mas é bem mais econômico em termos de transistores, permitindo que seja usado em maior quantidade (podendo chegar a 1 MB).

De uma forma geral, quanto mais rápido o cache, mais espaço ele ocupa e menos é possível incluir no processador. Em processadores multicore (vários núcleos), cada núcleo possui seu próprio cache L1 e L2.

Por fim, temos o cache L3, cada vez mais comum em processadores modernos, multi-núcleos. Ele é consideravelmente mais lento que os L1 e L2, e se diferencia dos demais principalmente por ser de uso compartilhado entre os núcleos e ter uma capacidade na ordem de vários megabytes.



A.7.3 HD

No final das contas, a memória RAM funciona como uma mesa de trabalho, cujo conteúdo é descartado a cada boot. Temos em seguida o disco rígido, também chamado de hard disk (o termo em Inglês), HD ou até mesmo de "disco duro" pelos nossos primos lusitanos. Ele serve como unidade de armazenamento permanente, guardando dados e programas.

O HD armazena os dados em discos magnéticos que mantêm a gravação por vários anos. Os discos giram a uma grande velocidade e um conjunto de cabeças de leitura, instaladas em um braço móvel faz o trabalho de gravar ou acessar os dados em qualquer posição

nos discos. Junto com o CD-ROM, o HD é um dos poucos componentes mecânicos ainda usados nos micros atuais e, justamente por isso, é o que normalmente dura menos tempo (em média de três a cinco anos de uso contínuo) e que inspira mais cuidados.

Na verdade, os discos magnéticos dos HDs são selados, pois a superfície magnética onde são armazenados os dados é extremamente fina e sensível. Qualquer grão de poeira que chegasse aos discos poderia causar danos à superfície, devido à enorme velocidade de rotação dos discos. Fotos em que o HD aparece aberto são apenas ilustrativas, no mundo real ele é apenas uma caixa fechada sem tanta graça.

Um fato importante é que, apesar de não ser volátil (ao contrário das RAMs que perdem todos os dados assim que o computador é desligado), o HD é muito mais lento que a memória RAM. Enquanto um simples módulo DDR2-533 (PC2-4200) comunica-se com o processador a uma velocidade teórica de 4200 megabytes por segundo, a velocidade de leitura sequencial dos HDs atuais (situação em que o HD é mais rápido) dificilmente ultrapassa a marca dos 100 MB/s, em média.

Para piorar as coisas, o tempo de acesso do HD (o tempo necessário para localizar a informação e iniciar a transferência) é absurdamente mais alto que o da memória RAM. Enquanto na memória falamos em tempos de acesso inferiores a 10 nanosegundos (milionésimos de segundo), a maioria dos HDs trabalha com tempos de acesso superiores a 10 milissegundos.

Aos poucos, os discos de estado sólido (SSD's), que são dispositivos formados por memória flash, aos poucos estão substituindo os HDs em alguns cenários, como para a instalação de sistemas operacionais. Sua popularização só não é mais rápida devido ao alto custo e capacidade significativamente inferior já que, na questão do desempenho, são indiscutivelmente mais rápidos.



A.7.4 PLACA DE VÍDEO

A placa de vídeo é um dos componentes mais importantes do PC. Originalmente, as placas de vídeo eram dispositivos simples, que se limitavam a mostrar o conteúdo da memória de vídeo no monitor. A memória de vídeo continha um simples bitmap da imagem atual, atualizada pelo processador, e o RAMDAC (um conversor digital-analógico que faz parte da placa de vídeo) lia a imagem periodicamente e a enviava ao monitor.

A resolução máxima suportada pela placa de vídeo era limitada pela quantidade de memória de vídeo. Na época, memória era um artigo caro, de forma que as placas vinham com apenas 1 ou 2 MB. As placas de 1 MB permitiam usar no máximo 800x600 com 16 bits de cor, ou 1024x768 com 256 cores. Estavam limitadas ao que cabia na memória de vídeo.

Em seguida, as placas passaram a suportar recursos de aceleração, que permitem fazer coisas como mover janelas ou processar arquivos de vídeo de forma a aliviar o processador principal. Esses recursos melhoram bastante a velocidade de atualização da tela (em 2D), tornando o sistema bem mais responsivo.

Finalmente, as placas deram o passo final, passando a suportar recursos 3D. Imagens em três dimensões são formadas por polígonos, formas geométricas como triângulos e retângulos em diversos formatos. Qualquer objeto em um game 3D é formado por um grande número destes polígonos. Cada polígono tem sua posição na imagem, um tamanho e cor específicos. O "processador" incluído na placa, responsável por todas estas funções é chamado de GPU (Graphics Processing Unit, ou unidade de processamento gráfico).

Apesar de o processador também ser capaz de criar imagens tridimensionais, trabalhando sozinho ele não é capaz de gerar imagens de qualidade a grandes velocidades (como as demandadas por jogos complexos), pois tais imagens exigem um número absurdo de cálculos e processamento. Para piorar ainda mais a situação, o processador tem que ao mesmo tempo executar várias outras tarefas relacionadas com o aplicativo.

As placas aceleradoras 3D, por sua vez, possuem processadores dedicados, cuja função é unicamente processar as imagens, o que podem fazer com uma velocidade incrível, deixando o processador livre para executar outras tarefas. Com elas, é possível construir imagens tridimensionais com uma velocidade suficiente para criar jogos complexos a um alto frame-rate.

Depois dos jogos e aplicativos profissionais, os próximos a aproveitarem as funções 3D das placas de vídeo foram os próprios sistemas operacionais. A idéia fundamental é que, apesar de toda a evolução do hardware, continuamos usando interfaces muito similares às dos sistemas operacionais do final da década de 80, com

janelas, ícones e menus em 2D. Embora o monitor continue sendo uma tela bidimensional, é possível criar a ilusão de um ambiente 3D, da mesma forma que nos jogos, permitindo criar todo tipo de efeitos interessantes e, em alguns casos, até mesmo úteis ;-).

No caso do Windows Vista/7 temos o Aero, enquanto no Linux a solução mais usada é o AIGLX, disponível na maioria das distribuições atuais.

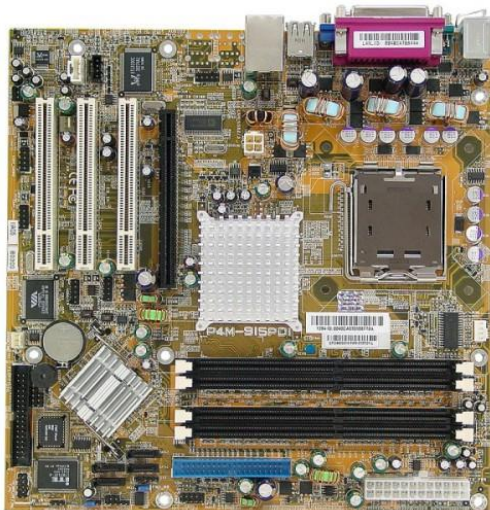
Com a evolução das placas 3D, os games passaram a utilizar gráficos cada vez mais elaborados, explorando os recursos das placas recentes. Isso criou um círculo vicioso, que faz com que você precise de uma placa razoavelmente recente para jogar qualquer game atual.

As placas 3D atuais são praticamente um computador à parte, pois além da qualidade generosa de memória RAM, acessada através de um barramento muito mais rápido que a do sistema, o chipset de vídeo é muito mais complexo e absurdamente mais rápido que o processador principal no processamento de gráficos.

As placas 3D offboard também incluem uma quantidade generosa de memória de vídeo (512 MB ou mais nos modelos mais recentes), acessada através de um barramento muito rápido. O GPU (o chipset da placa) é também muito poderoso, de forma que as duas coisas se combinam para oferecer um desempenho monstruoso.

Longe do mundo brilhante das placas de alto desempenho, temos as placas onboard, que são de longe as mais comuns. Elas são soluções bem mais simples, onde o GPU é integrado ao chipset da placa-mãe (ou, nas arquiteturas mais modernas, dentro do próprio processador) e, em vez de utilizar memória dedicada, como nas placas offboard, utiliza parte da memória RAM principal, que é "roubada" do sistema.

De uma forma geral, as placas de vídeo onboard (pelo menos os modelos que dispõem de drivers adequados) atuais atendem bem às tarefas do dia-a-dia, com a grande vantagem do custo. Elas também permitem rodar os games mais antigos, apesar de, naturalmente, ficarem devendo nos lançamentos recentes. As placas mais caras são reservadas a quem realmente faz questão de rodar os games recentes com uma boa qualidade. Existem ainda modelos de placas 3D específicos para uso profissional, como as nVidia Quadro.



A.7.5 PLACA-MÃE

A placa-mãe é o componente mais importante do micro, pois é ela a responsável pela comunicação entre todos os componentes. Pela enorme quantidade de chips, trilhas, capacitores e encaixes, a placa-mãe também é o componente que, de uma forma geral, mais dá defeitos. É comum que um slot PCI pare de funcionar (embora os outros continuem normais), que instalar um pente de memória no segundo soquete faça o micro passar a travar, embora o mesmo pente funcione perfeitamente no primeiro e assim por diante.

A maior parte dos problemas de instabilidade e travamentos são causados por problemas diversos na placa-mãe, por isso ela é o componente que deve ser escolhido com mais cuidado. Em geral, vale mais a pena investir numa boa placa-mãe e economizar nos demais componentes, do que o contrário.

A qualidade da placa-mãe é de longe mais importante que o desempenho do processador. Você mal vai perceber uma diferença de 20% no clock do processador, mas com certeza vai perceber se o seu micro começar a travar ou se a placa de vídeo onboard não tiver um bom suporte no Linux, por exemplo.

Ao montar um PC de baixo custo, economize primeiro no processador, depois na placa de vídeo, som e outros periféricos. Deixe a placa-mãe por último no corte de despesas.

Antigamente existia a polêmica entre as placas com ou sem componentes onboard. Hoje em dia isso não existe mais, pois todas as placas vêm com som e rede onboard. Apenas alguns modelos não trazem vídeo onboard, atendendo ao público que vai usar uma placa 3D offboard e prefere uma placa mais barata ou com mais slots PCI do que com o vídeo onboard que, de qualquer forma, não vai usar.

Os conectores disponíveis na placa estão muito relacionados ao nível de atualização do equipamento. Placas atuais incluem conectores PCI Express x16, usados para a instalação de placas de vídeo offboard, slots PCI Express x1 e slots PCI, usados para a conexão de periféricos diversos. Placas antigas não possuem slots PCI Express nem portas SATA, oferecendo no lugar um slot AGP para a conexão da placa de vídeo e duas ou quatro portas IDE para a instalação dos HDs e drives ópticos.

Temos ainda soquetes para a instalação dos módulos de memória, o soquete do processador, o conector para a fonte de alimentação e o painel traseiro, que agrupa os encaixes dos componentes onboard, incluindo o conector VGA ou DVI do vídeo, conectores de som, conector da rede e as portas USB.

O soquete (ou slot) para o processador é a principal característica da placa-mãe, pois indica com quais processadores ela é compatível. O soquete é na verdade apenas um indício de diferenças mais "estruturais" na placa, incluindo o chipset usado, o layout das trilhas de dados, etc. É preciso desenvolver uma placa quase que inteiramente diferente para suportar um novo processador.

Existem dois tipos de portas para a conexão do HD: as portas IDE tradicionais, de 40 pinos (chamadas de PATA, de "Parallel ATA") e os conectores SATA (Serial ATA), que são muito menores. Muitas placas recentes incluem um único conector PATA e quatro conectores SATA. Outras incluem as duas portas IDE tradicionais e dois conectores SATA, e algumas já passam a trazer apenas conectores SATA, deixando de lado os conectores antigos.

Existem ainda algumas placas "legacy free", que eliminam também os conectores para o drive de disquete, portas seriais e porta paralela, incluindo apenas as portas USB. Isso permite simplificar o design das placas, reduzindo o custo de produção para o fabricante.

A.7.6 HARDWARE X SOFTWARE

Os computadores são muito bons em armazenar informações e fazer cálculos, mas não são capazes de tomar decisões sozinhos. Sempre existe um ser humano orientando o computador e dizendo a ele o que fazer a cada passo. Seja você mesmo, teclando e usando o mouse, ou, num nível mais baixo, o programador que escreveu os programas que você está usando.

Chegamos então aos softwares, gigantescas cadeias de instruções que permitem que os computadores façam coisas úteis. É aí que entra o sistema operacional e, depois dele, os programas que usamos no dia-a-dia.

Um bom sistema operacional é invisível. A função dele é detectar e utilizar o hardware da máquina de forma eficiente, fornecendo uma base estável sobre a qual os programas que utilizamos no cotidiano possam ser usados. Como diz Linus Torvalds, as pessoas não usam o sistema operacional, usam os programas instalados. Quando você se lembra que está usando um sistema operacional, é sinal de que alguma coisa não está funcionando como deveria.

O sistema operacional permite que o programador se concentre em adicionar funções úteis, sem ficar se preocupando com que tipo de placa de vídeo ou placa de som você tem. O programa diz que quer mostrar uma janela na tela e ponto; o modelo de placa de vídeo que está instalado e que comandos são necessários para mostrar a janela são problema do sistema operacional.

Para acessar a placa de vídeo, ou qualquer outro componente instalado, o sistema operacional precisa de um driver, que é um pequeno programa que trabalha como um intérprete, permitindo que o sistema converse com o dispositivo. Cada placa de vídeo ou som possui um conjunto próprio de recursos e comandos que permitem usá-los. O driver converte esses diferentes comandos em comandos padrão, que são entendidos pelo sistema operacional.

Embora as duas coisas sejam igualmente importantes, existe uma distinção entre o "hardware", que inclui todos os componentes físicos, como o processador, memória, placa-mãe, etc. e o "software", que inclui o sistema operacional, os programas e todas as informações armazenadas.

REFERÊNCIAS BIBLIOGRÁFICAS

- **C Completo e Total**, 3ª Edição, Herbert Schildt, Editora Makron Books.
- **Curso de Linguagem C**, UFMG.
- **Wikipedia [PT]**
 - Algoritmo
 - Algoritmo de Ordenação
 - Linguagem de Programação
 - Paradigma de Programação
 - Pesquisa Binária
- **Apostila de Lógica de Programação - Algoritmos**, Profª Flávia Pereira de Carvalho, Faculdade de Informática de Taquara.
- **Programação (I) - Planejamento e Otimização**, Edvaldo Silva de Almeida Júnior, www.vivaolinux.com.br.
- **Modularização: funções e procedimentos**, <http://iscte.pt/~mms/courses/ip/1998-1999/teoricas/capitulo-3.htm>
- **cplusplus.com** (www.cplusplus.com)
 - printf – C++ Reference, <http://www.cplusplus.com/reference/clibrary/cstdio/printf/>
 - scanf – C++ Reference, <http://www.cplusplus.com/reference/clibrary/cstdio/scanf/>
- **Programação em C – Capítulo 1**, Miguel Pimenta Monteiro, Faculdade de Engenharia da Universidade do Porto.
- **Guia do Hardware**, www.hardware.com.br, <acessado em 20/02/2011>
- **Lógica Temporizada**, <http://wwwusers.rdc.puc-rio.br/rmano/comp0clk.html>, <acessado em 20/02/2011>
- **Professor Raymundo de Oliveira**, <http://www.raymundodeoliveira.eng.br/binario.html>, <acessado em 09/06/2011>
- **Apostila de Estruturas de Dados**, Profs. Waldemar Celes e José Lucas Rangel, PUC-RIO, 2002.
- **Aposta de Algoritmos e Estruturas de Dados**, Prof. Mateus Conrad Barcellos da Costa, IFES/Serra, 2008.
- **Material de Estrutura de Dados**, Prof. José Geraldo Orlandi, IFES/Serra

HISTÓRICO DE VERSÕES

[2012-05-16] - v2.0.0

- Observações a respeito do aprendizado de algoritmos [Cap. 1].
- Comparação C vs. Assembly com exemplo "Hello, World" [Cap. 2].
- Explicações adicionais a respeito do tipo 'char' [Cap. 3].
- Tabela de precedência de operadores resumida [Cap. 3]
- Maiores detalhes a respeito de ponteiros e posições de memória [Cap. 8].
- Seção a respeito de Estruturas reescrita [Cap. 10]
- Capítulo sobre Arquivos completamente reestruturado e acrescentada seção sobre "Estruturação de Dados em Arquivo Texto" [Cap. 11]
- Acrescentada a Parte III, cobrindo uma Introdução aos Tipos Abstratos de Dados.
- Acrescentado capítulo sobre Abstração de Dados [Cap. 12]
- Acrescentado capítulo sobre TAD Lista [Cap. 13]
- Adição do Apêndice A: Bases Numéricas, Sinais Digitais e o PC.
- Tradução/Adaptação de diversos exemplos para melhor entendimento.
- Diversas correções gramaticais/estéticas ao longo dos capítulos.

[2011-08-02] - v1.4.1

- Acrescentada uma importante observação na seção envolvendo passagem de matrizes como parâmetro, no Cap. 9. Corrigido também o exemplo envolvendo vetores, nesta mesma seção.
- Pequenas correções gramaticais.

[2011-03-09] - v1.4.0

- Acrescentada uma árvore "genealógica" exibindo a influência de uma linguagem de programação sobre a outra, no Cap. 2.
- Acrescida uma discussão sobre o uso de C ou C++, na seção C vs C++, do Cap. 2.
- Substituição dos exemplos de função recursiva visando maior inteligibilidade.
- Explicações mais detalhadas a respeito da passagem de matrizes como parâmetro de funções.
- Melhor detalhada, com exemplos e diagramas, a relação existente entre matrizes (2D) e ponteiros, bem como a disposição dos dados linearmente na memória.

[2010-08-09] - v1.2.0

- Correção de pequenos erros ortográficos.
- Explicações sobre o loop for adicionadas como comentário

- Uso da função `fflush(stdin)` para limpeza do buffer de teclado
- Matrizes 2D como argumentos de funções
- Expandida a seção sobre Indireção Múltipla (Ponteiro para Ponteiro)
- Adicionada seção sobre Alocação Dinâmica de Estruturas

[2010-03-23] - v1.0.1

- Correção de pequenos erros ortográficos.

[2010-03-16] - v1.0.0

- Versão Inicial