Curso de Linguagem C
Em Construção
v0.001

Adriano Joaquim de Oliveira Cruz
Instituto de Matemática
Núcleo de Computação Eletrônica
UFRJ

28 de Dezembro de 2007

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Listings

Capítulo 1

Introdução

1.1 Sucessos e Fracassos da Computação

Os objetivos principais deste Capítulo são mostrar para o aluno iniciante alguns aspectos da história da computação e definir, informalmente, termos e palavras-chave que os profissionais da área de computação usam no seu dia a dia. Adriano Cruz ©.

A história do desenvolvimento dos computadores tem sido impressionante. O avanço da tecnologia e a presença da computação na nossa vida são inegáveis. Embora a história deste fantástico desenvolvimento seja recente e bem documentada, há lacunas e controvérsias impressionantes sobre diversos pontos. Neste capítulo iremos ver histórias de espionagem e brigas na justiça por roubo de idéias. Há oportunidades perdidas e gente que soube aproveitar a sua chance. Há verdades estabelecidas que tiveram de ser revistas.

O avanço na tecnologia dos computadores se deu em passos tão largos que os primeiros computadores parecem tão distantes no tempo quanto a Pré-História. O aumento de velocidade, desde os anos 40, foi da ordem de várias ordens de grandeza, enquanto que o custo dos computadores caiu de milhões de dólares para valores em torno de centenas de dólares. As primeiras máquinas tinham milhares de válvulas, ocupavam áreas enormes e consumiam quilowatts de energia. O microprocessador Pentium, lançado em 1993, tinha em torno de 3,1 milhões de transistores, ocupava uma área de aproximadamente 25 cm2 e consumia alguns watts de energia, custando aproximadamente 1000 dólares, somente o microprocessador. A Figura 1.1 mostra a imagem de um circuito integrado de microprocessador Pentium.

No entanto, esta história de redução de tamanho, aumento de velocidade e diminuição de gasto de potência, pode, para alguns pesquisadores, já ter uma data fixada para terminar. Em 1965, Gordon Moore, um dos fundadores da Intel, fabricante do Pentium e uma dos maiores fabricantes de circuitos integrados

Figura 1.1: Fotografia de um circuito integrado de microprocessador Pentium.

do mundo, enunciou o que ficou conhecido como a Lei de Moore: "Cada novo circuito integrado terá o dobro de transistores do anterior e será lançado dentro de um intervalo de 18 a 24 meses."Moore achava que esta lei seria válida somente até 1975, no entanto, ela continua válida até hoje. Na tabela 1.1, pode-se observar a evolução dos microprocessadores usados em nossos computadores.

Ano	**Processador**	**Transistores**
1971	4004	2.250
1972	8008	2.500
1974	8080	5.000
1982	80286	120.000
1985	80386	275.500
1989	80486 DX	1.180.000
1993	Pentium	3.100.000
1997	Pentium II	7.500.000
1999	Pentium III	24.000.000
2000	Pentium 4	42.000.000

Tabela 1.1: Transistores por chip nos microprocessadores da Intel

Os transistores, que compõem os circuitos eletrônicos, estão diminuido de tamanho, e estamos nos aproximando da fronteira final, os elétrons. Já se houve falar em tamanho de transistores medidos em números de elétrons. Devemos nos lembrar que toda a tecnologia atual está baseada em fluxo de elétrons, ou seja uma corrente elétrica. Os fios conduzem correntes de elétrons e os transistores controlam este fluxo. Se o tamanho diminuir além dos elétrons estaremos em outro domínio.

No entanto, na história da computação, muitas promessas não foram cumpridas e falhas gigantescas aconteceram. Como em diversas questões, artistas geniais, apontam o que não conseguimos ou não queremos ver e mostram que o rei está nu. Há uma frase de Picasso que diz: "Computadores são estúpidos,

eles somente conseguem responder perguntas". Esta frase expõe com ironia um fracasso da comunidade de computação que havia prometido criar rapidamente computadores inteligentes, computadores que poderiam questionar-se e nos questionar. Muitos acreditaram nesta promessa e muitos livros de ficção científica foram publicados em que este tipo de computador estaria disponível em um futuro muito próximo. Com notável exemplo podemos citar o filme "2001 - Uma Odisséia no Espaço", de Stanley Kubrik que estreou em 1968 e foi baseado no conto "The Sentinel", escrito em 1950 por Arthur Clark, um dos mestres da ficção científica. Neste filme o enlouquecido computador HAL 9000, que era capaz de ver, falar, raciocinar etc, mata quase todos os tripulantes de uma nave espacial. Ora, já passamos por 2001 e não existe a menor possibilidade de se ver um computador como o HAL ou tão louco de pedra como ele.

Na computação, um exemplo fantástico de sucesso é a Internet. A Inernet está se tornando essencial para o funcionamento do mundo moderno. Freqüentemente ouvimos dizer que ela é o meio de comunicação que mais rapidamente se difundiu pelo mundo. Pode parecer verdade, já que conhecemos tantos internautas e as empresas estão se atropelando para fazer parte desta onda e aproveitar as suas possibilidades. Esta corrida provocou alguns acidentes e muitos sonhos de riqueza se esvaíram no ar. Hoje, pode-se fazer quase tudo pela Internet, namorar, comprar, pagar contas, fazer amigos, estudar, jogar etc. Quem sabe, em um futuro próximo, voltaremos à Grécia Antiga e nos reuniremos em uma enorme praça virtual para, democraticamente, discutir nossas leis, dispensando intermediários.

1.2 Um Pouco da História da Computação

1.2.1 O Início

A primeira tentativa de se criar uma máquina de contar foi o ábaco. A palavra vem do árabe e significa pó. Os primeiros ábacos eram bandejas de areia sobre as quais se faziam figuras para representar as operações. Aparentemente, os chineses foram os inventores do ábaco de calcular. No entanto, há controvérsias, e os japoneses também reivindicam esta invenção, que eles chamam de *soroban*. Além disso há os russos, que inventaram um tipo mais simples, chamado de *tschoty* São conhecidos exemplares de ábaco datados de 2500 A.C. A Figura 1.2 ilustra um exemplar com as suas contas e varetas.

Em 1901 mergulhadores, trabalhando perto da ilha grega de Antikythera, encontraram os restos de um mecanismo, parecido com um relógio, com aproximadamente 2000 anos de idade. O mecanismo parece ser um dispositivo para calcular os movimentos de estrelas e planetas.

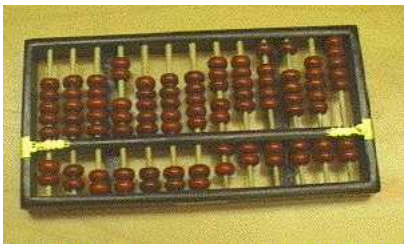

Figura 1.2: Imagem de um ábaco.

1.2.2 A Era Moderna

Em 1614 John Napier, matemático escocês, inventou um dispositivo feito de marfim para demonstrar a divisão por meio de subtrações e a multiplicação por meio de somas. A semelhança entre marfim e ossos, fez com que o dispositivo fosse conhecido como os ossos de Napier.

Um dos primeiros instrumentos modernos de calcular, do tipo mecânico, foi construído pelo filósofo, matemático e físico francês Blaise Pascal (Figura 1.3). Em 1642 aos 19 anos, na cidade de Rouen. Pascal desenvolveu uma máquina de calcular, para auxiliar seu trabalho de contabilidade. A engenhoca era baseada em 2 conjuntos de discos interligados por engrenagens: um para a introdução dos dados e outro para armazenar os resultados. A máquina utilizava o sistema decimal para calcular, de maneira que quando um disco ultrapassava o valor 9, retornava ao 0 e aumentava uma unidade no disco imediatamente superior.

Figura 1.3: Blaise Pascal

Pascal recebeu uma patente do rei da França, o que lhe possibilitou o lançamento de sua máquina no mercado. A comercialização das calculadoras não foi satisfatória devido a seu funcionamento pouco confiável, apesar dele ter construído cerca de 50 versões. As máquinas de calcular, derivadas da Pascalina, como ficou conhecida sua máquina, ainda podiam ser encontradas em lojas até alguns poucos anos atrás. Antes de morrer, aos 39 anos, em 1662, Pascal que contribuíra em vários campos da Ciência, ainda teve tempo de criar uma variante de sua máquina, a caixa registradora.

Em 1666, Samuel Morland adaptou a calculadora de Pascal para resolver multiplicações por meio de uma série de somas sucessivas. Independentemente, em 1671 Leibniz projetou uma outra calculadora que somava e multiplicava. Esta calculadora só foi concluída em 1694.

O primeiro computador de uso específico começou a ser projetado em 1819 e terminou em 1822, ou seja, há mais de 180 anos atrás, pelo britânico Charles Babbage (1791-1871, Figura 1.4), que o batizou de *Difference Engine* (Figura 1.5). A motivação de Babbage era resolver polinômios pelo método das diferenças. Naquele tempo as tábuas astronômicas e outras tabelas eram calculadas por humanos, em métodos tediosos e repetitivos.

Figura 1.4: Charles Babbage

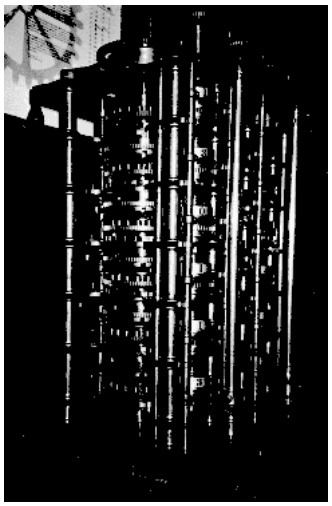

Figura 1.5: Fotografia da *Difference Engine*

Em 1823, ele iniciou o projeto de construir uma outra máquina mais avançada e capaz de calcular polinômios de até sexta ordem. Ele esperava terminar esta

máquina em três anos, mas a construção se arrastou até 1834. Este projeto que não foi completado, usou dinheiro do governo inglês e possivelmente a maior parte da fortuna pessoal de Babbage. A máquina, inteiramente mecânica, teria as seguintes características:

- Arredondamento automático;
- Precisão dupla;
- Alarmes para avisar fim de cálculo;
- Impressão automática de resultados em placas de cobre.

Em 1834 ele tinha completado os primeiros desenhos da máquina que denominou *Analytical Engine* que tinha as seguintes características:

- 50 dígitos decimais de precisão;
- Memória para 1000 destes números (165000 bits);
- Controle por meio de cartões perfurados das operações e endereços dos dados;
- Tempo de soma e subtração igual a 1 segundo; tempo de multiplicação e divisão igual a 1 minuto;
- Sub-rotinas;
- Arredondamento automático e detecção de transbordo (*overflow*);

Babagge nunca conseguiu terminar este ambicioso projeto. No entanto, os mais importantes conceitos de computação, que somente vieram a tona nos anos 40 do século vinte, já tinham sido considerados por Charles Babbage em o seu projeto.

Um fato curioso é que entre os auxiliares de Babagge estava Augusta Ada Byron, Countess of Lovelace. Considera-se, hoje, que ela escreveu para Charles Babbage o primeiro programa para computadores. Ada que mudou seu nome para Augusta Ada King, após seu casamento, estudava Matemática com De-Morgan que provou um dos teoremas básicos da Álgebra Booleana, que é a base matemática sobre a qual foram desenvolvidos os projetos dos modernos computadores. No entanto, não havia nenhuma ligação entre o trabalho do projetista dos primeiros computadores e o do matemático que estudava o que viria a ser o fundamento teórico de todo a computação que conhecemos hoje.

1.2.3 O Desenvolvimento durante as Grandes Guerras

Antes da Segunda Grande Guerra, em vários países, cientistas estavam trabalhando em projetos que visavam construir computadores eletro-mecânicos, ou sejam usavam relés. Alguns destes computadores eram de uso geral, outros tinha finalidades específicas. Alguns destes projetos estão listados a seguir.

Na Alemanha

Em 1934 na Alemanha Konrad Zuze, engenheiro projetista de aviões, concebeu uma máquina de somar para resolver os cálculos que deveria realizar em seus projetos. Em 1938, ele concluiu o Z1, um calculador mecânico com uma unidade aritmética que usava a base 2 para representar os números, base que hoje os computadores modernos empregam. Em 1938 ele melhorou o desempenho do Z1, graças aos relés.

O governo alemão patrocinou os trabalhos de Zuze e em 1941 estava pronto o Z2, uma máquina eletromecânica capaz de receber instruções por meio de uma fita de papel. Em 1941 foi introduzido o Z3, que calculava três a quatro adições por segundo, uma multiplicação em 4 ou 5 segundos e era capaz de extrair a raiz quadrada.

Nos Estados Unidos

Em 1944 a IBM e H. Haiken da Universidade de Harvard, concluiam a construção de um verdadeiro computador: o Harvard Mark I, que operava em base 10. O Mark I efetuava as quatro operações fundamentais, mais o cálculo de funções trigonométricas, exponenciais e logarítmicas. As instruções eram fornecidas por meio de fitas de papel e os dados lidos de cartões perfurados. Os resultados eram fornecidos em forma de cartões perfurados ou impressos por meio de máquinas de escrever.

Em 1943 na Universidade da Pensilvânia, J. Eckert e J. Mauchly iniciaram a construção de um computador à válvulas, ou seja eletrônico. O projeto foi concluído em 1946 e usado na segunda guerra mundial. O ENIAC podia ser reprogramado para executar diversas operações diferentes através de ligações por meio de fios e conectores.

Durante muitos anos o computador ENIAC foi considerado o primeiro computador eletrônico construído. A máquina projetada pelos Drs. Eckert and Mauchly era gigantesca quando comparada com os computadores pessoais atuais. Quando foi terminado, o ENIAC (Figura 1.6) enchia um laboratório inteiro, pesava trinta toneladas, e consumia duzentos quilowatts de potência.

Ele gerava tanto calor que teve de ser instalado em um dos poucos espaços da Universidade que possuia sistemas de refrigeração forçada. Mais de 19000 válvulas, eram os elementos principais dos circuitos do computador. Ele também tinha quinze mil relés e centenas de milhares de resistores, capacitores e indutores. Toda esta parafernália eletrônica foi montada em quarenta e dois painéis com mais 2,70 metros de altura, 60 centímetros de largura e 30 centímetros de comprimento, montados na forma da letra U. Uma leitora de cartões perfurados e uma perfuradora de cartões eram usados para entrada e saída de dados.

Os tempos de execução do ENIAC são mostrados na Tabela 1.2. Compare estes tempos com os tempos dos computadores atuais que estão na ordem de nano segundos, ou 10^{-9} segundos.

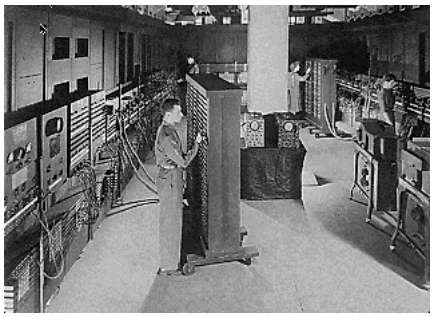

Figura 1.6: Computador Eniac

Operação	**Tempo**
soma	200 μs
multiplicação	2,8 ms
divisão	6,0 ms

Tabela 1.2: Tempo de execução das instruções aritméticas no ENIAC

Em 1939 John Vincent Atanasoff e Clifford E. Berry, na Universidade Estadual de Iowa construíram um protótipo de computador digital eletrônico, que usa aritmética binária. Em 19 de outubro de 1973, o juiz federal Earl R. Larson assinou uma decisão, em seguida a uma longa batalha judicial, que declarava a patente do ENIAC de Mauchly e Eckert inválida e atribuía a Atanasoff a invenção computador eletrônico digital, o ABC ou *Atanasoff-Berry Computer*.

Na Inglaterra

János von Neuman, emigrante húngaro que vivia nos EUA, sugeriu que a memória do computador deveria ser usada para armazenar as instruções do computador de maneira codificada, o conceito de programa armazenado. Esta idéia foi fundamental para o progresso da computação. Os primeiros computadores, como o ENIAC, eram programados por fios que os cientistas usavam para conectar as diversas partes. Quando um programa terminava, estes cientistas trocavam os fios de posição de acordo com a nova tarefa a ser executada. Com o programa armazenado na memória, juntamente com os dados, não era mais necessário interromper as atividades. Carregava-se o programa na memória, uma tarefa extremamente rápida, junto com os dados e dava-se partida no programa. Ao término da execução do programa passava-se imediatamente para a próxima tarefa sem interrupções para troca de fios.

Em 1949, na Inglaterra, dois computadores que usavam a memória para armazenar tanto programas como dados foram lançados. Na Universidade de

Cambridge foi lançado o EDSAC, (Electronic Delay Storage Automatic Calculator) e em Manchester o computador chamado de Manchester Mark I. O EDSAC é considerado o primeiro computador de programa armazenado a ser lançado. Curiosamente, a Universidade de Manchester reinvindica que o primeiro computador de programa armazenado foi o chamado "Baby", um protótipo do Mark I, que começou a operar onze meses antes do EDSAC.

Outro fato curioso em relação à Inglaterra e que foi divulgado recentemente relata que um computador chamado COLOSSUS entrou em operação secretamente na Inglaterra em 1943. Este computador foi usado para auxiliar na quebra dos códigos de criptografia alemães durante a segunda grande guerra.

1.2.4 As Gerações

Costumava-se dividir os projetos de computadores em gerações. Hoje em dia como a taxa de evolução é muito grande não se usa mais este tipo de terminologia. No entanto é interessante mencionar estas divisões.

Primeira Geração: Os computadores construídos com relés e válvulas são os da primeira geração. Estes computadores consumiam muita energia e espaço.

Segunda Geração: Os computadores da segunda geração foram construídos com transistores, os quais tinham a vantagem de serem mais compactos e consumirem muito menos energia. Por gerarem menos calor eram máquinas mais confiáveis.

Terceira Geração: Com o advento dos circuitos integrados, que são componentes em que vários transistores são construídos em uma mesma base de semicondutor, chegamos aos computadores de terceira geração. Com a integração o tamanho dos computadores e seu consumo diminuiu ainda mais e aumentou a capacidade de processamento.

Quarta Geração: Os computadores de quarta geração utilizavam circuitos com a tecnologia VLSI (*Very Large Scale Integration*).

1.3 O Hardware

Um típico diagrama em blocos de um computador digital monoprocessado esta mostrado na Figura 1.7. A **Unidade Central de Processamento** (UCP), em inglês *Central Processing Unit* (CPU), como o próprio nome diz, é a unidade onde os dados são processados, ou seja alterados, no computador. Ou seja, dentro das UCPs os dados são somados, subtraídos etc. A UCP também controla a movimentação dos dados dentro de todo o sistema. Os modulos que constituem a UCP são os seguintes:

Unidade de Controle (UC): comanda a operação do computador. Esta unidade lê da memória tanto as instruções como os dados e comanda todos os circuitos para executar cada instrução lida da memória. Atualmente as unidades de controle são capazes de executar mais de uma instrução por ciclo de máquina, o que caracteriza o processamento paralelo. A técnica mais comum para conseguir este paralelismo é conhecida como *pipelining*, que será detalhada mais adiante.

Unidade Aritmética e Lógica (UAL): local onde as transformações sobre os dados acontecem. Atualmente esta unidade é bastante sofisticada e diversos métodos são empregadas para acelerar a execução das instruções. Alguns processadores duplicam circuitos para permitir que mais de uma operacão aritmética seja executada por vez. É muito comum, por exemplo, a existência de uma unidade aritmética para executar instruções que operam sobre números inteiros e outra sobre números reais, chamada de unidade de ponto flutuante. As vezes a UCP conta com mais de uma de cada uma destas unidades.

Unidade de Entrada e Saída (UES): controla a comunicação com os usuários do computador e os equipamentos periféricos tais como discos e impressoras. Em alguns computadores mais simples esta unidade não existe independentemente, sendo distribuída pela Unidade Central de Processamento. Em computadores mais poderosos ao contrário as Unidades de Entrada e Saída são computadores completos que foram programados para controlar a comunicação com os periféricos.

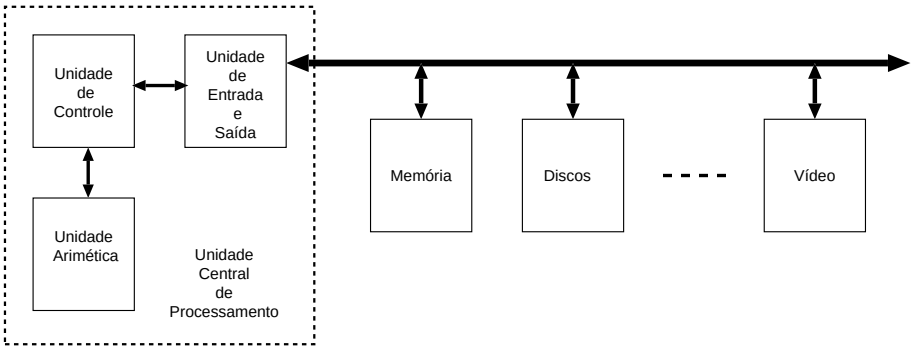

Figura 1.7: Diagrama Básico de um Computador Digital

O termo *pipelining* que mencionamos acima se refere a um dos modos de como o processador pode paralelizar a execução de instruções. Este termo pode ser traduzido por linha de montagem, porque é exatamente isto que a técnica faz, uma linha de montagem de instruções. Por exemplo, em uma linha de montagem de uma fábrica de automóveis, mais de um automóvel é montado

ao mesmo tempo. No início da linha o carro não existe. Em cada estágio da linha de montagem os operários vão adicionando partes e ao fim da linha sai um carro novo. Se você olhar a linha poderá ver carros em diversos estágios da montagem. Repare que a linha não pára, e a montagem de um carro não espera que o que começou a ser montado antes dele esteja terminado. Nas linhas de montagem muitos carros são montados ao mesmo tempo. O efeito final é que cada carro continua levando bastante tempo para ser montado, mas como vários vão sendo montados ao mesmo tempo, alguém que se colocasse no final da linha de montagem teria a impressão que cada carro leva muito pouco tempo para ser montado. Isto porque no final da linha de montagem sai um carro a cada poucos minutos. O mesmo acontece com a linha de montagem de instruções.

1.3.1 Microcomputadores

Uma UCP integrada em um único circuito (*chip*) é comumente chamada de **microprocessador**. Os microprocessadores atuais incluem outros circuitos que normalmente ficavam fora da UCP, tais como processadores de ponto flutuante e memórias cache. Alguns exemplos de microprocessadores são mostrados na Tabela 1.3

Microprocessador	**Empresa**
Arquitetura Intel X86	Intel, AMD
PowerPC	Consórcio Apple/IBM/Motorola
Power	IBM
MIPS	MIPS Technologies
ARM	ARM Technologies
SPARC	SUN

Tabela 1.3: Exemplos de Microprocessadores

Usualmente se chama de computador, o processador mais os seus periféricos e os sistemas para controlar estes periféricos, ou seja todo o sistema de processamento de dados. Os periféricos são os dispositivos usados para fazer a entrada e saída dos dados que serão processados.

Os **microcomputadores** são computadores baseados em **microprocessadores**. As assim chamadas placas mãe dos microprocessadores atuais incluem diversos componentes que formam o microcomputador. Por exemplo, uma placa mãe pode incluir o microprocessador e seus circuitos de suporte, que, no conjunto são conhecidos como o *chipset*. Além disso a placa mãe pode incluir também a memória principal e as placas de controle de periféricos, como a placa de vídeo, e os conectores para os periféricos, enfim, quase todo o sistema.

1.3.2 Memórias

Os dados no computador podem ser armazenados em diversos níveis de memória semicondutoras ou em periféricos (discos, fitas, etc). Quanto mais rápida a memória, usualmente mais cara ela é. A idéia por traz da separação em níveis é colocar mais perto do processador, em memórias rápidas e mais caras, os dados que o processador irá precisar mais freqüentemente. A medida que vamos nos afastando do processador as memórias vão, ao mesmo tempo, ficando mais baratas, aumentando de capacidade e diminuindo de velocidade. A Figura 1.8 ilustra esta hierarquia.

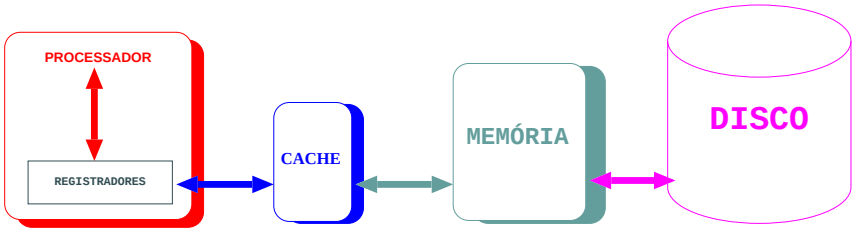

Figura 1.8: Níveis de hierarquia da memória de um computador.

No nível mais próximo do processador estão os registradores, que, na verdade, ficam internamente ao processador. São poucos e extremamente rápidos e, portanto, não podem armazenar grandes quantidades de dados. Somente os dados que são necessários ao processador com rapidez extraordinária devem ser colocados nos registradores.

Durante o processamento normal, é na memória principal onde o processador busca instruções e dados de um programa para executar. Além da memória principal estão os discos. Devido a sua velocidade menor que a da memória principal e a sua grande capacidade, os discos são considerados dispositivos de armazenamento secundário. Os discos são memórias de armazenamento permanente, isto é, quando os computadores são desligados o seu conteúdo se mantém. Ao contrário, a memória principal e os registradores são memórias semicondutoras e perdem seus conteúdos quando a energia elétrica é desligada.

Para acelerar o acesso aos dados freqüentemente usados, os computadores dispõem de memórias mais rápidas, porém de menor capacidade, que ficam entre os registradores e a memória principal. Estas funcionam como as bolsas ou carteiras em que carregamos documentos e outros itens que precisamos freqëntemente. Este tipo de memória é conhecido como memória cache. O cache opera de forma invisível para o processador. Ao pedir um dado para memória, circuitos especiais verificam se este dada está no cache, caso esteja, ele é repassado para o processador. Para o processador o que aconteceu é que a memória entregou o dado com uma rapidez maior do que o normal.

Uma memória é como se fosse uma série de cofres numerados capazes de

armazenar os dados, como está ilustrado na Figura 2.3. Os dados e instruções na memória são apontados ou referenciados por estes números, conhecidos como *endereços*. Ou seja, para ler um dado da memória é necessário fornecer um endereço para que a memória possa encontrar e devolver o conteúdo pedido. Isto é similar ao o que ocorre quando enviamos uma carta, o endereço faz com que o carteiro saiba onde ele deve entregar a correspondência. Em operação normal, toda vez que o processador precisa de um dado ele envia um pedido de leitura à memória junto com o endereço da memória onde o dado está. Nas escritas o processador envia o endereço, o dado e pedido de escrita. Normalmente a memória somente pode atender um pedido de cada vez. Portanto, para ler 1000 números o processador terá de fazer 1000 acessos sequencialmente.

Os dois tipos básicos de memória mais comuns são ROM e RAM. Estas siglas têm diversas variações (PROM, EPROM, DRAM, etc), mas os princípios básicos são os mesmos. Estas siglas indicam os dois tipos básicos de memória que são usadas em computadores. A sigla básica ROM significa *Read Only Memory*, ou seja memória de somente de leitura e RAM (*Random Access Memory*) que significa memória de acesso randômico, ou seja memória que se pode ler em qualquer endereço.

A sigla RAM é muito confusa porque em uma memória ROM também se pode ler em qualquer endereço. A diferença real é que nas RAMs se pode ler e escrever com a mesma velocidade em qualquer endereço, enquanto que na ROM, o acesso é rápido somente para leituras, a escrita é uma história mais complicada. A ROM normalmente contém dados que não podem ser modificados durante o funcionamento do computador. Outro tipo de dados armazenados em ROMs são os que não devem ser perdidos quando o computador é desligado. Exemplos de uso de ROM são as memórias que armazenam os programas que são executados quando os computadores são ligados, os famosos BIOS (*Basic Input Output System*). Um computador ao ser ligado deve ter um programa mínimo capaz de iniciar o seu funcionamento normal, caso contrário seria como uma pessoa que perdeu totalmente a memória. Para isto são escritos programas simples que fazem acesso aos periféricos em busca do Sistema Operacional da máquina.

As primeiras memórias do tipo ROM eram gravadas nas fábricas e nunca mais eram modificadas. Isto trazia algumas dificuldades, por exemplo, quando um programa precisava ser atualizado. Para resolver este tipo de problemas surgiram as PROMs, que são ROMs programáveis. Ou seja é possível desgravar o conteúdo antigo e gravar novos programas nesta memória. Antigamente este era um processo complicado e exigia que a memória fosse retirada fisicamente do circuito e colocada em dispositivos especiais capazes de apagar o conteúdo antigo. Em seguida um circuito programador de PROMs era usado para gravar o novo conteúdo e somente após tudo isto a memória era recolocada no local. O computador ficava literalmente sem a memória dos programas iniciais. Hoje em dia existm PROMs que podem ser apagadas e regravadas muito facilmente. Por exemplo, as EEPROMs (Eletricaly Erasable PROMs), que são memórias que podem ser apagadas eletricamente sem a necessidade de serem retiradas dos circuitos. *Flash memory* é uma forma de memória não volátil que pode

ser apagada e reprogramada eletricamente. Diferentemente das EEPROMs, ela deve ser apagada em blocos de endereços. Este tipo de memória custa menos do que EEPROMs e portanto são preferidas quando é necessário usar memória não volátil em forma de circuitos integrados.

As memórias RAMs são as memórias onde os nossos programas comuns rodam. Elas são modificáveis e de acesso rápido tanto na leitura quanto na gravação. Muitas siglas aparecem e desaparecem quando falamos de memórias RAM. Existem as DRAM, memórias EDO, SIMM, etc. Tudo isto ou se refere ao método de acesso dos dados na memória ou a tecnologia de construção ou a outra caracterítica acessória. O certo é que todas elas tem como caracterítica básica o fato dos acessos de leitura e escrita poderem ser feitos na mesma velocidade.

1.3.3 Bits e Bytes

A memória do computador é composta de **bits**, a menor unidade de informação que o computador armazena. Um bit pode conter o valor 0 ou 1, que são os dígitos usados na base dois, a base usada nos computadores. Um conjunto de 8 bits forma o **byte**. Uma **palavra de memória** é um conjunto de bytes. Atualmente a maioria dos computadores tem palavras de memória com largura de 32 (4 bytes) ou 64 (8 bytes) bits. Na Figura 1.9 mostramos os diversos tamanhos dos dados.

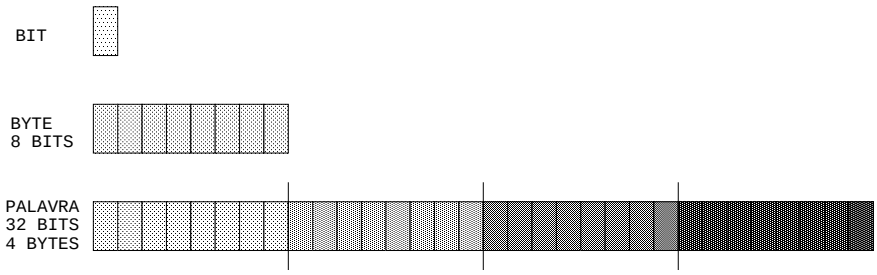

Figura 1.9: Tamanho de Bits, Bytes e Palavras

Observar que estamos falando de dados na memória e não do tamanho dos dados que o computador pode processar. Considere que este tamanho é relacionado com a quantidade máxima de algarismos que um número pode ter para ser processado. Um computador pode ter capacidade de processar 64 bits de cada vez. Caso sua memória tenha palavras de 32 bits o processador deverá, então, ler duas palavras da memória para poder processar um número. Lembre-se que as duas leituras são atendidas uma de cada vez. Da mesma forma o computador pode processar 32 bits de cada vez e a memória ter largura 64 bits. Isto pode acelerar o processamento, já que o processador está se adiantando e recebendo o que poderá ser o próximo dado a ser processado, ou seja economizando uma leitura.

Devido a base 2 o fator kilo tem um significado diferente em computação. Por exemplo 1 Kbyte de memória corresponde a 2 elevado a 10 (2^{10}), ou seja 1024 bytes. Da mesma forma 1 Megabyte corresponde a 1024 x 1024 bytes e 1 Gigabyte é igual a 1024 x 1024 x 1024 bytes. Na Tabela 1.4 estão mostradas as diversas abreviações usadas quando se fazem referências às memórias.

Nome	**Símbolo**	**Multiplicador**
Kilobyte	Kb	$2^{10} = 1024$
Megabyte	MB	2^{20}
Gigabyte	GB	2^{30}
Terabyte	TB	2^{40}
Petabyte	PB	2^{50}
Exabyte	PB	2^{60}

Tabela 1.4: Abreviações usadas em referências às memórias.

1.3.4 Periféricos

Como já mencionamos antes, os dados não ficam guardados somente na memória, há também os periféricos. Há periféricos de entrada, outros de saída e alguns que servem tanto para entrada como saída de dados. Periféricos não servem somente para armazenar dados. Há periféricos que são usados para permitir a interação entre os usuários e o computador. A tabela 1.5 ilustra alguns destes periféricos.

Entrada	**Saída**	**Ambos**
Teclados	Impressoras	Discos Rígidos
Mouse	Vídeo	Fitas Magnéticas
CD-ROM	Plotter	Disquetes
Scanner	Alto-falantes	Discos Zip

Tabela 1.5: Exemplos de periféricos

1.4 O Software

Tudo isto que sobre o que acabamos de escrever constitui o *hardware* do computador, o que se vê e o que se toca. A partir de agora falaremos brevemente no *software*, o que não se vê nem se toca, mas também está lá.

Para que um computador execute alguma tarefa primeiro se desenvolve um algoritmo, que é a receita de como o problema deve ser resolvido. Em seguida

este algoritmo deve ser traduzido para uma linguagem que possa ser entendida pelo computador ou que possa ser traduzida para esta linguagem. No início da computação eletrônica com programas armazenados, estes eram escritos diretamente em linguagem de máquina que é a linguagem que o computador realmente ëntende". Estas instruções são conjuntos de bits indicando a operação que deve ser executada e, caso necessário, onde como achar os os dados que serão operados. Por esta razão também costuma-se dizer que são programas escritos em binário.

Com a evolução da computação os programas passaram a ser escritos em *assembly*, que é uma representação em mnemônicos das instruções de máquina. Deste modo era é mais fácil escrever os algoritmos. Por exemplo, um fragmento de um programa escrito em *assembly* do processador PowerPC é:

```
li r3,4      * O primeiro numero a ser somado e 4.
li r4,8      * 8 e o segundo numero
add r5,r4,r3 * Some os conteúdos de r3 (4) e r4 (8)
             * e armazene o resultado em r5
```

Este pequeno trecho de programa armazena os numeros 4 e 5 em registradores internos do processador em seguida os soma e armazena o resultado em um terceiro registrador. As informações após os asteriscos são comentários usados para explicar o que o programa está fazendo naquela instrução.

O PowerPC é um microprocessador criado em 1991 por um consórcio formado pela IBM, Apple e Motorola Os microprocessadores PowerPC podem ser usados para equipar desde sistemas embutidos até computadores de alto desempenho. A Apple usou este microprocessador para equipar suas máquinas até 2006.

Um programa escrito em *assembly* deve ser traduzido para a representação binária, tarefa que normalmente se chama de *montar* o programa. A palavra *assembler* frequentemente é usada erradamente para significar a linguagem e não o programa que traduz o programa de *assembly* para linguagem binária de máquina. Este tipo de programação pode levar a se escrever programas muito eficientes, devido ao controle quase que total do programador sobre a máquina. No entanto devido ao fato de ser uma linguagem próxima do computador e afastada da maneira de raciocionar do ser humano é mais dificil de ser usada. Além deste fato há outros problemas tais como: dificuldade de leitura por humanos, dificuldade de manutenção dos programas, maior tempo de desenvolvimento etc.

Para evitar estes problemas foram desenvolvidas as linguagens de programação chamadas de linguagens de alto nível, por estarem mais próximas da lingugagem natural empregada pelos serem humanos. Alguns exemplos de linguagens de programação são:

Fortran: Usada em programação científica e engenharia;

Pascal: Usada em ensino de linguagens e desenvolvimento de sistemas;

COBOL: Usada em ambientes comerciais;

Basic: O nome diz tudo, básica;

C: Mesmas características do Pascal com facilidades que permitem mais controle do computador;

C++: Linguagem originária do C com metodologia de orientação à objetos;

Java: Linguagem também baseada na sintaxe do C e também seguindo o modelo de orientação à objetos.

Delphi: Linguagem originária do Pascal com metodologia de orientação à objetos;

Lisp e Prolog: Linguagens usadas para desenvolver programas de Inteligência Artificial.

Aplicativos importantes para os programadores são os compiladores. Estes programas traduzem programas escritos em linguagens de alto nível para a linguagem de máquina, de modo que o computador possa executá-los. De maneira geral um compilador é um programa que traduz um programa de uma linguagem para outra.

Podemos resumir os passos necessários para criar um programa em uma linguagem de programação, por exemplo `C`, nos passos descritos a seguir. A Figura 1.10 ilustra a ordem em que se desenvolvem estes passos.

Criação do Algoritmo: neste passo é criado o algoritmo que irá resolver o problema. As diversas maneiras de descrever um algoritmo serão apresentadas no próximo capítulo.

Codificação do Algoritmo: O algoritmo preparado no passo anterior é escrito em uma linguagem de programação. Neste passo o programador conta, normalmente, com a ajuda de um editor de textos (não processador de textos). Para esta edição qualquer editor pode ser usado. Hoje em dia muitos ambientes de desenvolvimento integram todas as ferramentas necessárias para criar um programa, inclusive o editor, em um único aplicativo.

Compilação do Programa: O arquivo texto contendo o programa passa por um programa especial chamado compilador que gera, caso não hajam erros, uma saída que é quase o programa executável, ou seja o programa em código binário do processador em que será executado. Os erros mais comuns nesta etapa são erros de uso correto da linguagem de programação. Estes erros são chamados de erros de compilação. As linguagens de programação são baseadas em regras gramaticais muito rígidas e qualquer violação destas regras pode implicar em erro. No caso de erros serem encontrados o programador deve voltar ao passo de codificação para a correção dos erros.

Ligação: Em inglês esta passo é conhecido por *link edition*. Um programa completo é composto por vários módulos que podem ter sido criados pelo próprio programador ou por outras programadores. Por exemplo, em `C` os trechos de programa que interagem com os usuários, os comandos de entrada e saída de dados, normalmente vêm com o programa compilador. Estes trechos podem estar guardados em bibliotecas de programas e são ligados ao programa do usuário para completar o programa.

Depuração e Testes: Nesta etapa o programa será testado para a retirada dos possíveis erros de lógica que o programador cometeu. Caso algum erro seja encontrado o programador deve reelaborar o que estiver errado no algoritmo e em seguida ir para a etapa de codificação do algoritmo. Este ciclo pode repetir-se inúmeras vezes até que o desenvolvedor acredite que os erros foram corrigidos.

Uso do Programa: O programa foi entregue aos seus usuários para ser usado. Durante o uso, erros que não foram encontrados durante o desenvolvimento do programa podem ser descobertos e precisam ser corrigidos. A correção pode ser feita pelos mesmos programadores que desenvolveram o programa ou por outro grupo devidamente treinado. Costuma-se chamar esta correção de manuntenção do programa.

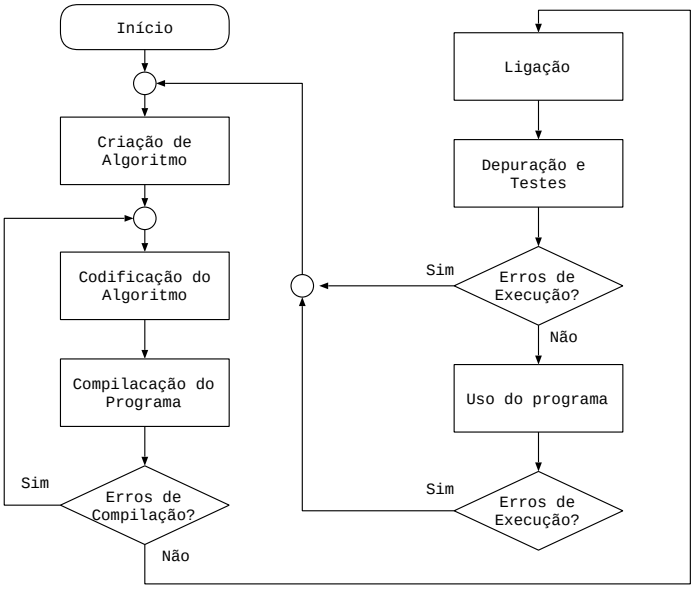

Figura 1.10: Ciclo de desenvolvimento de um programa.

Algumas linguagens de programação não são compiladas e sim interpretadas. Isto significa que o programa não para ser executado não precisa ser traduzido

para linguagem de máquina. Em um programa interpretado um aplicativo lê o programa instrução por instrução diretamente na própria linguagem de alto nível, traduz cada uma destas intruções para linguagem de máquina e as executa. Não há, portanto, o processo de tradução antecipada do programa. A interpretação de um programa funciona como o processo de tradução simultânea do discurso de um orador. A medida que ele pronuncia seu discurso um tradutor repete as frases na linguagem destino. Um programa compilado funciona como se primeiro, o tradutor traduzisse todo o discurso e depois o lesse. A linguagem Basic é uma linguagem interpretada. Em Java ocorre um processo um pouco diferente. Um programa em Java é traduzido para uma linguagem intermediária e depois interpretado por meio de uma chamada máquina virtual. Não há efetivamente uma compilação para linguagem de máquina. A execução de um programa escrito em uma linguagen interpretada é mais lenta, já que o processo de interpretação e execução ao mesmo tempo é mais lento do que a simples execução de um programa traduzido antecipadamente.

Hoje em dia a maior parte dos usuários de computadores não são programadores e sim pessoas que usam programas para resolver seus problemas do dia a dia. Aplicativos típicos que rodam nos computadores são: editores de texto, processadores de texto, planilhas eletrônicas, compiladores, bancos de dados, jogos, etc.

Para gerenciar os recursos do computador existe um programa especial normalmente chamado de Sistema Operacional (S. O.). Por exemplo, considere o problema de gerenciar como os diversos programas que um usuário normalmente utiliza partilharão o processador do computador. Um usuário pode estar ouvindo música, digitando um texto e imprimindo um outro documento ao mesmo tempo. Portanto, os computadores são capazes de executar um número de tarefas muito maior do que o número de processadores disponíveis. Atualmente a maior parte dos computadores possui somente um processador. O Sistema Operacional controla a alocação de recursos tais como: comunicação com os usuários, espaço em discos, uso de memória, tempo que cada programa pode rodar etc. Alguns dos sistemas operacionais conhecidos são os baseados no padrão UNIX, por exemplo o LINUX. Outros sistemas muito usados são os da família Windows.

Compilando Programas Simples em `C`

Para resolver os exercícios deste livro você irá precisar de um compilador para a linguagem `C` e de um editor de textos simples (não processador do como o Word). O editor pode ser tão simples quanto o *Notepad*, na verdade recomendamos fortemente que o editor seja simples para que você possa ter contacto com todas as etapas do processo de desenvolvimento de um programa. Para compilar empregaremos o compilador `gcc` que é gratuito e pode ser obtido na Internet como veremos adiante. Não será necessário nenhum ambiente mais complexo, tal como um Integrated Development Environment (IDE).

A coleção de compiladores da GNU (*GNU Compiler Collection*) usualmente abreviada por `gcc`, é uma coleção de compiladores produzidos pelo projeto GNU. A abreviação `gcc`, originalmente, significava GNU C Compiler. Este aplicativo é distribuído gratuitamente pela Free Software Foundation (FSF) sob a licença GNU GPL e GNU LGPL. Este é o compilador padrão para os sistemas operacionais livres do tipo Unix, como o LINUX, e diversos sistemas operacionais proprietários como o Apple Mac OS X. Atualmente o `gcc` pode compilar `C++`, Objective-C, Java, Fortran e ADA, entre outras linguagens. Vamos considerar, como exemplo, um programa chamado `teste.c`. Para compilar e gerar o executável para este programa digitamos o comando

```
gcc -o teste teste.c -Wall
```

em uma janela de comandos no sistema Windows ou em um terminal nos sistemas Unix. O sufixo `.c` no nome do programa normalmente é usado para indicar que o arquivo é de um programa `C`. Este comando deve ser digitado no diretório onde está o arquivo fonte `teste.c`. O arquivo executável será armazenado no mesmo diretório.

Nos sistemas Unix normalmente o `gcc` faz parte da distribuição padrão e nada precisa ser feito. No Windows uma maneira fácil de obter uma versão do `gcc` é instalar o MinGW (Minimalist GNU for Windows). MinGW é uma coleção de arquivos e bibliotecas distribuídas livremente as quais combinadas com outras ferramentas da GNU permitem que programas para Windows sejam produzidos sem a necessidade de bibliotecas extras e pagas. O MinGW dispõe de um programa instalador que facilita enormemente o processo. Este programa pode ser obtido no sítio oficial do MinGW. Caso após a instalação, o comando indicado não funcione uma das razões para a falha pode ser que o sistema operacional não sabe onde se encontra o compilador `gcc`. Suponha que o programa `gcc` foi instalado no diretório `C:\MinGW\bin`. Uma solução é digitar o caminho completo do compilador. Neste caso o comando se torna

```
C:\MinGW\bin\gcc -o teste teste.c -Wall
```

Para que não seja necessário digitar o caminho completo, é preciso adicionar este caminho à variável `PATH` do Windows. Consulte o manual para obter informações de como fazer este passo no seu sistema Windows.

1.5 Um programa em C

Vamos terminar este capítulo mostrando um exemplo simples de programa escrito em `C`(Listagem 1.1). A única coisa que este programa faz é imprimir `Alo Mundo!` e terminar.

A primeira linha do programa avisa ao compilador que irá usar funções de entrada e saída de dados guardadas na biblioteca `stdio`. Neste caso a função usada é `printf`. A segunda linha é o início real do programa. A linha indica que esta é a função `main` que todo programa `C` deve conter, pois é nesta função que o programa obrigatoriamente começa sua execução. A função vai retornar um

valor inteiro (`int`) ao final de sua execução e não vai precisar receber nenhum argumento para sua execução (`void`). As chaves ({ `e` }) marcam o início e o fim da função. Para imprimir o texto `Alo Mundo!` o programa usa a função `printf`. O início e o fim do texto a ser impresso são marcados pelo caractere `"`. A função termina com o comando `return 0`, que avisa ao sistema operacional, que foi quem iniciou a execução do programa, que o programa terminou sem problemas. Este programa simples ilustra alguns das estruturas básicas que serão usadas nos programas `C` que serão apresentados neste livro.

Listing 1.1: Exemplo de Programa em C.

```
#include <stdio.h>
int main (void)
{
    printf("Alo Mundo!\n");
    return 0;
}
```

Exercícios

1.1: O que é o hardware do computador?

1.2: Quais os principais componentes de um computador?

1.3: Quais as diferenças entre um microprocessador e o microcomputador?

1.4: Dê exemplos de microprocessadores e de microcomputadores.

1.5: Qual o número exato de bytes em 64Kbytes?

1.6: Se você já usa computadores, liste alguns aplicativos que você normalmente usa.

1.7: Defina Sistema Operacional.

1.8: Qual a diferença básica entre memórias ROM e RAM?

1.9: Procure em manuais, internet e outras fontes quais são os tempos de acesso das memórias RAMs atuais.

1.10: Faça três listas, uma de periféricos de entrada, outra de periféricos de saída e finalmente uma de periféricos de entrada e saída.

1.11: Procure nomes de linguagens de programação não listadas no texto e diga quais são as suas características principais.

Capítulo 2

Algoritmos

2.1 Introdução

O objetivo deste capítulo é fazer uma breve introdução ao conceito de algoritmos e apresentar algumas formas mais comuns de representar algoritmos para facilitar o entendimento dos demais capítulos deste livro. Iremos apresentar as construções mais comuns empregadas no desenvolvimento de algoritmos e apresentaremos exemplos básicos de algoritmos usando algumas destas formas de representação e construções.

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 seqüência de passos que permitam que o problema possa ser resolvido de maneira automática e repetitiva. Além disto é preciso definir como os dados que serão processados serão armazenados no computador. Portanto, a solução de um problema por computador é baseada em dois pontos: a seqüência de passos e a forma como os dados serão armazenados no computador. Esta seqüência de passos é chamada de algoritmo.

Usamos algoritmos em diversas atividades que realizamos diariamente. Uma grande parte destas atividades não estão relacionadas com computação. Um exemplo simples e prosaico, de como um problema pode ser resolvido caso forneçamos uma seqüência de passos que mostrem a maneira de obter a solução, é uma receita para preparar um bolo.

Uma vez que foi criado um algoritmo para resolver um determinado problema usando computadores passamos para a próxima fase que é a escrita deste algoritmo em alguma linguagem de programação.

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.

No seu livro Fundamental Algorithms vol. 1 Donald Knuth apresenta uma versão para a origem desta palavra. Ela seria derivada do nome de um famoso matemático persa chamado Abu Ja´far Maomé ibn Mûsâ al-Khowârism (825) que traduzido literalmente quer dizer Pai de Ja´far, Maomé, filho de Moisés, de Khowârizm. Khowârizm é hoje a cidade de Khiva, na ex União Soviética. Este autor escreveu um livro chamado *Kitab al jabr w´al-muqabala* (Regras de Restauração e Redução). O título do livro deu origem também a palavra Álgebra.

O significado da palavra é muito similar ao de uma receita, procedimento, técnica, rotina. **Um algoritmo é um conjunto finito de regras que fornece uma seqüência de operações para resolver um problema específico**. Segundo o dicionário do prof. Aurélio Buarque de Holanda um algoritmo é um:

> Processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, regras formais para a obtenção de resultado ou de solução de problema.

Um algoritmo opera sobre um conjunto de entradas (no caso do bolo, farinha ovos, fermento, etc.) de modo a gerar uma saída que seja útil (ou agradável) para o usuário (o bolo pronto).

Um algoritmo computacional tem cinco características importantes:

Finitude: Um algoritmo deve sempre terminar após um número finito de passos.

Definição: Cada passo de um algoritmo deve ser precisamente definido. As ações devem ser definidas rigorosamente e sem ambiguidades.

Entradas: Um algoritmo deve ter zero ou mais entradas, isto é quantidades que são lhe são fornecidas antes do algoritmo iniciar.

Saídas: Um algoritmo deve ter uma ou mais saídas, isto é quantidades que tem uma relação específica com as entradas.

Efetividade: Um algoritmo deve ser efetivo. Isto significa que todas as operações devem ser suficientemente básicas de modo que possam ser, em princípio, executadas com precisão em um tempo finito por um humano usando papel e lápis.

2.2 Primeiros Passos

É claro que todos nós sabemos construir algoritmos. Se isto não fosse verdade, não conseguiríamos sair de casa pela manhã, ir ao trabalho, decidir qual o melhor caminho para chegar a um lugar, voltar para casa, etc. Para que tudo isto seja feito é necessário uma série de entradas do tipo: a que hora acordar, que hora sair de casa, qual o melhor meio de transporte etc. Um fator importante é que pode haver mais de um algoritmo para resolver um determinado problema. Por exemplo, para ir de casa até o trabalho, posso escolher diversos meios de transporte em função do preço, conforto, rapidez, etc. A escolha será feita em função do critério que melhor se adequar as nossas necessidades. Um exemplo de algoritmo pode ser as instruções que um professor passa aos seus alunos em uma academia de ginástica, mostrado no Algoritmo 2.1. Observar que nesta representação do algoritmo cada linha contém uma instrução.

Algoritmo 2.1: Exemplo de Algoritmo.

início
  enquanto *não fez 10 vezes* **faça**
    Levantar e abaixar braço direito
    Levantar e abaixar braço esquerdo
    Levantar e abaixar perna esquerda
    Levantar e abaixar perna direita
  fim enqto
fim

Computadores são máquinas muito eficientes na resolução de problemas matemáticos ou que envolvam números. Como exemplo de um algoritmo matemático, vamos considerar o problema de resolver uma equação do primeiro grau da forma

$$ax + b = 0$$

A solução desta equação é

$$x = -b/a$$

se o valor de a for diferente de 0. Caso a seja igual a 0, a equação não possui solução, já que não é possível dividir por 0. Este algoritmo escrito (Algoritmo 2.2) em uma pseudo-linguagem de programação ficaria da seguinte maneira:

As instruções do algoritmo são executadas passo a passo e uma intrução somente é executada quando a anterior terminou sua tarefa. Os dois primeiros passos do algoritmo servem para o algoritmo obter os valores dos coeficientes a e b. Os valores podem, por exemplo, serem digitados em um teclado pelo usuário que está usando o algoritmo. O valor digitado vai para uma posição da memória do computador, que para facilitar o manuseio do dado, recebe um nome. Neste exemplo demos os nomes a, b e x as posições de memória usadas pelo programa para armazenar dados. Após os dois primeiros passos o algoritmo executa uma

Algoritmo 2.2: Algoritmo para resolver uma equação do primeiro grau.

Entrada: Coeficientes a e b da equação $ax + b = 0$
Saída: Resultado x da Equação
início
  ler a
  ler b
  se $a \neq 0$ **então**
    imprimir ‘‘A equação nao tem solução’’
  senão
    $x \leftarrow -b/a$
    imprimir ‘‘A raiz da equação vale ’’, x
  fim se
fim

instrução de teste para verificar se o valor de a é diferente de 0. Neste caso podemos ter duas respostas e o computador irá escolher entre dois caminhos independentes e exclusivos. Caso a seja diferente a zero, o algoritmo executa as instruções entre a palavra `então` e `senão`, e portanto, imprime uma mensagem de aviso para o usuário e termina. Esta mensagem normalmente aparece em um monitor de vídeo. No caso de a ser diferente de zero, o algoritmo executa as instruções entre `senão` e `fim se`. Isto significa calcular o resultado da equação e atribuir este resultado à x. O último passo, desta opção é a impressão do resultado da equação.

2.3 Representação

As formas mais comuns de representação de algoritmos são as seguintes:

Linguagem Natural: Os algoritmos são expressos diretamente em linguagem natural, como nos exemplos anteriores.

Fluxograma Convencional: Esta é um representação gráfica que emprega formas geométricas padronizadas para indicar as diversas ações e decisões que devem ser executadas para resolver o problema.

Pseudo-linguagem: Emprega uma linguagem intermediária entre a linguagem natural e uma linguagem de programação para descrever os algoritmos.

Não existe consenso entre os especialistas sobre qual seria a melhor maneira de representar um algoritmo. Atualmente a maneira mais comum de representar-se algoritmos é através de uma pseudo-linguagem ou pseudo-código. Esta forma de representação tem a vantagem de fazer com que o algoritmo seja escrito de uma forma que está mais próxima de uma linguagem de programação de computadores.

2.3.1 Linguagem Natural

A representação em linguagem natural tem a desvantagem de colocar uma grande distância entre a solução encontrada e o resultado final do processo que é um programa em linguagem de programação. O Algoritmo 2.3 mostra um algoritmo, escrito em linguagem natural, para calcular a média de um aluno que faz três provas e precisa de obter média acima de 5.0 para ser aprovado.

Algoritmo 2.3: Algoritmo para calcular a média das notas de um aluno.

Entrada: Notas n_1, n_2 e n_3.
Saída: Resultado *media* do aluno e se ele foi aprovado ou não.
início
Obter as notas n_1, n_2 e n_3
Calcular média. Usar a fórmula $((n_1 + n_2 + n_3)/3.0)$.
Se a média for maior que 5.0 imprimir que o aluno foi aprovado
Caso contrário imprimir que o aluno foi reprovado.
Imprimir a média.
fim

2.3.2 Fluxogramas

Esta forma de representação de algoritmos emprega várias formas geométricas para descrever cada uma das possíveis ações durante a execução do algoritmos. Existem algumas formas geométricas que são empregadas normalmente e que estão mostradas na Figura 2.1. Cada uma destas formas se aplica a uma determinada ação como está indicado. Estas formas são alguns exemplos e existem outras formas que podem ser aplicadas, no entanto nesta apostila estas serão suficientes para os exemplos que serão mostrados.

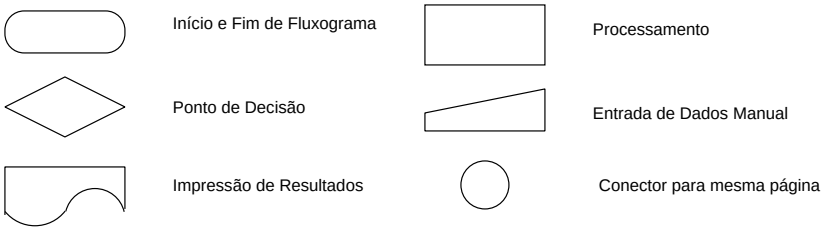

Figura 2.1: Símbolos mais comumente usados em fluxogramas.

Como exemplo de um algoritmo escrito em forma de fluxograma, vamos considerar o algoritmo 2.2 para resolver uma equação do primeiro grau da forma $ax + b = 0$. A Figura 2.2 mostra um fluxograma para resolver este problema.

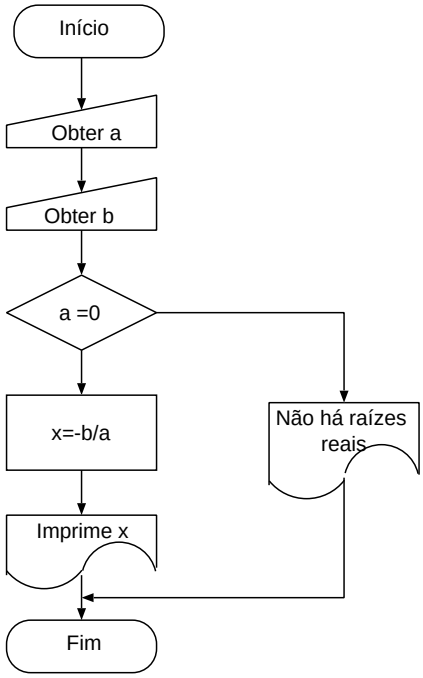

Figura 2.2: Fluxograma para resolver uma equação do primeiro grau.

Os dois primeiros passos do algoritmo lêem os valores dos coeficientes a e b da equação. Em seguida há um teste para descobrir se o valor de a é igual a zero. Se o valor de a for igual a zero o algoritmo manda que seja impressa uma mensagem informando que a equação não tem solução. Caso o valor de a seja diferente os próximos passos calculam o valor da solução e em seguida imprimem este resultado

2.3.3 Pseudo-Linguagem

Este modo de representar algoritmos procura empregar uma linguagem que esteja o mais próximo possível de uma linguagem de programação de computadores de alto nível mas evitando de definir regras de construção gramatical muito rígidas. A idéia é usar as vantagens do emprego da linguagem natural, mas restringindo o escopo da linguagem. Normalmente estas linguagens são versões ultra reduzidas de linguagens de alto nível do tipo Pascal ou C. O algoritmo 2.2 foi escrito em uma pseudo-linguagem. A maioria destas linguagens são muito parecidas e têm a vantagem de serem facilmente entendidas. Vamos apresentar agora um outro exemplo (Algoritmo 2.4) escrito em pseudo-linguagem. Este algoritmo serve para descobrir qual é a maior nota de um grupo de três notas de um aluno. O algoritmo inicialmente lê a primeira nota e guarda esta nota

como a maior nota. Em seguida, lê cada uma das outras notas e compara com a nota guardada como a maior nota. Caso a nota lida seja maior substitui o valor anterior pelo novo valor.

Algoritmo 2.4: Algoritmo para calcular a maior nota de um grupo de notas.

Entrada: Três notas de um aluno, (*notaAluno*).
Saída: Maior das notas do aluno, (*maiorNota*)
início
    `-- Lê primeira nota`
    ler *notaAluno*
    maiorNota ← *notaAluno*
    `-- Lê segunda nota`
    ler *notaAluno*
    se *notaAluno* > *maiorNota* **então**
        maiorNota ← *notaAluno*
    fim se
    `-- Lê terceira nota`
    ler *notaAluno*
    se *notaAluno* > *maiorNota* **então**
        maiorNota ← *notaAluno*
    fim se
    imprimir "A maior nota das notas é ", *maiorNota*
fim

2.4 Modelo de von Neumann

Algoritmos para computadores se baseiam em alguns conceitos básicos e em um modelo de computador e memória que deve ser bem entendido para que se possa criar algoritmos eficientes. Este modelo foi proposto pelo matemático húngaro Neumann János Lajos Margittai. Em húngaro o nome de família aparece antes. Assim em português o seu nome seria János Lajos Margittai Neumann. O seu pai comprou um título e ele passou a se chamar János Lajos Margittai von Neumann. No modelo de computador proposto por von Neumann as instruções e os dados ficam juntos na memória. O processador busca na memória e executa uma instrução de cada vez.

Para ilustrar como este modelo funciona vamos analisar passo a passo a execução de um algoritmo simples. Na Figura 2.3 mostramos nos endereços `0`, `1` e `2`, de uma memória, um grupo de instruções formando parte de um algoritmo. As instruções também podem ser acompanhadas no Algoritmo 2.5. Vamos assumir que o computador começe executando o algoritmo a partir da instrução que está no endereço 0. A UC irá continuar buscando e executando uma instrução de cada vez a partir dos endereços seguintes, a não ser que haja

uma ordem para desviar o fluxo das instruções. É importante observar que o computador executa uma instrução de cada vez, e como veremos adiante, também um dado é buscado de cada vez. Portanto, as transferências entre a memória e o processador são feitas passo a passo. O ciclo normal da execução de um programa é então:

1. Busca instrução;
2. Decodifica instrução;
3. Executa instrução;
4. Volta para o passo 1 buscando a instrução seguinte na memória.

Portanto, após a instrução do endereço 0 ser lida da memória e trazida para a UCP, ela é decodificada pela UC, que descobre que a instrução manda carregar o valor inteiro 2 na posição de memória 10, que é chamada de a para facilitar. A instrução seguinte ordena a carga do valor inteiro 8 na posição de memória chamada de b, que é a posição 11. A última instrução ordena a soma dos valores armazenados em a e b e que o resultado seja armazenado na posição chamada c.

Algoritmo 2.5: Modelo de memória e funcionamento de um algoritmo

```
-- Armazena 2 na memória no lugar chamado a
a ← 2
-- Armazena 8 na memória no lugar chamado b
b ← 8
-- Soma a com b e armazena no lugar chamado c
c ← a + b
```

Observe que no modelo de von Neumann instruções e dados ficam na memória. Considere a última instrução do programa que pede para que os dados da posição 10 (a) e 11 (b) sejam somados e o resultado armazenado na posição 12 (c). Para executar esta instrução, o computador faz as seguintes operações na memória:

1. ler a instrução no endereço 2;
2. ler o dado a na posição 10;
3. ler o dado b na posição 11;
4. escrever o resultado da soma no endereço 12.

Portanto, temos 3 leituras e uma escrita. Destas operações, a primeira é uma leitura de instrução e as restantes operações com dados.

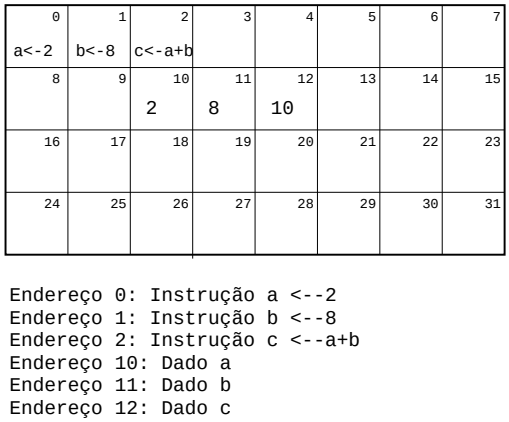

Figura 2.3: Modelo de memória

2.5 Estruturas Básicas de Algoritmos

Para ilustrar como criar algoritmos para computadores neste modelo, vamos discutir alguns tipos básicos de estruturas usados nesta tarefa. Para isto iremos usar a representação que for apropriada no momento. Não iremos neste livro discutir em detalhes estes tópicos nem apresentar de uma maneira formal qualquer uma destas formas. O interesse é apenas apresentar e discutir alguns algoritmos para ilustrar o pensamento usado pelos programadores quando criam um algoritmo para resolver um problema específico. Estas estruturas são importantes e serão reapresentadas quando formos apresentar a linguagem `C`. Para o programador iniciante esta discussão serve como introdução.

2.5.1 Comandos de leitura

Estes comandos servem para obter dados do mundo exterior, normalmente digitados por um usuário em um teclado. Outros exemplos de lugares de onde podem ser obtidos dados são os arquivos em discos rígidos, disquetes e fitas magnéticas. Estes dados são lidos e guardados na memória para posterior uso. Normalmente os lugares para onde vão estes dados recebem um nome para facilitar o seu manuseio durante a execução do algoritmo.

Por exemplo, o comando

ler a

significa que o algoritmo irá obter um dado do teclado e irá armazená-lo em uma posição de memória que passará a ser conhecida pelo nome a. Estas posições são conhecidas por variáveis em computação. Costuma-se dizer então que a é uma variável do algoritmo. Apesar de a no algoritmo 2.2 representar uma das

constantes da equação do primeiro grau no algoritmo ela é uma das variáveis. Observe que cada vez que o algoritmo é executado a pode assumir um valor diferente, portanto varia de acordo com a execução do algoritmo.

O comando pode ser seguido por uma lista de nomes separados por vírgulas. Por exemplo o comando

ler a, b

lê dois valores do teclado e os atribui as variáveis a e b. Observe que a ordem em que os valores foram digitados determina como os valores serão atribuídos. O primeiro valor lido é atribuído a primeira variável, no caso a. O segundo valor lido é atribuído a segunda variável (b). Os valores normalmente são digitados separados por um ou mais espaços em branco ou em linhas diferentes.

2.5.2 Comandos de escrita

Após a obtenção dos resultados do algoritmo, estes devem ser apresentados ao usuário, e para isto usamos os comandos de escrita. Por exemplo o comando

imprimir x

imprime o valor atual que está na memória representada pelo nome x.

Da mesma forma que nos comandos de leitura é possível colocar uma lista de nomes de variáveis após o comando. Por exemplo, o comando

imprimir $x1$, $x2$

imprime os valores das variáveis $x1$ e $x2$. O meio de apresentação dos resultados, normalmente, é um monitor de vídeo.

O comando **imprimir** pode ser usado para mandar mensagens de texto para o usuário do algoritmo das formas mais variadas. Alguns exemplos de comandos de impressão são os seguintes:

---

imprimir "Entre com o valor do coeficiente"
imprimir "O valor de x é ", x
imprimir "O valor de x1 é ", $x1$, "e o de x2 é ", $x2$

Notar que os textos entre aspas indicam um texto que deve ser impresso no periférico de saída sem nenhuma modificação. Vamos considerar que as variáveis destes exemplos valem $x = 10$, $x1 = 5$ e $x2 = 8$. Os três comandos mostrariam no periférico de saída os seguintes resultados:

```
Entre com o valor do coeficiente.
O valor de x é 10
O valor de x1 é 5 e o de x2 é 8
```

2.5.3 Expressões

Expressões são usadas para definir os cálculos requeridos pelo algoritmo, por exemplo $-b/a$. Iremos discutir dois tipos básicos de expressões: expressões aritméticas e expressões lógicas.

Expressões manipulam dados dentro dos algoritmos. Uma pergunta importante neste momento é: que tipo de dados poderemos manipular? As linguagens de programação normalmente estabelecem regras precisas para definir que tipos de dados elas irão manipular. Nesta discussão vamos estabelecer, ainda que informalmente, algumas regras que limitam os conjuntos de dados existentes na Matemática e estabelecem que dados poderão ser manipulados pelos algoritmos. Isto ocorre porque os computadores possuem limitações que os impedem de manipular todos os tipos de dados que um ser humano pode tratar. Mais adiante, quando formos estudar a linguagem `C` iremos apresentar mais formalmente as regras desta linguagem para estas representações. Existem três tipos básicos de dados que iremos discutir:

Dados numéricos: como o nome indica são os números que serão operados.

Dados alfa-numéricos: são os dados representados por caracteres. Como caracteres podem ser letras, algarismos e sinais diversos estes dados recebem este nome.

Dados Logicos: estes dados podem assumir dois valores `verdadeiro` e `falso`. Estes dados resultam de expressões do tipo $x > 0$.

Os dados numéricos que os algoritmos que iremos construir podem manipular são de dois tipos: inteiros e reais. São exemplos de números inteiros:

```
+3
 3
-324
-50
```

São exemplos de números reais:

```
+0.5
 0.5
-8.175
 2.0
```

Dados alfa-numéricos servem para tratamento de textos e normalmente são compostos por uma seqüência de caracteres contendo letras, algarismos e caracteres de pontuação. Nos algoritmos são normalmente representados por uma seqüência de caracteres entre aspas, por exemplo:

```
‘‘Linguagem de programação’’
‘‘Qual é o seu nome?’’
‘‘12345’’
```

Dados lógicos são intensamente aplicados durante o processo de tomada de decisões que o computador frequentemente é obrigado a fazer. Em muitos textos este tipo de dados também é chamado de tipo de dados booleanos, devido a George Boole, matemático que deu ao nome à álgebra (álgebra booleana) que manipula este tipo de dados. Os dados deste tipo somente podem assumir dois valores: `verdadeiro` e `falso`. Computadores tomam decisões, durante o processamento de um algoritmo, baseados nestes dois valores. Por exemplo, considere a decisão abaixo:

---

se $a \neq 0$ **então**
  imprimir ‘‘A equação nao tem solução’’
senão
  $x \leftarrow -b/a$
  imprimir ‘‘A raiz da equação vale ’’, x
fim se

Nesta instrução aparece a expressão $a \neq 0$, que procura descobrir se o valor de raiz é maior que 0. Esta expressão somente pode ter como resultado os valores: verdadeiro ou falso. Nos nossos algoritmos estes valores serão representados por `verdadeiro` e `falso`.

Expressões Aritméticas

Para os nossos exemplos iniciais iremos adotar os operadores aritméticos binários mostrados na Tabela 2.5.3. A coluna prioridade indica a prioridade relativa dos operandos. O menor número indica a maior prioridade. Quando temos dois operandos de mesma prioridade o computador irá executar primeiro a operação mais à esquerda. Em computação as expressões devem ser escritas em linhas e para alterar a prioridade deve-se usar parênteses. Maiores detalhes sobre expressões serão apresentados no Capítulo XXX. No momento estamos apresentando alguns conceitos básicos para facilitar a discussão de alguns algoritmos.

Para ilustrar o uso de operadores aritméticos vamos considerar alguns exemplos de expressões. Os exemplos a seguir mostram como converter uma expressão matemática para a forma que usaremos em pseudo-linguagem.

Operador	Descrição	Prioridade
`/`	Divisão	0
`*`	Multiplicação	0
`%`	Módulo (resto da divisão de operandos inteiros)	0
`+`	Soma	1
`-`	Subtração	1

Tabela 2.1: Operadores Aritméticos.

Expressão Matemática	Expressão em Pseudo-linguagem
$\frac{a}{b+c}$	`a/(b+c)`
$\frac{a+b}{c+d}$	`(a+b)/(c+d)`
$b^2 - 4ac$	`b*b-4*a*c`
$\frac{1}{1+\frac{1}{a+b}}$	`1/(1 + 1/(a+b))`

2.5.4 Comandos de atribuição

Servem para atribuir valores à posições de memória. Normalmente estes valores podem ser constantes ou resultados de expressões dos diversos tipos. Exemplos de comandos de atribuição são mostrados a seguir.

$x \leftarrow -b/a$
$media \leftarrow (n1 + n2)/2$
$inicio \leftarrow 0$
$nome \leftarrow$ ``Ze Sa''

2.5.5 Comandos de controle

São usados para controlar o fluxo de execução das instruções. Nas linguagens de programação existem diversos tipos de diferentes de comandos de controle. Para estes exemplos iniciais vamos mostrar somente o comando mais básico que serve para o computador escolher entre dois possíveis caminhos qual o algoritmo deve seguir. Este comando representado em fluxograma pode ser visto na Figura 2.4 e em pseudo linguagem tem a forma mostrada no algoritmo 2.6.

Um exemplo de uso desta contrução escolhido a partir da vida real pode ser o seguinte. Tenho que decidir o que fazer em um domingo de folga. Se estiver chovendo irei ao cinema, caso contrário irei à praia. Observe que para ir para a praia basta apenas que não esteja chovendo, nenhum outro teste foi feito. Esta decisão em forma de pseudo-linguagem fica da maneira mostrada no Algoritmo 2.7.

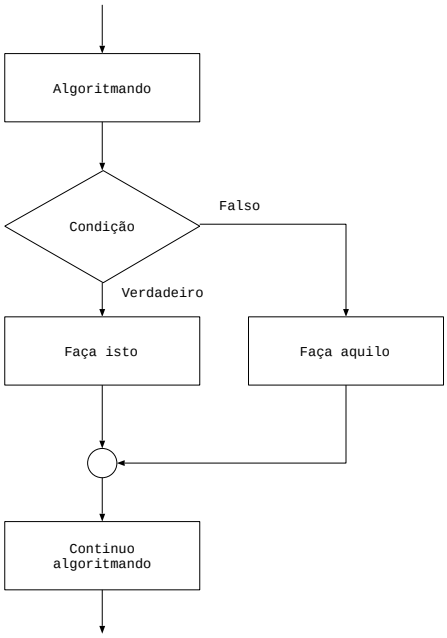

Figura 2.4: Fluxograma do comando `se ... então ... senão`.

Algoritmo 2.6: Comando se em pseudo-linguagem

se *Condição sendo testada* **então**
    Faça isto
senão
    Faça aquilo
fim se

Nem sempre nos algoritmos precisamos de duas alternativas. As vezes precisamos somente de decidir se devemos fazer algo ou não Por exemplo, vamos assumir que decidimos ir ao cinema no domingo chova ou faça sol. No entanto, preciso decidir se levo um guarda-chuva. O algoritmo para este caso está mostrado no Algoritmo 2.8. Observe que no caso de não estar chovendo nada precisa ser feito. A Figura 2.5 mostra esta decisão em forma de fluxograma.

2.5.6 Comandos de repetição

As linguagens de programação normalmente possuem diversos comandos que permitem que um trecho de algoritmo seja repetido um número de vezes. Para estes exemplos de algoritmos iremos apresentar um comando de repetição que

Algoritmo 2.7: Algoritmo para decidir o que fazer no domingo.

se *está chovendo* **então**
    vou ao cinema
senão
    vou à praia
fim se

Algoritmo 2.8: Algoritmo para decidir se deve levar um guarda-chuva.

Vestir para ir ao cinena
se *está chovendo* **então**
    pego guarda-chuva
fim se
Vou ao cinema

é suficientemente geral para substituir todos os outros. Este comando, que chamaremos de comando `enquanto` tem a forma mostrada na Figura 2.6.

O comando funciona da seguinte maneira:

Passo 1: Testar se a condição é verdadeira. Caso seja verdade executar o bloco de comandos situados entre o início do comando e o final do comando. O final do comando `enquanto` normalmente é marcado de alguma forma. Em nossa pseudo-linguagem marcaremos o fim pelas palavras `fim enquanto`.

Passo 2: Executar o bloco de comandos até o fim do `enquanto`. Quando chegar ao fim retornar automaticamente para o início do comando e refazer o passo 1.

Como exemplo consideremos o caso em que precisamos ler um conjunto de 10 números e imprimir se cada um dos números lidos é par ou não. Para descobrir se o número é par vamos dividi-lo por 2 e testar o resto. Lembrar que caso o resto da divisão seja igual a zero o número é par. Neste algoritmo sabemos a quantidade de números a serem lidos e portanto o número de repetições é pré-determinado. Mais adiante faremos um exemplo onde o número de repetições não é pré-determinado. O Algoritmo 2.9 mostra como seria implementado este problema.

Vamos mostrar um exemplo onde o número de repetições não é conhecido. Considere no exemplo anterior que o total de números a ser lido não é conhecido. Mas então, como o algoritmo irá terminar de ler números? Usaremos o que costuma-se chamar de sentinela em computação. O algoritmo irá se manter lendo números enquanto os números forem positivos. No momento que for lido um número negativo o algoritmo pára. A sentinela que indica o final da lista de números é um número negativo. O Algoritmo 2.10 mostra como fica em pseudo-linguagem o algoritmo modificado. Observe que neste algoritmo

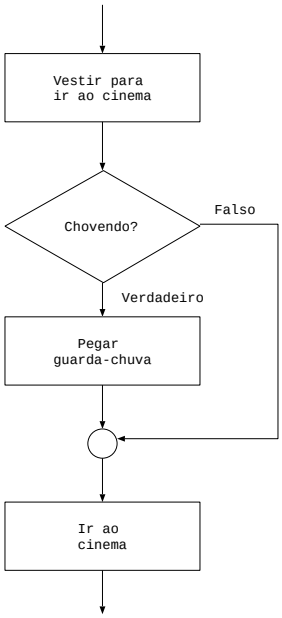

Figura 2.5: Fluxograma para decidir se deve levar um guarda-chuva.

o primeiro número tem de ser lido antes do comando `enquanto`. Isto porque assumimos que o primeiro número que for digitado pode ser negativo e portanto a lista de números não tem nenhum número.

2.6 Exemplos de Algoritmos

Nesta seção iremos apresentar uma série de algoritmos escritos na pseudo-linguagem que acabamos de apresentar.

Exemplo 2.1: Este algoritmo serve para descobrir qual é a maior nota de uma turma de alunos. Neste algoritmo iremos considerar que as notas podem variar entre 0.0 e 10.0 e que a turma tem 25 alunos. O algoritmo 2.11 inicialmente inicia a $maiorNota$ com zero, depois compara cada nota com esta $maiorNota$ caso ela seja maior substitui o valor anterior pelo novo valor. Observar que criamos uma variável para armazenar a quantidade de alunos. Você poderia perguntar porque não usar o número 25 toda vez que for necessário. A razão é simples. Suponha que você gostaria de usar este algoritmo para calcular a maior nota de uma turma de 40 alunos. Neste algoritmo bastaria substituir o número 25 por 40. No entanto, existe uma solução mais geral ainda que permite que o algoritmo possa ser usado para qualquer tamanho de turma. Este problema está apresentado na lista de exercícios deste capítulo.

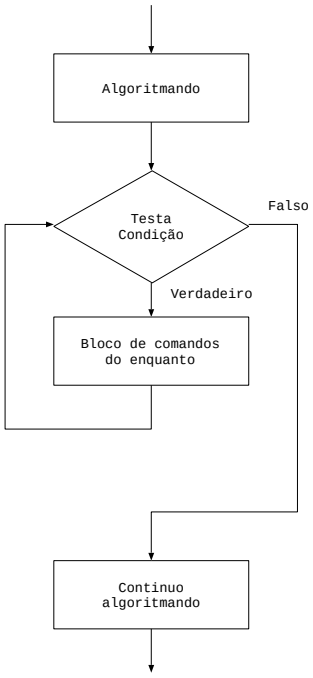

Figura 2.6: Fluxograma do comando `enquanto`.

Exemplo 2.2: Vamos mostrar outro exemplo de algoritmo muito usado. Precisamos ler as notas de uma turma de alunos e calcular a média destas notas. Vamos assumir que a turma tem 25 alunos e as notas estão entre 0 e 10. O algoritmo está mostrado em Algoritmo 2.12.

Exemplo 2.3: Neste exemplo considere o seguinte problema. Um escritório de previsão do tempo armazena diariamente a temperatura média de uma determinada região. A tarefa é descobrir qual é a maior temperatura do ano passado. Assuma que foram armazenadas 365 temperaturas, uma para cada dia do ano. Neste caso não podemos aplicar o algoritmo 2.11 usado para descobrir a maior nota da turma. Antes de continuar procure encontrar a razão. Como dica, considere que estamos no polo sul e portanto todas as temperaturas lidas são negativas.

Uma solução possível para este exemplo está mostrada no algoritmo 2.13. Este algoritmo faz o seguinte. Pega a primeira temperatura e a anota como a menor já encontrada. A partir daí o algoritmo fica repetidamente lendo temperaturas dos registros do escritório comparando com a temperatura que no momento consta como a menor de todas. Se a temperatura tirada dos arquivos for menor que a menor atual, o algoritmo joga fora a temperatura anotada e guarda a que foi lida como a nova menor temperatura. Quando não houver

Algoritmo 2.9: Algoritmo para ler 10 números e imprimir se são pares ou não.

```
Entrada: 10 números, (numero).
Saída: Se o número é par ou não
início
    totalNumeros ← 10
    enquanto totalNumeros > 0 faça
        ler numero
        se numero%2 = 0 então
            imprimir numero, "par"
        senão
            imprimir numero, "impar"
        fim se
        totalNumeros ← totalNumeros − 1
    fim enqto
fim
```

mais temperaturas para ler a que estiver anotada como a menor é a menor verdadeiramente.

Exercícios

2.1: Uma empresa paga R$10.00 por hora normal trabalhada e R$ 15.00 por hora extra. Escreva um algoritmo que leia o total de horas normais e o total de horas extras trabalhadas por um empregado em um ano e calcule o salário anual deste trabalhador.

2.2: Assuma que o trabalhador do exercício anterior deve pagar 10% de imposto se o seu salário anual for menor ou igual a R$ 12000.00. Caso o salário seja maior que este valor o imposto devido é igual a 10% sobre R$ 12000.00 mais 25% sobre o que passar de R$ 12000.00. Escreva um programa que calcule o imposto devido pelo trabalhador.

2.3: Escreva um algoritmo que descubra a maior nota de uma turma de alunos. O tamanho da turma deve ser o primeiro dado pedido ao usuário.

2.4: Modifique o algoritmo anterior de modo que ele imprima também quantas vezes a maior nota aparece.

2.5: Nos exercícios anteriores assumimos que os usuários sempre digitam uma nota entre 0 e 10. Vamos assumir agora que o usuário sempre digita um número, mas este número pode estar fora do intervalo 0 a 10. Ou seja, poderemos ter uma nota menor que zero ou maior que 10. Modifique o algoritmo anterior

Algoritmo 2.10: Algoritmo para ler números e imprimir se são pares ou não.

Entrada: números, (*numero*). O algoritmo pára quando um número negativo é lido
Saída: Se o número é par ou não
início
  ler *numero*
  enquanto *numero* > 0 **faça**
    se *numero* % $2 = 0$ **então**
      imprimir *numero*, "par"
    fim se
    imprimir *numero*, "impar"
    ler *numero*
  fim enqto
fim

para que ele verifique a nota digitada e, caso o aluno tenha digitado uma nota inválida, uma mensagem avisando o usuário seja impressa e uma nova nota seja pedida. O algoritmo deve insistir até que o usuário digite um valor válido.

2.6: Escreva um algoritmo que leia três números e os imprima em ordem crescente.

2.7: Escreva um algoritmo que leia um número inteiro entre 100 e 999 e imprima na saída cada um dos algarismos que compõem o número. Observe que o número é lido com um valor inteiro, e, portanto, ele tem de ser decomposto em três números: os algarismos das centenas, dezenas e unidades.

2.8: Escreva um algoritmo que leia uma hora em horas, minutos e segundos e some um segundo a hora lida.

2.9: Escreva um algoritmo que leia duas datas em dia, mes e ano e imprima a data mais recente.

Algoritmo 2.11: Algoritmo para calcular a maior nota de uma turma de 25 alunos.

Entrada: Nota de cada um dos dos 25 alunos da turma, (*notaAluno*).
Saída: Maior das notas dos alunos, (*maiorNota*)
início
    $totalAlunos \leftarrow 25$
    $maiorNota \leftarrow 0.0$
    enquanto $totalAlunos > 0$ **faça**
        ler *notaAluno*
        se $notaAluno > maiorNota$ **então**
            $maiorNota \leftarrow notaAluno$
        fim se
        $totalAlunos \leftarrow totalAlunos - 1$
    fim enqto
    imprimir "A maior nota das notas é ", *maiorNota*
fim

Algoritmo 2.12: Algoritmo para calcular a nota média de uma turma de 25 alunos.

Entrada: Nota de cada um dos dos 25 alunos da turma, (*notaAluno*).
Saída: Média das notas dos alunos, (*mediaNotas*)
início
    $totalAlunos \leftarrow 25$
    $somaNotas \leftarrow 0.0$
    enquanto $totalAlunos > 0$ **faça**
        ler *notaAluno*
        $somaNotas \leftarrow somaNotas + notaAluno$
        $totalAlunos \leftarrow totalAlunos - 1$
    fim enqto
    $mediaNotas \leftarrow somaNotas/25$
    imprimir "A média das notas é ", *mediaNotas*
fim

Algoritmo 2.13: Algoritmo para calcular a maior temperatura do ano.

Entrada: Temperaturas registrada em ano, (*temperatura*).
Saída: Maior das temperaturas, (*maiorTemperatura*)
início
  $totalTemperaturas \leftarrow 365$
  ler *temperatura*
  Já li uma temperatura
  $totalTemperaturas \leftarrow totalTemperaturas - 1$
  `-- A primeira temperatura é a maior temperatura`
  $maiorTemperatura \leftarrow temperatura$
  enquanto $totalTemperaturas > 0$ **faça**
    ler *temperatura*
    se $temperatura > maiorTemperatura$ **então**
      $maiorTemperatura \leftarrow temperatura$
    fim se
    $totalTemperaturas \leftarrow totalTemperaturas - 1$
  fim enqto
  imprimir "A maior nota das notas é ", *maiorTemperatura*
fim

Capítulo 3

Tipos de Dados, Constantes e Variáveis

3.1 Introdução

Variáveis e constantes são os elementos básicos que um programa manipula. Uma variável corresponde a um espaço reservado na memória do computador para armazenar um determinado tipo de dado. Variáveis devem receber nomes para poderem ser mais facilmente referenciadas e modificadas sempre que necessário. Muitas linguagens de programação exigem que os programas declarem todas as variáveis antes que elas possam ser usadas. Estas declarações especificam de que tipo são as variáveis usadas pelos programas e as vezes um valor inicial. Tipos podem ser por exemplo: inteiros, reais, caracteres, etc. As expressões combinam variáveis e constantes para calcular novos valores.

3.2 Tipos de Dados

3.2.1 Tipos Básicos

Os dados em `C` podem assumir cinco tipos básicos que são os seguintes:

char: O valor armazenado é um caracter. Caracateres geralmente são armazenados em códigos (usualmente o código ASCII). A Tabela A.1 mostra este código. Caracteres são armazenados em um byte.

int: O valor armazenado é um número inteiro e o tamanho do subconjunto que pode ser representado pelo computador normalmente depende da máquina em que o programa está rodando. Atualmente em `C` os números inteiros são armazenados em 32 bits.

float: Número em ponto flutuante de precisão simples, normalmente 32 bits. São conhecidos como números reais, no entanto, os computadores somente podem armazenar e trabalhar com uma pequena parte do conjunto dos números reais.

double: Número em ponto flutuante de precisão dupla, com isto a precisão e as vezes a excursão dos números aumenta. Este tipo é armazenado em 64 bits.

void: Este tipo serve para indicar que um resultado não tem um tipo definido. Uma das aplicações deste tipo em `C` é criar um tipo vazio que pode posteriormente ser modificado para um dos tipos anteriores.

3.2.2 Modificadores de tipos

Modificadores podem ser aplicados a estes tipos. Estes modificadores são palavras que alteram o tamanho do conjunto de valores que o tipo pode representar. Por exemplo, um modificador permite que possam ser usados mais bits para armazenar números inteiros. Um outro modificador obriga que só números inteiros sem sinal possam ser armazenados pela variável. Deste modo não é necessário guardar o bit de sinal do número e somente números positivos são armazenados. O resultado prático é que o conjunto praticamente dobra de tamanho.

Os modificadores de tipos são os seguintes:

unsigned: Este modificador pode ser aplicado aos tipos **int** e **char** e faz com o bit de sinal não seja usado, ou seja o tipo passa a ter um bit a mais.

signed: Este modificado também pode ser aplicado aos tipos **int** e **char**. O uso de **signed** com **int** é redundante.

long: Modificador que pode ser aplicado aos tipos **int** e **double** aumentando o número de bytes reservado para armazenamento de dados.

É possível combinar estes modificadores de diversas maneiras como está mostrado na Tabela 3.1 que lista os tipos básicos definidos no padrão ANSI e a sua excursão.

3.3 Constantes Numéricas

Constantes são valores que o programa não pode modificar durante a execução de um programa. Elas são usadas em expressões para representar vários tipos de valores. Em `C` existem regras rígidas para determinar como devem ser escritos estes valores. A seguir iremos mostrar estas regras.

Para escrever constantes numéricas vamos usar as seguintes definições:

Tipo	Bytes	Faixa Mínima
char	1	-127 a 127
unsigned char	1	0 a 255
signed char	1	-127 a 127
int	4	-2.147.483.648 a 2.147.483.647
unsigned int	4	0 a 4.294.967.295
signed int	4	-2.147.483.648 a 2.147.483.647
short int, short	2	-32.768 a 32.767
unsigned short int	2	0 a 65.535
signed short int	2	-32.768 a 32.767
long int, long	4	-2.147.483.648 a 2.147.483.647
signed long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4.294.967.295
long long int **long long**	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
signed long long int **signed long long**	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
unsigned long long int **unsigned long long**	8	0 a 18.446.744.073.709.551.615
float	4	oito dígitos de precisão
double	8	16 dígitos de precisão
long double	12	16 dígitos de precisão

Tabela 3.1: Tipos de dados definidos pelo Padrão ANSI C.

dígito: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`

dígito_sem_zero: `1, 2, 3, 4, 5, 6, 7, 8, 9`

dígito_octal: `0, 1, 2, 3, 4, 5, 6, 7`

dígito_hexa: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F`

sinal: `+, -`

ponto_decimal: `.`

sufixo: `sufixo_sem_sinal, sufixo_longo`

sufixo_sem_sinal: `u, U`

sufixo_longo: `l, L`

sufixo_flutuante: `f, F, l, L`

O uso destas caracteres será mostrado com exemplos nas seções seguintes.

3.3.1 Constantes Inteiras na base 10

São valores numéricos sem ponto decimal, precedidos ou não por um sinal. Não é possível separar, por espaços em branco, o sinal do valor numérico. Podemos descrever este formato com uma notação simplificada da seguinte maneira:

`[sinal]dígito_sem_zero{dígito}[sufixo_sem_sinal|sufixo_longo]`

Esta notação deve ser entendida da seguinte maneira. Colchetes indicam opção, portanto, o fato de `sinal` (`+ ou -`) estar entre colchetes significa que um número inteiro pode ou não ter sinal, isto é o sinal é opcional. Em seguida temos um `dígito_sem_zero` que é obrigatório. Isto é dados inteiros devem começar por, pelo menos, um algarismo entre 1 e 9. A seguir temos a palavra `dígito` entre chaves. As chaves indicam que o fator entre elas pode ser repetido zero ou mais vezes. Portanto, um número inteiro, após o algarismo inicial obrigatório, pode ser seguido por uma seqüência de zero ou mais algarismos. Como sufixo podemos ter opcionalmente as letras `u` (`U`) ou `l` (`L`) ou uma mistura, em qualquer ordem, das duas. Para constantes inteiras o sufixo `U` (`u`) representa o modificador **unsigned**. O sufixo `L` (`l`) representa o modificador **long**. Um par de `L´s` (`l´s`) indica que a constante é do tipo **long long**. A tabela 3.2 mostra exemplos de números inteiros:

Tipo	**Constantes**			
int	1997	-3	+5	7
unsigned int	1997U	45u	12345U	0U
long int	1234L	1997L	-3l	+0L
unsigned long int	1997UL	45Lu	23Ul	0LU
long long	134LL	1997Ll	-3ll	+0LL
unsigned long long	1997ULL	45LLu	23Ull	0LLU

Tabela 3.2: Constantes Inteiras na Base 10

Alguns exemplos de erros na escrita de constantes inteiras são:

- `1.0` (Não é possível usar ponto decimal.)
- `- 345` (Não é possível colocar um espaço entre o sinal e o valor numérico.)
- 2^3 (Não é possível usar notação de expoentes.)

Nos compiladores modernos o número de bytes usados para armazenar os valores inteiros é o mesmo tanto para tipos inteiros (**int**) quanto para tipos inteiros longos (**long int**), como está mostrado na Tabela 3.1. Por esta razão, a diferença entre estes dois tipos de constantes perdeu a razão de ser. Alguns exemplos de constantes inteira longas estão mostrados na Tabela 3.2.

3.3.2 Constantes Inteiras Octais

São constantes representadas na base 8. Normalmente são representadas sem sinal e devem começar com um 0. Usando a notação apresentada na seção anterior podemos definir a forma que deve ter uma constante octal como:

`0{dígito_octal}[sufixo_sem_sinal|sufixo_longo]`

Na Tabela 3.3 mostramos exemplos de constantes octais e o seu valor na base 10. Números escritos na base 8 somente podem ser escritos com algarismos entre 0 e 7 inclusive.

Base 8	**Base 10**
025	21
077	63
011	9
010ul	8
0175	125

Tabela 3.3: Constantes octais

3.3.3 Constantes Inteiras Hexadecimais

São constantes representadas na base 16. São iniciadas com um 0x ou 0X. Usando a notação podemos definir uma contante hexadecimal como:

`[0x|0X]dígito_hexa{dígito_hexa}[sufixo_sem_sinal|sufixo_longo]`

Na Tabela 3.4 mostramos exemplos de constantes hexadecimais e o seu valor na base 10. Para escrever constantes na base 16 usamos todos os algarismos e ainda as letras `A` (ou `a`), `B` (ou `b`), `C` (ou `c`), `D` (ou `d`), `E` (ou `e`), `F` (ou `f`), que representam respectivamente os seguintes valores 10, 11, 12, 13, 14 e 15.

Base 16	**Base 10**
0xF	15
0X25	37
0XAB	171
0XBEEF	48879

Tabela 3.4: Constantes hexadecimais

3.3.4 Conversão entre Bases

A conversão de números inteiros entre a base 8 e a base 10 tem uma fórmula simples, que pode ser estendida para converter números entre qualquer base e a base 10. Vamos considerar que um número $(N)_8$ escrito na base 8 tenha a seguinte forma

$$(N)_8 = d_{n-1}d_{n-2}\ldots d_1 d_0$$

onde $7 \leq d_i \leq 0$

A fórmula para converter um número da base 8 para a base 10 é a seguinte

$$N_{10} = d_{n-1} \times 8^{n-1} + d_{n-2} \times 8^{n-2} + \cdots + d_1 \times 8^1 + d_0 \times 8^0 \qquad (3.1)$$

Esta equação está escrita na base 10. Por exemplo, aplicando a equação 3.1 para converter o número `0175` da base 8 para a base 10 ficamos com

$$(0175)_8 = 1 \times 8^2 + 7 \times 8^1 + 5 \times 8^0 = (125)_{10}$$

A fórmula para conversão da base 8 para a base 10 pode se estendida para uma base qualquer com a substituição do algarismo 8. Considere uma base qualquer representada por b. Nesta base os dígitos d_i ficam no intervalo $b-1 \leq d_i \leq 0$. A equação 3.2 mostra a fórmula para converter um número em uma base b qualquer para a base 10.

$$N_{10} = d_{n-1} \times b^{n-1} + d_{n-2} \times b^{n-2} + \cdots + d_1 \times b^1 + d_0 \times b^0 \qquad (3.2)$$

Vamos considerar a contante inteira $(3AB)_{16}$. Aplicando a fórmula 3.2 temos

$$(3AF)_{16} = 3 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = (943)_{10}$$

O algoritmo para converter um número inteiro da base 10 para uma determinada base b é feito por um conjunto de divisões sucessivas do número pela base até que o resultado da divisão seja 0. O Algoritmo 3.1 mostra como converter um número $(N)_{10}$ para uma base b. É importante notar que os algarismos na base b vão sendo impressos na ordem inversa, do menos significativo para o mais significativo. Por exemplo, caso forneçamos para o algoritmo o número $(943)_{10}$ e a base 16, o algoritmo iria imprimir os resultados 15, 10 e 3 nesta ordem. Isto corresponderia aos seguintes dígitos da base 16: `F`, `A` e `3` e, portanto, a resposta seria $(3AF)_{16}$.

3.3.5 Constantes em Ponto Flutuante

Constantes em ponto flutuante são usadas para representar números reais. O nome ponto flutuante é devido ao modo como os valores são armazenados pelo computador. Constantes de ponto flutuante podem ser do tipo **float, double, long** ou **long double**. Constantes sem nenhum sufixo são consideradas do tipo

Algoritmo 3.1: Algoritmo para converter inteiros na base 10 para uma base b.

Entrada: número, (*numero*) e base b (*baseb*).
Saída: Dígitos do número na base b
início
  ler *numero*
  ler *base*
  enquanto *numero* > 0 **faça**
    $resto \leftarrow numero \ \% \ base$
    $numero \leftarrow numero \ / \ base$
    imprimir *resto*
  fim enqto
fim

double. Caso seja usado o sufixo `F` ou o `f` a constante será considerada como do tipo **float**. O sufixo `L` ou o `l` torna a constante **long double**.

Uma constante em ponto flutuante pode ser definida de duas maneiras. Você pode escrever um número com ponto decimal (`1.5`) ou na chamada forma científica, em que um expoente é usado (`0.15E1`). Na segunda forma o número é igual a 0.15×10^1. É possível omitir ou os dígitos antes do ponto (a parte inteira) ou após (a parte fracionária), mas nunca os dois grupos. É possível escrever um número em ponto flutuante sem ponto, desde que um expoente seja usado. Portanto, os números `.8`, `1234.`, `1E1` são números de ponto flutuante.

Para mostrar mais formalmente como se deve escrever as constantes de ponto flutuante vamos usar a mesma notação usada até aqui, mas usando uma hierarquia de definições para facilitar o entendimento. Primeiro damos uma definição mais geral que vai sendo mais detalhada a medida que avançamos. Lembrar que termos entre chaves podem ser repetidos 0 ou mais vezes e termos entre colchetes são opcionais.

Portanto, usando a forma hierárquica, uma constante de ponto flutuante (`CPF`) pode ser definida das seguintes maneiras:

```
CPF = [sinal]fração[expoente][sufixo_flutuante]
CPF = [sinal]seq_dígitos expoente[sufixo_flutuante]
```

A seguir definimos cada um dos componentes. Uma `fração` é definida como:

```
fração = [seq_digitos] ponto_decimal seq_dígitos ou
fração = seq_dígitos ponto_decimal
```

O expoente e a sequência de dígitos são definidos como:

```
expoente = [e | E][sinal]seq_dígitos
seq_dígitos = dígito{dígito}
```

A Tabela 3.5 mostra exemplos de constantes em ponto flutuante.

Descrição	Número
sinal fração expoente	+23.45e-10
fração	123.45
fração expoente	123.45E+10
fração sufixo	123.45F
seq_dígitos ponto_decimal	123.

Tabela 3.5: Constantes em ponto flutuante

3.4 Constantes Caracteres

Uma constante caractere é um único caractere escrito entre ', como em 'a'. Além disso, uma constante de tamanho igual a um byte pode ser usada para definir um caracter, escrevendo-se, por exemplo, `'\ddd'`, onde `ddd` é uma constante com entre um e três dígitos octais. Em `C`, caracteres podem participar normalmente de expressões aritméticas. O valor que entra na expressão é o do código usado para representar o caractere. Exemplos de constantes do tipo caractere são mostrados na Tabela 3.6.

Caractere	Significado
`'a'`	caractere a
`'A'`	caractere A
`'\0141'`	Constante octal correspondente ao caractere 'a'
`'('`	caractere abre parênteses
`'9'`	algarismo 9
`'\n'`	Nova linha, posiciona o cursor no ínicio da nova linha.

Tabela 3.6: Exemplos de constantes caractere

Certos caracteres que não são visíveis podem ser representados antepondo-se o caractere `'\'` (barra invertida), como no exemplo nova linha da Tabela 3.6. Este caractere é também conhecido como caractere de escape. Exemplos são mostrados na Tabela 3.7.

3.4.1 Constantes Cadeias de Caracteres

Neste livro vamos usar em alguns casos a palavra cadeia para significar cadeia de caracteres (*string* em inglês). Uma constante do tipo cadeia de caracteres é uma seqüência de qualquer número de caracteres entre `"` como no exemplo: `"alo mundo!!!"`.

E importante notar que a linguagem `C` insere automaticamente ao final de uma cadeia de caracteres um caractere `null` (`'\0'`). Este caractere será usado

Caractere	Significado
'\n'	Passa para uma nova linha.
'\t'	Tabulação horizontal, move o cursor para a próxima parada de tabulação.
'\b'	Retorna um caractere.
'\f'	Salta uma página.
'\r'	*Carriage return*, posiciona o cursor no ínicio da linha atual.
'\a'	Alerta, faz soar a campainha do sistema.
'\0'	Null, caractere que em C termina uma cadeia de caracteres.

Tabela 3.7: Exemplos de caracteres invisíveis.

em diversos algoritmos como sinal de fim de cadeia. Os caracteres '\' (caractere escape) e '"' (início e fim de cadeia) têm significados especiais em cadeias de caracteres e para serem representados precisam ser antecedidos pelo caractere escape. Portanto, \\ e \" devem ser usados dentro de cadeias de caracteres para representar \ e " respectivamente. Por exemplo,

```
"Estas são \" (aspas) dentro de cadeias."
```

As aspas no meio da cadeia não indicam o fim, já que elas estão precedidas do caractere de escape.

3.5 Variáveis

Variáveis são nomes dados para posições de memória a fim de facilitar o manuseio dos dados durante a criação dos programas. Os dados podem ser de qualquer dos tipos definidos para a linguagem C.

3.5.1 Nomes das Variáveis

Existem algumas regras básicas que regulam o batismo de variáveis. Estas regras são as seguintes:

- Nomes de variável só podem conter letras, dígitos e o caractere '_';
- Todo primeiro caractere deve ser sempre uma letra ou o caractere '_';
- Letras maiúsculas e minúsculas são consideradas caracteres diferentes, isto é, C diferencia a caixa das letras;

- Palavras reservadas não podem ser usadas como nome de variáveis. Palavras reservadas são palavras usadas para indicar os comandos da linguagem, tipos de dados ou outras funções. O Anexo B mostra as palavras reservadas da linguagem `C`.

É boa política escolher nomes que indiquem a função da variável. Por exemplo:

```
soma         total           nome           raio
mediaNotas   salarioMensal   taxa_imposto   _inicio
```

Em `C` nomes como `raio`, `Raio` e `RAIO` referem-se a diferentes variáveis. No entanto, para afastar confusões, evite diferenciar nomes de variáveis por letras maiúsculas e minúsculas. Normalmente, os programadores usam letras maiúsculas para representar constantes.

Observe que em alguns nomes combinamos duas palavras para melhor indicar o dado armazenado na variável. Note também que o caractere espaço não pode ser usado em nomes de variáveis. Os programadores ao longo do tempo desenvolveram algumas regras informais para fazer esta combinação. Por exemplo, usa-se o caractere `'_'` para separar as palavras que compõem o nome, como em `taxa_imposto`. Outra maneira é usar letras maiúsculas para indicar quando começa uma palavra, como em `mediaNotas`. Alguns programadores usam a convenção de não começar nomes de variáveis por letras maiúsculas. Não existem regras formais para definir como nomes devem ser criados. O melhor é analisar as regras que programadores mais experientes usam ou os padrões que empresas adotam, para então escolher o que mais lhe agrada e segui-lo. Uma vez adotado um padrão ele deve ser seguido para evitar incoerências.

3.5.2 Declaração de variáveis

Para serem usadas, as variáveis precisam ser declaradas de modo que o compilador possa reservar espaço na memória para o valor a ser armazenado. A forma geral de uma declaração é:

tipo `lista_de_variáveis;`

onde uma `lista_de_variáveis` é uma lista de nomes de variáveis separadas por vírgulas. Por exemplo:

```
int i;
unsigned int a, b, c;
unsigned short int dia, mes, ano;
float raio, diametro;
double salario;
```

3.5.3 Atribuição de valores

Após ser declarada, uma variável pode receber valores. O operador de atribuição = indica que o resultado da expressão à direita do operador será atribuído à variável. Nada se pode afirmar sobre o conteúdo de uma uma variável que já foi declarada mas ainda não recebeu um valor.

A seguir são mostrados exemplos de atribuições de valores às variáveis durante as declarações.

```
int i = 0, j = 10;
float raio = 2.54;
char c = 'd';
double precisao = 0.00001L;
```

A seguir mostramos um trecho de programa com exemplos de atribuição de valores após a definição das variáveis.

```
int i, j;
float raio;
char c;
i = 0;
j = 10;
raio = 2.54;
c = 'd';
```

Exercícios

3.1: Indique os nomes de váriaveis que são válidos. Justifique os nomes inválidos.

(a)	`tempo`	(e)	`2dias`
(b)	`nota_final`	(f)	`teste 1`
(c)	`us$`	(g)	`raio.do.circulo`
(d)	`char`	(h)	`DiaHoje`

3.2: Indique quais dos números abaixo são constantes inteiras (longas ou não) válidas. Justifique suas respostas.

(a)	`100`	(e)	`- 234`
(b)	`2 345 123`	(f)	`0L`
(c)	`3.0`	(g)	`21`
(d)	`-35`	(h)	`0xF1`

3.3: Qual o valor na base 10 das constantes abaixo?

(a) `025`

(b) `0123`

(c) `0xD`

(d) `0x1D`

3.4: Considere um computador que armazene números inteiros em 32 bits.

(a) Caso um bit seja reservado para o sinal diga qual é o menor número inteiro negativo que este computador pode armazenar?

(b) Para os números sem sinal, qual é o maior número positivo?

Capítulo 4

Entrada e Saída pelo Console

4.1 Introdução

Neste capítulo vamos apresentar conceitos básicos de entrada e saída de dados para que os exemplos e exercícios iniciais possam ser construídos. Um programa que não fornece resultados nem pede valores para operar não deve ter grande utilidade. A entrada de dados será feita pelo teclado e a saída poderá ser vista na tela do computador. Em `C`, quando um programa se inicia, normalmente três fluxos (arquivos) de dados são abertos para operações de entrada e saída: um para entrada, um para saída e um para imprimir mensagens de erro ou diagnóstico. Normalmente o fluxo de entrada está conectado ao teclado, enquanto que o fluxo de saída e o de mensagens de erro, para serem visualizados, estão conectados ao monitor. Estas configurações podem ser alteradas de acordo com as necessidades dos usuários e estas operações são chamadas de redirecionamento. O fluxo de entrada é chamado de entrada padrão (*standard input*); o fluxo de saída é chamado de saída padrão (*standard output*) e o fluxo de erros é chamado de saída padrão de erros (*standard error output*). Estes termos são substituídos pelas suas formas abreviadas: `stdin, stdout` e `stderr`.

4.2 Biblioteca Padrão

Na linguagem `C` não existem comandos de entrada e saída. As operações de entrada e saída são executadas com auxílio de variáveis, macros e funções especiais. Para termos acesso à biblioteca que contém estas ferramentas o programa deve conter a declaração

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

no início do programa.

A diretiva **#include** instrui o compilador a ler o arquivo indicado entre `'<'` e `'>'`, e processá-lo como se ele fosse parte do arquivo original e seu conteúdo estivesse no ponto onde a diretiva foi escrita. Se o nome do arquivo estiver entre os sinais de maior e menor, como no exemplo, ele será procurado em um diretório específico de localização pré-definida, onde estão os arquivos de inclusão. Quando se usa aspas o arquivo é procurado de maneira definida pela implementação, isso pode significar procurar no diretório de trabalho atual, ou em um diretório indicado no comando usado para compilar o programa. Normalmente os programadores usam maior e menor para incluir os arquivos de cabeçalho padrão e aspas para a inclusão de arquivos do próprio projeto.

4.3 Saída - A Função `printf`

A função `printf` faz com que dados sejam escritos na saída padrão, que normalmente é a tela do computador. O protótipo da função é:

```
int printf(controle, arg1, arg2, ...);
```

onde os argumentos `arg1, arg2, ...` são impressos de acordo com o formato indicado pela cadeia de caracteres que compõe `controle`. O formato é ao mesmo tempo de uso simples e bastante flexível, permitindo que os resultados possam ser apresentados de diversas maneiras. A função retorna o número de caracteres impressos, não incluindo o `nulo` em vetores de caracteres. No caso de um erro de saída um valor negativo é retornado.

Um exemplo simples pode tornar a explicação mais clara. O programa 4.1 imprime o valor da variável `ano`.

Listing 4.1: Exemplo de impressão de resultados

```
#include <stdio.h>
int main (void)
{
    int ano = 1997;

    /* Imprime o valor do ano */
    printf("Estamos no ano %d", ano);
    return 0;
}
```

Este programa irá imprimir na tela do computador:

```
Estamos no ano 1997
```

Como `controle` é uma cadeia ele aparece entre `" "`. Ele define como serão impressos os valores representados pelos argumentos. No `controle` podem existir dois tipos de informações: caracteres comuns e códigos de formatação. Os

caracteres comuns, como no exemplo o texto `Estamos no ano`, são escritos na tela sem nenhuma modificação. Os códigos de formatação, aparecem precedidos por um caractere`'%'` e são aplicados aos argumentos na ordem em que aparecem. Deve haver um código de formatação para cada argumento. O código `%d` indica que o valor armazenado em `ano` deve ser impresso na notação inteiro decimal. É importante notar que o campo de `controle` aparece somente uma vez na função `printf` e sempre no início.

4.3.1 Códigos de Conversão

Os códigos de conversão estão mostrados na tabela 4.3.1.

Código	Comentário
`%c`	Caracter simples
`%d`	Inteiro decimal com sinal
`%i`	Inteiro decimal com sinal
`%E`	Real em notação científica com E
`%e`	Real em notação científica com e
`%f`	Real em ponto flutuante
`%G`	`%E` ou `%f`, o que for mais curto
`%g`	`%g` ou `%f`, o que for mais curto
`%o`	Inteiro em base octal
`%s`	Cadeia Caracteres
`%u`	Inteiro decimal sem sinal
`%x`	Inteiro em base hexadecimal (letras minúsculas)
`%X`	Inteiro em base hexadecimal (letras maiúsculas)
`%p`	Endereço de memória
`%%`	Imprime o caractere %

Tabela 4.1: Códigos de Conversão para leitura e entrada de dados.

Entre o caractere `%` e o `código de conversão` podem ser inseridos caracteres que alteram o formato. A seguir são mostrados a ordem de inserção destes caracteres e o seu significado:

`%[modificadores][largura][.precisão][comprimento]código`

`modificadores:` Usados logo após o caractere `%`.

`'-'` Um sinal de menos serve para especificar que o argumento deve ser justificado à esquerda no seu campo de impressão. Caso nenhum sinal seja usado o argumento será ajustado à direita. O programa 4.2 ilustra os dois tipos de justificação.

`'+'` Força que o resultado seja precedido por sinal de menos ou de mais, mesmo para números positivos. O padrão é que somente negativos sejam precedidos por sinal de menos.

`espaço` Caso nenhum sinal vá ser escrito, um espaço é inserido antes do valor.

`'#'` Usado com `o, x` ou `X` precede o valor com `0, 0x` ou `0X` respectivamente para valores diferentes de zero. Usado com `e, E` e `f`, força que a saída contenha um ponto decimal mesmo que não haja parte fracionária. Por padrão, se não há parte fracionária o ponto decimal não é escrito. Usado com `g` ou `G` o resultado é o mesmo que com `e` ou `E`, mas os zeros finais não são retirados.

`'0'` Completa o campo, pela esquerda, com zeros (0) ao invés de espaços, sempre que a opção para completar seja especificada (ver especificador de largura do campo).

`largura:` Caso seja usado um número inteiro, este especifica o tamanho mínimo do campo onde o argumento será impresso. Na listagem 4.2 o número especifica que 8 espaços são reservados para imprimir o resultado. Os espaços livres serão completados com espaços em branco. Se o argumento precisar de mais espaço que o especificado ele será escrito normalmente e o tamanho mínimo é ignorado.

`.precisão` Este número tem diferentes significados dependendo do código usado.

caracteres: No caso de impressão de cadeia de caracteres (`s`), este número especifica o número máximo de caracteres de uma cadeia de caracteres a serem impressos.

ponto flutuante: No caso de formato (`e, E, f`) é o número de dígitos a serem impressos a direita do ponto, ou seja o número de casas decimais. Para o formato `g` ou `G` é o número máximo dígitos significativos.

inteiros: No formatos inteiros (`d, i, o, u, x, X`) a precisão especificou o número máximo de dígitos a serem impressos. Se o número de caracteres a serem impressos é menor que este o resultado é completado com brancos. O valor não é truncado

`comprimento:` Modifica os formatos da seguinte maneira:

`l` Aplicado aos formatos de tipo `d`, `i`, `o`, `u`, `x` e `X` indicando que o dado é do tipo `long int` e não `int`.

`h` Modifica o dado, nos formatos `d`, `i`, `o`, `u`, `x` e `X` para tipo `short int`.

`L` Nos formatos `e`, `E`, `f`, `g` e `G` o argumento é modificado para `long double`.

Listing 4.2: Exemplo de justificação de resultados.

```
#include <stdio.h>
int main( void)
{
     int ano = 1997;

     printf("Justificado para direita Ano = %8d\n", ano);
     printf("Justificado para esquerda Ano = %-8d\n", ano);

     return 0;
 }
```

Listing 4.3: Exemplo de uso de especificador de precisão.

```
#include <stdio.h>
int main()
{
     float r = 1.0/3.0;
     char s[] = "Alo Mundo";

     printf("O resultado e = %9.3f\n", r);
     printf("%9.3s\n", s);

     return 0;

 }
```

O programa 4.2 irá imprimir o seguinte resultado:

```
Justificado para direita Ano =     1997
Justificado para esquerda Ano = 1997
```

O programa exemplo 4.3 imprimirá o seguinte resultado:

```
O resultado e = 0.333
.     Alo
```

Nos exemplos anteriores verifique que `'\n'` não é impresso. A barra inclinada é chamada de sequência de escape, indicando que o próximo caractere não é para ser impresso mas representa caracteres invisíveis ou caracteres que não estão representados no teclado. Esta seqüência de escape indica que o programa deve passar a imprimir na próxima linha.

4.4 Entrada - A Função `scanf`

A função `scanf` pode ser utilizada para entrada de dados a partir do teclado e seu protótipo é:

```
int scanf(controle, arg1, arg2, ...);
```

Uma diferença fundamental que existe entre esta função e a função `printf` está nos argumentos que vêm depois do `controle`. No caso de `scanf` os argumentos são os endereços das variáveis que irão receber os valores lidos e não, como em `printf`, as próprias variáveis. A indicação que estamos referenciando um endereço e não a variável se faz pelo operador `&`. Por exemplo, o comando

```
scanf(%d %d, &a, &b);
```

espera que dois valores inteiros sejam digitados no teclado. O primeiro é armazenado na variável `a` e o segundo em `b`. Os valores serão armazenados diretamente nos endereços indicados por `&a` e `&b` respectivamente.

Um outro exemplo incluindo variáveis reais é:

```
int i;
float x;
scanf("%d %f", &i, &x);
```

Assumindo que a linha de entrada no teclado fosse

```
34 56.43
```

a execução do exemplo iria terminar com o valor inteiro `34` sendo armazenado na variável `i` e o valor real `56.43` em `x`.

Usualmente o campo de controle só contém especificações de conversão, como os listados na Tabela 4.3.1, que são utilizadas para interpretar os dados que serão lidos, no entanto, como em `printf`, outros caracteres podem aparecer. O campo de controle pode conter:

Caracteres branco: A função lê e ignora todos os caracteres `branco` e/ou `<enter>` e/ou `tab` que aparecerem antes de qualquer caractere diferente destes.

Caracteres comuns: (não `%`) que devem casar com o próximo caractere diferente de branco da entrada. Isto significa que qualquer caractere que não for igual a branco e/ou `<enter>` e/ou `tab` ou parte de um especificador de formato faz com que a função leia o próximo caractere da entrada (`stdin`) e se for igual a este ele é descartado. Caso os caracteres sejam diferentes a função falha e retorna deixando os caracteres seguintes não lidos.

Especificações de conversão: Um especificador de conversão de formato seguindo um modelo similar ao da função `printf`.

O modelo é o seguinte:

`%{*}{largura}{modificadores}tipo`

O caracteres entre chaves são opcionais. O asterisco indica que o dado será lido de `stdin` mas ignorado. A largura especifica o número máximo de caracteres a serem lidos.

Os modificadores alteram o tamanho do especificadores de tipo que vêm logo a seguir. Existem os seguintes modificadores:

h: Os tipos `d, i` e `n`, que são **int** passam a ser **short int** e os tipos `o, u` e `x`, também **int** passam a ser **unsigned short int**.

l: Os tipos `d, i` e `n` passam a ser **long int** e os tipos `o, u` e `x` passam a **unsigned long int**. Os tipos `e, f` e `g` passam de **float** para **double**.

L: Os tipos `e, f` e `g` passam de **float** para **long double**.

Por exemplo, para que os valores digitados sejam separados por vírgulas, o comando deveria ser escrito da seguinte maneira:

```
scanf("%d, %f", &i, &x);
```

Observar que deve haver uma correspondência exata entre os caracteres não brancos do controle e os caracteres digitados. Neste caso a entrada deveria ser:

```
35, 46.3
```

O programa 4.4 mostra exemplos de uso da função `scanf`.

Listing 4.4: Exemplo de uso de `scanf`.

```
#include  <stdio.h>
int main ()
{
     char c;
     int num1, num2;

     printf("Entre com um caractere qualquer.\n");
     scanf("%c", &c);
     printf("Codigo ASCII do caractere %c vale %d.\n", c, c);
     printf("Agora dois inteiros separados por espaco.\n");
     scanf("%d %d", &num1, &num2);
     printf("A soma destes numeros vale %d.\n", num1+num2);
     return 0;
}
```

O resultado da execução deste programa é:

```
Entre com um caractere qualquer.
d
Codigo ASCII do caractere d vale 100.
Agora dois inteiros separados por espaco.
2 4
A soma destes numeros vale 6.
```

A função `scanf` retorna o número de itens lidos com sucesso. Este número pode ser usado para verificar se todos os valores pedidos foram lidos. No caso de ocorrer uma falha antes da leitura se iniciar a constante **EOF** é retornada.

4.5 Lendo e Imprimindo Caracteres

4.5.1 Funções `getchar` e `putchar`

Para ler e escrever caracteres do teclado as funções de entrada e saída mais simples são `getchar` e `putchar`, que estão na biblioteca `stdio.h` e cujos protótipos são os seguintes:

```
int getchar(void);
int putchar(int c);
```

Apesar da função `getchar` retornar um parâmetro inteiro é possível atribuir este valor a uma variável do tipo `char` porque o código do caractere está armazenado no byte ordem mais baixa. O mesmo acontece com a função `putchar` que recebe um inteiro, mas somente o byte de ordem mais baixa é passado para a tela do computador. A função `putchar` retorna o caractere que foi escrito e **EOF** em caso de erro. O programa da listagem 4.5 mostra exemplos de uso destas funções, e o seu resultado é:

```
Entre com um algarismo entre 0 e 9.
7
O caractere lido foi o 7
```

Listing 4.5: Exemplo de uso de `getchar` e `putchar`.

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

     printf("Entre com um algarismo entre 0 e 9.\n");
     c = getchar();

     printf("O caractere lido foi o ");
     putchar(c);

     return 0;
}
```

Observar que, normalmente, quando algum dado é fornecido pelo teclado termina-se a digitação com a tecla `<enter>`. No entanto, o `<enter>` é um caractere também, e isto pode causar problemas. Vamos analisar o que acontece quando antes do comando `getchar`, se lê um dado do tipo inteiro, por exemplo. O comando `scanf` lê o número inteiro mas não o `<enter>` digitado. Deste modo, quando logo em seguida o programa executar a função `getchar`, o que será lido é o `<enter>` digitado ao final do número. A listagem 4.6 é um exemplo de programa onde isto pode ocorrer. Considere que o usuário digitou `35<enter>` como resposta ao comando `scanf`. O comando `getchar` irá ler o `<enter>` e em

seguida o programa irá imprimir o número 35, lido no scanf, e apenas uma linha em branco correspondente ao caractere <enter>, lido pelo getchar, como está indicado a seguir. Mais adiante mostraremos como resolver este problema.

```
Entre com um numero inteiro.
35
Agora um caractere.
Numero lido 35
Caractere lido

```

Listing 4.6: Exemplo de uso de getchar e putchar.

```
#include  <stdio.h>
int main (void)
{
     char c;
     int i;

     printf("Entre com um numero inteiro.\n");
     scanf("%d", &i);

     printf("Agora um caractere.\n");
     c = getchar();

     printf("Numero lido %d\n", i);
     printf("Caractere lido %c\n", c);
     return 0;
}
```

4.5.2 Lendo e Imprimindo Cadeias de Caracteres

Uma cadeia de caracteres (*string*) em C é um vetor de caracteres. Vetores, que serão vistos mais adiante no Capítulo 7, são conjuntos de caracteres em que cada um deles pode ser acessado independentemente dos outros por meio de um endereço. Nesta etapa iremos apresentar rapidamente alguns conceitos que nos permitirão criar alguns exemplos simples com cadeias de caracteres. Para usar cadeias é preciso primeiro definir um espaço para armazená-las. Para isto é preciso declarar o nome, o tamanho e o tipo do vetor. Considere que precisamos armazenar uma cadeia de caracteres chamada nome com 40 caracteres. A definição desta cadeia ficaria da seguinte maneira:

```
char nome[41];
```

Quando definir o tamanho do vetor de caracteres, observar que toda cadeia em C termina com o caractere NULL ('\0'), que é automaticamente inserido pelo compilador. Portanto o vetor nome deve ser definido com um espaço a

mais. Após este passo, o vetor `nome` pode ser usado durante a execução do programa.

4.5.3 Lendo e Imprimindo cadeias com `scanf` e `printf`

O programa 4.7 mostra como ler e imprimir um cadeia usando os comandos `scanf` e `printf` respectivamente.

Listing 4.7: Exemplo de uso de `printf` e `scanf` na leitura de cadeias.

```
#define DIM 40
#include<stdio.h>
int main (void)
{
     char nome[DIM]; /* linha de caracteres lidos */

     /* Entrada de dados do vetor */
     printf("Por favor, qual o seu nome?\n");
     scanf("%s", nome);
     printf("Sou um computador. Posso ajuda-lo %s?\n", nome);
     return 0;
}
```

Considere que este programa se chama `util`. Uma possível interação entre este programa e um usuário poderia ser da seguinte maneira.

```
$ util
Por favor, qual o seu nome?
Ze Sa
Sou um computador.  Posso ajuda-lo Ze?
```

O símbolo $ é o *prompt* típico dos sistemas `Unix`. Aparentemente o computador se tornou íntimo do usuário `Ze Sa` e o tratou apenas pelo primeiro nome. A explicação para esta intimidade está no modo de leitura. Quando se usa `scanf` para ler uma cadeia deve-se empregar o código de conversão `%s`. Este comando não lê o nome todo, mas encerra a leitura dos caracteres quando encontra um caractere espaço (ou branco), ou seja o separador de cadeias no comando `scanf` é o caractere espaço. Mas como ler para um vetor um nome inteiro, ou um cadeia que contenha brancos? Para isto deve-se usar a função `gets` que será nosso próximo assunto.

4.5.4 Lendo e Imprimindo cadeias com `gets` e `puts`

Diferentemente do comando `scanf` a função `gets` lê toda a cadeia até que a tecla `<enter>` seja digitada. No vetor são colocados todos os códigos dos caracteres lidos excetuando-se o da tecla `<enter>`, que não é armazenado sendo substituído

pelo código NULL. Caso a função scanf do exemplo anterior fosse substituída pela gets o programa imprimiria

```
Posso ajuda-lo Ze Sa?
```

O comando que substitui o scanf é gets(nome). O protótipo da função gets é o seguinte:

```
#include<stdio.h>
char *gets (char *str);
```

A função gets retorna str caso nenhum erro ocorra. Caso o final do arquivo seja encontrado antes de qualquer caractere ser lido, o vetor permanece inalterado e um ponteiro nulo é retornado. Caso um erro ocorra durante a leitura, o conteúdo do array fica indeterminado e novamente um ponteiro nulo é retornado.

A função puts tem o seguinte protótipo:

```
#include<stdio.h>
int puts (const char *str);
```

Ela imprime a cadeia apontado por str. O programa 4.8 é semelhante ao exemplo anterior com as funções printf substituídas por puts. Observe que a impressão sempre termina e passa para a próxima linha. A função puts retorna um valor positivo caso nenhum erro ocorra. Em caso de erro é retornado um valor negativo.

```
Entre com o seu nome, por favor.
Ze Sa
Alo
Ze Sa
Eu sou um computador, em que posso ajuda-lo?
```

Listing 4.8: Exemplo de uso de puts e gets na leitura de cadeias.

```
#define DIM 41
#include <stdio.h>
int main ( void )
{
     char nome[DIM];  /* linha de caracteres lidos */
     /* Entrada de dados do vetor */
     puts("Entre com o seu nome, por favor.");
     gets( nome);
     puts("Alo ");
     puts(nome);
     puts("Eu sou um computador, em que posso ajuda-lo?");
     return 0;
}
```

4.5.5 A Função `fgets`

A função `gets` pode abrir porta para invasões de computadores pelo fato dela não controlar o número de caracteres lido de `stdin`. Apesar do usuário definir um tamanho máximo para o vetor que irá armazenar os caracteres a função ignora o limite e continua lendo valores até que o usuário digite o caractere `<enter>`.

Para evitar este problema recomenda-se o emprego da função `fgets` cujo protótipo é

```
#include<stdio.h>
int *fgets (const char *str, int tam, FILE *fluxo);
```

A função `fgets` lê no máximo um caractere a menos que o número de caracteres especificado no parâmetro `tam` a partir do fluxo de entrada de dados definido por `fluxo`. No caso de leitura do teclado, como temos feito, fluxo é igual a `stdin`. A leitura é interrompida quando um caractere `<enter>` é encontrado ou o final do arquivo foi atingido. Diferentemente do que ocorre na função gets, aqui o caractere `<enter>` é armazenado no vetor onde os demais caracteres estão sendo guardados. O caractere nulo é adicionado após o último caractere lido.

A função retorna `str` caso seja bem sucedida. Se o final do arquivo for atingido e nenhum caractere tiver sido lido, o vetor `str` permanece inalterado e um ponteiro `nulo` é retornado. Caso ocorra um erro de leitura o conteúdo do vetor fica indeterminado e um ponteiro `nulo` é retornado.

Exercícios

4.1: Escreva um programa que declare variaveis do tipo `int, char` e `float`, inicialize-as, e imprima os seus valores.

4.2: Escreva um programa que defina variáveis do tipo `int` e armazene nelas constantes octais e hexadecimais e imprima o seu conteúdo no formato original e em formato decimal.

4.3: Faça um programa que leia um valor inteiro no formato decimal e escreva, na tela, este mesmo valor nas bases hexadecimal e octal.

```
Exemplo de Entrada e Saída:
Entre com o valor:
10
Hexadecimal:  A
Octal:  12
```

4.4: Faça um programa capaz de ler um valor real e escrevê-lo com apenas uma casa decimal.

4.5: Faça um programa que leia três palavras de até 10 letras e reescreva estas palavras alinhadas à direita da tela.

4.6: Sabendo que os argumentos da função `printf` podem ser expressões (`a+b`, `a/b`, `a*b`, `3*a`...), e não somente argumentos, faça um programa capaz de ler um valor inteiro e escrever seu triplo, seu quadrado, e a sua metade.

```
Exemplo de Entrada e Saída:
Valor:
6
Triplo:  18
Quadrado:  36
Meio:  3
```

4.7: Escreva um programa que leia 3 números reais e imprima a média aritmética destes números.

4.8: Escreva um programa que pegue o valor de uma conta de restaurante e imprima o valor total a ser pago, considerando que o restaurante cobra 10% de taxa para os atendentes.

4.9: Faça um programa que peça ao usuário a quilometragem atual, a quilometragem anterior, os litros consumidos e informe a taxa de consumo (quilômetros por litro) de um automóvel.

4.10: Escreva um programa que converta uma temperatura de Farenheit para Celsius.

4.11: Escreva um programa que, dado o perímetro de um círculo, calcule sua área.

4.12: Faça um programa que utilize a função `gets` para ler duas cadeias de tamanho até 20 e em seguia às reescreva na linha de baixo, uma ao lado da outra e separadas por "/-/ ";

Capítulo 5

Operadores e Expressões

5.1 Introdução

O objetivo deste capítulo é apresentar os operadores existentes na linguagem `C` e a forma correta de construir expressões que envolvam estes operadores, constantes e variáveis.

5.2 Operador de Atribuição

Este é o operador usado para transferir o resultado de uma expressão para uma variável. Em `C` este operador é o sinal de igual (`=`). Esta escolha do sinal de igual para servir de operador de atribuição pode causar problemas. Isto porque este sinal não está representando que o resultado da expressão do lado direito é igual ao resultado do lado esquerdo e sim uma atribuição. Observe que o comando de atribuição termina em ponto e vírgula. Isto faz parte das regras da linguagem `C`, que determina que comandos terminam com este caractere. Por exemplo:

```
soma = a + b;
pi = 3.1415;
```

É possível fazer-se várias atribuições em uma única linha, como no exemplo a seguir:

```
a = b = c = 1.0;
```

as três váriaveis recebem o mesmo valor. As atribuições são feitas na seguinte ordem:

1. `c = 1.0;` `c` recebe o valor `1.0`.
2. `b` recebe o resultado da expressão à sua direita, que é o valor atribuído à `c`, ou seja `1.0`.

3. `a` recebe o resultado da expressão à sua direita, que é o valor atribuído à `b`, ou seja `1.0`.

5.3 Operadores Aritméticos

A Tabela 5.3 mostra os operadores aritméticos e as suas ordens de precedência.

Operador	`Descrição`	`Prioridade`
`+`	Mais unário	0
`-`	Menos unário	0
`++`	Incremento	1
`--`	Decremento	1
`*`	Multiplicação	2
`/`	Divisão	2
`%`	Resto da divisão	2
`+`	Soma	3
`-`	Subtração	3

Tabela 5.1: Operadores aritméticos.

Os símbolos mostrados na Tabela 5.3 são os únicos que podem ser usados para representar as operações acima listadas. Expressões aritméticas em `C` devem ser escritas no formato linear para facilitar a digitação dos programas e também porque alguns símbolos usados em Matemática não existem nos teclados. O exemplo mais comum deste formato é a operação de divisão que deve ser escrita `a/b`.

Parênteses têm um papel importante nas expressões e permitem que a ordem das operações seja alterada. Expressões entre parênteses são calculadas em primeiro lugar, portanto eles conferem o maior grau de prioridade as expressões que eles envolvem. Podemos ter pares de parênteses envolvendo outros pares. Dizemos que os parênteses estão aninhados. Neste caso as expressões dentro dos parênteses mais internos são avaliadas primeiro.

Outro ponto importante são as regras de precedência que determinam que operação deve ser executada primeiro. Na tabela os operadores estão listados em ordem decrescente de prioridade. Para os operadores aritméticos a operação de mais alta precedência é o `-` unário, vindo em seguida `++`, `--` com a mesma prioridade. Os operadores de multiplicação, divisão e módulo tem a mesma prioridade. O operador menos unário multiplica seu operador por -1. Quando duas operações de mesmo nível de prioridade têm de ser avaliadas, a operação mais à esquerda será avaliada primeiro.

Um ponto importante que deve ser sempre levado em consideração quando uma expressão for calculada são os tipos das variáveis, porque eles alteram radicalmente os resultados das expressões. Por exemplo, a divisão entre operandos

do tipo inteiro tem como resultado um valor inteiro. Portanto, se o resultado possuir uma parte fracionária ela será truncada. Não é possível aplicar a operação de módulo a operandos do tipo `float` e `double`. Algumas regras de conversão simples existem e serão discutidas em detalhes mais adiante. Por exemplo a operação `1/3` em `C` fornece como resultado o valor 0, enquanto que 1 % 3 é igual a 3.

A seguir mostramos alguns exemplos de expressões aritméticas escritas na notação da linguagem `C`. Observe o uso de parênteses para evitar ambiguidades que poderiam fazer com que a expressão fosse calculada erradamente.

Exemplo 5.4:

1. $a + \frac{b}{b+c} \Longrightarrow$ `a + b/(b+c)`
2. $b^2 + c^2 \Longrightarrow$ `b*b + c*c`
3. $\frac{x}{a+\frac{b}{c}} \Longrightarrow$ `x/(a+b/c)`

5.4 Operadores Relacionais e Lógicos

5.4.1 Operadores Relacionais

Os operadores relacionais estão mostrados na Tabela 5.2. Nesta tabela mostramos somente a ordem de precedência destes operadores. A ordem de precedência que inclui todos os operadores está mostrada na Tabela 5.10.

Operador	Descrição	Prioridade
`>=`	Maior ou igual a	0
`>`	Maior que	0
`<=`	Menor ou igual a	0
`<`	Menor que	0
`==`	Igual a	1
`!=`	Diferente de	1

Tabela 5.2: Operadores Relacionais.

Os operadores `>`, `>=`, `<` e `<=` têm a mesma precedência e estão acima de `==` e `!=`. Estes operadores têm precedência menor que os aritméticos, portanto expressões como `( i < limite - 1)` e `i < (limite -1)` têm o mesmo significado.

5.4.2 Operadores Lógicos

Os operadores lógicos definem as maneiras como as relações acima podem ser conectadas. Por exemplo podemos querer testar se ao mesmo tempo uma nota é maior ou igual a 5.0 e a taxa de presença é maior que 75%.

Para simplificar a apresentação destes operadores serão usadas variáveis para substituir as relações. Neste caso a expressão acima seria representada como `p e q`, onde `p` está representando nota maior ou igual a 5.0 e `q` taxa de presença maior que 75%. Estas expressões podem ter dois resultados `verdadeiro` e `falso`. Observar que, assim como em operações aritméticas, podemos ter combinações de mais de duas relações em uma única expressão. Por exemplo, podemos ter a seguinte combinação: ano maior que 2000 e mês menor que 6 e dia maior que 15. Nas linguagens de programação os valores `verdadeiro` e `falso` podem ser representados de diversas maneiras. Uma das maneiras mais comum é representar `verdadeiro` por `true` e `falso` por `false`. Em `C` o valor `falso` é representado por `0` e `verdadeiro` por qualquer valor diferente de `0`. A seguir iremos mostrar os operadores lógicos existentes na linguagem `C`.

E lógico

O símbolo usado para representar o operador `E` lógico é `&&`. A Tabela 5.3 mostra a tabela verdade do operador. O resultado da expressão é `verdadeiro` se e somente se todas as variáveis forem iguais a `verdadeiro`. Por exemplo, considere o seguinte trecho de programa:

```
int i = 3, j = -5;
float z = 3.0;
int resultado;

resultado = (10 > 5) && ( i > -5) && (z != 0);
printf("O resultado e vale %d.", resultado);
```

O resultado deste trecho é a impressão de um valor diferente de `0`, ou seja o valor correspondente a `verdadeiro`. Isto porque 10 é maior que 5 `E i` é maior que -5 `E z` é diferente de 0.

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

Tabela 5.3: Operador Lógico `E`.

OU lógico

O símbolo usado para representar o operador `OU` lógico é `||`. A Tabela 5.4 mostra a tabela verdade do operador. Para que o resultado da expressão seja verdade basta que uma das variáveis seja verdade. Por exemplo, considere o seguinte trecho de programa:

```
float x = 3.0;
int n = 55, i = 0;
int resultado;

resultado = (i != 0) || (x == 0) || (n < 100);
printf("O resultado e \%d", resultado);
```

O resultado deste trecho é a impressão do valor `1`. Isto porque, apesar de `i` não ser diferente de 0 e `x` não ser diferente de zero, temos que `n` é menor que 100. Como basta um dos testes ser verdade para o resultado ser verdade será impresso um valor diferente de 0.

p	q	p || q
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 5.4: Operador Lógico `OU`.

Não lógico

O símbolo usado para representar o operador `E` lógico é `!`. A Tabela 5.5 mostra a tabela verdade do operador. Este operador é unário e quando aplicado à uma variável ele troca seu valor. Por exemplo, considere o seguinte trecho de programa:

```
int dia = 25, ano = 1959;
int resultado;

resultado = ! ( (dia < 30) && (ano > 1950) )
printf ("O resultado vale \%d.", resultado);
```

Este trecho de programa imprime `0 (falso)`, porque `dia` é menor que 30 `E` `ano` é maior que 1950. Portanto, o resultado do parênteses vale 1 (`verdadeiro`). No entanto, o operador `!` nega este valor que vira 0.

A tabela 5.6 mostra, em ordem decrescente, a precedência dos operadores lógicos e relacionais.

p	!p
0	1
1	0

Tabela 5.5: Operador Lógico NÃO.

Operador	Prioridade
!	0
>, >=, <, <=	1
==, !=	2
&&	3
||	4

Tabela 5.6: Precedência dos operadores lógicos e relacionais.

5.5 Operadores com Bits

Para operações com bits, a linguagem C dispõe de alguns operadores que podem ser usados nos tipos char, int, long e long long mas não podem ser usados em float, double, long double e void. A diferença entre estes operadores e os lógicos é que estes operam em pares de bits enquanto que os operadores lógicos anteriores consideram a palavra toda. Por exemplo, para um valor int ser falso é necessário que todos os 32 bits sejam iguais a zero. Os operadores em bits estão mostrados na Tabela 5.7.

Operador	Descrição	Prioridade
&	E	
|	OU	
^	Ou exclusivo	
~	Não	
>>	Desloca para direita	
<<	Desloca para esquerda	

Tabela 5.7: Operadores com bits.

Os operadores &, | e ~ têm a mesma tabela verdade que os operadores &&, || e ! respectivamente. O operador ^ (OU Exclusivo) está descrito pela Tabela 5.8. O resultado da operação é verdadeiro se e somente se os dois operandos são diferentes.

Os operandos de deslocamento têm os seguintes modos de operação:

operando >> vezes: o operando é deslocado vezes bits para a direita.

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

Tabela 5.8: Operador Lógico `OU`.

operando << vezes: o `operando` é deslocado `vezes` bits para a esquerda.

Observações:

- Nos deslocamentos à direita em variáveis `unsigned` e nos deslocamentos à esquerda, os bits que entram são zeros;
- Nos deslocamentos à direita em variáveis `signed`, os bits que entram correspondem ao sinal do número (1= sinal negativo, 0 = sinal positivo).
- Um deslocamento para a direita é equivalente a uma divisão por 2. Deslocamento para a esquerda é equivalente a uma multiplicação por 2. Assim `a = a * 2;` e `a = a << 1;` são equivalentes.

O exemplo 5.1 ilustra o uso dos operandos de deslocamento:

Listing 5.1: Exemplo de operadores de deslocamento.

```
#include<stdio.h>
int main (void)
{
     unsigned int c = 7;
     int d = -7;

     c = c<<1; printf("%3d = %08X\n", c, c);
     c = c>>1; printf("%3d = %08X\n", c, c);

     d = d<<1; printf("%3d = %08X\n", d, d);
     d = d>>1; printf("%3d = %08X\n", d, d);

     return 0;
}
```

Este programa teria como resposta os seguintes resultados.

```
 14 = 0000000E
  7 = 00000007
-14 = FFFFFFF2
 -7 = FFFFFFF9
```

Os resultados mostram que o número 7 após o primeiro deslocamento de 1 bit para a esquerda ficou igual a 14, portanto um 0 entrou no número. Quando o número foi deslocado para direita 1 bit, ele retornou ao valor original. Observe que quando o número -14 foi deslocado para a direita entrou um bit 1, que é igual ao sinal negativo.

5.6 Operadores de Atribuição Composta

Em `C` qualquer expressão da forma:

```
variavel = variavel operador expressao
```

pode ser escrita como:

```
variavel operador= expressao
```

Por exemplo:

```
ano = ano + 10;
```

é equivalente a

```
ano += 10;
```

Outros exemplos são:

```
raiz = raiz * 4;
raiz *= 4;
soma = soma / ( a + b);
soma /= (a + b);
a = a >> 1;
a >>= 1;
i = i % 2;
i %= 2;
```

5.7 Operador vírgula

O operador vírgula (`,`) é usado para separar duas ou mais expressões que são escritas onde somente uma é esperada. Quando o conjunto de expressões tem de ser reduzido a somente um valor, somente a expressão mais à direita é considerada. Por exemplo, considere o seguinte trecho de código:

```
y = (x=5, x+2);
```

A expressão começa a ser avaliada da esquerda para a direita. Portanto, primeiro seria atribuído o valor `5` a variável `x`. Em seguida atribui `x+2` para a variável `y`. Ao final a variável `x` contém o valor `5` e `y` o valor `7`.

5.8 Operador sizeof()

O operador `sizeof()` é um operador unário que retorna o tamanho em bytes da expressão ou tipo fornecido entre parênteses. Por exemplo, suponha que o tipo `float` tenha quatro bytes então o operador `sizeof(float)` retorna o valor 4. Para se calcular o tamanho de bytes de uma expressão não é necessário o uso de parênteses. No exemplo 5.2 ilustramos alguns exemplos de uso do operador `sizeof()`.

Listing 5.2: Exemplo do operador `sizeof`.

```
#define DIM 10
#include<stdio.h>
#include<conio.h>
int main()
{
     int i=0;
     float f=3.0;
     char c='a';
     int v[DIM];

     printf("Tamanho em bytes de alguns tipos\n");
     printf("Tamanho de int %d\n", sizeof i);
     printf("Tamanho do float %d\n", sizeof f);
     printf("Tamanho do double %d\n", sizeof (double));
     printf("Tamanho do char %d\n", sizeof c);
     printf("Tamanho do vetor de %d inteiros \%d\n",
               DIM, sizeof(v));
     return 0;
}
```

Este programa imprime os seguintes resultados:

```
Tamanho em bytes de alguns tipos
Tamanho de int 4
Tamanho do float 4
Tamanho do double 8
Tamanho do char 1
Tamanho do vetor de 10 inteiros 40
```

5.9 Conversão de Tipos

Quando operandos de tipos diferentes aparecem em expressões são convertidos para um tipo comum, que permita o cálculo da expressão da forma mais eficiente. Por exemplo, uma operação que envolva um tipo `int` e um `float`, o valor `int` é convertido para `float`.

Observar que conversões ocorrem somente quando necessário. Por exemplo, em uma divisão de inteiros o resultado é do tipo inteiro. Isto pode causar surpresas desagradáveis para programadores iniciantes. A expressão `1/3*3` tem como resultado o valor inteiro 0. Já que a primeira expressão executada `1/3` tem como resultado 0.

Operandos do tipo `char` e `int` podem ser livremente misturados em expressões aritméticas. Os tipos `char` são convertidos para `int`. Caso o conjunto de caracteres esteja codificado segundo a tabela ASCII, esta facilidade permite a realização de algumas transformações interessantes. Por exemplo, a conversão de uma letra maiúscula para minúscula pode ser facilmente implementada com o comando:

```
l = l - 'A' + 'a';
```

A letra maiúscula armazenada na variável `l` é subtraída do código da letra maiúscula `'A'`, fornecendo a posição desta letra no alfabeto. Em seguida este valor é somado ao código da letra minúscula `'a'`, resultando da conversão para minúscula.

Portanto, conversões aritméticas ocorrem de maneira quase que natural. Em operações binárias as seguintes conversões ocorrem quando diferentes tipos estão envolvidos:

- `char` é convertido para `int`;
- `float` é convertido para `double`.

Então, se algum dos operandos é `double` o outro é convertido para `double` e o resultado é `double`. Caso contrário, se algum dos operandos é `long`, o outro é convertido para `long` e o resultado é `long`. Caso contrário, se algum dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado é deste tipo. Caso contrário os operandos são `int` e o resultado é `int`. Note que todos os `floats` em uma expressão são convertidos para `double` e a expressão é avaliada em `double`.

O resultado de uma expressão é convertido para o tipo da variável onde o resultado será armazenado. Um resultado `float` ao ser carregado em uma variável do tipo `int` causa o truncamento da parte fracionária, porventura existente.

A conversão de inteiro para caractere é bem comportada, mas o contrário nem sempre ocorre convenientemente. A linguagem não especifica se o tipo `char` é um tipo com sinal ou não. Quando um caractere é armazenado em uma variável do tipo inteiro podem ocorrer problemas com caracteres que têm o bit mais à esquerda igual a 1. Isto porque algumas arquiteturas podem extender este bit e outras não.

5.10 Regras de Precedência

A Tabela 5.10 mostra, em ordem decrescente de prioridade, as regras de precedência dos operadores em C. Os operadores que estão na mesma linha da tabela e com a mesma ordem têm a mesma prioridade. Alguns dos operadores listados na Tabela somente serão mostrados nos capítulos seguintes.

Pri	Operador	Descrição
0	`() [] -> .`	Agrupamento; acesso vetor; acesso membro
1	`! ~ ++ -- * &`	Unárias lógicas, aritméticas e com ponteiros;
1	(tipo) sizeof()	Conformação de tipo; tamanho
2	`* / %`	Multiplicação, divisão e módulo
3	`+ -`	soma e subtração
4	`>> <<`	Deslocamento de bits à direita e esquerda
5	`< <= >= >`	Operadores relacionais
6	`== !=`	Igualdade e diferença
7	`&`	E bit a bit
8	`^`	Ou exclusivo bit a bit
9	`|`	Ou bit a bit
10	`&&`	E
11	`||`	Ou
12	`? () : ()`	Ternário
13	`= += -= *= /= %=`	Atribuições
13	`>>= <<= &= |=`	Atribuições
14	`,`	Separador de expressões

Tabela 5.9: Precedência dos operadores.

Exercícios

5.1: Escreva as expressões `C` abaixo na sua forma matemática usual:

1. `(a/b)*(c/d)`
2. `(a/b*c/d)`
3. `(a/(b*c)/d)`
4. `a*x*x+b*x+c`

5.2: Escreva as expressões matemáticas na linguagem `C`.

1. $b^2 - 4 \cdot b \cdot c$
2. $\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}$
3. $\frac{a+b}{c+d}$
4. $a \times \frac{x}{c+d}$

5.3: Diga a ordem de cálculo e o resultado das expressões abaixo:

1. `x = 5 * 4 / 6 + 7;`
2. `x = 5 * 4.0 / 6 + 7;`
3. `x = 5 * 4 % 6 + 7;`
4. `x = ((4 / 2) + (3.0 * 5));`

5.4: Escreva um programa que imprima a tabela verdade da função ou exclusivo.

5.5: Escreva um programa que calcule o produto entre um valor x e 2^n, onde n e x são inteiros. Utilize operadores binários.

5.6: Escreva um programa que leia um ângulo em segundos e imprima quantos graus, minutos e segundos há neste ângulo.

5.7: Escreva um programa que leia um tempo em segundos e imprima quantas horas, minutos e segundos há neste tempo.

5.8: Escreva um programa que leia um comprimento em centímetros e imprima quantos metros, decímetros e centímetros há neste comprimento.

Capítulo 6

Comandos de Controle

6.1 Introdução

Este capítulo tem por objetivo apresentar os comandos de controle da linguagem C. Estes comandos servem para controlar o fluxo de execução das instruções de um programa. Estes comandos permitem que o computador tome decisões independentemente do usuário que está rodando o programa.

6.2 Blocos de Comandos

Blocos de comando são grupos de comandos que devem ser tratados como uma unidade lógica. O início de um bloco em C é marcado por uma chave de abertura ({) e o término por uma chave de fechamento (}). O bloco de comandos serve para agrupar comandos que devem ser executados juntos. Por exemplo, usa-se bloco de comandos quando em comandos de teste deve-se escolher entre executar dois blocos de comandos. Um bloco de comandos pode ser utilizado em qualquer trecho de programa onde se pode usar um comando C. É interessante observar que um bloco de comandos pode ter zero comandos C. Um bloco de comandos com 0 ou 1 comando pode dispensar as chaves. Um bloco de comandos é mostrado a seguir.

```
/* bloco_de_comandos */
{
    i = 0;
    j = j + 1;
    printf("%d %d\n", i, j);
}
```

6.3 Comandos de Teste

Os comandos de teste permitem ao computador decidir o caminho a seguir, durante a execução do programa, indepentemente do usuário. Estes testes são baseados em estados internos disponíveis ao processador. Estes estados podem ser resultantes de uma operação aritmética anterior, de uma operação anterior etc.

6.3.1 Comando if

O comando `if` é utilizado quando for necessário escolher entre dois caminhos. A forma geral do comando `if` é a seguinte:

```
if (expressão)
  bloco_de_comandos1;
else
  bloco_de_comandos2;
```

Neste comando a expressão é avaliada, e caso o resultado seja `verdadeiro` (qualquer resultado diferente de zero) o `bloco_de_comandos1` é executado, caso contrário o `bloco_de_comandos2` é executado. Pela definição do comando a expressão deve ter como resultado um valor diferente de zero para ser considerada verdade. Observar que somente um dos dois blocos será executado. Como a cláusula `else` é opcional a forma abaixo do comando `if` é perfeitamente válida.

```
if (expressão)
  bloco_de_comandos;
```

Lembrar que os blocos de comandos devem ser delimitados pelas chaves, a não ser quando o bloco é composto por 0 ou 1 comando. A seguir mostramos alguns exemplos de uso de comando `if`:

```
scanf("%d", &dia);
if ( dia > 31 && dia < 1 )
    printf("Dia invalido\n");

scanf("%d", &numero);
if ( numero >= 0 )
  printf("Numero positivo\n");
else
  printf("Numero negativo\n");

scanf("%f", &salario);
if (salario < 800.00)
{
  printf("Aliquota de imposto = 0.1\n");
  imposto = salario * 0.1;
}
else
```

```
{
  printf("Aliquota de imposto = 0.25\n");
  imposto = salario * 0.25;
}
```

Uma construção que pode aparecer são os comandos `if's` em escada, cuja forma geral é a seguinte:

```
if (expressão)
     bloco_de_comandos
else       if (expressão1)
           bloco_de_comandos1
     else       if (expressão2)
                bloco_de_comandos2
                ...
           else bloco_de_comandosn
```

O programa 6.1 mostra um exemplo com `if's` em escada e aninhados. Um exemplo de uso deste programa é o seguinte:

```
Este programa simula uma calculadora simples.
Por favor entre com os dois operandos.
3 5
Qual a operacao
+
O resultado da + vale 8.000000.
```

Para evitar que o recuo da escada seja muito profundo o comando `if` em escada foi escrito da seguinte maneira:

```
if (expressão)
  bloco_de_comandos;
else if (expressão)
  bloco_de_comandos;
else if (expressão)
  bloco_de_comandos;
  ...
else bloco_de_comandos;
```

6.3.2 Comando `switch`

O comando `if`, em todas suas formas, é suficiente resolver problemas de seleçao de comandos. Porém em alguns casos, como no exemplo 6.1 o programa se torna mais trabalhoso para ser escrito e entendido. O comando `switch` facilita a escrita de trechos de programa em que a seleção deve ser feita entre várias alternativas.

A forma geral do comando `switch` é a seguinte:

Listing 6.1: Programas com if´s em escada e aninhados.

```
#include  <stdio.h>
int main (void)
{
     float     num1,          /* primeiro operando */
          num2,          /* segundo operando */
          res;      /* resultado da operacao */
     char oper;          /* caracter que define a operacao */

     printf("\nEste programa simula uma calculadora simples.\n");
     printf("Por favor entre com os dois operandos.\n");
     scanf("%f %f", &num1, &num2); getchar(); /* tirar o cr */
     printf("Qual a operacao? \n");
     oper = getchar();
     if (oper == '+')
          res = num1 + num2;
     else if (oper == '-')
          res = num1 - num2;
     else if (oper == '*')
          res = num1 * num2;
     else if (oper == '/')
    {
          if (num2 == 0.0)
          {
               printf("Operacao de divisao por 0 invalida!\n");
               return 1;
          }
          else res = num1 / num2;
     }
     else
     {
          printf("Operacao invalida!\n");
          return 1;
     }
     printf("O resultado da %c vale %f.\n", oper, res);
     return 0;
}
```

```
switch (expressão)
{
     case constante1:
          seqüência_de_comandos;
          break;
     case constante2:
          seqüência_de_comandos;
          break;
     case constante3:
          seqüência_de_comandos;
          break;
          ...
     default:
     seqüência_de_comandos;
}
```

Uma seqüência de comandos é diferente de um bloco de comandos. Um bloco de comandos inicia com uma chave e termina com uma chave, enquanto que uma seqüência é apenas uma série de comandos. Por exemplo, uma vez que um bloco de comandos foi selecionado por um comando `if` ele será executado até a última instrução do bloco, a menos que haja um comando de desvio. Uma série de comandos são apenas comandos colocados um após outro. A execução do comando `switch` segue os seguintes passos:

1. A expressão é avaliada;
2. O resultado da expressão é comparado com os valores das constantes que aparecem nos comandos `case`;
3. Quando o resultado da expressão for igual a uma das constantes, a execução se inicia a partir do comando associado com esta constante. A execução continua até o fim do comando `switch`, ou até que um comando `break` seja encontrado;
4. Caso não ocorra nenhuma coincidência os comandos associados ao comando `default` são executados. O comando `default` é opcional, e se ele não aparecer nenhum comando será executado.

O comando `break` é um dos comandos de desvio da linguagem `C`. O `break` é usado dentro do comando `switch` para interromper a execução da seqüência de comandos e pular para o comando seguinte ao comando `switch`.

Há alguns pontos importantes que devem ser mencionados sobre o comando `switch`.

- O resultado da expressão deve ser um tipo enumerável, por exemplo o tipo `int`. Também podem ser usados tipos compatíveis com `int`, isto é, expressões com resultados tipo `char` podem ser usadas;

- Notar que caso não apareça um comando de desvio, todas as instruções seguintes ao teste `case` que teve sucesso serão executadas, mesmo as que estejam relacionadas com outros testes `case`;
- O comando switch só pode testar igualdade;
- Não podem aparecer duas constantes iguais em um `case`;

O programa 6.2 mostra um exemplo de uso de comandos `switch`.

6.3.3 Comando Ternário

O comando ternário tem este nome porque necessita de três operandos para ser avaliado. O comando ternário tem a seguinte forma:

```
expressão1 ?  expressão2 :  expressão3
```

Para avaliar o resultado total da expressão, primeiro av`expressão1` é avaliada. Caso este resultado seja correspondente ao valor `verdadeiro` então o resultado da expressão será igual ao resultado da `expressão2`. Caso contrário a `expressão3` é avaliada e se torna o resultado. O programa 6.3 mostra um exemplo de uso de comando ternário.

6.4 Laços de Repetição

Estes comandos permitem que trechos de programa sejam repetidos um certo número de vezes controlado pelo programa. O número de vezes que um laço será executado pode ser fixo ou depender de condições que mudam durante a execução do laço.

6.4.1 Comando `for`

Este comando aparece em várias linguagens de programação, mas na linguagem `C` ele apresenta uma grau maior de flexibilidade. A idéia básica do comando for é a seguinte. Uma variável de controle, geralmente um contador, recebe um valor inicial. O trecho de programa que pertence ao laço é executado e ao final a variável de controle é incrementada ou decrementada e comparada com o valor final que ela deve alcançar. Caso a condição de término tenha sido atingida o laço é interrompido. A forma geral do comando `for` é a seguinte:

```
for (expressão1; expressão2; expressão3)
     blocodecomandos;
```

As três expressões geralmente têm os seguintes significados:

1. A `expressão1` é utilizada para inicializar a variável de controle do laço;

Listing 6.2: Exemplo de switch.

```
#include  <stdio.h>
int main (void)
{
     float     num1,          /* primeiro operando */
          num2,          /* segundo operando */
          res;      /* resultado da operacao */
     char oper;          /* caracter que define a operacao */

     printf("\nEste programa simula uma calculadora simples.\n");
     printf("Por favor entre com os dois operandos.\n");
     scanf("%f %f", &num1, &num2); getchar();
     printf("Qual a operacao \n");
     oper = getchar();
     printf("A operacao e %c\n", oper);
     switch (oper) {
          case '+':
               res = num1 + num2;
            break;
          case '-':
               res = num1 - num2;
               break;
          case '*':
               res = num1 * num2;
               break;
          case '/':
               if (num2 == 0.0){
                    printf("Divisao por zero e uma opcao invalida.\n");
                    return 1;
            }
               else {
                    res = num1 / num2;
                    break;
            }
          default:
               printf("Operacao invalida!\n");
               return 2;
     }
     printf("O resultado da %c vale %f.\n", oper, res);
     return 0;
}
```

Listing 6.3: Exemplo de comando ternário.

```
#include  <stdio.h>
int main (void)
{
     float     num1,          /* primeiro operando */
          num2,          /* segundo operando */
          max;      /* resultado da operacao */

     printf("Imprime o maior valor de dois numeros.\n");
     printf("Por favor entre com os dois mumeros.\n");
     scanf("%f %f", &num1, &num2);
     max = (num1>num2)?num1:num2;
     printf("O maior dos numeros lidos e %f.\n", max);
     return 0;
}
```

2. A `expressão2` é um teste que controla o fim do laço;

3. A `expressão3` normalmente faz um incremento ou decremento da variável de controle.

A execução do comando `for` segue os seguintes passos:

1. A `expressão1` é avaliada;

2. A `expressão2` é avaliada para determinar se o comando deve ser executado;

3. Se o resultado da `expressão2` for `verdadeiro` o bloco de comandos é executado, caso contrário o laço é terminado;

4. A expressão3 é avaliada;

5. Voltar para o passo 2.

O trecho a seguir imprime todos os números entre 1 e 100.

```
for (i = 1; i <= 100; i++)
{
     printf("Numero %d\n", i);
}
```

O programa 6.4 mostra como se pode calcular o fatorial de um número usando-se o comando `for`.

Listing 6.4: Exemplo de comando for.

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

int main()
{
     int numero, fat=1, i;

     printf("\nEntre com um numero positivo.");
     scanf("%d", &numero);
     for (i=numero; i>1; i--) fat = fat * i;
     printf("O fatorial de %u vale %u.", numero, fat);
     return 0;
}
```

Laços `for` com mais de um comando por expressão

Outra possibilidade que o comando `for` em `C` permite é a inclusão de vários comandos, separados por vírgulas, nas expressões. O trecho de programa a seguir mostra um exemplo de uso de comando `for` com vários comandos nas expressões.

```
int i,j;
for ( i=1, j = 10; i <= 10; i++, j += 10)
{
 printf ("i =  %d, j = %d\n", i, j);
}
```

Laços `for` com testes usando outras variáveis

A expressão de controle não precisa necessáriamente envolver somente um teste com a variável que controla o laço. O teste de final do laço pode ser qualquer expressão relacional ou lógica. No programa 6.5 o laço pode terminar porque a variável de controle já chegou ao seu valor limite ou foi batida a tecla `'*'`, e neste caso o laço termina antecipadamente.

Laços `for` com expressões faltando

Um outro ponto importante do for é que nem todas as expressões precisam estar presentes. No exemplo 6.6 a variável de controle não é incrementada. A única maneira do programa terminar é o usuário bater o número -1.

É possível omitir qualquer uma das expressões. Por exemplo, se a `expressão2` for omitida o programa assume que ela é sempre verdade de modo que o laço só termina com um comando de desvio como o `break`. O programa do exemplo

Listing 6.5: Exemplo de comando for com testes sobre outras variáveis.

```
#include<stdio.h>
int main()
{
     char c = ' ';
     int i;
     for (i=0 ; (i<5) && (c != '*'); i++ )
     {
          printf("%c\n", c);
          c = getchar();
     }
     return 0;
}
```

Listing 6.6: Exemplo de comando for sem alteração da variável de controle.

```
#include<stdio.h>
int main()
{
     int i;
     for (i=0 ; i != -1 ; )
     {
          printf("%d\n",i );
          scanf ("%d", &i);
     }
    return 0;
}
```

6.7 para quando o valor da variável de controle for igual a 5. Neste caso o teste será verdade o laço termina por meio do `break`.

Laço infinito

Uma construção muito utilizada é o laço infinito. No laço infinito o programa pára quando se executa o comando `break`. O trecho de programa a seguir somente pára quando for digitada a tecla `'s'` ou `'S '`.

```
for ( ; ; )
{
     printf("\nVoce quer parar?\n");
     c = getchar();
     if (c == 'S' || c == 's') break;
}
```

Listing 6.7: Exemplo de comando for sem teste de fim.

```
#include<stdio.h>
int main()
{
     int i;
     for (i = 0; ; i++)
     {
          printf("numero %d\n", i);
          if (i == 5) break;
     }
     return 0;
}
```

Laços `for` aninhados

Uma importante construção aparece quando colocamos como comando a ser repetido um outro comando `for`. Esta construção pode aparecer quando estamos trabalhando com matrizes. O exemplo 6.8 mostra um programa que imprime uma tabuada.

Listing 6.8: Comando for aninhados.

```
#include<stdio.h>
int main(void)
{
     int i, j;

     printf("Imprime tabuada de multiplicacao.\n");
     for (i=1 ; i<10 ; i++)
     {
          printf("Tabuada de %d\n", i);
          for (j=1; j<10; j++)
          {
               printf("%d x %d = %d\n", i, j, i * j);
          }
     }
     return 0;
}
```

6.4.2 Comando `while`

O comando `while` tem a seguinte forma geral:

```
while (expressão)
    bloco_de_comandos
```

A expressão pode assumir o valor `falso` (igual a 0) ou `verdade` (diferente de 0). Os passos para execução do comando são os seguintes:

1. A expressão é avaliada;
2. Se o resultado for `verdadeiro` então o bloco de comandos é executado, caso contrário a execução do bloco é terminada;
3. Voltar para o passo 1.

Uma característica do comando `while`, como pode ser visto dos passos acima, é que o bloco de comandos pode não ser executado caso a condição seja igual a `falso` logo no primeiro teste.

O trecho de abaixo imprime os 100 primeiros números usando um comando `while`.

```
i = 1;
while (i <= 100)
{
     printf("Numero %d\n", i);
     i++;
}
```

A expressão do comando pode incluir chamadas de função. Lembrar que qualquer atribuição entre parênteses é considerada como uma expressão que tem como resultado o valor da atribuição sendo feita. Por exemplo, o programa 6.9 repete um bloco de comandos enquanto o usuário usar a tecla `'c'` para continuar, qualquer outra tecla o bloco é interrompido.

Listing 6.9: Comando while com uma função.

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

     puts("Tecle c para continuar.\n");
     while ((c=getchar()) == 'c')
     {
          puts("Nao Acabou.\n");
          getchar(); /* tira o enter */
     }
     puts("Acabou.\n");
     return 0;
}
```

6.4.3 Comando `do-while`

A forma genérica do comando é a seguinte:

```
do
    bloco_de_comandos
while (expressão);
```

Observar que neste comando a expressão de teste está após a execução do comando, portanto o bloco de comandos é executado pelo menos uma vez. A execução do comando segue os seguintes passos:

1. Executa o comando;
2. Avalia a expressão;
3. Se o resultado da expressão for `verdadeiro` então volta para o passo 1, caso contrário interrompe o `do-while`

O exemplo de comando `for` para imprimir os 100 primeiros números escrito com comando `do-while` fica da seguinte maneira:

```
i = 1;
do   {
     printf("Numero %d\n", i);
     i++;
} while (i <= 100);
```

6.5 Comandos de Desvio

6.5.1 Comando `break`

O comando `break` pode ser tanto usado para terminar um teste `case` dentro de um comando `switch` quanto interromper a execução de um laço. Quando o comando é utilizado dentro de um comando `for` o laço é imediatamente interrompido e o programa continua a execução no comando seguinte ao comando `for`. No trecho de programa abaixo o comando for deve ler 100 números inteiros positivos. No entanto, se for digitado um número negativo o comando `for` é interrompido imediatamente sem que o número seja impresso.

```
for (i = 0; i < 100; i++)
{
     scanf("%d", &num);
     if (num < 0) break;
     printf("%d\n", num);
}
```

6.5.2 Comando `continue`

O comando `continue` é parecido com o comando `break`. A diferença é que o comando `continue` simplesmente interrompe a execução da iteração corrente passando para a próxima iteração do laço, se houver uma. No comando `for` o controle passa a execução da `expressão3`. Nos comandos `while` e `do-while` o controle passa para a fase de testes.

No trecho de programa abaixo o laço lê 100 números inteiros, caso o número seja negativo, um novo número é lido.

```
for (i = 0; i < 100; i++)
{
      scanf("%d", &num);
      if (num < 0) continue;
      printf("%d\n", num);
}
```

6.5.3 Comando `goto`

O comando `goto` causa um desvio incondicional para um outro ponto da função em que o comando está sendo usado. O comando para onde deve ser feito o desvio é indicado por um rótulo, que é um identificador válido em `C` seguido por dois pontos. É importante notar que o comando `goto` e o ponto para onde será feito o desvio pode estar em qualquer ponto dentro da mesma função. A forma geral deste comando é:

```
goto rótulo;
...
rótulo:
```

Este comando durante muito tempo foi associado a programas ilegíveis. O argumento para esta afirmação se baseia no fato de que programas com comandos `goto` perdem a organização e estrutura porque o fluxo de execução pode ficar saltando erraticamente de um ponto para outro. Atualmente as restrições ao uso do comando tem diminuído e seu uso pode ser admitido em alguns casos.

6.5.4 Função `exit()`

A função exit provoca a terminação de um programa, retornando o controle ao sistema operacional. O protótipo da função é a seguinte:

```
void exit (int codigo);
```

Observar que esta função interrompe o programa como um todo. O código é usado para indicar qual condição causou a interrupção do programa. Usualmente o valor 0 indica que o programa terminou sem problemas. Um valor diferente de 0 indica um erro.

6.5.5 Comando `return`

O comando `return` é usado para interromper a execução de uma função e retornar um valor ao programa que chamou esta função. Caso haja algum valor associado ao comando `return` este é devolvido para a função, caso contrário um valor qualquer é retornado. A forma geral do comando é:

```
return expressão;
```

Notar que a expressão é opcional. A chave que termina uma função é equivalente a um comando `return` sem a expressão correspondente. É possível haver mais de um comando `return` dentro de uma função. O primeiro que for encontrado durante a execução causará o fim da execução. Uma função declarada como do tipo `void` não pode ter um comando `return` que retorne um valor. Isto não faz sentido, já que funções deste tipo não podem retornar valores.

Exercícios

6.1: Escreva um programa que calcule `x` elevado a `n`. Assuma que n é um valor inteiro.

6.2: Escreva um programa que exiba as opções
`‘‘1-multiplicar’’` e `‘‘2-somar’’`
de um menu, leia a opção desejada, leia dois valores, execute a operação (utilizando o comando `if`) e exiba o resultado.

6.3: Utilizando `if's` em escada, inclua, no programa do exercício anterior, as opções `‘‘3-Subtrair’’` e `‘‘4-Dividir’’`.

6.4: Simplifique os programas anteriores da seguinte forma:

- Reescreva o programa do exercício 1 substituindo o comando `if` pelo comando ternário.
- Reescreva o programa do exercício 2 substituindo os `if's` em escada pelo comando `switch`.

6.5: Utilizando um laço `for` dentro de outro, escreva um programa que exiba as tabuadas de multiplicação dos números de 1 à 9.

6.6: Escreva um programa com menu de 5 opções que utilize o comando de desvio `goto` para executar a opção desejada e só saia do programa caso a opção `‘‘5-Sair’’` seja selecionada.

6.7: Escreva um programa que tenha um número (inteiro) como entrada do usuário e escreva como saída a sequencia de bits que forma esse numero. Por exemplo, após digitado o número `10`, a saida deve ser `0000000000001010`.

6.8: Escreva um programa que imprima todos os numeros pares entre 0 e 50 e em seguida imprima todos os impares. Deixar um espaco entre os numeros.

6.9: Escreva um programa que leia 10 numeros. O programa deve imprimir a media, o maior e o menor deles.
Obs: Os numeros devem ser entre 0 e 10.

6.10: Escreva um programa que leia 10 numeros. O programa deve imprimir a media, o maior e o menor deles.
Obs: Considere agora que os números podem ser quaisquer.

6.11: Escreva um programa que exibe a tabela ascii.

6.12: Crie um programa para verificar se um número dado é primo.

6.13: Escreva um programa que leia um numero do teclado e ache todos os seus divisores.

6.14: Escreva um programa que imprima a seqüência
`‘‘9876543218765432176543216543215432143213211’’`

Não use nenhuma constante, use apenas variáveis. Em outra linha imprima as letras maiúsculas de `A` até `Z (ABCD...)`.

6.15: Escreva um programa que conte de 100 a 999 (inclusive) e exiba, um por linha, o produto dos três digitos dos numeros. Por exemplo, inicialmente o programa irá exibir:

```
0 (1*0*0)
0 (1*0*1)
0 (1*0*2)
(...)
0 (1*1*0)
1 (1*1*1)
2 (1*1*2)
...
9*9*9=729
```

Faça seu programa dar uma pausa a cada 20 linhas para que seja possível ver todos os números pouco a pouco. Solicite que seja pressionada alguma tecla para ver a próxima sequencia de números.

Capítulo 7

Vetores e Cadeias de Caracteres

7.1 Introdução

Vetores são usados para tratamento de conjuntos de dados que possuem as mesmas características. Uma das vantagens de usar vetores é que o conjunto recebe um nome comum e elementos deste conjunto são referenciados através de índices. Pelo nome vetor estaremos referenciando estruturas que podem ter mais de uma dimensão, como por exemplo matrizes de duas dimensões. Neste capítulo estaremos mostrando vetores de tamanhos fixos. Somente após apresentarmos ponteiros iremos abordar alocação de memória para vetores.

7.2 Declaração de Vetores Unidimensionais

A forma geral da declaração de vetores de uma dimensão é:

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

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

Exemplos de declarações de vetores são:

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

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

```
espaço = tamanho * sizeof(tipo)
```

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

O programa 7.1 ilustra como se declara um vetor, inicializa seus valores e imprime o conteúdo. Notar o uso da diretiva `#define DIM 5` para definir uma constante, que posteriormente foi usada para estabelecer o tamanho do vetor. Esta constante passa a ser usada nas referências ao vetor, por exemplo no comando de geração do conjunto de dados armazenado no vetor. Caso seja necessário trocar o tamanho do vetor basta alterar o valor da constante e recompilar o programa.

Listing 7.1: Exemplo de vetores.

```
#define DIM 5
#include <stdio.h>
int main(void)
{
     int vetor[DIM];
     unsigned int i, num;

     puts("Este programa gera um vetor de inteiros.\n");
     puts("Entre com o numero inicial do conjunto. ");
     scanf("%d", &num);

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

     /* Impressao do conjunto */
     for (i = 0; i < DIM; i++)
          printf("Elemento %d = %d\n", i, vetor[i]);

     return 0;
}
```

O programa 7.2 calcula o produto escalar de dois vetores inteiros. Observar como na leitura dos elementos do vetor usa-se o operador de endereço `&` antes do nome de cada elemento.

O programa 7.3 ilustra o método da bolha para ordenação em ordem crescente de um vetor de inteiros. Neste método a cada etapa o maior elemento é movido para a sua posição. A cada iteração os elementos do vetor são comparados dois a dois, sendo trocados caso seja necessário. Ao término da primeira passada pelo vetor, o maior elemento é levado para a sua posição, no final do vetor. Portanto, ele não precisa ser mais considerado, daí o valor da variável

Listing 7.2: Produto escalar de dois vetores.

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

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

     printf("Entre com um vetor de %d elementos\n", DIM);
     for (i = 0; i < DIM; i++)
     {
          printf("Elemento %d ", i);
          scanf("%d", &vetor1[i]);
     }
     printf("Entre com outro vetor de %d elementos\n", DIM);
     for (i = 0; i < DIM; i++)
     {
          printf("Elemento %d ", i);
          scanf("%d", &vetor2[i]);
     }

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

     printf("O produto vale %d", prod);

     return 0;
}
```

Operação	v[0]	v[1]	v[2]	v[3]	v[4]
Passo 1					
v[0] > v[1]?	20	15	8	12	5
Trocar v[0] e v[1]	15	20	8	12	5
v[1] > v[2]?	15	20	8	12	5
Trocar v[1] e v[2]	15	8	20	12	5
v[2] > v[3]?	15	8	20	12	5
Trocar v[2] e v[3]	15	8	12	20	5
v[3] > v[4]?	15	8	12	20	5
Trocar v[3] e v[4]	15	8	12	5	20
Passo 2					
v[0] > v[1]?	15	8	12	5	20
Trocar v[0] e v[1]	8	15	12	5	20
v[1] > v[2]?	8	15	12	5	20
Trocar v[1] e v[2]	8	12	15	5	20
v[2] > v[3]?	8	12	15	5	20
Trocar v[2] e v[3]	8	12	5	15	20
Passo 3					
v[0] > v[1]?	8	12	5	15	20
v[1] > v[2]?	8	12	5	15	20
Trocar v[1] e v[2]	8	5	12	15	20
Passo 4					
v[0] > v[1]?	8	5	12	15	20
Trocar v[0] e v[1]?	5	8	12	15	20

Tabela 7.1: Passos executados durante o algoritmo da bolha.

que aponta para o final do vetor (`fim`) é diminuída de 1. O processo é repetido até que todos os elementos sejam levados para as suas posições ou que nenhuma troca seja realizada. Quando nenhuma troca é realizada o vetor está ordenado. A Tabela 7.1 mostra os passos executados pelo algoritmo até ordenar o vetor.

7.3 Cadeias de Caracteres

Um cadeia de caracteres (*string*) é um conjunto de caracteres terminado por um caractere nulo, que é representado como `'\0'`. Para especificar um vetor para armazenar um cadeia deve-se sempre reservar um espaço para este caracter. Por exemplo, para armazenar um cadeia de 40 caracteres deve-se reservar um vetor de 41 de caracteres. Em `C` é possível haver constantes cadeia, que são definidas como uma lista de caracteres entre aspas. Por exemplo,

Listing 7.3: Ordenação pelo método da bolha.

```
#define DIM 5
#define FALSO 0
#define VERDADE 1

#include  <stdio.h>
int main (void)
{
     int vetor[DIM], i;
     int trocou = FALSO, fim=DIM, temp;

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

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

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

     return 0;
}
```

```
"programando em C"
```

Não é necessário a colocação do caracter nulo ao final da cadeia. Em `C` não há o tipo cadeia (*string*) e, portanto, conjuntos de caracteres teriam de ser tratados como conjuntos de números inteiros, por exemplo. Para facilitar a programação foram criadas algumas funções para manipular cadeias. Algumas das funções mais comuns estão resumidamente descritas a seguir:

Nas definições a seguir, `size_t` é o tipo inteiro sem sinal que volta como resultado do operador **sizeof**.

- **char** `*strcat(char *dest, const char *orig)`: Concatena cadeia `orig` ao final de `dest`. O primeiro caracter de `orig` substitui o caracter nulo de `dest`. A função retorna o valor de `dest`.

- **char** `*strncat (char *dest, const char *orig, size_t n)`: Concatena cadeia `orig` ao final de `dest`, usando no máximo `n` caracteres de `orig`. O primeiro caracter de `orig` substitui o caracter nulo de `dest`. A função retorna o valor de `dest`.

- **char** `*strcmp (const char *cad1, const char *cad2)`: Compara lexicograficamente as duas cadeias. Retorna zero se as cadeias são iguais, menor que 0 se `cad1 < cad2`, maior que 0 se `cad1 > cad2`.

- **char** `*strncmp (const char *cad1, const char *cad2, size_t n)`: Compara lexicograficamente até `n` caracteres das duas cadeias. Retorna zero se as cadeias são iguais, menor que 0 se `cad1 < cad2`, maior que 0 se `cad1 > cad2`.

- `size_t strlen(const char *cad)`: Calcula o comprimento da cadeia sem contar o caracater nulo. O comprimento da cadeia é determinado pelo caractere nulo. Não confundir o tamanho da cadeia com o tamanho do vetor que armazena a cadeia.

- **char** `*strcpy(char *dest, const char *orig)`: Copia cadeia `orig` para `dest`. A cadeia destino deve ter espaço suficiente para armazenar `orig`. O valor de `dest` é retornado.

Estas funções estão na biblioteca `string.h`. O programa 7.4 mostra exemplos de uso de algumas das funções de cadeia. Neste exemplo, o programa primeiro lê um nome e em seguida um sobrenome. O programa irá então concatenar as duas cadeias. Observe que sempre é colocado um branco ao final do nome para separá-lo do sobrenome. Este branco é inserido usando a função `strcat`, e esta é razão das aspas, ou seja, uma cadeia de um caractere apenas. A seguir mostramos um resultado da execução do programa 7.4.

```
Entre com um nome Ze
Ze
Entre com um sobrenome Sa
Sa

Ze Sa

Qual caracter?  a
O caractere aparece na posicao 4
```

Listing 7.4: Exemplos de funções para cadeias.

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

int main( void )
{
     char c, nome[81], sobrenome[41];
     int i;

     printf("Entre com um nome ");
     fgets(nome, 41, stdin);
     nome[strlen(nome)-1] = '\0'; /* tira cr do fim */
     puts(nome);

     printf("Entre com um sobrenome ");
     fgets(sobrenome, 41, stdin);
     sobrenome[strlen(sobrenome)-1] = '\0';
     puts(sobrenome);

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

     printf("Qual caracter? ");
     c = getchar();

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

7.4 Declaração de Vetores Multidimensionais

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

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

onde dimI é o tamanho da dimensão I. Deve-se tomar cuidado com armazenamento de matrizes multidimensionais, por que a memória necessária para guardar estes dados é igual a

```
sizeof(tipo)*dim1*dim2*dim3*...*dimN
```

Por exemplo a declaração

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

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

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

armazena o dobro do elemento que está na quarta linha e nona coluna na variável c. Observar que o primeiro índice indica a linha e o segundo a coluna. Lembrar que o número da primeira linha (coluna) é igual a 0. O programa 7.5 lê uma matriz de três linhas e cinco colunas e imprime os valores lidos.

Listing 7.5: Leitura de uma matriz.

```
#define DIML 3
#define DIMC 5
#include <stdio.h>
int main( void )
{
     int i, j;
     int matriz[DIML][DIMC];

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

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

A matriz é armazenada na memória linha a linha e a Figura 7.4 ilustra esta idéia com uma matriz de números inteiros de três por três. Estamos assumindo que cada número inteiro ocupa quatro bytes, o endereço aponta um byte e a matriz está armazenada a partir do endereço 1000.

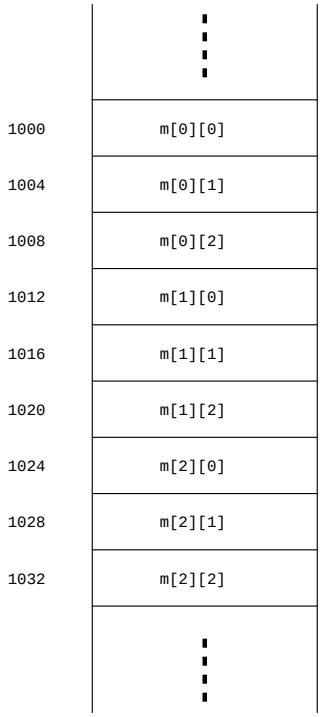

Figura 7.1: Mapa de memória de uma matriz.

Uma operação muito comum em matemática é a multiplicação de matrizes. Considere a matriz $M1$ com $L1$ linhas e $C1$ colunas e a matriz $M2$ com $L2$ linhas e $C2$ colunas. O número de colunas $C1$ de $M1$ deve ser igual ao número de linhas $L2$ de $M2$. O elemento MR_{ij} da matriz resultado MR do produto destas matrizes é definido pela equação 7.1. O programa 7.6 multiplica duas matrizes de acordo com a fórmula.

$$MR_{ij} = \sum_{k=1}^{C1} M1_{ik} \times M2_{kj} \tag{7.1}$$

7.5 Vetores de Cadeias de Caracteres

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

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

Listing 7.6: Multiplicação de duas matrizes.

```
#include<stdio.h>

#define L1 3
#define L2 3
#define C1 3
#define C2 3

int main ( void )
{
     float m1[L1][C1], m2[L2][C2];
     float mr[L1][C2], m;
     int i, j, k;

     for (i=0; i<L1; i++)
     {
          for (j=0; j<C1; j++)
          {
               printf ("%d, %d ", i,j);
               scanf("%f", &m1[i][j]);
          }
     }
     for (i=0; i<L2; i++)
     {
          for (j=0; j<C2; j++)
          {
               printf ("%d, %d ", i,j);
               scanf("%f", &m2[i][j]);
          }
     }

     for (i=0; i<L1; i++)
     {
          for (j=0; j<C2; j++)
          {
               m = 0;
               for (k=0; k<C1; k++)
               {
                    m += m1[i][k]*m2[k][j];
               }
               mr[i][j] = m;
          }
     }

     for (i=0; i<L1; i++ )
     {
          for (j=0; j<C2; j++)
          {
               printf("%.3f ", mr[i][j]);
          }
          printf("\n");
     }
     return 0;
}
```

O Programa 7.7 mostra um programa que lê uma matriz de nomes e imprime os seus conteudos. É importante notar que para ler um nome o programa não lê um caracter de cada vez mas usa a função `fgets`. Como cada linha da matriz é uma cadeia de caracteres, o programa lê o nome que está na linha `i` como `fgets(nomes[i], DIMC-1, stdin)`.

Listing 7.7: Leitura de um vetor de nomes.

```
#define DIML 5
#define DIMC 41
#include <stdio.h>
#include <string.h>
int main( void )
{
     int i;
     char nomes[DIML][DIMC];

     for (i=0; i < DIML; i++)
     {
          printf("Entre com a linha %d ", i);
          fgets(nomes[i], DIMC-1, stdin);
          nomes[i][strlen(nomes[i])-1] = '\0';
     }

     for (i=0; i < DIML; i++)
     {
          printf("O nome %d e ", i);
          puts(nomes[i]);
     }

     return 0;
}
```

7.6 Inicialização de Vetores e Matrizes

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

```
tipo nome[dim] = {lista_de_valores};
```

onde `lista_de_valores` é um conjunto de valores separados por vírgulas. Por exemplo, a declaração abaixo inicializa um vetor inteiro de cinco posições.

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

Observe que nesta declaração é necessário que o tamanho do conjunto seja conhecido antecipadamente. No entanto, também é possível inicializar vetores

em que não se conhece o seu tamanho. Neste caso, então, é importante que o programador preveja um modo de indicar o fim do vetor.

O Programa 7.8 mostra os casos ilustrados acima. No primeiro exemplo o tamanho do vetor é conhecido e foi definido pela constante `DIM`. Para descobrir o como parar de processar o vetor, quando desconhecemos seu tamanho, apresentamos duas soluções possíveis. No primeiro caso a condição de fim do vetor é o número negativo -1. Neste caso uma posição do vetor é perdida para armazenar esta condição. No segundo caso é usado o operador `sizeof` para descobir o tamanho do vetor. Observe que `sizeof` calcula o tamanho do vetor em bytes e por esta razão é necessário uma divisão pelo tamanho em bytes do tipo de cada elemento.

Listing 7.8: Exemplos de tratamento de vetores.

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

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

     printf("Este programa imprime vetores ");
     printf("contendo numeros inteiros e\n");
     printf("que foram inicializados durante ");
     printf("a sua declaracao.\n");

     /* Impressao dos conjuntos */
     printf("\nVetor com tamanho pre-definido\n");
     for (i=0; i  < DIM; i++)
          printf("Elemento %d = %d\n", i, vetor[i]);

     printf("\nVetor terminando por -1\n");
     for (i=0; vetor1[i]>0; i++)
          printf("Elemento %d = %d\n", i, vetor1[i]);

     tam = sizeof (vetor2) / sizeof (int);
     printf("\nDescobrindo o tamanho do Vetor\n");
     for (i=0; i  <  tam ; i++)
          printf("Elemento %d = %d\n", i, vetor2[i]);

     return 0;
}
```

É possível inicializar matrizes multidimensionais e neste caso é necessário especificar todas as dimensões menos a primeira, para que o compilador possa

reservar memória de maneira adequada. A primeira dimensão somente especifica quantos elementos o vetor irá armazenar e isto lendo a inicialização o compilador pode descobrir.

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

```
float mat[][4] = { {1.0, 2.0, 3.0, 4.0},   // linha 0
                   {8.0, 9.0, 7.5, 6.0},   // linha 1
                   {0.0, 0.1, 0.5, 0.4} }; // linha 2
```

O Programa 7.9 ilustra a definição de um vetor de cadeia de caracteres, que nada mais é do que uma matriz de caracteres. Note que as cadeias são separadas uma das outras por vírgulas.

Listing 7.9: Exemplos de tratamento de vetores.

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

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

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

     return 0;
}
```

Exercícios

7.1: Escreva um programa que leia uma linha de até 80 caracteres do teclado e imprima quantos caracteres foram lidos.

7.2: Escreva um programa que leia uma linha de caracteres do teclado e imprima quantas vezes um caracter, também fornecido pelo teclado, aparece nesta linha. O programa também deve imprimir em que posições o caracter foi encontrado.

7.3: Escreva um programa que leia uma linha do teclado e em seguida um par de caracteres. O programa deve procurar este par na linha e imprimir em que posições o par foi encontrado. Obs. Não use funções da biblioteca de `strings` do `C`

7.4: Escreva um programa que leia uma linha do teclado e imprima todas as vogais encontradas no texto e o total de vezes que elas aparecem.
Obs: Tamanho maximo da linha deve ser 40 caracteres.

7.5: Oimperador romano Cesar usava um sistema simples para codificar as mensagens que enviava aos seus generais. Neste sistema cada letra era substituída por três letras à frente no alfabeto. A sua missão é mais simples ainda, escrever um programa que converta cada letra, e somente as letras, de uma mensagem de até 80 caracteres para a letra imediatamente posterior. Note que a letra `'z'` deve ser convertida para a letra `'a'`, e a letra `'Z'` para `'A'`.

7.6: Escreva um programa que leia uma frase de 80 caracteres e a imprime retirando os espaços em branco.

7.7: Escreva um programa que leia uma linha de caracteresdo teclado de tamanho 80. A linha somente contém letras. Divida a linha em blocos de 5 letras. Dentro de cada bloco o seu programa deve trocar a primeira letra pela letra seguinte no alfabeto, a segunda letra por duas letras adiante no alfabeto, a terceira por três letras adiante e assim até a quinta. Os espaços em branco devem ser retirados da frase. Considere o seguinte exemplo.

1. Frase lida:
   `EVA VIU A UVA`
2. Retirada dos espaços em branco:
   `EVAVIUAUVA`
3. Divisão em blocos de 5 (blocos indicados por tipos diferentes):
   `EVAVI`*UAUVA*
4. Criptografia:
   `FYDANVCYAF`

Portanto, o que será impresso pelo programa é:
`FYDANVCYAF`

7.8: Escreva um programa que leia uma matriz de 3x3 que contém somente caracteres 0 e X e procure linhas que contenham somente um dos dois caracteres. O caracter a ser procurado deve ser lido do teclado.

7.9: Escreva um programa que leia uma linha de caracteres do teclado e converta o primeiro caracter de cada palavra para maiúsculas. Assuma que as palavras são sempre separadas por um branco.

7.10: Escreva um programa que leia para um vetor um conjunto de números inteiros. Assuma que o conjunto de números lidos é menor que o tamanho do vetor. O programa deve inserir no vetor, em uma posição especificada pelo usuário, um número lido do teclado. Assuma que a posição especificada pelo usuário corresponde ao índice do vetor.

7.11: Faça um programa que inverta uma cadeia de caracteres. O programa deve ler a cadeia com `gets` e armazená-la invertida em outra cadeia. Use o comando `for` para varrer a cadeia até o seu final.

7.12: Escreva um programa que leia um conjunto de nomes para uma matriz e imprima estes nomes em ordem alfabética. Assuma que os nomes serão lidos somente em letras maiúsculas. Assuma também que os nomes têm no máximo 40 caracteres e serão lidos 10 nomes ao todo.

Capítulo 8

Funções

8.1 Introdução

Em `C`, diferentemente de outras linguagens como Pascal, todas as ações ocorrem dentro de funções. Na linguagem `C` não há conceito de um programa principal, o que existe é uma função chamada `main` que é sempre a primeira a ser executada. Um programa pode ser escrito apenas com a função `main` e mais as funções existentes nas bibliotecas da linguagem `C`. No entanto o uso de funções pode facilitar o desenvolvimento de programas de diversas maneiras.

Em primeiro lugar temos as vantagens do reuso de código desenvolvido por outros programadores. As funções de entrada e saída são o exemplo mais direto deste reuso. Em `C` não existem estes tipos de comandos como na maioria das linguagens. Programas escritos em `C` usam funções de entrada e saída escritas e testadas por outros programadores. Este reuso de código apresenta várias vantagens. Primeiro, diminui o tempo de desenvolvimento do programas. Em segundo lugar, como estas funções foram testadas por diversos usuários, a quantidade de erros é bastante reduzida. Estes fatores contribuem para a redução dos custos de desenvolvimento dos projetos.

Uma outra vantagem do uso de funções e a maior facilidade na divisão do trabalho necessário para construir um aplicativo. Funções podem ser desenvolvidas por programadores trabalhando independentemente. Para isto basta que alguns acordos sejam feitos entre os programadores que irão programar a função e os que irão usá-las. Estes acordos precisam definir que parâmetros a função irá receber, que resultados irá fornecer e que operações ela deve realizar sobre estes parâmetros para obter os resultados necessários. Esta divisão do trabalho concorre para acelerar o desenvolvimento dos programas e na redução dos custos deste desenvolvimento.

A divisão de um programa em funções também permite que os testes do sistema completo sejam feitos mais facilmente e com mais garantia de correção.

Os programadores podem testar suas funções separadamente em testes menos complexos, já que as funções normalmente são simples e têm requisitos menos complicados de serem avaliados. Isto permite que muitos erros do sistema completo possam ser retirados antes que ele esteja completo. Normalmente testar um programa complexo requer testes complexos.

Mesmo quando um programa é desenvolvido por um único programador a sua divisão em funções traz vantagens, por dividir um trabalho complexo em diversas fatias menores permitindo ao programador se concentrar a cada vez em problemas mais simples.

8.2 Forma Geral

A forma geral de uma função em `C` é a seguinte:

```
tipo nome (tipo nome1, tipo nome2, ..., tipo nomeN )
{
    declaração das variáveis
    corpo da função
}
```

Uma função recebe uma lista de argumentos (`nome1, nome2, ..., nomeN`), executa comandos com estes argumentos e pode retornar ou não um resultado para a função que chamou esta função. A lista de argumentos, também chamados de parâmetros, é uma lista, separada por vírgulas, de variáveis com seus tipos associados. Não é possível usar uma única definição de tipo para várias variáveis. A lista de argumentos pode ser vazia, ou seja, a função não recebe nenhum argumento. O nome da função pode ser qualquer identificador válido. O `tipo` que aparece antes do nome da função especifica o tipo do resultado que será devolvido ao final da execução da função. Caso nenhum tipo seja especificado o compilador assume que um tipo inteiro é retornado. O tipo **void** pode ser usado para declarar funções que não retornam valor algum.

Há basicamente duas maneiras de terminar a execução de uma função. Normalmente usa-se o comando **return** para retornar o resultado da função. Portanto, quando o comando

```
return expressão;
```

for executado, o valor da `expressão` é devolvido para a função que chamou. Quando não há valor para retornar o comando **return** não precisa ser usado e a função termina quando a chave que indica o término do corpo da função é atingido.

Os parâmetros são valores que a função recebe para realizar as tarefas para as quais foi programada. Por exemplo, uma função que calcule a raiz quadrada de um número do tipo **float**, deve declarar como parâmetro uma variável deste tipo para receber o valor.

É importante notar que diferentemente de declarações de variáveis onde podemos associar vários nomes de variáveis a uma declaração como em

```
int a, dia, mes, i;
```

na lista de parâmetros é necessário associar um tipo a cada variável como no exemplo a seguir:

```
float media (float n1, float n2, float n3);
```

Neste exemplo, uma função chamada `media` que é do tipo **float**, isto é retorna um resultado **float**, recebe três argumentos (`n1, n2, n3`) também do tipo **float**.

Um ponto importante é como usar a função. Suponha que uma determinada função, `A`, deseje usar uma outra função, `B`. A função `A` deve colocar no local desejado o nome da função (`B`) e a lista de valores que deseja passar. Por exemplo, um programador que deseje usar a função `media` em seu programa para calcular a média de três valores, `nota1, nota2` e `nota3`, deve escrever no local onde quer que a média seja calculada o seguinte comando:

```
resultado = media(nota1, nota2, nota3);
```

onde `resultado` é a variável que vai receber a média calculada.

É importante notar que o nome da função pode aparecer em qualquer lugar onde o nome de uma variável apareceria. Além disso os tipos e o número de parâmetros que aparecem na declaração da função e na sua chamada devem estar na mesma ordem e ser tipos equivalentes. Se os tipos são incompatíveis, o compilador não gera um erro, mas podem ser gerados avisos na compilação e resultados estranhos.

Outro ponto importante a ser notado e que será detalhado mais adiante é que os nomes das variáveis nos programas que usam a função `media` podem ser diferentes dos nomes usados na definição da função.

8.3 Protótipos de Funções

O padrão ANSI estendeu a declaração da função para permitir que o compilador faça uma verificação mais rígida da compatibilidade entre os tipos que a função espera receber e àqueles que são fornecidos. Protótipos de funções ajudam a detectar erros antes que eles ocorram, impedindo que funções sejam chamadas com argumentos inconsistentes. A forma geral de definição de um protótipo é a seguinte:

```
tipo nome (tipo nome1, tipo nome2, ..., tipo nomeN);
```

O exemplo 8.1 mostra a declaração de uma função e seu protótipo.

Também é possível declarar um protótipo sem dar os nomes das variáveis somente os tipos das funções. No exemplo 8.1 o protótipo da função `soma` pode ser declarada da seguinte maneira

```
int soma (int, int)
```

Listing 8.1: Exemplo de protótipos.

```
#include<stdio.h>

/* Prototipo da funcao */
int soma (int a, int b);

/* Funcao Principal */
int main()
{
     int a=5, b=9;
     printf("%d\n", soma(a,b));
      return 0;
}

/* Definicao da funcao */
int soma(int a, int b)
{
    return a+b;
}
```

8.4 Escopo de Variáveis

Variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou pode ocorrer que variáveis precisem ser acessíveis à diversas funções diferentes. Por esta razão, temos que apresentar os locais onde as variáveis de um programa podem ser definidas e a partir destes locais podermos inferir onde elas estarão disponíveis. As variáveis podem ser declaradas basicamente em três lugares:

- dentro de funções,
- fora de todas as funções,
- na lista de parâmetros das funções.

As variáveis definidas dentros das funções são chamadas de variáveis locais, as que aparecem fora de todas as funções chamamos de variáveis globais e aquelas que aparecem na lista de parâmetros são os parâmetros formais. É importante notar que em `C` todas as funções estão no mesmo nível, por esta razão não é possível definir uma função dentro de outra função.

8.4.1 Variáveis Locais

As variáveis locais são aquelas declaradas dentro de uma função ou um bloco de comandos. Elas passam a existir quando do início da execução do bloco de

comandos ou função onde foram definidas e são destruídas ao final da execução do bloco. Uma variável local só pode ser referenciada, ou seja usada, dentro da função (ou bloco) onde foi declarada. Outro ponto muito importante é que como as variáveis locais deixam de existir ao final da execução da função (ou bloco), elas são invisíveis para outras funções do mesmo programa. O código que define uma função e os seus dados são particulares da função (do bloco).

No programa 8.2 podemos ver o uso de variáveis locais. A variável `i` é definida em cada uma das funções do programa (`pares`, `impares`). Os valores da variável não podem ser acessados a partir de nenhuma outra função e as modificações feitas dentro da função somente valem enquanto a função está sendo executada.

Listing 8.2: Exemplos de variáveis locais.

```
#include <stdio.h>

void pares(void)
{
     int i;
     for (i = 2; i <= 10; i += 2)
     {
          printf("%d: ", i);
     }
}
void impares(void)
{
     int i;
     for (i = 3; i <= 11; i += 2)
     {
          printf("%d: ", i);
     }
}
int main(int argc, char *argv[])
{
     pares();
     printf("\n");
     impares();

     return 0;
}
```

Alguns autores usam o termo *variáveis automáticas* para se referir as variáveis locais. Em `C` existe a palavra chave **auto** que pode ser usada para declarar que variáveis pertencem à classe de armazenamento padrão. No entanto, como todas as variáveis locais são por definição automáticas raramente se usa esta palavra chave.

Observe que um bloco de comandos se inicia em um ‘‘{’’ e termina em

um ‘‘}’’. O bloco de comandos, dentro do qual mais comumente se define uma variável é a função. Todas as variáveis que serão usadas dentro de um bloco de comandos precisam ser declaradas antes do primeiro comando do bloco. Declarações de variáveis, incluindo sua inicialização, podem vir logo após o abre chaves que inícia um bloco de comandos, não somente o que começa uma função. O programa 8.3 ilustra este tipo de declaração.

Listing 8.3: Definição de variável dentro de um bloco.

```
#include<stdio.h>
int main()
{
    int i;
    for (i=0; i<10; i++)
     {
         int t = 2;
         printf("%d\n", i*t);
    }
     return 0;
}
```

Existem algumas vantagens em se declarar variáveis dentro de blocos. Como as variáveis somente existem durante a execução do bloco, o programa pode ocupar menos espaço de memória. Por exemplo, se a execução do bloco for condicional a variável pode nem ser alocada. Outra vantagem é que como a variável somente existe dentro do bloco, pode-se controlar melhor o uso da variável, evitando erros de uso indevido da variável.

8.5 Variáveis Globais

As variáveis globais são definidas fora de qualquer função e são portanto disponíveis para qualquer função. Este tipo de variável pode servir como uma canal de comunicação entre funções, uma maneira de transferir valores entre elas. Por exemplo, se duas funções tem de partilhar dados, mais uma não chama a outra, uma variável global tem de ser usada.

O programa 8.4 ilustra este tipo de declaração. O resultado da execução deste programa é o seguinte:

```
Funcao soma1:  i = 1
Funcao sub1:  i = 9
Funcao main:  i = 1
```

Observe que a variável global `i` recebe o valor `0` no início da função `main`. A função `soma1` ao executar um comando que aumenta o valor de `i` em uma unidade está aumentando a variável global. Em seguida vemos que a função `sub1` define uma variável local também chamada `i` e, portanto, a alteração feita por esta

função somente modifica esta variável. Finalmente, a função `main` imprime o valor final da variável global.

Listing 8.4: Definição de variável global.

```
#include <stdio.h>

int i;

void soma1(void)
{
     i += 1;
     printf("Funcao soma1: i = %d\n", i);
}
void sub1(void)
{
     int i = 10;
     i -= 1;
     printf("Funcao sub1: i = %d\n", i);
}
int main(int argc, char *argv[])
{
     i = 0;
     soma1();
     sub1();
     printf("Funcao main: i = %d\n", i);
     return 0;
}
```

8.6 Parâmetros Formais

As variáveis que aparecem na lista de parâmetros da função são chamadas de parâmetros formais da função. Eles são criados no início da execução da função e destruídos no final. Parâmetros são valores que as funções recebem da função que a chamou. Portanto, os parâmetros permitem que uma função passe valores para outra. Normalmente os parâmetros são inicializados durante a chamada da função, pois para isto foram criados. No entanto, as variáveis que atuam como parâmetros são iguais a todas as outras e podem ser modificadas, operadas, etc, sem nenhuma restrição. Parâmetros podem ser passados para funções de duas maneiras: passagem por valor ou passagem por referência.

8.6.1 Passagem de Parâmetros por Valor

Na passagem por valor uma cópia do valor do argumento é passado para a função. Neste caso a função que recebe este valor, ao fazer modificações no

parâmetro, não estará alterando o valor original que somente existe na função que chamou.

O exemplo 8.5 mostra o uso de passagem de parâmetros por valor. Observe que as função `Eleva` recebe dois parâmetros (`a`,`b`) e opera usando os valores recebidos. O resultado da função é retornado por meio da variável local `res`.

Listing 8.5: Exemplo de passagem por valor.

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

float Eleva(float a, int b)
{
     float res = 1.0;

    for ( ; b>0; b--) res *= a;
    return res;
}

int main()
{
     float numero;
     int potencia;
     char linha[80];

     puts("Entre com um numero");
     gets(linha); numero = atof(linha);
     puts("Entre com a potencia");
     gets(linha); potencia = atoi(linha);
     printf("\n%f Elevado a %d e igual a %f\n",
               numero, potencia, Eleva(numero, potencia));

     return 0;
}
```

Para ilustrar o fato de que somente o valor é passado vamos usar o exemplo 8.6. Neste programa as variáveis `a` e `b` recebem os valores `10` e `20` respectivamente. Na função `trocar` estes valores são recebidos e são trocados localmente. Após o retorno da função, o programa imprime os valores originais das variáveis, já que estes não sofreram nenhuma alteração. O resultado da execução deste programa é o seguinte:

```
a = 10, b = 20
```

8.6.2 Passagem de Parâmetros por Referência

Na passagem por referência o que é passado para a função é o endereço do parâmetro e, portanto, a função que recebe pode, através do endereço, modificar

Listing 8.6: Uso indevido de variáveis locais.

```
#include <stdio.h>
void trocar(int a, int b)
{
     int temp;
     temp = a; a = b; b = temp;
}

int main(int argc, char *argv[])
{
     int a = 10, b = 20;
     trocar(a, b);
     printf("a = %d, b = %d\n", a, b);

     return 0;
}
```

o valor do argumento diretamente na função que chamou. Para a passagem de parâmetros por referência é necessário o uso de ponteiros. Este assunto será discutido no próximo capítulo e portanto, por agora, usaremos somente funções com passagem por valor.

8.6.3 Passagem de Vetores e Matrizes

Matrizes são um caso especial e excessão a regra que parâmetros são passados sempre por valor. Como veremos mais adiante, o nome de um vetor corresponde ao endereço do primeiro elemento do array, Quando um vetor é passado como parâmetro, apenas o endereço do primeiro elemento é passado.

Existem basicamente três maneiras de declarar um vetor como um parâmetro de uma função. Na primeira ele é declarado como tem sido apresentado em todos os exemplos até agora. O exemplo 8.7 mostra um programa que usa uma função para descobrir quantas vezes um caracter ocorre em um vetor. Observe que na definição da função a dimensão do vetor foi declarada explicitamente.

Uma outra maneira, leva em conta que apenas o endereço do vetor é passado. Neste modo o parâmetro é declarado como um vetor sem dimensão. Isto é perfeitamente possível porque a função somente precisa receber o endereço onde se encontra o vetor. Além disso `C` não confere limites de vetores e portanto a função precisa do endereço inicial do vetor e uma maneira de descobrir o final do vetor. Esta maneira pode ser, por exemplo, uma constante, ou o caracter `'\0'` em um vetor de caracteres. O exemplo 8.8 mostra este modo de passar vetores com um programa que inverte o conteúdo de um vetor.

A terceira maneira de passagem de parâmetros implica no uso de ponteiros, o que somente iremos ver no próximo capítulo.

Listing 8.7: Passagem de vetor com dimensões.

```
#include<stdio.h>
#include<conio.h>
#define DIM 80

char conta (char v[], char c);

int main()
{
     char c, linha[DIM];
     int maiusculas[26], minusculas[26];

     puts("Entre com uma linha");
     gets (linha);
     for (c = 'a'; c <= 'z'; c++)
         minusculas[c-'a'] = conta(linha, c);
     for (c = 'A'; c <= 'Z'; c++)
         maiusculas[c-'A'] = conta(linha, c);
     for (c = 'a'; c <= 'z'; c++)
         if (minusculas[c-'a'])
            printf("%c apareceu %d vezes\n", c, minusculas[c-'a']);
     for (c = 'A'; c <= 'Z'; c++)
         if (maiusculas[c-'A'])
            printf("%c apareceu %d vezes\n", c, maiusculas[c-'A']);
     return 0;
}

char conta (char v[DIM], char c)
{
     int i=0, vezes=0;

     while (v[i] != '\0')
           if (v[i++] == c) vezes++;
     return vezes;
}
```

Listing 8.8: Passagem de vetores sem dimensões.

```
#include<stdio.h>
#include<conio.h>
#define DIM 6
void Le_vetor (int v[], int tam);
void Imprime_vetor (int v[], int tam);
void Inverte_vetor (int v[], int tam);

int main()
{
     int v[DIM];

     Le_vetor(v, DIM);
     Imprime_vetor (v, DIM);
     Inverte_vetor (v, DIM);
     Imprime_vetor (v, DIM);
     return 0;
}

void Le_vetor (int v[], int tam)
{
     int i;

     for ( i = 0; i < tam; i++)
     {
          printf("%d = ? ", i);
          scanf("%d", &v[i]);
     }
}

void Imprime_vetor (int v[], int tam)
{
     int i;

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

void Inverte_vetor (int v[], int tam){
     int i, temp;

     for (i = 0; i < tam/2; i++){
         temp = v[i];
         v[i] = v[tam-i-1];
         v[tam-i-1] = temp;
     }
}
```

8.7 O Comando `return`

O comando **return** é usado para retornar o valor calculado para a função que chamou. Qualquer expressão pode aparecer no comando, que tem a a seguinte forma geral:

return `expressão`

A função que chamou é livre para ignorar o valor retornado. Além disso a função pode não conter o comando e portanto nenhum valor é retornado e neste caso a função termina quando o último comando da função é executado. Quando o comando **return** não existe o valor de retorno é considerado indefinido. As funções que não retornam valores devem ser declaradas como do tipo **void**. É importante observar que funções que são declaradas com um tipo válido podem ser incluídas em qualquer expressão válida em `C`.

8.8 Recursão

Funções em `C` podem ser usadas recursivamente, isto é uma função pode chamar a si mesmo. É como se procurássemos no dicionário a definição da palavra recursão e encontrássemos o seguinte texto:

recursão: s.f. Veja a definição em recursão

Um exemplo simples de função que pode ser escrita com chamadas recursivas é o fatorial de um número inteiro. O fatorial de um número, sem recursão, é definido como

$$n! = n * (n-1) * (n-2) * \cdots * 2 * 1$$

A partir desta definição podemos escrever a função fatorial como

```
unsigned long int fat (unsigned long int num)
{
    unsigned long int fato=1, i;

    for (i=num; i>1; i--) fato = fato * i;

    return fato;
}
```

Alternativamente, o fatorial pode ser defindo como o produto deste número pelo fatorial de seu predecessor, ou seja

$$n! = n * (n-1)!$$

Deste modo podemos escrever uma função recursiva em que cada chamada da função que calcula o fatorial chama a própria função fatorial. O exemplo, mostrado a seguir, mostra como a função pode ser escrita recursivamente.

```
unsigned long int fat (unsigned long int num)
{
     if (num == 0)
       return 1;
    else
        return num * fat (num-1);
}
```

Quando a função fatorial recursiva é chamada, primeiro é verificado se o número recebido como parâmetro vale 0 e neste caso a função retorna o valor 1. No caso contrário ela devolve o valor da expressão `num * fat(num-1)`, ou seja o produto do número pelo valor do fatorial do número predecessor. Ou seja para retornar um valor a função precisa chamar ela mesma passando como parâmetro o valor do número menos 1. Este processo continua se repetindo até que o valor passado é igual a 0, o que indica o final da recursão. Neste ponto o processo se reverte e as chamadas começam a ser respondidas.

Um ponto importante é que toda função recursiva deve prever cuidadosamente como o processo de recursão deve ser interrompido. No caso da função `fat` o processo é interrompido quando o valor do número passado como parâmetro vale 0. Se este teste não tivesse sido incluído na função as chamadas continuariam indefinidamente com valores negativos cada vez menores sendo passados como parâmetro.

Quando uma função chama a si mesmo recursivamente ela recebe um conjunto novo de variáveis na pilha que é usada para transferência de valores entre funções. É importante notar que recursão não traz obrigatoriamente economia de memória porque os valores sendo processados tem de ser mantidos em pilhas. Nem será mais rápido, e as vezes pode ser até mais lento porque temos o custo de chamada as funções. As principais vantagens da recursão são códigos mais compactos e provalvemente mais fáceis de serem lidos.

Um outro exemplo simples de função que pode ser resolvida por recursão é x^n, assumindo que $n \geq 0$. Esta função pode escrita na sua forma recursiva como

$$x^n = x * x^{(n-1)}$$

que nos leva a escrever a função da maneira mostrada no exemplo 8.9. Na função consideramos que x é do tipo **float**.

8.9 Argumentos - `argc` e `argv`

A função `main` como todas as funções podem ter parâmetros. Como a função `main` é sempre a primeira a ser executada, os parâmetros que ela recebe são fornecidos pela linha de comando ou pelo programa que iniciou a sua execução. No caso da função `main` são usados dois argumentos especiais **int** `argc` e **char** `**argv`.

O primeiro argumento, `argc`, é uma variável inteira que indica quantos argumentos foram fornecidos para a função. Observar que `argc` vale sempre pelo

Listing 8.9: Função recursiva para calcular x^n.

```
float Elevar(float x, int n)
{
     if (n <= 1)
     {
          return x;
     }
     else
     {
          return x * Elevar(x, b-1);
     }
}
```

menos 1, porque o nome do programa é sempre o primeiro argumento fornecido ao programa. A partir do segundo argumento em diante é que aparecem os outros argumentos.

O outro parâmetro é um vetor de cadeias de caracteres, e portanto, caso sejam fornecidos números, estes devem ser convertidos para o formato requerido. Cada um dos argumentos do programa é um elemento deste vetor. A primeira linha da função `main` pode ter a seguinte forma

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

O programa exemplo 8.10 calcula o fatorial dos números fornecidos como argumentos.

Os nomes `argc` e `argv` são comumente usados mas o programador é livre para escolher os nomes mais apropriados.

Exercícios

8.1: Escrever um programa que declare, inicialize e imprima um vetor de 10 inteiros. O vetor deve conter os 10 primeiros múltiplos de 5. A inicialização do vetor e a sua impressão devem ser feitas por funções.

8.2: Escreva um programa para declarar um vetor de caracteres de tamanho 26 e imprimir o seu conteúdo. O vetor deve ser inicializado com as letras minúsculas do alfabeto. A inicialização do vetor e a sua impressão devem ser feitas por funções.

8.3: Escreva um programa que armazene em uma matriz três nomes de pessoas e em seguida os imprima. Assuma que o tamanho máximo de cada nome é 40 caracteres. Neste programa a leitura dos nomes dever ser feita por uma função e a impressão dos nomes por outra.

Listing 8.10: Uso de argc e argv.

```
#include <stdio.h>
#include <stdlib.h>
unsigned long int fat (unsigned long int num)
{
    if (num == 0)
          return 1;
    else
          return num * fat (num-1);
 }
int main(int argc, char *argv[])
{
     unsigned long int numero, fatorial, i;

     if (argc < 2)
     {
          printf("Para rodar: %s num1 num2 ... .\n", argv[0]);
          return 1;
     }
     for (i=1; i<argc; i++)
     {
          numero = (unsigned long int) (atoi(argv[i]));
          fatorial = fat(numero);
          printf("O fatorial de %lu vale %lu.\n", numero, fatorial);
     }
      return 0;
}
```

8.4: Escreva um programa que imprima o codigo ASCII de todos os caracteres, das seguintes maneiras:

1. caractere a caractere, a escolha do usuario;
2. a tabela inteira, a partir de um determinado valor decimal.

Cada item deste exercício deve corresponder a uma função.

8.5: Escreva um programa que crie uma tabela de temperaturas Celsius - Fahrenheit. O programa deve usar uma função que converta de Celsius para Fahrenheit. A tabela deve iniciar na temperatura 0 graus Celsius e terminar na temperatura 100 graus Celsius.

8.6: Escreva um programa, usando funções, que gere um vetor a partir de uma matriz. Cada elemento do vetor é igual a soma dos elementos de uma das linhas da matriz. Considere que a matriz tenha tamanho 10 por 10.

8.7: Escreva um programa usando recursividade para gerar a seqüência do Fibonacci. A seqüência de Fibonacci é definida como:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

O programa deve ler um numero e exibir o valor dele na sequencia de Fibonacci. Exemplos de entrada e saída do programa são mostrados abaixo.

```
Entre com um numero inteiro:  0
Fibonacci (0) = 0
Entre com um numero inteiro:  1
Fibonacci (1) = 1
Entre com um numero inteiro:  2
Fibonacci (2) = 1
Entre com um numero inteiro:  3
Fibonacci (3) = 2
Entre com um numero inteiro:  6
Fibonacci (6) = 8
```

Capítulo 9

Ponteiros

9.1 Introdução

Ponteiros são usados em situações em que é necessário conhecer o endereço onde está armazenada a variável e não o seu conteúdo. Um ponteiro é uma variável que contém um endereço de uma posição de memória e não o conteúdo da posição. A memória de um computador pode ser vista como uma sequência de bytes cada um com seu próprio e único endereço. Não há dois bytes com o mesmo endereço. O primeiro endereço é sempre 0 e o último geralmente é uma potência de 2. Por exemplo um computador com memória igual a 512 Mbytes tem 512x1024x1024 bytes. A Figura 9.1 mostra o mapa de um trecho de memória que contém duas variáveis inteiras (`num, res`) ocupando 4 bytes cada uma e mais um ponteiro (`pint`), que também ocupa 4 bytes. Observar que os endereços estão pulando de quatro em quatro bytes devido ao espaço que cada um destas variáveis ocupa.

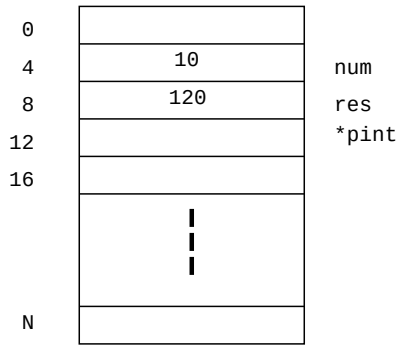

Figura 9.1: Mapa de memória com duas variáveis e ponteiro.

Ponteiros são importantes, por exemplo, quando se deseja que uma função retorne mais de um valor. Neste caso uma solução é a função receber como argumentos não os valores dos parâmetros mas sim ponteiros que apontem para seus endereços. Assim esta função pode modificar diretamente os conteúdos destas variáveis, que após o fim da função estarão disponíveis para a função que chamou. Neste caso os argumentos podem funcionar como entrada e saída de dados da função.

Uma outra aplicação importante de ponteiros é apontar para áreas de memória que devem ser gerenciadas durante a execução do programa. Com ponteiros, é possível reservar as posições de memória necessárias para armazenamento destas áreas somente quando for necessário e não quando as variáveis são declaradas. Neste esquema o programador pode reservar o número exato de posições que o programa requer. A Figura 9.2 ilustra como um ponteiro faz referência para uma área de memória. Na figura a variável ponteiro `pi` aponta para a área de memória que contém um vetor de 10 inteiros. Com ponteiros, o programador precisa, no início, definir a variável ponteiro e seu tipo. Durante a execução do programa, após descobrir o tamanho do vetor, reserva a área necessária para guardar os dados. Observe a diferença do que ocorre quando se usa vetores de tamanho fixo. Neste caso a definição do tamanho do vetor é dada na declaração do vetor e é mantida até o final da execução do programa.

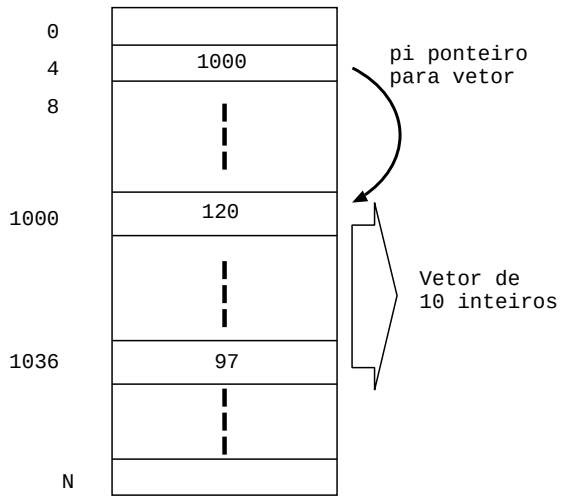

Figura 9.2: Ponteiro apontando para área de memória contendo vetor.

9.2 Operações com Ponteiros

9.2.1 Declaração de Ponteiros

Antes de serem usados os ponteiros, como as variáveis, precisam ser declarados. A forma geral da declaração de um ponteiro é a seguinte:

```
tipo *nome;
```

Onde **tipo** é qualquer tipo válido em `C` e `nome` é o nome da variável ponteiro. Por exemplo:

```
int *res; /* ponteiro para inteiro */
float *div; /* ponteiro para ponto flutuante */
```

Como as variáveis, os ponteiros devem ser inicializados antes de serem usados. Esta inicialização pode ser feita na declaração ou através de uma atribuição. Após a declaração o que temos é um espaço na memória reservado para armazenamento de endereços. O valor inicial da memória é indefinido como acontece com variáveis. A Figura 9.3 ilustra esta situação. Um ponteiro pode ser inicializado com um endereço ou com o valor `NULL`. O valor `NULL`, que é equivalente a `0`, é uma constante definida no arquivo `<stdio.h>` e significa que o ponteiro não aponta para lugar nenhum. A atribuição de inteiros a ponteiros não faz sentido a não ser em aplicações muito especiais e o único valor inteiro que se pode atribuir a um ponteiro é o `0`. Esta tipo de atribuição não faz sentido porque na maioria das aplicações é o sistema operacional que aloca e gerencia a posição dos programas na memória e, portanto, o usuário não tem controle sobre estes endereços.

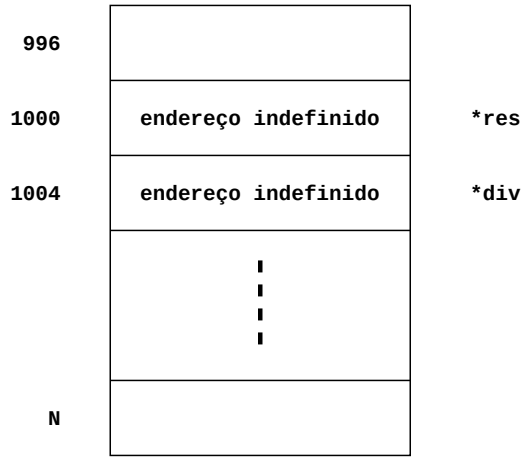

Figura 9.3: Declaração de ponteiros.

9.2.2 Os Operadores Especiais para Ponteiros

Existem dois operadores especiais para ponteiros: `*` e `&`. Os dois operadores são unários, isto é requerem somente um operando. O operador `&` devolve o endereço de memória do seu operando. Considere a Figura 9.1. Após a execução da instrução

```
pint = &num; /*o endereco de num e carregado em pint */
```

a variável ponteiro `pint` termina com o valor `4`, como está mostrado na Figura 9.4. Lembre-se que o valor `4` não tem sentido prático na maioria das aplicações. O fato importante é que o ponteiro `pint` passou a apontar para a variável `num`

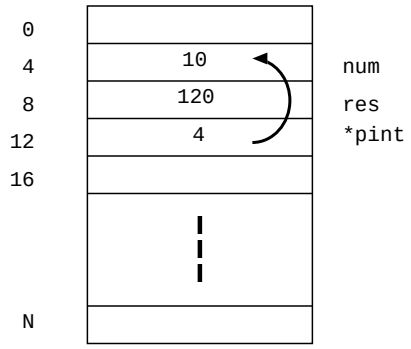

Figura 9.4: Atribuição de endereço de uma variável a um ponteiro.

O operador `*` é o complemento de `&`. O operador `*` devolve o valor da variável localizada no endereço apontado pelo ponteiro. Por exemplo, considere que o comando `res = *pint;` foi executado logo após `pint = &num;`. Isto significa que a variável `res` recebe o valor apontado por `pint`, ou seja a variável `res` recebe o valor `10`, como está mostrado na Figura 9.5.

Estes operadores não devem ser confundidos com os já estudados em capítulos anteriores. O operador `*` para ponteiros não tem nada a ver com o operador multiplicação `*`. O operador ponteiro `*` é unário e, como o operador `&`, tem precedência maior que do que todos os operadores aritméticos.

9.2.3 Atribuição de Ponteiros

Da mesma maneira que ocorre com uma variável comum, o conteúdo de um ponteiro pode ser passado para outro ponteiro do mesmo tipo. Por exemplo, uma variável ponteiro declarada como apontador de dados inteiros deve sempre apontar para dados deste tipo. Observar que em `C` é possível atribuir qualquer endereço a uma variável ponteiro. Deste modo é possível atribuir o endereço de uma variável do tipo **float** a um ponteiro do tipo **int**. No entanto, o programa não irá funcionar da maneira correta. O programa 9.1 mostra exemplos de

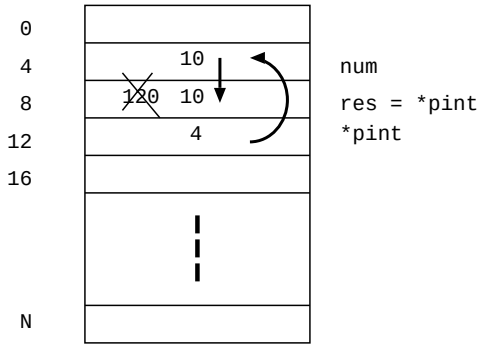

Figura 9.5: Uso de um ponteiro para copiar valor de uma variável.

atribuições de ponteiros. Neste exemplo o endereço do terceiro elemento do vetor `v` é carregado em `p1` e o endereço da variável `i` é carregado em `p2`. A Figura 9.6 a situação da memória após estas operações. Além disso no final o endereço apontado por `p1` é carregado em `p2`. Os comandos `printf` imprimem os valores apontados pelos ponteiros respectivos, mostrando os seguintes valores:

```
30
100
30
```

Listing 9.1: Exemplo de atribuição de ponteiros.

```
#include <stdio.h>
int main(void)
{
     int vetor[] = { 10, 20, 30, 40, 50 };

     int *p1, *p2;
     int i = 100;

     p1 = &vetor[2];
     printf("%d\n", *p1);
     p2 = &i;
     printf("%d\n", *p2);
     p2 = p1;
     printf("%d\n", *p2);
     return 0;
}
```

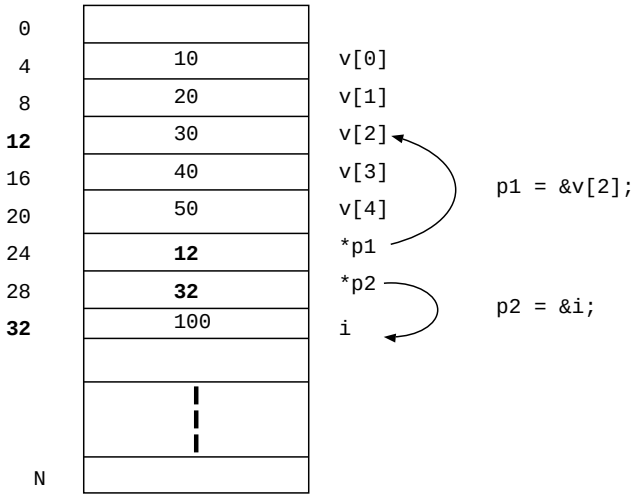

Figura 9.6: Exemplos de atribuições de ponteiros.

9.2.4 Incrementando e Decrementando Ponteiros

O exemplo 9.2 mostra que operações de incremento e decremento podem ser aplicadas em operandos. O primeiro `printf` imprime `30`, que é o elemento de índice igual a `2` no vetor `vetor`. Após o incremento do ponteiro o segundo `printf` imprime `40` e o mesmo acontece com o terceiro `printf` que imprime `50`.

Listing 9.2: Exemplos de operações com ponteiros.

```
int main(void)
{
     int vetor[] = { 10, 20, 30, 40, 50 };
     int *p1;

     p1 = &vetor[2];
     printf("%d\n", *p1);
     p1++;
     printf("%d\n", *p1);
     p1 = p1 + 1;
     printf("%d\n", *p1);
     return 0;
}
```

Pode parecer estranho que um ponteiro para um número inteiro, que é armazenado em quatro bytes, seja incrementado por um e passe para apontar para o próximo número inteiro. A primeira vista, para que passasse a apontar para o próximo endereço, seria necessário aumentar o endereço em quatro. Ou seja,

sempre que um ponteiro é incrementado (decrementado) ele passa a apontar para a posição do elemento seguinte (anterior). O compilador interpreta o comando `p1++` como: *passe a apontar para o próximo número inteiro* e, portanto, aumenta o endereço do número de bytes correto. Este ajuste é feito de acordo com o tipo do operando que o ponteiro está apontando. Do mesmo modo, somar três a um ponteiro faz com que ele passe apontar para o terceiro elemento após o atual. Portanto, um incremento em um ponteiro que aponta para um valor que é armazenado em `n` bytes faz que `n` seja somado ao endereço. É então possível somar-se e subtrair-se inteiros de ponteiros. A operação abaixo faz com que o ponteiro `p` passe a apontar para o terceiro elemento após o atual.

```
p = p + 3;
```

Também é possível usar-se o seguinte comando

```
*(p+1)=10;
```

Este comando armazena o valor 10 na posição seguinte àquela apontada por `p`. A operação é realizada nos seguintes passos:

1. A expressão `p+1` é calculada e o seu resultado aponta para o próximo endereço de dado inteiro;

2. A expressão do lado direito do sinal de atribuição, é calculada e fornece como resultado o valor `10`;

3. Este resultado é atribuído ao endereço calculado no primeiro passo.

A diferença entre ponteiros fornece quantos elementos do tipo do ponteiro existem entre os dois ponteiros. No exemplo 9.3 é impresso o valor 2.

Listing 9.3: Exemplo de subtração de ponteiros.

```
#include <stdio.h>
int main(void)
{
     float vetor[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
     float *p1, *p2;

     p1 = &vetor[2]; /* endereco do terceiro elemento */
     p2 = vetor;         /* endereco do primeiro elemento */
     printf("Diferenca entre ponteiros %d\n", p1-p2);
     return 0;
}
```

Não é possível multiplicar ou dividir ponteiros, e não se pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.

9.2.5 Comparação de Ponteiros

É possível comparar ponteiros em uma expressão relacional. No entanto, só é possível comparar ponteiros de mesmo tipo. O Programa 9.4 ilustra um exemplo deste tipo de operação.

Listing 9.4: Exemplo de comparação de ponteiros.

```
#include<stdio.h>
int main (void)
{
     char *c, *v, a, b;

     scanf("%c %c", &a, &b);
     c = &a;
     v = &b;

     if (c == v)
          printf("As variáveis estao na mesma posicao.");
     else
          printf("As variaveis nao estao na mesma posicao.");

     return 0;
}
```

9.3 Ponteiros e Vetores

Ponteiros e Vetores estão fortemente relacionados na linguagem `C`. O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor. A declaração **int** `vetor[100]` cria um vetor de inteiros de 100 posições e permite que algumas operações com ponteiros possam ser realizadas com a variável `vetor`.

No entanto, existe uma diferença fundamental entre declarar um conjunto de dados como um vetor ou através de um ponteiro. Na declaração de vetor, o compilador automaticamente reserva um bloco de memória para que o vetor seja armazenado. Quando apenas um ponteiro é declarado a única coisa que o compilador faz é alocar um ponteiro para apontar para a memória, sem que espaço seja reservado. O nome de um vetor é chamado de ponteiro constante e, portanto, não pode ter o seu valor alterado. O nome de um ponteiro constante não pode aparecer em expressões no lado esquerdo do sinal de igual, ou seja, não pode receber valores diferentes do valor inicial atribuído na declaração da variável. Assim, os comandos que alteram o ponteiro `list`, mostrados no exemplo 9.5, não são válidos:

O conteúdo de vetores pode ser acessado usando-se o operador `*` para obter o conteúdo do vetor. O Programa 9.6 mostra a notação que usa índices para

Listing 9.5: Exemplo de alterações inválidas sobre ponteiros.

```
int list[5], i;

/* O ponteiro list nao pode ser modificado
       recebendo o endereco de i */
list = &i
/* O ponteiro list nao pode ser incrementado */
list++;
```

buscar o elemento de um vetor e o uso do operador * de ponteiros. Neste programa o conteúdo do vetor é impresso usando-se estas duas notações.

Listing 9.6: Exemplo de notações de vetores.

```
#include<stdio.h>
int main(void)
{
     float v[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
     int i;
     for (i = 0; i < 7; i++)
          printf("%.1f ", v[i]);
     printf("\n");
     for (i = 0; i < 7; i++)
          printf("%.1f ", *(v+i));
     return 0;
}
```

Para percorrer um vetor além da maneira mostrada no programa 9.6 é possível usar um ponteiro variável como ilustrado no Programa 9.7. Observe como o ponteiro `p` recebe seu valor inicial e a maneira como ele é incrementado.

9.4 Ponteiros e Cadeias de Caracteres

Uma cadeia de caracteres constante é escrita como no exemplo:

```
"Esta e uma cadeia de caracteres."
```

Até agora um dos usos mais comuns de cadeias de caracteres constantes tem sido na função `printf`, como no exemplo abaixo

```
printf("Acabou o programa.\n");
```

Quando uma cadeia de caracteres como esta é enviada para a função, o que é passado é o ponteiro para a cadeia. É possível então carregar o endereço da cadeia em um ponteiro do tipo **char**, como no exemplo 9.8. Neste programa é contado o número de caracteres de uma cadeia. Observe o ponteiro `*(s+tam++)` apontando caracter a caracter.

Listing 9.7: Exemplo de ponteiro variável.

```
#include <stdio.h>
int main(void)
{
     float v[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
     int i;
     float *p;

     for (i = 0; i < 7; i++) printf("%.1f ", v[i]);
     printf("\n");
     for (i = 0; i < 7; i++) printf("%.1f ", *(v+i));
     printf("\n");
     for (i = 0, p = v; i < 7; i++, p++) printf("%.1f ", *p);
     return 0;
}
```

Listing 9.8: Exemplo de ponteiro para cadeia de caracteres.

```
#include <stdio.h>
int main(void)
{
    char *s, *lista="1234567890";
     int tam=0;

     s = lista;
     while(*(s + tam++) != '\0');
     tam--;
     printf("O tamanho do string \"%s\" e %d caracteres.\n",
                 lista, tam);
     return 0;
}
```

Um outro exemplo (Programa 9.9) mostra uma função que copia um cadeia de caracteres para outra.

9.5 Alocação Dinâmica de Memória

O uso de ponteiros e vetores exige que após a definição da variável ponteiro uma área de memória deve ser reservada para armazenar os dados do vetor. Para obter esta área o programa deve usar funções existentes na biblioteca `stdlib`. Estas funções pedem ao sistema operacional para separar pedaços da memória e devolvem ao programa que pediu o endereço inicial deste local. As funções básicas de alocação de memória que iremos discutir são:

void `*malloc(size_t size);` Reserva espaço na memória para algum item

Listing 9.9: Exemplo de cópia de cadeias de caracteres.

```
#include<stdio.h>
int strcop(char *d, char *o);
int main(void)
{
     char destino[20];
     char *origem="cadeia de caractere de origem";
     strcop(destino, origem);
     printf("%s\n", origem);
     printf("%s\n", destino);
     return 0;
}
int strcop(char *d, char *o)
{
     while ((*d++ = *o++) != '\0');
     return 0;
}
```

de um programa. O tamanho em bytes reservado é definido pela variável `size`. O valor armazenado no espaço é indefinido. A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.

void `*calloc(size_t num, size_t size);` Reserva espaço na memória para um vetor de `num` itens do programa. Cada item tem tamanho `size` e todos os bits do espaço são inicializados com 0. A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.

void `free(void *pont);` O espaço apontado por `pont` é devolvido ao sistema para uso. Caso `pont` seja um ponteiro nulo nenhuma ação é executada. No caso do ponteiro não ter sido resultado de uma reserva feita por meio de uma das funções `calloc, realloc` ou `malloc` o resultado é indefinido.

void `realloc(void *pont, size_t size);` A função altera o tamanho do objeto na memória apontado por `pont` para o tamanho especificado por `size`. O conteúdo do objeto será mantido até um tamanho igual ao menor dos dois tamanhos, novo e antigo. Se o novo tamanho requerer movimento, o espaço reservado anteriormente é liberado. Caso o novo tamanho for maior, o conteúdo da porção de memória reservada a mais ficará com um valor sem especificação. Se o tamanho `size` for igual a 0 e `pont` não é um ponteiro nulo o objeto previamente reservado é liberado.

Estas funções podem ser encontradas na biblioteca `stdlib.h`. O Programa 9.10 ilustra o uso das função `calloc` e `free`.

Um outro exemplo, agora empregando a função `malloc()` está mostrado no Programa 9.11. Observe que neste programa também mostramos exemplos onde um endereço de variável foi passado para uma função de modo que a função `main` possa receber um valor (`vezes`).

Listing 9.10: Exemplo de uso de `calloc` e `free`.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
     float *v;
     int i, tam;
     printf("Qual o tamanho do vetor? ");
     scanf("%d", &tam);
     v = calloc(tam, sizeof(float));
     if (!v)
     {
          printf("Nao consegui alocar memoria.");
          return 1;
     }
     for (i=0; i<tam; i++)
     {
          printf("Elemento %d ?", i);
          scanf("%f", v+i);
          printf("Li valor %f \n", *(v+i));
     }
     free(v);
     return 0;
}
```

9.6 Ponteiros e Matrizes

Um ponteiro aponta para uma área de memória que é endereçada de maneira linear. Portanto, vetores podem ser fácilmente manipulados com ponteiros. No entanto, quando usamos estruturas de dados com maior dimensionalidade, como matrizes, por exemplo, que são arranjos bidimensionais de dados, é necessário mapear o espaço bidimensional (ou de maior ordem) para uma dimensão. No caso das matrizes é necessário mapear o endereço de cada elemento na matriz, que é definido por um par *(linha, coluna)* em um endereço linear.

Considere uma matriz chamada `matriz` de tamanho `LIN,COL` que poderia ser declarada e ter um de seus elementos lidos da maneira mostrada no trecho de programa listado em 9.12. Caso o programa utilizasse ponteiros ao invés de notação de matrizes, poderíamos usar uma solução que mapeasse a matriz que é um objeto de duas dimensões em um vetor que tem apenas uma. Neste caso o programa deve fazer a translação de endereços toda vez que precisar ler ou escrever na matriz. O trecho de programa ficaria como mostrado no exemplo 9.13. A expressão `matriz+(i*COL+j)` calcula a posição do elemento `matriz[i][j]` a partir do primeiro elemento da matriz que está no endereço inicial `matriz`.

No entanto, esta solução ainda não é a melhor já que o usuário necessita escrever diretamente uma expressão para mapear o endereço bidimensional da

Listing 9.11: Exemplo de uso de `malloc`.

```
#include <stdio.h>
#include <stdlib.h>
void LeVetor (float *v, int tam);
float  ProcuraMaior (float *v, int tam, int *vezes);
int main(void)
{
     float *v, maior;
     int i, tam, vezes;

     printf("Qual o tamanho do vetor? ");
     scanf("%d", &tam);
     v = (float *) malloc(tam * sizeof(float));
     if (v)
     {
          LeVetor(v, tam);
          maior = ProcuraMaior (v, tam, &vezes);
          printf(Maior = %f e aparece %d vezes.\n.",
                                   maior, vezes);
          free(v);
     }
     else
     {
          printf("Nao consegui alocar memoria.");
          return 1;
     }
     return 0;
}
void LeVetor (float *v, int tam)
{
     int i;
     for (i=0; i<tam; i++)
     {
          printf("Elemento %d ?", i);
          scanf("%f", v+i);
          printf("Li valor %f \n", *(v+i));
     }
}
float ProcuraMaior (float *v, int tam, int *vezes)
{
     int i;
     float maior;

     maior = v[0]; *vezes = 1;
     for (i=1; i<tam; i++)
     {
          if (v[i] &gt; maior)
          {
               maior = v[i];
               *vezes = 1;
          }
          else if (maior == v[i]) *vezes=*vezes+1;
     }
     return maior;
}
```

Listing 9.12: Exemplo de matriz normal sem uso de ponteiros.

```
#define LIN 3
#define COL 4

      int matriz[LIN][COL];
      for(i=0; i<LIN; i++)
      {
            for (j=0; j<COL; j++)
            {
                  printf("Elemento %d %d = ", i, j);
                  scanf("%d", &matriz[i][j]);
            }
      }
```

Listing 9.13: Exemplo de matriz mapeada em um vetor.

```
#define LIN 3
#define COL 4

      int *matriz;
      int i, j;

      matriz = malloc(LIN*COL*sizeof(int));
      if (!matriz)
      {
            printf("Erro.\n");
            return 1;
      }
      for(i = 0; i < LIN; i++)
      {
            for (j = 0; j < COL; j++)
            {
                  printf("Elemento %d %d = ", i, j);
                  scanf("%d", matriz+(i*COL+j));
            }
}
```

matriz `matriz[i][j]` em um endereço linear. O ideal é usar uma notação que use somente ponteiros. Esta notação será discutida nas seções seguintes.

9.7 Vetores de Ponteiros

Uma possibilidade mais interessante é utilizar vetores de ponteiros. Esta não é a notação ideal, mas é um passo na direção da notação mais efetiva. Neste caso cada linha da matriz corresponde a um vetor que é apontado por um ponteiro armazenado no vetor de ponteiros. Como ponteiros também são variáveis é possível então criar vetores de ponteiros e utilizá-los. O exemplo mostrado em 9.14 mostra um programa onde é utilizado um vetor de ponteiros para várias linhas de caracteres. Observe na função `main` a declaração **char** `*linha[LINHAS];` que define um vetor de tamanho `LINHAS`. Este vetor contem ponteiros e não valores. Até este momento do programa temos apenas posições reservadas para armazenar ponteiros. A alocação de espaço e a inicialização dos ponteiros é feito no primeiro comando **for**. Como cada elemento do vetor `linha` é um ponteiro temos que o endereço retornado pela função `malloc` é armazenado em cada um dos elementos deste vetor. A leitura das linhas de caracteres é feita pela função `gets(linha[i])` que passa o elemento do vetor onde os caracteres serão armazenados.

9.8 Ponteiros para Ponteiros

No exemplo anterior podemos observar que o número de linhas da matriz é fixa, e portanto, há uma mistura de notação de ponteiros com matrizes. Vamos considerar um exemplo onde tanto o número de linhas como o de colunas é desconhecido. Neste exemplo iremos criar um vetor de ponteiros que irá armazenar o endereço inicial de cada linha. Portanto, para obter um elemento da matriz primeiro devemos descobrir onde está a linha no vetor que armazena os endereços das linhas, em seguida procuramos na linha o elemento. A Figura 9.7 ilustra como será feito o armazenamento desta matriz.

O Programa 9.15, listado a seguir, irá pedir ao usuário que digite o número de linhas e colunas da matriz. Em seguida lerá todos os elementos da matriz e por último irá trocar duas linhas da matriz de posição. Observe que agora foi criado um ponteiro para ponteiro chamado de `**matriz`. O programa primeiro pergunta o número de linhas da matriz para poder alocar espaço para armazenar os ponteiros para cada uma das linhas. Em seguida é alocado espaço para armazenar cada uma das linhas. O comando

```
matriz = (int **) malloc (lin * sizeof(int *));
```

foi usado pelo programa para reservar espaço para armazenar `lin` linhas de ponteiros para ponteiros. Observe que o comando **sizeof(int** `*)` calcula o espaço para armazenar um ponteiro na memória. Note também que o valor retornado

Listing 9.14: Exemplo de uso de vetor de ponteiros.

```
#include<stdio.h>
#include<stdlib.h>
#define LINHAS 10
#define COLUNAS 60

int main(void)
{
     char *linha[LINHAS];
    int i;

     for (i = 0; i < LINHAS; i++)
     {
          if (!(linha[i] = malloc(COLUNAS*sizeof(char))))
          {
               printf("Sem memória para vetor %d.\n", i);
               return i;
          }
     }

     for (i = 0; i < LINHAS; i++)
     {
          printf("Entre com a linha %d.\n", i);
          gets(linha[i]);
     }

    for (i = 0; i < LINHAS; i++)
     {
          printf("Linha %d %s.\n", i, linha[i]);
     }
     return 0;
}
```

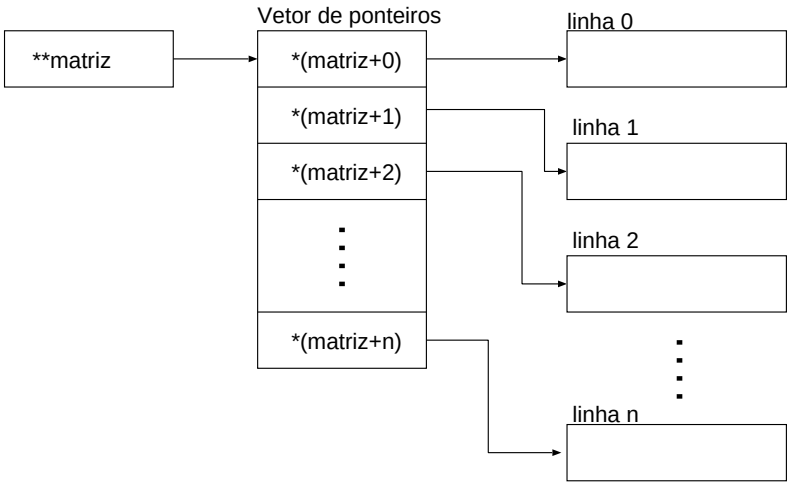

Figura 9.7: Armazenamento de matrizes com vetores de ponteiros.

pela função `malloc` foi conformado ao tipo ponteiro para ponteiro pela operação `(int **)`. O interessante do programa é que a troca de linhas da matriz involve simplesmente a troca de dois ponteiros e não a troca de todos os elementos das linhas. Esta solução é muito mais rápida do que trocar elemento a elemento, especialmente para matrizes grandes.

A seguir mostramos o programa nas listagens 9.16 e 9.17 que é o exemplo anterior modificado para utilizar funções. O propósito é mostrar como ficam as chamadas e as definições das funções que utilizam ponteiros para ponteiros.

Listing 9.15: Exemplo de uso de ponteiros para ponteiros.

```
#include<stdio.h>
#include<stdlib.h>
int main (void)
{
     int **matriz; /* ponteiro para os ponteiros */
     int lin, col; /* número de linhas e colunas */
     int i, j;
     int linha1, linha2; /* linhas que serao trocadas */
     char linha[80]; /* linha de caracteres com os dados */
     int *temp;

     puts("Qual o numero de linhas?");
     gets(linha); lin = atoi(linha);
     matriz = (int **) malloc (lin * sizeof(int *));
     if (!matriz)
     {
          puts("Nao há espaço para alocar memória");
          return 1;
     }
     puts("Qual o numero de colunas?");
     gets(linha); col = atoi(linha);
     for (i=0; i<lin; i++)
     {
          *(matriz +i) = (int *) malloc(col * sizeof (int));
          if (! *(matriz+i) )
          {
               printf("Sem espaço para alocar a linha %d", i);
               return 1;
          }
     }
     puts("Entre com os dados");}
     for (i=0; i<lin; i++)
     {
          printf("Entre com a linha %d\n", i);
          for (j=0; j<col; j++)
          {
               printf("Elemento %d %d\n", i, j);
               scanf("%d", *(matriz +i) +j);
          }
     }
     puts("Qual a primeira linha a ser trocada?");
     gets(linha); linha1=atoi(linha);
     puts("Qual a segunda linha a ser trocada?");
     gets(linha); linha2=atoi(linha);
     temp = *(matriz + linha1);
     *(matriz + linha1) = *(matriz + linha2);
     *(matriz + linha2) = temp;
     puts("Dados trocados.");
     for (i=0; i<lin; i++)
     {
          for (j=0; j<col; j++)
          {
               printf("%7d ", *(*(matriz +i) +j));
          }
          printf("\n");
     }
     return 0;
}
```

Listing 9.16: Exemplo de uso de ponteiros para ponteiros usando funções.

```
#include<stdio.h>
#include<stdlib.h>}
int **aloca_linhas(int );
void aloca_colunas (int **, int, int);
void le_dados(int **, int, int );
void imprime_matriz(int **, int, int );
void troca_linhas (int **, int, int);
int main (void)
{
     int **matriz;
     int lin, col;  
     int linha1, linha2;
     char linha[80];

     puts("Qual o numero de linhas?");
     gets(linha); lin = atoi(linha);
     matriz = aloca_linhas(lin);
     puts("Qual o numero de colunas?");
     gets(linha); col = atoi(linha);

     printf("Alocando espaço para linhas.\n");
     aloca_colunas(matriz, lin, col);
     le_dados(matriz, lin, col);
     imprime_matriz(matriz, lin, col);
     puts("Qual a primeira linha a ser trocada?");
     gets(linha); linha1=atoi(linha);
     puts("Qual a segunda linha a ser trocada?");
     gets(linha); linha2=atoi(linha);
     troca_linhas(matriz, linha1, linha2);
     imprime_matriz(matriz, lin, col);
     return 0;
}

int **aloca_linhas(int lin)
{
     int **m;}
     m = (int **) malloc (lin * sizeof(int *));
     if (!m)
     {
          puts("Sem espaço para alocar memória");
          return 1;
     }
     return m;
}
void aloca_colunas(int **matriz, int lin, int col)
{
     int i;
     for (i=0; i<lin; i++)
     {
          *(matriz +i) = (int *) malloc(col * sizeof (int));
          if (! *(matriz+i) )
          {
               printf("Sem espaço para linha %d", i);
               return 1;
          }
     }
}
```

Listing 9.17: Continuação do exemplo 9.16.

```
void le_dados (int **matriz, int lin, int col)
{
     int i, j;
     puts("Entre com os dados");
     for (i=0; i<lin; i++)
     {
          printf("Entre com a linha %d\n", i);
          for (j=0; j<col; j++)
          {
               printf("Elemento %d %d\n", i, j);
               scanf("%d", *(matriz +i) +j);
          }
     }
}
void imprime_matriz (int **matriz, int lin, int col)
{
     int i, j;
     for (i=0; i<lin; i++)
     {
          for (j=0; j<col; j++)
          {
               printf("%7d ", *(*(matriz +i) +j));
          }
          printf("\n");
     }
}
void troca_linhas ( int **matriz, int linha1, int linha2)
{
     int *temp;
     temp = *(matriz + linha1);
     *(matriz + linha1) = *(matriz + linha2);
     *(matriz + linha2) = temp;
}
```

Exercícios

9.1: Escreva um programa que gere um vetor de três dimensões (X, Y e Z) em que cada posição guarda a soma de suas coordenadas. As dimensões da matriz deverão ser determinadas em tempo de execução e o programa deverá informar os valores gerados.

9.2: Escreva um programa que leia uma frase de até 80 caracteres do teclado e imprima a frequência com que aparece cada uma das letras do alfabeto na frase.

9.3: Escreva um programa que leia uma frase de até 80 caracteres e a imprima em ordem reversa convertendo todos os caracteres minúsculos para maiúsculos.

9.4: Escreva um programa que leia uma matriz e a imprima. O programa deve ler o numero de colunas e linhas do teclado. O programa deve ainda trocar duas linhas da matriz de posição. Os números das linhas a serem trocadas devem ser lidos do teclado.

9.5: Escreva um programa que simule uma pilha usando vetores. O programa deve implementar as seguintes operações na pilha:

- Inserir
- Remover
- Listar

9.6: Escreva uma função que receba um ponteiro para uma cadeia de caractere e troque todo o caracter após um branco pelo seu equivalente maiúsculo.

9.7: Escreva um programa que leia seu nome completo e pergunte quantas letras tem o seu primeiro nome. Assuma que a letra 'a' tem índice 0, a letra 'b' índice 1 e assim por diante. O programa deve imprimir quantas letras iguais a letra cujo índice é o número de letras do seu primeiro nome existem no seu nome completo.

9.8: Escreva um programa que leia seu nome completo e pergunte quantas letras tem o seu primeiro nome. Faça uma função que verifique se a expressão apontada por substr está presente na cadeia apontada por str. A função possui o protótipo abaixo:¡br¿ `int posicao(char *substr, char *str);` e retorna a posição em que a sub-cadeia aparece em cadeia.

9.9: Escreva um programa que procure em uma matriz elementos que sejam ao mesmo tempo o maior da linha e o menor coluna. As dimensões da matriz devem ser pedidas ao usuário.

9.10: Escreva um programa que leia duas cadeias de caracteres e concatene a segunda cadeia ao final da primeira.

Capítulo 10

Estruturas

10.1 Introdução

Uma estrutura é um conjunto de uma ou mais variáveis, que podem ser de tipos diferentes, agrupadas sob um único nome. O fato de variáveis agrupadas em uma estrutura poderem ser referenciadas por um único nome facilita a manipulação dos dados armazenados nestas estruturas. Um exemplo poderia ser uma estrutura que armazenasse as diversas informações sobre os alunos de uma Universidade. Nesta estrutura estariam armazenadas, sob o mesmo nome, informações do tipo: nome, registro, data de nascimento, data de ingresso, CPF, etc. Uma estrutura pode incluir outras estruturas além de variáveis simples. As estruturas facilitam manipular estes agrupamentos complexos de dados. Por exemplo, considere o problema de ordenar as informações sobre os alunos da Universidade exemplo. A ordenação pode ser efetuada como se todos os dados que compõem a estrutura fossem uma entidade única.

10.2 Definições Básicas

Uma estrutura, então, é uma coleção de variáveis, de tipos diversos ou não, agrupadas sob um único nome. As variáveis que compõem a estrutura são os seus membros, elementos ou campos. Normalmente os elementos da estrutura tem alguma relação semântica. Por exemplo: alunos de uma universidade, discos de uma coleção, elementos de uma figura geométrica, etc. Vamos considerar o exemplo do aluno e assumir que estaremos armazenando o seu nome, registro, ano de entrada e curso. Para este fim podemos criar uma estrutura como a descrita abaixo:

```
struct aluno
{
     char nome[40];
```

```
     int registro;
     int ano_entrada;
     char curso[20];
};
```

A palavra chave **struct** inicia a declaração da estrutura, em seguida pode aparecer um identificador (`aluno`), que subsequentemente pode ser usado como abreviação da definição da estrutura. A declaração continua com a lista de declarações entre chaves e termina com um `;`. Um membro da estrutura e uma váriavel não membro da estrutura podem ter o mesmo nome, já que é possível distingui-las por contexto.

A declaração acima ainda não alocou espaço de memória já que nenhuma variável foi realmente definida. Esta declaração é apenas um modelo de como estruturas do tipo `aluno` devem ser construídas. Para definir estruturas do tipo aluno podemos usar a seguinte declaração.

```
struct aluno paulo, carlos, ana;
```

Nesta declaração três estruturas do tipo aluno foram criadas. Esta declaração alocou espaço para armazenar os dados dos três alunos. A declaração acima é idêntica, na forma, a declaração de variáveis de um tipo pré-definido, como por exemplo:

```
int a, b, c;
```

É possível declarar ao mesmo tempo o modelo da estrutura e as variáveis do programa. Por exemplo,

```
struct aluno
{
     char nome[40];
     int registro;
     int ano_entrada;
     char curso[20];
} paulo, carlos, ana;
```

Para referenciar um elemento da estrutura usa-se o nome da variável do tipo da estrutura seguida de um ponto e do nome do elemento. Por exemplo,

```
paulo.ano_entrada = 1999;
```

armazena o ano em que aluno `paulo` entrou na universidade. Para ler o nome do curso que paulo cursa pode-se usar o comando

```
gets(paulo.curso);
```

Estruturas podem conter outras estruturas como membros. Por exemplo, vamos definir uma estrutura para armazenar uma data com a seguinte definição:

```
struct data
{
     int dia, mes, ano;
}
```

Agora vamos modificar a estrutura aluno de modo que ela inclua a data de nascimento do aluno. A estrutura fica com a seguinte definição:

```
struct aluno
{
     char nome[40];
     int registro;
     int ano_entrada;
     char curso[20];
     struct data data_nascimento;
};
```

Para se referir ao mês de nascimento de uma variável `paulo` do tipo estrutura `aluno` usamos a declaração

```
paulo.data_nascimento.mes
```

o operador `.` associa da esquerda para a direita.

10.3 Atribuição de Estruturas

É possível atribuir o conteúdo de uma estrutura a outra estrutura do mesmo tipo, não sendo necessário atribuir elemento por elemento da estrutura. O programa 10.1 ilustra como podemos atribuir uma estrutura a outra.

Listing 10.1: Atribuição de Estruturas.

```
#include<stdio.h>
int main ()
{
     struct empregado
     {
          char nome[40];
          float salario;
     } temp, emp1;

     puts("Entre com nome.");
     gets(emp1.nome);
     puts("Qual o salario?"); scanf("%f", &emp1.salario);
     temp=emp1;
     printf("O salario de %s e %.2f\n",
               temp.nome, temp.salario);
     return 0;
}
```

10.4 Matrizes de Estruturas

Estruturas aparecem freqüentemente na forma de matrizes. A declaração abaixo define uma matriz de 100 estruturas do tipo aluno.

```
struct aluno turma[100];
```

Para se imprimir ao nome do terceiro aluno (início em 0) usamos o comando

```
printf("%s\n", turma[2].nome);
```

O exemplo 10.2 mostra atribuições entre estruturas e operações aritméticas envolvendo membros de estruturas. O programa coloca um vetor de estruturas em ordem crescente usando como chave de ordenação um dos membros da estrutura (`nota`).

10.5 Estruturas e Funções

Primeiro vamos considerar o caso de passar elementos da estrutura para funções. Caso os elementos da estrutura sejam variáveis de um dos tipos já vistos, a passagem é efetuada da maneira normal. Por exemplo, considere a estrutura

```
struct ponto
{
     float x, y;
     p1, p2;
}
```

Para passar a coordenda `x` do ponto `p1` para uma função chamada `positivo` poderíamos a seguinte declaração:

```
if ( positivo(p1.x) == 0 )
   puts("Eixo y);
else if ( positivo(p1.x)>0 )
   puts("Eixo positivo dos x");
else
  puts("Eixo negativo dos x");
```

A função que recebe este parâmetro está preparada para receber uma variável de ponto flutuante simples. Caso seja necessário passar o endereço de um dos membros ou elementos da estrutura basta colocar o operador `&` antes do nome da estrutura. Por exemplo, para trocar os valores das coordenadas `x` dos pontos `p1` e `p2` usaríamos chamadas da seguinte forma.

```
troca_x (&p1.x, &p2.x);
```

Para trabalhar com endereços é necessário usar ponteiros dentro da função `troca_x`, mas isto veremos no próximo item. Antes vamos verificar como é possível passar uma estrutura inteira para uma função.

Listing 10.2: Ordenação de Estruturas.

```
#define MAX 4
#include <stdio.h>
#include <string.h>
int main ()
{
     struct aluno
     {
          char nome[40];
          float n1, n2, media;
     } turma[MAX], turma2[MAX];
     int i, j, pos;

     puts("Lendo dados da turma");
     for (i=0; i<MAX; i++)
     {
          printf("Dados do aluno %d\n", i);
          puts("Nome?"); gets(turma[i].nome);
          puts("Primeira nota?"); scanf("%f", &turma[i].n1);
          puts("Segunda nota?"); scanf("%f", &turma[i].n2);
          getchar();
          turma[i].media=(turma[i].n1+turma[i].n2)/2.0;
     }
     puts("Imprimindo dados lidos da turma.");
     puts("Digite qualquer coisa para continuar."); getchar();
     for (i=0; i<MAX; i++)
     {
          printf("\nDados do aluno %d\n", i);
          printf("Nome: %s\n",turma[i].nome);
          printf("Primeira nota: %.1f\n", turma[i].n1);
          printf("Segunda nota: %.1f\n", turma[i].n2);
          printf("Media: %.1f\n", turma[i].media);
     }
     for (i=0; i<MAX; i++)
     {
          pos = 0;
          for (j=0; j<MAX; j++)
               if (turma[i].media > turma[j].media) pos++;
                    turma2[pos] = turma[i];
     }

     for (i=0; i<MAX; i++) turma[i]=turma2[i];

     puts("Imprimindo dados ordenados da turma.");
     puts("Digite qualquer coisa para continuar."); getchar();
     for (i=0; i<MAX; i++)
     {
          printf("\nDados do aluno %d\n", i);
          printf("Nome: %s\n",turma[i].nome);
          printf("Primeira nota: %.1f\n", turma[i].n1);
          printf("Segunda nota: %.1f\n", turma[i].n2);
          printf("Media: %.1f\n", turma[i].media);
     }
     return 0;
}
```

Estruturas, quando passadas para funções, se comportam da mesma maneira que as variáveis dos tipos que já estudamos. Ao passar uma estrutura para uma função estaremos passando os valores armazenados nos membros da estrutura. Como este tipo de passagem é feito por valor, alterações nos membros da estrutura não modificam os valores da estrutura na função que chamou. A passagem de estruturas para funções é ilustrada no exemplo 10.3 onde o comprimento da reta que liga dois pontos `P1` e `P2` é calculado e impresso.

Listing 10.3: Passagem de estruturas para funções.

```
#include<stdio.h>
#include<math.h>
struct ponto
{
     float x, y;
} P1, P2;
float comp (struct ponto, struct ponto);
int main ()
{
     puts ("Coordenadas do ponto 1");
     printf("x1 = ? "); scanf("%f", &P1.x);
     printf("y1 = ? "); scanf("%f", &P1.y);
     puts ("Coordenadas do ponto 2");
     printf("x1 = ? "); scanf("%f", &P2.x);
     printf("y1 = ? "); scanf("%f", &P2.y);
     printf("\nComprimento da reta = %.2f",
     comp(P1, P2));
     return 0;
}

float comp(struct ponto p1, struct ponto p2)
{
     return sqrt (pow(p2.x-p1.x,2)+pow(p2.y-p1.y,2));
}
```

O exemplo mostrado nas listagens 10.4 e 10.5 apresenta um programa que implementa um banco de dados simples sobre uma turma de alunos. Neste exemplo mostramos a passagem de matrizes de estruturas para funções.

10.6 Ponteiros para Estruturas

Para definir ponteiros para estruturas a declaração é similar a declaração de um ponteiro normal. O exemplo abaixo mostra a definição de um ponteiro chamado `maria` para uma estrutura chamada `aluno`.

```
struct aluno
{
```

Listing 10.4: Ordenação de Estruturas.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAXNOME 40
#define MAXALUNOS 4
#define VERDADE 1
#define FALSO 0

struct RegAluno
{
     char nome[MAXNOME];
     float n1, n2, m1;
};
void insere(struct RegAluno turma[]);
void remover(struct RegAluno turma[]);
void lista (struct RegAluno turma[]);
void inicia(struct RegAluno turma[]);
char escolhe();

int main(void)
{
     struct RegAluno turma[MAXALUNOS];
     char c, continua;

     inicia(turma);
     do {
          continua = VERDADE;
          c = escolhe();
          switch (c)
          {
               case 'i':
                    insere(turma);
                    break;
               case 'r':
                    remover(turma);
                    break;
               case 'l':
                    lista(turma);
                    break;
               default:
                    continua = !continua;
          };
     } while (continua);
     printf("Acabou.\n");
}

char escolhe()
{
     char c;

     printf("\nEscolha uma operacao.\n");
     printf("[I]nserir um aluno.\n");
     printf("[R]emover um aluno.\n");
     printf("[L]istar a turma.\n");
     printf("Qualquer outra tecla para terminar.\n");
     c = tolower(getch());
     return c;
}
```

Listing 10.5: Ordenação de Estruturas (continuação).

```
void insere (struct RegAluno turma[])
{
     int i=0;
     char achou=FALSO;

     while (i<MAXALUNOS && !achou)
     {
          if (*turma[i].nome == ' ')
          {
               achou = !achou;
               printf("\nForneca os dados sobre o aluno.\n");
               fflush(stdin);
               printf("Nome. "); gets(turma[i].nome);
               printf("Nota 1. "); scanf("%f", &turma[i].n1);
               printf("Nota 2. "); scanf("%f", &(turma[i].n2));
               turma[i].m1 = (turma[i].n1+turma[i].n2)/2.0;
          }
          i++;
     }
     if (!achou)
          printf("\nDesculpe, nao ha espaco para inserir um novo aluno.\n");
}

void remover (struct RegAluno turma[])
{
     int i=0;
     char nome[MAXNOME];
     char achou = FALSO;

     fflush(stdin);
     printf("Qual o nome a remover?");
     gets(nome);
     while (i<MAXALUNOS && !achou)
     {
          if (!strcmp(turma[i].nome, nome))
          {
               achou = !achou;
               *turma[i].nome = ' ';
               printf("\nAluno %s removido.\n", nome);
          }
          i++;
     }
     if (!achou) printf("\nDesculpe, mas nao ha este aluno.\n");
}

void lista  (struct RegAluno turma[])
{
     int i, aluno=0;

     for (i=0; i<MAXALUNOS; i++)
     {
          if (*turma[i].nome != ' ')
          {
               printf("\nDados sobre o aluno %d.\n", aluno++);
               printf("Nome.   "); printf("%s\n", turma[i].nome);
               printf("Nota 1. "); printf("%.1f\n", turma[i].n1);
               printf("Nota 2. "); printf("%.1f\n", turma[i].n2);
               printf("Media.  "); printf("%.1f\n", turma[i].m1);
          }
```

```
    char nome[40];
    int ano_entrada;
    float n1, n2, media;
} *maria;
```

Ponteiros são uteis quando passamos estruturas para funções. Ao passar apenas o ponteiro para estrutura economizamos tempo e memória. O espaço de memória, é economizado por que se evita passar os dados que compõem a estrutura um por um. O tempo é economizado porque não é necessário gastar o tempo de empilhar e desempilhar todos os elementos da estrutura no processo de passagem para a função. Empilhar e desempilhar se referem a pilha de dados usada para transferir os dados entre funções.

Para acessar elementos da estrutura apontada por um ponteiro usa-se o chamado operador seta (`->`). Por exemplo para imprimir a média da aluna `maria` usaríamos o comando

```
printf ("A media vale %.1f", maria->media);
```

Para alocar espaço para estruturas apontadas por ponteiros é necessário usar o operador unário **sizeof**, isto porque o tamanho de uma estrutura é sempre igual ou maior que a soma dos tamanhos dos seu componentes. Para explicar esta fato devemos considerar como os dados são armazenados na memória dos computadores.

Algumas arquiteturas de computadores endereçam os dados na mémoria por bytes, isto é cada endereço de memória se refere a um byte. No entanto, estas arquiteturas lêem sempre uma palavra inteira da memória. Usualmente, palavras podem ser compostas de dois bytes e começam em endereços pares, como está mostrado na figura abaixo. Sabemos que existem variáveis que ocupam mais de um byte, por exemplo inteiros que são compostos de dois bytes.

Imagine então uma estrutura composta de um caracter (1 byte) e um número de inteiro (2 bytes). Caso a memória do computador seja organizada em palavras de 16 bits ou 2 bytes a estrutura acima ocuparia 3 bytes ou uma palavra e meia. Para ler o número inteiro o programa deveria ler duas palavras. Lembrar que se os dados fossem sempre armazenados sequencialmente, metade do número inteiro estaria em uma palavra e a metade restante na outra, como está indicado na figura abaixo (parte a). Para facilitar o acesso às variáveis, alguns compiladores armazenam as variáveis de acordo com o que está indicado na figura (parte b). Observar que agora a estrutura ocupa quatro bytes. Neste caso o acesso ao número inteiro será sempre feito em um passo e portanto ganhou-se em tempo de acesso ao custo de gasto de memória. Este é uma troca constante em computação.

Vimos então que embora o total de bytes dos elementos da estrutura fosse três o compilador pode armazenar a estrutura em quatro bytes, daí a necessidade de sempre usar o operador **sizeof** quando alocar espaço.

O programa 10.6 mostra como utilizar ponteiros para estruturas e a forma mais segura de alocar espaço para os dados.

Listing 10.6: Ponteiros para de estruturas.

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

#define TAMNOME 40

typedef struct _func
{
     char nome[TAMNOME];
     float salario;
     float imposto;
} Tfunc ;

void le (Tfunc *, int );
void imprime (Tfunc *, int );
float imposto (Tfunc *, int);

int main ()
{
     Tfunc *cadastro;
     int funcionarios;

     puts("Quantos funcionarios tem a empresa?");
     scanf ("%d", &funcionarios);

     cadastro=(Tfunc *) malloc(funcionarios*sizeof(Tfunc));
     if (!cadastro)
     {
          puts("Nao ha espaco para alocar memoria.");
          exit (1);
     }
     le(cadastro, funcionarios);

     printf("O imposto total a ser recolhido vale %.2f\n",
     imposto(cadastro, funcionarios));
     imprime(cadastro, funcionarios);
     exit(0);
}

void le (Tfunc *cadastro, int funcionarios)
{
     int i;

     puts("Lendo dados dos funcionarios.");
     for (i=0; i<funcionarios; i++)
     {
          printf ("Dados dos funcionario %d\n", i);
          puts ("Nome ?");
          getchar();
          fgets((cadastro+i)->nome, TAMNOME, stdin);
          puts ("Salario ?");
          scanf("%f", &((cadastro+i)->salario));
     }
}

void imprime (Tfunc *cadastro, int funcionarios)
{
     int i;

     puts("Imprimindo dados dos funcionarios.");
```

Observar que neste exemplo usamos as duas notações possíveis para representar elementos de uma estrutura. Nas funções `le` e `imprime` usamos a notação de ponteiros e na função `imposto` usamos o fato de que após a alocação de espaço para o vetor de estruturas podemos usar a notação de vetores. Reproduzimos abaixo as funções com as duas notações para ficar mais claro o que estamos procurando destacar.

Função `le` (notação ponteiros)

```
gets((cadastro+i)->nome);
scanf("%f", &((cadastro+i)->salario));
```

Função `imprime` (notação ponteiros)

```
printf("Nome = %s\n", (cadastro+i)->nome);
printf("Salario = %.2f\n", (cadastro+i)->salario);
```

Função `imposto` (notação vetorial)

```
if (cadastro[i].salario > 1000.00)
     func += cadastro[i].salario * 0.25;
else
     func += cadastro[i].salario * 0.10;
```

Exercícios

10.1: Considere que uma empresa precisa armazenar os seguintes dados de um cliente:

- Nome completo com no máximo 50 caracteres;
- renda mensal do do cliente;
- ano de nascimento;
- possui ou não carro.

Defina um tipo e uma estrutura para armazenarem estes dados e escreva um programa que leia estes dados armazene-os em uma variável e em seguida os imprima.

10.2: Considerando a mesma estrutura do exercício anterior, escreva um programa que leia os dados de 100 clientes e imprima:

- quantos clientes têm renda mensal acima da média;
- quantos clientes têm carro;
- quantos clientes nasceram entre 1960 (inclusive) e 1980 (exclusive).

10.3: Reescreva o programa 10.2 empregando funções para implementar as diversas tarefas do programa. A função `main` deve ficar da maneira indicada na Listagem 10.7.

10.4: Escrever um programa que utilize structs e ponteiro para struct e imprima o conteudo das variaveis da struct.

10.5: Escrever um programa que utilize enumeradores com as materias do seu periodo. Inicialize cada materia com um numero. Depois imprime os valores das variaveis enumeradas.

10.6: Escrever um programa que utilize union. Inicialize as variaveis com valores diferentes e imprima o conteudo delas.

10.7: Fazer um programa que simule uma pilha push pop, usando structs. Um exemplo de entrada poderia ser o seguinte:

empilha C
empilha B
empilha A
desempilha A
desempilha B

Listing 10.7: Listagem do exercicio 3.

```
int main (void)
{
     struct aluno turma[MAX];

     le(turma);
     puts("Imprimindo dados lidos da turma.");
     puts("Digite qualquer coisa para continuar.");
     getchar();
     imprime(turma);

     ordena_medias(turma);
     puts("Imprimindo dados ordenados da turma.");
     puts("Digite qualquer coisa para continuar.");
     getchar();
     imprime(turma);
     getchar();
}
```

desempilha C

10.8: Escreva um programa que solicite o nome e telefone de uma pessoa e grave essas informações num vetor de uma estrutura que contem esses dados (nome e telefone). O programa deve ter três opcoes apenas: uma que adiciona um novo dado, outra que lista todos os dados atualmente armazenados na memória e outra que sai do programa. Esse vetor de estrutura deve ter apenas 10 elementos e fornecer uma mensagem de erro caso o usuário tente adicionar mais pessoas que este máximo permitido.

10.9: Escreva uma estrutura similar as strings do Delphi (possuem um campo armazenando o tamanho da string e um ponteiro para o primeiro caracter da string) e crie as funcões strcpy e strcat para strings nesse formato.

10.10: Escreva um programa fazendo o uso de estruturas. Você deverá criar uma estrutura chamada Ponto, contendo apenas a posição x e y (inteiros) do ponto. Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles. Apresente no final a distância entre os dois pontos.

10.11: Crie uma estrutura chamada retangulo, que possua duas estruturas ponto (o ponto superior esquerdo e o ponto inferior direito). Faça um programa que receba as informações acerca de um retângulo (as coordenadas dos dois pontos), e informe a área, o comprimento da diagonal e o comprimento de cada aresta

10.12: Escreva um programa que use as mesmas estruturas do exercício anterior para descobrir se um ponto está dentro de um retângulo.

10.13: Considere que foi definida a seguinte estrutura:

```
typedef struct _frac
{
     int numerador, denominador;
} FRACAO;
```

Escreva um programa em C que calcule as quatro operações usando frações definidas com estruturas do tipo `FRACAO`. O programa deve ler duas frações e imprimir o resultado de cada uma das quatro operações.

Capítulo 11

Entrada e Saída por Arquivos

11.1 Introdução

Em `C` não existem comandos de Entrada e Saída especiais como em outras linguagens de programação, sendo estas tarefas executadas por funções especialmente criadas para esta finalidade e armazenadas em bibliotecas específicas.

11.2 Fluxos de Dados

Para isolar os programadores dos problemas de manipular os vários tipos de dispositivos de armazenamento e seus diferentes formatos a linguagem `C` utiliza o conceito de fluxo (*stream*) de dados. Todos os diferentes sistemas de arquivos se comportam da mesma maneira quando manipulados como um fluxo contínuo de dados. Dados podem ser manipulados em dois diferentes tipos de fluxos: fluxos de texto e fluxos binários.

11.2.1 Fluxos de Texto

Um fluxo de texto é composto por uma seqüência de caracteres, que pode ou não ser dividida em linhas terminadas por um caractere de final de linha. Um detalhe que deve ser considerado é que na última linha não é obrigatório o caracter de fim de linha.

Nem sempre a tradução entre a representação do caracter no fluxo de texto e no sistema de arquivos do computador hospedeiro é um para um. Por exemplo, entre UNIX e DOS há uma diferença na representação de final de linha que causa problemas na impressão de arquivos. Em UNIX um final de linha é

representado pelo caracter de alimentação de linha (LF). Em DOS um final de linha é representado pelo par retorno de carro/alimentação de linha (CR/LF). Deste modo quando um arquivo gerado em UNIX vai para uma impressora que espera final de linha no modo DOS surge o que é comumente chamado de efeito escada. A impressão continua na linha seguinte mas sem voltar para o início da linha porque em UNIX o caracter de retorno de carro não é inserido no fluxo de texto.

Até agora temos trabalhado com os fluxos de dados padrão: `stdin`, para entrada de dados e `stdout` para saída de dados. Todo programa em `C` ao iniciar é automaticamente associado a estes dois fluxos sem necessitar de nenhuma intervenção do programador.

11.2.2 Fluxo Binário

Um fluxo binário é composto por uma seqüência de bytes lidos, sem tradução, diretamente do dispositivo externo. Existe uma correspondência um para um entre os dados do dispositivo e os que estão no fluxo que o programa manipula.

A Figura 11.1 ilustra estes dois tipos de fluxos. No fluxo de texto os dados são armazenados como caracteres sem conversão para representação binária. Cada um dos caracteres ocupa um byte. O numero `12` ocupa dois bytes e o número `113` ocupa 3. Um caractere em branco foi inserido entre cada um dos números para separá-los, de modo que a função de entrada e saída possa descobrir que são dois números inteiros (`12` e `113`) e não o número `12113`.

No fluxo binário cada número inteiro ocupa 32 bits e é armazenado na forma binária. Os caracteres do exemplo estão armazenados em binário seguindo a tabela ASCII. Observe que não há necessidade de separar os números já que eles sempre ocupam 32 bits.

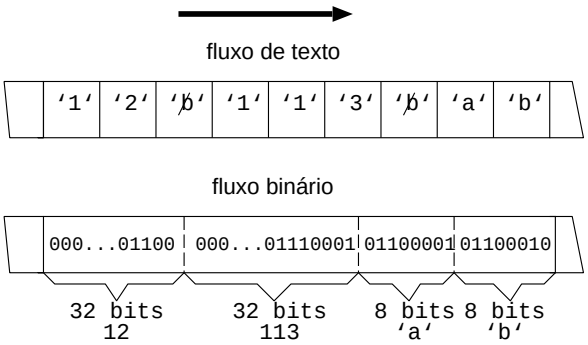

Figura 11.1: Fluxos de dados.

11.2.3 Arquivos

Arquivo pode ser qualquer dispositivo de Entrada e Saída como por exemplo: impressora, teclado, disquete, disco rígido etc. Programas vêem os arquivos através de fluxos. Para que um determinado arquivo seja associado a um determinado fluxo é necessário que o arquivo seja "aberto". Após esta operação, o programa pode utilizar os dados armazenados no arquivo.

Operações comuns em arquivos são:

- abertura e fechamento de arquivos;
- apagar um arquivo;
- leitura e escrita de um caracter;
- procurar saber se o fim do arquivo foi atingido;
- posicionar o arquivo em um ponto determinado.

Obviamente algumas dessas funções não se aplicam a todos os tipos de dispositivos. Por exemplo, para uma impressora não é possível usar a função que reposiciona o arquivo no ínicio. Um arquivo em disco permite acesso aleatório enquanto um teclado não.

Ao final das operações necessárias o programa deve fechar o arquivo. Ao final da execução de um programa todos os arquivos associados são fechados automaticamente e os conteúdos dos *buffers* são descarregados para o dispositivo externo. Caso o arquivo seja de entrada o conteúdo do *buffer* é esvaziado.

11.3 Funções de Entrada e Saída

As funções de Entrada e Saída normalmente utilizadas pelos programadores estão armazenadas na biblioteca `stdio.h`. As funções mais comuns estão mostradas na tabela 11.1.

Para ter acesso aos dados em um arquivo é necessário a definição de um ponteiro do tipo especial `FILE`. Este tipo também está definido na biblioteca `stdio.h`. Um ponteiro deste tipo permite que o programa tenha acesso a uma estrutura que armazena informações importantes sobre o arquivo. Para definir uma variável deste tipo o programa deve conter a seguinte declaração

```
FILE *arq;
```

onde `arq` é o ponteiro que será usado para executar as operações no arquivo.

Função	Descrição
`fopen()`	Abre um arquivo
`fputc()`	Escreve um caracter em um arquivo
`getc()`, `fgetc()`	Lê um caracter de um arquivo
`fprintf()`	Equivalente a `printf()`
`sscanf()`	Equivalente a `scanf()`. Lê de uma cadeia de caracteres
`fscanf()`	Equivalente a `scanf()`
`fseek()`	Posiciona o arquivo em um ponto específico
`rewind()`	Posiona o arquivo no início
`feof()`	Retorna verdade se chegou ao fim do arquivo
`ferror()`	Verifica a ocorrência de um erro
`fflush()`	Descarrega o *buffer* associado ao arquivo
`fread()`	Leitura de dados no modo binário
`fwrite()`	Escrita de dados no modo binário

Tabela 11.1: Exemplos de funções de Entrada e Saída.

11.4 Início e Fim

As operações mostradas a seguir mostram operações que devem ser realizadas antes e depois de usar um arquivo (`fopen()` e `fclose()`). As outras duas funções servem para que o usuário possa detectar o fim de um arquivo ou voltar para seu início.

11.4.1 Abrindo um Arquivo

Antes de qualquer operação ser executada com o arquivo, ele deve ser "aberto". Esta operação associa um fluxo de dados a um arquivo. Um arquivo pode ser aberto de diversas maneiras de acordo com as operações que deverão ser executadas: leitura, escrita, leitura/escrita, adicão de texto etc. A função utilizada para abrir o arquivo é chamada `fopen()` e tem o seguinte protótipo:

```
FILE *fopen (const char *parq, const char *modo)
```

onde `parq` é um ponteiro de arquivo para o arquivo a ser manipulado e `modo` é um ponteiro para uma cadeia de caracteres que define a maneira como o arquivo vai ser aberto. Este ponteiro não deve ser modificado e a função retorna um ponteiro nulo (`NULL`) se o arquivo não puder ser aberto. A seguir listamos os diversos modos que podem ser usados para abrir um arquivo.

"r": Abre um arquivo para leitura, o arquivo deve existir ou um erro ocorre.

"w": Cria um arquivo vazio para escrita, caso um arquivo com o mesmo nome exista o seu conteúdo é apagado.

"a": Adiciona ao final de um arquivo. O arquivo é criado caso ele não exista.

"r+": Abre um arquivo para leitura e escrita. O arquivo deve existir ou um erro ocorre.

"w+": Cria um arquivo vazio para leitura e escrita. Se um arquivo com o mesmo nome existe o conteúdo é apagado.

"a+": Abre um arquivo para leitura e adição. Todas as operações de escrita são feitas no final do arquivo. É possível reposionar o ponteiro do arquivo para qualquer lugar em leituras, mas as escritas moverão o ponteiro para o final do arquivo. O arquivo é criado caso não exista.

Observar que se um arquivo for aberto com permissão de escrita todo o seu conteúdo anterior será apagado. Caso o arquivo não exista ele será criado.

O trecho de programa abaixo ilustra os passos necessários para abrir um arquivo para escrita. Primeiro é declarado o ponteiro `pa` para o arquivo. Em seguida a função `fopen` é chamada para associar o nome externo do programa (`arquivo.txt`) no modo `escrita` ao ponteiro `pa`. Um teste para ponteiro `nulo` é feito para verificar se ocorreu algum problema com a operação de abertura do arquivo.

```
FILE *pa; /* declaracao do ponteiro para arquivo */

/* nome externo associado ao interno */
pa = fopen ("arquivo.txt", "w");
if (pa == NULL)  /* verifica erro na abertura */
{
     printf("Arquivo nao pode ser aberto.");
     return 1;
}
```

Lembrar que abrir, para escrita, um arquivo que já existe, implica em apagar todo o conteúdo anterior e a preparação do arquivo para receber dados a partir de seu ponto inicial. Se o programador deseja acrescentar dados ao final de um arquivo já existente o modo de abertura deve ser `a`.

11.4.2 Fechando um Arquivo

Um arquivo aberto por meio da função `fopen()` deve ser fechado com a função `fclose()` cujo protótipo é

```
int fclose (FILE *parq);
```

onde `parq` é um ponteiro de arquivo para o arquivo que deve ser fechado. Todos os *buffers* internos associados com o fluxo de dados do arquivo são descarregados. O conteúdo de qualquer *buffer* não escrito é escrito e dados não lidos de

buffers são perdidos. Este ponto é importante de ser considerado porque em muitos sistemas operacionais uma operação de escrita em um arquivo não ocorre imediatamente a emissão da ordem de escrita. O sistema operacional pode executar a ordem no momento que achar mais conveniente. Um valor zero de retorno significa que a operação foi executada com êxito, qualquer outro valor implica em erro.

11.4.3 Fim de Arquivo

A função `feof()` indica que um arquivo chegou ao seu final. A pergunta que pode surgir é a seguinte - *Se já existe o valor `EOF` para indicar o final de arquivo, por que precisamos de uma função extra do tipo `feof()`?* O problema é que `EOF` é um valor inteiro e ao ler arquivos binários este valor pode ser lido como parte do arquivo e não por ser o final do arquivo. A função `feof()` serve para indicar que o final de um arquivo binário foi encontrado. Naturalmente esta função pode ser aplicada também a arquivos texto. O protótipo da função é o seguinte:

```
int feof(FILE *parq)
```

Um valor diferente de zero é retornado no caso de ter sido atingido o final do arquivo. O valor zero indica que ainda não se chegou ao final do arquivo.

O exemplo 11.1 mostra um programa que lê um caractere do teclado e o mostra na tela. Neste exemplo a leitura termina quando o usuário digita o caracter `<ctl>+D`, que indica final de arquivo pelo teclado em Unix (no outro sistema é `<ctl>+Z`).

Listing 11.1: Uso da função `feof()`.

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

    c = getchar();
    while (!feof(stdin))
    {
        putchar(c);
        c = getchar();
    }
    return 0;
}
```

11.4.4 Volta ao Início

A função `rewind()` recoloca o indicador de posição de arquivo no inicio do arquivo. Uma operação semelhante ao que fazemos em uma fita cassete de música

ou vídeo. O protótipo da função é o seguinte:

```
void rewind(FILE *parq)
```

é importante observar que o arquivo deve estar aberto em um modo que permita a execução das operações desejadas. Por exemplo, um arquivo aberto somente para "escrita" e em seguida reposicionado para o início, não irá permitir outra operação que não seja "escrita".

11.5 Lendo e Escrevendo Caracteres

As operações mais simples em arquivos são a leitura e escrita de caracteres. Para ler um caracter de um arquivo, que foi previamente aberto, pode-se usar as funções `getc()` e `fgetc()`, que são equivalentes. O protótipo de `fgetc` é:

```
int fgetc (FILE *parq);
```

As funções `getc()` e `fgetc()` são equivalentes e muitos compiladores implementam `getc()` como uma macro do seguinte modo:

```
#define getc(parq) fgetc(parq)
```

A função lê o caracter como um **unsigned char** mas retorna o valor como um inteiro, onde o byte mais significativo vale zero. O apontador do arquivo avança um caractere e passa a apontar para o próximo caractere a ser lido. A função devolve o código `EOF` ao chegar ao final do arquivo ou caso um erro ocorra. O valor `EOF` também é um inteiro válido e portanto ao usar arquivos binários é necessário que a função `feof()` seja utilizada para verificar o final do arquivo. A função `ferror()` pode ser usada para determinar se um erro ocorreu.

Para escrever caracteres há duas funções definidas `putc()` e `fputc()`. O protótipo da função `fputc` é o seguinte:

```
int fputc(int ch, FILE *parq)
```

onde `parq` é um ponteiro de arquivo para o arquivo que foi previamente aberto por meio da função `fopen()` e `ch` e o caracter a ser escrito.

O programa 11.2 mostra como um arquivo pode ser criado para leitura e escrita. Em seguida um conjunto de caracteres lido do teclado é escrito no arquivo. O próximo passo é a leitura do arquivo que é iniciada após uma chamada a função `rewind()`, fazendo com que o indicador de posição do arquivo volte a apontar para seu início.

Uma outra alternativa mostrada em 11.3 mostra um exemplo onde o arquivo é criado para escrita em seguida é fechado e reaberto para leitura ficando automaticamente posicionado no início para a leitura.

Listing 11.2: Exemplo de leitura e escrita de caracteres.

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

int main (void )
{
     int c;
     FILE *pa;
     char *nome = "texto.txt";

     /* Abre o arquivo para leitura e escrita */
     if (( pa = fopen(nome, "w+")) == NULL)
     {
          printf("\n\nNao foi possivel abrir o arquivo.\n");
          exit(1);
     }
     /* Cada caracter digitado ser gravado no arquivo */
     c = getchar();
     while (!feof(stdin))
     {
          fputc(c, pa);
          c = getchar();
     }

     rewind(pa);  /* volta ao inicio do arquivo */
     printf("\nTerminei de escrever, agora vou ler.\n");
     c = fgetc(pa);
     while (!feof(pa))
     {
          putchar(c);
          c = fgetc(pa);
     }
    fclose(pa);

     return 0;
}
```

Listing 11.3: Exemplo de leitura e escrita de caracteres.

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

int  main (void )
{
     int c;
     FILE *pa;
     char *nome = "texto.txt";

     if (( pa = fopen(nome, "w+")) == NULL)
     {
          printf("\n\nErro ao abrir o arquivo - escrita.\n");
          exit(1);
     }
     /* Cada caracter digitado ser gravado no arquivo */
     c = getchar();
     while (!feof(stdin))
     {
          fputc(c, pa);
          c = getchar();
     }

     fclose(pa);
     printf("\nTerminei de escrever, agora vou ler.\n");

     if (( pa = fopen(nome, "r")) == NULL)
     {
          printf("\n\nErro ao abrir o arquivo - leitura.\n");
          exit(1);
     }

     c = fgetc(pa);
     while (!feof(pa))
     {
          putchar(c);
          c = fgetc(pa);
     }
    fclose(pa);
     return 0;
}
```

11.6 Testando Erros

A função `ferror(FILE *parq)` serve para verificar se ocorreu um erro associado ao fluxo de dados sendo usado. Um valor diferente de zero é a indicação do erro, que ocorre geralmente quando a operação previamante executada no fluxo falhou. O parâmetro `parq` é um ponteiro para o fluxo a ser testado. O programa 11.4 abre um arquivo para leitura, mas tenta escrever um caractere o que provoca um erro que é testado pela função `ferror`.

Listing 11.4: Uso da função `ferror()`.

```
#include <stdio.h>
int main ()
{
  FILE *pArq;
  pArq=fopen("MeusDados.txt","r");

  if (pArq==NULL)
  {
    printf("Erro abrindo arquivo.");
    return 1;
  }
  else
  {
    fputc ('x',pArq);
    if (ferror (pArq))
    {
      printf ("Erro escrevendo arquivo\n");
      fclose (pArq);
      return 1;
    }
  }
  return 0;
}
```

11.7 Lendo e Escrevendo Cadeias de Caracteres

As funções `fgets()` e `fputs()` servem para ler e escrever cadeias de caracteres em arquivos. Os protótipos das funções são:

```
int fputs(char *str, FILE *parq);

int fgets(char *str, int comp, FILE *parq);
```

A função `fputs()` escreve a cadeia de caracteres apontada por `str` no fluxo apontado por `parq`. O código nulo ao final da cadeia não é copiado para o fluxo.

O código correspondente à `EOF` será retornado se ocorrer um erro e um valor não negativo em caso de sucesso.

A função `fgets()` lê uma cadeia de caracteres do fluxo especificado por `parq` até que um caracter de nova linha seja encontrado ou `comp-1` caracteres sejam lidos. O caractere de nova linha interrompe a leitura. Observar que diferentemente de `gets()` o caracter de nova linha encontrado passa a fazer parte da cadeia que recebe um caracter nulo ao seu final. O ponteiro `str` é retornado caso a leitura ocorra sem erro. No caso de erro o ponteiro `str` recebe o valor `NULL`. Se o fim do arquivo for encontrado e nenhum caractere foi lido, o conteúdo de `str` é mantido e um `NULL` é retornado. O exemplo 11.5 mostra um exemplo de uso destas funções para ler e escrever cadeias de caracteres em um arquivo.

11.8 Entrada e Saída Formatada

As funções `fprintf()` e `fscanf()` são equivalentes as funções `printf()` e `scanf()` usadas até agora, sendo a única modificação o fato de que elas trabalham com fluxos de dados (arquivos). Os protótipos das duas funções são os seguintes:

```
int fprintf(FILE *parq, const char *formatacao, ...);
int fscanf(FILE *parq, const char *formatacao, ...);
```

onde `parq` é um ponteiro de arquivo recebido após uma chamada a `fopen()`.

Em leituras, a função retorna o número de itens lidos com sucesso. Esta contagem pode igualar o número esperado de leituras ou ser menor no caso de falha. Caso ocorra uma falha antes de que uma leitura possa ser feita com sucesso, `EOF` é retornado.

Em escritas, caso a operação de escrita tenha sucesso, o número total de caracteres escrito é retornado. Um número negativo é retornado em caso de falha.

Embora estas duas funções, por sua semelhança com `printf()` e `scanf()`, sejam maneiras convenientes de escrecer e ler dados de arquivos, elas têm a desvantagem de serem mais lentas do que uso de arquivos binários. A perda de tempo é devido ao fato dos dados serem gravados em ASCII, o que obriga a uma conversão dos dados a cada operação realizada. Em alguns casos o fato dos dados serem gravados em ASCII pode ser considerado um vantagem que se sobrepõe a desvantagem da redução de velocidade. Dados gravados em ASCII podem ser facilmente verificados pelos usuários, o que não acontece com dados em binário.

O exemplo 11.6 mostra o uso destas funções para ler e escrever vários tipos de dados em um arquivo.

Listing 11.5: Exemplo de leitura e escrita de cadeias de caracteres.

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

#define MAX 80

int main (void )
{
     char linha[MAX];
     FILE *pa;
     char *nome = "texto.txt";

     if (( pa = fopen(nome, "w+")) == NULL)
     {
          printf("\n\nNao foi possivel abrir o arquivo.\n");
          exit(1);
     }

     fgets(linha, MAX, stdin);
     while (!feof(stdin))
     {
          fputs(linha, pa);
          fgets(linha, MAX, stdin);
     }

     rewind(pa);  /* volta ao inicio do arquivo */
     printf("\nTerminei de escrever, agora vou ler.\n\n");
     fgets(linha, MAX, pa);
     while (!feof(pa))
     {
          puts(linha);
          fgets(linha, MAX, pa);
     }
    fclose(pa);

    return 0;
}
```

Listing 11.6: Exemplo de leitura e escrita de dados formatados.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 20

int main (void )
{
     char palavra[MAX];
    int i;
    float f;
     FILE *pa;
     char *nome = "format.txt";

     /* Abre o arquivo para leitura e escrita */
     if (( pa = fopen(nome, "w+")) == NULL)
     {
          printf("\n\nNao foi possivel abrir o arquivo.\n");
          exit(1);
     }
    puts ("Entre com uma palavra."); scanf ("%s", palavra);
    puts ("Entre com um numero inteiro."); scanf("%d", &i);
    puts ("Entre com um numero flutuante."); scanf("%f", &f);
    /* Escreve os dados no arquivo */
    fprintf(pa, "%s %d %f", palavra, i, f);

     rewind(pa);  /* volta ao inicio do arquivo */
     printf("\nTerminei de escrever, agora vou ler.\n");
     fscanf(pa, "%s %d %f", palavra, &i, &f);
    printf("Palavra lida: %s\n", palavra);
    printf("Inteiro lido: %d\n", i);
    printf("Float   lido: %f\n", f);
    fclose(pa);
     return 0;
}
```

11.9 Lendo e Escrevendo Arquivos Binários

As funções `fread` e `fwrite` são empregadas para leitura e escrita de dados em modo binário.

Os protótipos das funções são:

```
  size_t fread (void *ptr, size_t size, size_t nmemb, FILE *parq);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *parq);
```

A função `fread` lê `nmemb` objetos, cada um com `size` bytes de comprimento, do fluxo apontado por `stream` e os coloca na localização apontada por `ptr`. A função retorna o número de itens que foram lidos com sucesso. Caso ocorra um erro, ou o fim do arquivo foi atingido o valor de retorno é menor do que `nmemb` ou zero. Esta função não distingue entre um fim de arquivo e erro, portanto é aconselhavel o uso de `feof()` ou `ferror()` para determinar que erro ocorreu.

A função `fwrite` escreve `nmemb` elementos de dados, cada um com `size` bytes de comprimento, para o fluxo apontado por `stream` obtendo-os da localização apontada por `ptr`. `fwrite` retorna o número de itens que foram lidos com sucesso. Caso ocorra um erro, ou o fim do arquivo foi atingido o valor de retorno é menor do que `nmemb` ou zero.

O programa 11.7 ilustra como podemos escrever e ler dados de diferentes tipos em arquivos. Como um dos parâmetros da função é o número de bytes do dado a ser lido, é recomendado o uso de **sizeof**.

Uma das principais aplicações destas funções é a leitura e escrita de estruturas criadas pelos usuários. A gravação em binário da estrutura permite que o programador ao escrever ou ler do arquivo se preocupe somente com a estrutura como um todo e não com cada elemento que a compoõe. O programa 11.8 mostra um exemplo onde estruturas são gravadas e lidas de um arquivo. Neste exemplo é usado um laço para gravar uma estrutura de cada vez. No entanto, também é possível gravar todas as estruturas de uma vez mudando o terceiro parâmetro da função `fwrite()`. O laço seria substituído por

```
fwrite( &turma[i], sizeof (struct pessoa), MAX, pa);
```

Para testar erro basta verificar o valor retornado pela função. Caso ela tenha retornado um valor diferente de `MAX` ocorreu um erro.

Exercícios

11.1: Escreva um programa que abra um arquivo texto e conte o número de caracteres presentes nele. Imprima o número de caracteres na tela.

11.2: Considere um arquivo de dados do tipo texto com o seguinte conteúdo:

Listing 11.7: Exemplo de leitura e escrita na forma binária.

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

int main (void )
{
     int inum=10;
     float fnum=2.5;
     double pi=3.141516;
     char c='Z';
     FILE *pa;
     char *nome = "texto.bin";

     if (( pa = fopen(nome, "w+")) == NULL)
     {
          printf("\n\nErro ao abrir o arquivo para escrita.\n");
          perror("fopen");
          exit(1);
     }
     fwrite(&inum, sizeof(int), 1, pa);
     fwrite(&fnum, sizeof(float), 1, pa);
     fwrite(&pi, sizeof(double), 1, pa);
     fwrite(&c, sizeof(char), 1, pa);

     rewind(pa);

     fread(&inum, sizeof(int), 1, pa);
     fread(&fnum, sizeof(float), 1, pa);
     fread(&pi, sizeof(double), 1, pa);
     fread(&c, sizeof(char), 1, pa);

     printf("%d, %f, %f, %c\n", inum, fnum, pi, c);

     fclose(pa);
     return 0;
}
```

Listing 11.8: Exemplo de leitura e escrita de estruturas.

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

#define MAX 4

int main ()
{
     FILE *pa;
     char nome[40], linha[80];
     struct pessoa
      {
            char nome[40];
            int ano;
     } turma [MAX], back[MAX];
     int i;

     for (i=0; i<MAX; i++)
      {
         puts("Nome ? ");
         fgets(turma[i].nome, 40, stdin);
           turma[i].nome[strlen(turma[i].nome)-1]='\0';
         puts("Ano  ? "); fgets(linha, 80, stdin);
         sscanf(linha, "%d", &turma[i].ano);
     }

     for (i=0; i<MAX; i++)
      {
         printf("Nome = %s\n", turma[i].nome);
         printf("Ano  = %d\n\n", turma[i].ano);
     }

     puts("\nGravando\n");
     puts("Qual o nome do arquivo?"); fgets(nome, 40, stdin);
     nome[strlen(nome)-1]='\0';

     if (( pa = fopen(nome, "w+")) == NULL )
      {
        puts("Arquivo nao pode ser aberto");
        exit(1);
     }

     for (i=0; i<MAX; i++)
      {
         if (fwrite(&turma[i],sizeof(struct pessoa),1,pa) != 1)
            puts("Erro na escrita.");
     }

     rewind(pa);

     for (i=0; i<MAX; i++)
      {
         if (fread(&back[i],sizeof(struct pessoa),1,pa) != 1)
           {
            puts("Erro na escrita.");
            if (feof(pa)) break;
            puts("Erro na leitura.");
         }
     }
```

```
3
ZE SA
8.5
10.0
ANTONIO SANTOS
7.5
8.5
SEBASTIAO OLIVEIRA
5.0
6.0
```

O arquivo acima é um exemplo. Considere então que nestes arquivos a primeira linha contém o número de alunos no arquivo. As linhas seguintes contém os seguintes dados:

- nome do aluno com no máximo 50 caracteres;
- nota da primeira prova;
- nota da segunda prova.

Escreva um programa que imprima os nomes de todos os alunos que têm a média das duas notas menor que 7.0

11.3: Escreva um programa que grave os dados lidos no exercício anterior em um arquivo do tipo binário de acesso aleatório. O número que indica quantos alunos devem ser lidos (primeira linha do arquivo) não deve ser gravado no arquivo binário. Nesta questão o programa deve obrigatoriamente usar um vetor de estruturas do seguinte tipo:

```
typedef struc _ALUNO
{
    char nome[81];
    float n1, n2;
} ALUNO;
```

11.4: Escreva um programa que leia de um arquivo, cujo nome sera fornecido pelo usuario, um conjunto de numeros reais e armazena em um vetor. O tamanho máximo do vetor e dado pela constante `TAM_MAX`. A quantidade de numeros no arquivo varia entre 0 e `TAM_MAX`. O programa ao final calcula a media dos numeros lidos.

11.5: Faça um programa que leia 10 caracteres e armazene em um arquivo 10 cópias de cada um. Exiba o conteúdo

11.6: Crie uma função que receba duas strings como parâmetros, uma com um endereço de arquivo e outra com um texto qualquer, e adicione o texto no fim do arquivo.

11.7: Utilizando a função do exercício anterior faça um programa que gere 10 arquivos com o nome "Teste"e extensões "01", ..., "10". Cada um contendo o texto "Texto do arquivo [NÚMERO DO ARQUIVO]".

11.8: Escreva um programa para armazenar o telefone de 5 amigos. O programa deve obrigatoriamente usar a estrutura

```
typedef struct _PESSOA
{
     char nome[50];
     int idade;
     float altura;
     char telefone[10];
 } PESSOA;
```

a ser preenchida pelo usuário antes do armazenamento de cada registro.

11.9: Faça um programa que leia os dados do arquivo gerado no exercício anterior e salve-os num novo arquivo utilizando uma saída formatada como indicado abaixo.

FORMATO:
```
[nome] tem [idade] anos e [altura] de altura
Tel.:  [telefone]
```

11.10: Escreva um programa que leia um arquivo texto contendo linhas de dados. Em cada linha do arquivo há o nome de um aluno e duas notas. Estes dados estão separados por ponto e vírgula. Existe um ponto e vírgula ao final de cada linha. O formato dos dados e o seguinte:
```
ze sa; 10.0; 9.0;
antonio silva:  9.0; 7.0;
```

O programa deve ler estes dados e imprimir os valores lidos, a média das duas notas e se o aluno foi aprovado ou não ($media \geq 5$). O formato de saida é:
```
ze sa 10.0 8.0 9.0 aprovado
antonio silva 9.0 7.0 8.0 aprovado
```

11.11: Faça um programa que receba o nome de um arquivo e gere uma cópia.

11.12: Escreva um programa que compare dois arquivos especificados pelo usuário e imprima sempre que os caracteres dos dois arquivos coincidirem. Por exemplo:

```
arquivo1.c
Olá, pessoal!

arquivo2.c
Oi, como vai?
```

Neste caso, os caracteres na primeira e décima primeira posição são iguais nos dois arquivos. A saída do seu programa deve ser algo como:

```
1 - 0
11 - a
```

indicando que os primeiros caracteres dos arquivos são iguais (`0`) bem como o décimo primeiro (`a`)

Apêndice A

Tabela ASCII

A tabela ASCII (American Standard for Information Interchange) é usada por grande parte da indústria de computadores para a troca de informações e armazenamento de caracteres. Cada caracter é representado por um código de 8 bits. A Tabela A.1 mostra os códigos para a tabela ASCII de 7 bits. Existe uma table extendida para 8 bits que inclue os caracteres acentuados.

Para saber qual é o código de um caracter na base 10 junte o dígito da primeira coluna da tabela com o dígito da primeira linha da tabela. Por exemplo, o código da letra a minúscula é 97 na base 10.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	”	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	¡	=	¿	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	ˆ	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	—	}	˜	del		

Tabela A.1: Conjunto de caracteres ASCII

Os caracteres de controle listados acima, servem para comunicação com periféricos e controlar a troca de dados entre computadores. Eles têm o significado mostrado na Tabela A.2.

Carac	**Descrição**	**Carac**	**Descrição**
nul	Caractere nulo	soh	Começo de cabeçalho de transmissão
stx	Começo de texto	etx	Fim de texto
eot	Fim de transmissão	enq	Interroga
ack	Confirmação	bel	Sinal sonoro
bs	Volta um caractere	ht	Tabulação horizontal
lf	Passa para próxima linha	vt	Tabulação vertical
ff	Passa para próxima página	cr	Passa para início da linha
so	Shift-out	si	Shift-in
dle	Data line escape	dc1	Controle de dispositivo
dc2	Controle de dispositivo	dc3	Controle de dispositivo
dc4	Controle de dispositivo	nak	Negativa de confirmação
syn	Synchronous idle	etb	Fim de transmissão de um bloco
can	Cancela	em	Fim de meio de transmissão
sub	Substitui	esc	Escape
fs	Separador de arquivo	gs	Separador de grupo
rs	Separador de registro	us	Separador de unidade
sp	Espaço em branco		

Tabela A.2: Conjunto de códigos especiais ASCII e seus significados

Apêndice B

Palavras Reservadas

Palavras reservadas, também as vezes chamadas de palavra chave, servem para propósitos especiais nas linguagens de programação. Servem para declarar tipos de dados ou propriedades de um objeto da linguagem, indicar um comando além de várias outras funções. Palavras reservadas não podem ser usadas como nomes de variáveis ou funções.

asm: Indica que código escrito em *assembly* será inserido junto comandos `C`.

auto: Modificador que define a classe de armazenamento padrão.

break: Comando usado para sair incondicionalmente dos comandos `for`, `while`, `switch`, and `do...while`.

case: Comando usado dentro do comando `switch`.

char: O tipo de dados mais simples em `C`, normalmente usado para armazenar caracteres.

const: Modificados de dados que impede que uma variáveis seja modificada. Esta palavra não existia nas primeiras versões da linguagem `C` e foi introduzida pelo comitê ANSI `C`. Veja `volatile`.

continue: Comando que interrompe os comandos de repetição `for` , `while` , ou `do...while` e faz que eles passem para a próxima iteração.

default: É usado dentro do comando `switch` para aceitar qualquer valor não definido previamente com um comando `case`.

do: Comando de repetição usado em conjunto com o comando `while` . Pela definição do comando o laço é sempre executado pelo menos uma vez.

double: Tipo de dados usado para armazenar valores de ponto flutuante em precisão dupla.

else: Comando que indica um bloco de comandos alternativo que deve ser executado quando a condição testada pelo comando `if` foi avaliada como FALSA.

enum: Tipo definido pelo usuário que permite a definição de variáveis que irão aceitar somente certos valores.

extern: Modificador de dados que indica que uma variável irá ser declarada em outra área do programa.

float: Tipo usado para armazenar valores de ponto flutuante.

for: Comando de repetição que contém inicialização de variáveis, incremento e seções condicionais. Em `C` o comando `for` é um comando de repetição extremamente flexível, permitindo inúmeras possibilidades.

goto: Comando que causa um pulo para uma posição do programa marcada com um rótulo.

if: Comando de testes usado para mudar o fluxo do programa baseada em uma decisão VERDADE/FALSO.

int: Tipo de dados usado para armazenar valores inteiros.

long: Tipo de dados usado para armazenar valores inteiros com precisão maior do que o tipo `int`. Nos computadores modernos o tipo `long` tem a mesma precisão que o tipo `int` e são usados 4 bytes.

register: Especificador de classe de armazenamento que pede que, caso seja possível, uma variável deve ser armazenada nos registradores do processador.

return: Comando que causa o fluxo de instruções do programa abandonar a função em execução e retornar para a função que chamou. Também pode ser usado para retornar um único valor.

short: Tipo de dados usado para armazenar valores inteiros em precisão menor do que o tipo `int`. Neste tipo 2 bytes são usados para armazenar os dados.

signed: Modificador usado para indicar que uma variável pode armazenar tanto valores positivos como negativos.

sizeof: Operador que retorna o tamanho em bytes do item fornecido.

static: Modificador usado para significar que o compilador deve preparar o código de forma a reter o valor da variável.

struct: Usado para combinar `C` variáveis de tipos diferentes na mesma estrutura.

switch: Comando de desvio usado para permitir que o fluxo do programa possa ser mudado para várias direções diferentes. Usado em conjunto com o comando `case`.

typedef: Modificador usado para criar novos nomes para tipos já existentes.

union: Palavra chave usada para permitir múltiplas variáveis partilharem o mesmo espaço na memória.

unsigned: Modificador usado para significar que uma variável conterá somente valores positivos.

void: Palavra usada para significar que ou a função não retorna nada ou que um ponteiro deve ser considerado genérico ou ser capaz de apontar para qualquer tipo de dados.

volatile: Modificador que significa que uma variável pode ser alterada.

while: Comando de teste que executa uma seção de código enquanto uma condição retorna VERDADE.

Em adição a estas as seguintes palavras são reservadas em `C++`:

`catch`, `inline`, `template`, `class`, `new`, `this`, `delete`, `operator`, `throw`, `except`, `private`, `try`, `finally`, `protected`, `virtual`, `friend`, `public`.

Caso queira escrever programas que possam ser convertidas para a linguagem `C++` é aconselhável não usá-las.

Bibliografia