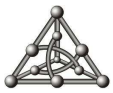
# Programação de Computadores I

Faculdade de Computação

UFMS



2009

# SUMÁRIO

AULA 0

# BOAS VINDAS!

---

Esta aula é reservada para ambientar o(a) estudante no laboratório, fazendo-o(a) *logar* em um computador, experimentar o ambiente de trabalho e a ferramenta de suporte a disciplinas da faculdade (SSD-FACOM), onde deverá cadastrar seu perfil de usuário(a) e se cadastrar nas disciplinas de seu semestre. Essa será a ferramenta de interface entre o(a) professor(a) e os(as) estudantes durante todo o curso.

Além disso, as informações gerais sobre a disciplina, tais como a forma de avaliação, datas de provas e trabalhos, critérios de avaliação e média final, além da conduta ética esperada dos(as) estudantes, são explicitadas nesse ambiente.

AULA 1

# BREVE HISTÓRIA DO COMPUTADOR

Nesta aula abordaremos uma pequena história do computador, desde seu surgimento até os nossos dias. Sabemos, no entanto, que esta é na verdade uma tentativa e que fatos históricos relevantes podem ter sido excluídos involuntariamente. Continuaremos a trabalhar no sentido de produzir um texto cada vez mais amplo e completo, buscando informações em outras fontes ainda desconhecidas.

Este texto é baseado principalmente nas páginas[1] da Rede Mundial de Computadores [2, 3, 9, 13, 18] e nos livros de Philippe Breton [1] e Russel Shackelford [14].

## 1.1 Pré-história

Quando começou a realizar tarefas mecânicas para sua comodidade e bem estar, o ser humano – predominantemente as mulheres – tinha de realizar, hora após hora, dia após dia, todos os cálculos repetitivos em tarefas das ciências, do comércio e da indústria. Tabelas e mais tabelas eram preenchidas com os resultados desses cálculos, para que não fossem sempre refeitos. No entanto, erros nesses cálculos eram freqüentes devido, em especial, ao tédio e à desconcentração. Além disso, esses cálculos não eram realizados de forma rápida. Assim, muitos inventores tentaram por centenas de anos construir máquinas que nos ajudassem a realizar essas tarefas.

Figura 1.1: Ábaco.

O ábaco[2] é provavelmente o mais antigo instrumento conhecido de auxílio ao ser humano em cálculos matemáticos. O ábaco mais antigo foi descoberto na Babilônia, hoje conhecida como Iraque, e data do ano de 300 A.C. Seu valor está especialmente no fato de auxiliar a memória humana durante uma operação aritmética. Alguém bem treinado no uso de um ábaco pode executar adições e subtrações com a mesma velocidade de quem usa uma calculadora eletrônica, mas outras operações básicas são mais lentas. Esse instrumento ainda é consideravelmente utilizado em países do Oriente.

[1] Essas páginas são muito legais! Vale a pena visitá-las!

[2] *Abacus* é uma palavra em latim que tem sua origem nas palavras gregas *abax* ou *abakon*, que quer dizer "tabela" e que, por sua vez, tem sua origem possivelmente na palavra semítica *abq*, que significa "areia" (do livro *The Universal History of Numbers* de Georges Ifrah, Wiley Press 2000).

John Napier[3], em seu trabalho *Mirifici Logarithmorum Canonis Descriptio* de 1614, inventou o *logaritmo*, uma ferramenta que permitia que multiplicações fossem realizadas via adições e divisões via subtrações. A "mágica" consistia na consulta do logaritmo de cada operando, obtido de uma tabela impressa. A invenção de John Napier deu origem à régua de cálculo, instrumento de precisão construído inicialmente na Inglaterra em 1622 por William Oughtred[4] e utilizado nos projetos Mercury, Gemini e Apollo da NASA, que levaram o homem à lua.

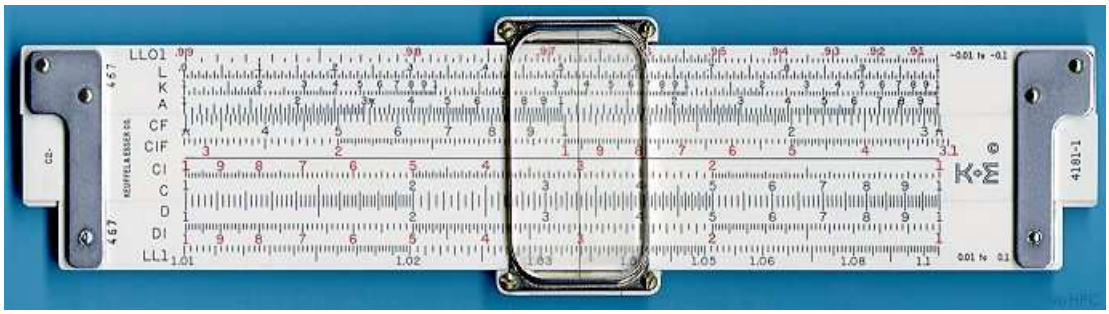



Figura 1.2: Régua de cálculo.

Leonardo da Vinci[5] e Wilhelm Schickard[6], nos séculos XVI e XVII, respectivamente, foram os primeiros a projetar máquinas de cálculo baseadas em engrenagens. A máquina proposta por da Vinci nunca saiu do papel, mas a máquina de Schickard é conhecida como a primeira calculadora mecânica automática ou o primeiro computador não programável. O sucesso de uma máquina como essa só foi obtido por Blaise Pascal[7] em 1642, que aos 19 anos construiu a *Pascaline* para ajudar seu pai, um recolhedor de taxas, a somar quantias. A Pascaline realizava apenas adições e não era muito precisa. Pouco tempo após Pascal ter proposto sua máquina de calcular, o alemão Gottfried Leibniz[8], co-inventor do *Cálculo* juntamente com Isaac Newton[9], planejou construir uma calculadora[10] com as quatro operações básicas e que, ao invés de engrenagens, empregava cilindros dentados com 10 dentes, possibilitando seu trabalho no sistema numérico decimal. O século seguinte tem poucas inovações diretamente relacionadas aos computadores, como a escala de temperaturas Fahrenheit[11], o telégrafo e a eletricidade, descoberta por Benjamin Franklin[12].

Joseph Jacquard[13] criou, em 1801, um tear que podia tecer a partir de um padrão lido automaticamente de cartões de madeira perfurados e conectados por cordões. Descendentes desses cartões perfurados são utilizados até hoje em algumas aplicações. A invenção de Jacquard incrementou a produtividade de tecidos da época, mas, em contrapartida, gerou grande de-

[3] John Napier (1550 – 1617), nascido na Escócia, matemático, físico, astrônomo e 8º Lorde de Merchistoun.
[4] William Oughtred (1575 – 1660), nascido na Inglaterra, matemático.
[5] Leonardo di ser Piero da Vinci (1452 – 1519), nascido na Itália, erudito, arquiteto, anatomista, escultor, engenheiro, inventor, matemático, músico, cientista e pintor.
[6] Wilhelm Schickard (1592 – 1635), nascido na Alemanha, erudito.
[7] Blaise Pascal (1623 – 1662), nascido na França, matemático, físico e filósofo.
[8] Gottfried Wilhelm Leibniz (1646 – 1716), nascido na Alemanha, erudito.
[9] Sir Isaac Newton (1643 – 1727), nascido na Inglaterra, físico, matemático, astrônomo e filósofo.
[10] Ele chamou sua máquina de *stepped reckoner*.
[11] Daniel Gabriel Fahrenheit (1686 – 1736), nascido na Alemanha, físico e engenheiro.
[12] Benjamin Franklin (1706 – 1790), nascido nos Estados Unidos, jornalista, editor, autor, filantropo, abolicionista, funcionário público, cientista, diplomata e inventor. Foi também um dos líderes da Revolução Norte-americana.
[13] Joseph Marie Jacquard (1752 – 1834), nascido na França, alfaiate e inventor.

semprego de operários da indústria têxtil.

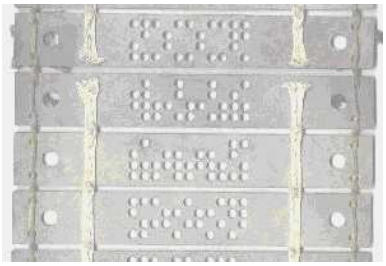


Figura 1.3: Cartões perfurados da máquina de Jacquard.

A *Máquina de Diferenças*[14], projetada por Charles Babbage[15], era uma máquina de calcular a vapor e foi criada com o objetivo de atender à estratégia expansionista do governo inglês, que tinha a ambição de se tornar o maior império do planeta. Esse foi o projeto mais caro financiado pelo governo da Inglaterra até então, mas após dez anos de tentativas infrutíferas, o projeto ruiu e a máquina não foi terminada. Todavia, Babbage não desistiu e projetou uma segunda máquina chamada *Máquina Analítica*[16] que seria alimentada por seis máquinas a vapor, teria o tamanho de uma casa e era uma máquina de propósito mais geral, já que seria programável por meio de cartões perfurados. Além disso, sua invenção também permitiria que os cartões perfurados, agora de papel, fossem empregados como um dispositivo de armazenamento. Essa máquina teria ainda um dispositivo que a distinguia das calculadoras e a aproximava do que conhecemos hoje como computadores: uma sentença condicional.

Ada Byron[17] tornou-se amiga de Charles Babbage e ficou fascinada com o as idéias da Máquina Analítica. Apesar de nunca ter sido construída, Ada escreveu diversos programas para essa máquina e entrou para a história como a primeira programadora de um computador. Ada inventou a sub-rotina e reconheceu a importância do laço como uma estrutura de programação.

Em 1890 o governo norte-americano tinha de realizar o censo da população dos Estados Unidos e a previsão era de que, da forma como tinha sido realizado o último, o censo de 1890 levaria cerca de 7 anos e meio até ser finalizado. Um prêmio foi então anunciado para o inventor que ajudasse a automatizar o censo de 1890. O prêmio foi vencido por Herman Hollerith[18], que propôs a Mesa de Hollerith[19], uma máquina que consistia de um leitor de cartões perfurados que era sensível aos buracos nos cartões, um mecanismo de engrenagens que podia contar e um grande expositor de indicadores para mostrar os resultados da computação. Essa máquina foi projetada com sucesso e o censo de 1890 terminou em somente 3 anos. A partir daí Hollerith fundou uma empresa chamada TABULATING MACHINE COMPANY. Em 1911 sua empresa se une a outras duas e se torna a COMPUTING TABULATING RECORDING CORPORATION – CTR. Sob a presidência de Thomas Watson[20], a empresa foi renomeada INTERNATIONAL BUSINESS MACHINES – IBM em 1924.

[14] Do inglês *Difference Engine*.
[15] Charles Babbage (1791 – 1871), nascido na Inglaterra, matemático, filósofo, engenheiro mecânico e precursor da Ciência da Computação.
[16] Do inglês *Analytic Engine*.
[17] Augusta Ada Byron (1815 – 1852), nascida na Inglaterra, conhecida como a primeira programadora da história. Depois de seu casamento passou a se chamar Augusta Ada King, Condessa de Lovelace.
[18] Herman Hollerith (1860 – 1929), nascido nos Estados Unidos, estatístico e empresário.
[19] Do inglês *Hollerith desk*.
[20] Thomas John Watson (1874 – 1956), nascido nos Estados Unidos, empresário.

Figura 1.4: Mesa de Hollerith.

## 1.2 Século XX

A explosão dos computadores ocorreu no século XX. Até a metade desse século, computadores eletromecânicos e os primeiros computadores totalmente eletrônicos foram projetados com fins militares, para realizar cálculos balísticos e decifrar códigos dos inimigos. Eminentes cientistas, que deram origem a quase tudo do que chamamos de Ciência da Computação, estiveram envolvidos nesses projetos. A partir da segunda metade do século, a explosão dos computadores eletrônicos se deu, quando o computador pessoal passou a fazer parte de nosso dia a dia.

### 1.2.1 Computadores Eletromecânicos

No início do século XX a IBM continuava produzindo calculadoras mecânicas que eram muito utilizadas em escritórios especialmente para realizar somas. No entanto, a demanda por calculadoras mecânicas que realizassem cálculos científicos começou a crescer, impulsionada especialmente pelas forças armadas norte-americanas e pelo seu envolvimento nas Primeira e Segunda Guerras Mundiais.

Konrad Zuse[21] propôs a série "Z" de computadores e, destacadamente, em 1941, o primeiro computador digital binário programável por fita, o Z3, de 22 bits de barramento, relógio interno com velocidade de 5 Hz e 2.000 relés. O Z4, construído em 1950, é o primeiro computador comercial, alugado pelo Instituto Federal de Tecnologia da Suíça (*Eidgenössische Technische Hochschule Zürich – ETH Zürich*). Zuse também projetou uma linguagem de programação de alto nível, a Plankalkül, publicada em 1948. Devido às circunstâncias da Segunda Guerra Mundial, seu trabalho ficou conhecido muito posteriormente nos Estados Unidos e na Inglaterra, em meados dos anos 60.

O primeiro computador programável construído nos Estados Unidos foi o Mark I, projetado pela universidade de Harvard e a IBM em 1944. O Mark I era um computador eletromecânico composto por interruptores, relés, engates e embreagens. Pesava cerca de 5 toneladas, incorporava mais de 800 quilômetros de fios, media 2,5 metros de altura por 15,5 metros de

[21] Konrad Zuse (1910 – 1995), nascido na Alemanha, engenheiro e pioneiro da computação.

comprimento e funcionava através de um motor de 5 cavalos-vapor. O Mark I podia operar números de até 23 dígitos. Podia adicionar ou subtrair esses números em $3/10$ de segundo, multiplicá-los em 4 segundos e dividi-los em 10 segundos[22]. Apesar de ser um computador enorme, com aproximadamente 750 mil componentes, o Mark I podia armazenar apenas 72 números e sua velocidade de armazenamento e recuperação era muito lenta, uma motivação e um fator preponderante para substituição posterior do computador eletromecânico pelo computador eletrônico. Apesar disso, o Mark I funcionou sem parar por quinze anos.

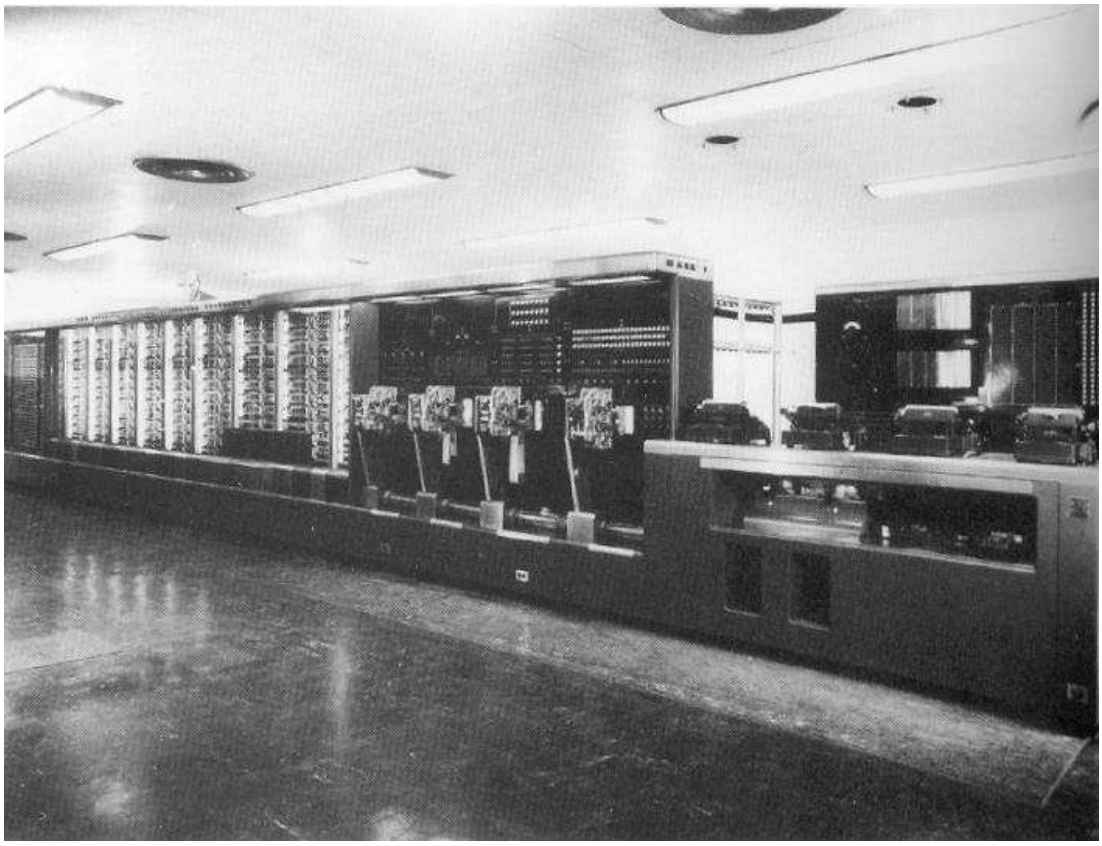


Figura 1.5: O computador eletromecânico Harvard Mark I.

Uma das primeiras pessoas a programar o Mark I foi uma mulher, Grace Hopper[23], que em 1953 inventou a primeira linguagem de programação de alto nível chamada Flow-matic, vindo a se tornar posteriormente a linguagem COBOL. Uma linguagem de programação de alto nível é projetada com o objetivo de ser mais compreensível ao ser humano, diferentemente da linguagem binária, de baixo nível, compreendida pelo computador. No entanto, uma linguagem de alto nível necessita de um programa – chamado compilador – que traduz um programa em linguagem de alto nível para um programa em linguagem de baixo nível, de modo que o Mark I pudesse executá-lo. Portanto, Grace Hopper também construiu o primeiro compilador conhecido.

[22] É espantoso observar que após 45 anos da fabricação do Mark I, um computador podia realizar uma operação de adição em $1/10^9$ de segundo.

[23] Grace Murray Hopper (1906 – 1992), nascida nos Estados Unidos, cientista da computação e oficial da Marinha dos Estados Unidos.

### 1.2.2 Computadores Eletrônicos

No princípio da era eletrônica, logo no começo do século XX, os computadores substituíram os interruptores e as engrenagens eletromecânicas pelas válvulas. Em seguida, a revolução da microeletrônica permitiu que uma quantidade enorme de fiação produzida de forma artesanal pudesse ser produzida industrialmente como um circuito integrado. A principal vantagem de um circuito integrado é que milhões de transistores podem ser produzidos e interconectados em um processo industrial de larga escala. Além disso, os transistores são minúsculos comparados às válvulas, além de muito mais confiáveis.

As contribuições também vieram da teoria. A Ciência da Computação não seria a mesma não fossem as contribuições de um matemático inglês nascido em 1912, conhecido como "Pai da Ciência da Computação". No início do século XX, Alan Turing[24] formalizou o conceito de algoritmo e computação através de um modelo teórico e formal chamado de *máquina de Turing* e formulou a tese que qualquer modelo de computação prático ou tem capacidades equivalentes às de uma máquina de Turing ou tem um subconjunto delas[25]. Turing, assim como os melhores cérebros da época, também se envolveu com a guerra. Na Segunda Guerra Mundial, no centro de criptoanálise de Bletchley Park, Inglaterra, trabalhou como criptoanalista na decifragem do código produzido pela máquina Enigma da Alemanha nazista. Em 1947 participou do desenvolvimento de um computador eletrônico chamado Manchester Mark I na Universidade de Manchester, Inglaterra. Em 1952 foi acusado de "atos de indecência nojenta" por assumir sua relação com um homem em Manchester. Turing morreu em 1956 antes de completar 44 anos após comer uma maçã envenenada. Sua morte foi registrada como suicídio.

Alguns dispositivos da primeira metade do século XX reclamam o título de primeiro computador digital eletrônico. O Z3 de Konrad Zuse, como já mencionado, era o primeiro computador eletromecânico de propósito geral. Foi o primeiro computador a usar aritmética binária, era Turing-completo e era totalmente programável por fita perfurada. No entanto, usava relés em seu funcionamento e, portanto, não era eletrônico. Entre 1937 e 1942 foi concebido o Computador Atanasoff-Berry (ABC), projetado por John Atanasoff[26] e Clifford Berry[27], que continha elementos importantes da computação moderna como aritmética binária e válvulas, mas sua especificidade e impossibilidade de armazenamento de seus programas o distinguem dos computadores modernos. O Colossus também foi um computador construído para fins militares entre os anos de 1943 e 1944. Tommy Flowers[28] o projetou para auxiliar criptoanalistas ingleses a decifrar mensagens criptografadas produzidas pelas máquinas Lorenz SZ 40 e 42 da Alemanha nazista na Segunda Guerra Mundial. O Colossus, apesar de utilizar a mais recente tecnologia eletrônica de sua época, não era um computador de propósito geral, era programável de forma limitada e não era Turing-completo.

Figura 1.6: Válvula.

[24] Alan Mathison Turing (1912 – 1954), nascido na Inglaterra, matemático, lógico e criptoanalista.
[25] Esta afirmação é chamada Tese de Church-Turing. Uma linguagem de programação ou um computador abstrato é *Turing-completo* se satisfaz essa tese.
[26] John Vincent Atanasoff (1903 – 1995), nascido nos Estados Unidos, físico.
[27] Clifford Edward Berry (1918 – 1963), nascido nos Estados Unidos, engenheiro elétrico.
[28] Thomas Harold Flowers (1905 – 1998), nascido na Inglaterra, engenheiro.

**De 1940 a 1960**

O título de primeiro computador digital de propósito geral e totalmente eletrônico é em geral dado ao ENIAC (*Electronic Numerical Integrator and Calculator*). Esse computador foi construído na Universidade da Pensilvânia entre 1943 e 1945 pelos professores John Mauchly[29] e John Eckert[30] obtendo financiamento do departamento de guerra com a promessa de construir uma máquina que substituiria "todos" os computadores existentes, em particular as mulheres que calculavam as tabelas balísticas para as armas da artilharia pesada do exército. O ENIAC ocupava uma sala de 6 por 12 metros, pesava 30 toneladas e usava mais de 18 mil tubos a vácuo, que eram muito pouco confiáveis e aqueciam demasiadamente.

Figura 1.7: O primeiro computador eletrônico ENIAC.

Apesar de suas 18 mil válvulas, o ENIAC podia armazenar apenas 20 números por vez. No entanto, graças à eliminação de engrenagens, era muito mais rápido que o Mark I. Por exemplo, enquanto uma multiplicação no Mark I levava 6 segundos, no ENIAC levava 2,8 milésimos de segundo. A velocidade do relógio interno do ENIAC era de 100 mil ciclos por segundo[31]. Financiado pelo exército dos Estados Unidos, o ENIAC tinha como principal tarefa verificar a possibilidade da construção da bomba de hidrogênio. Após processar um programa armazenado em meio milhão de cartões perfurados por seis semanas, o ENIAC infelizmente respondeu que a bomba de hidrogênio era viável.

[29] John William Mauchly (1907 – 1980), nascido nos Estados Unidos, físico e pioneiro da computação.
[30] John Adam Presper Eckert Jr. (1919 – 1995), nascido nos Estados Unidos, físico e pioneiro da computação.
[31] Diríamos que o ENIAC era um computador com velocidade de 100 KHz.

O ENIAC mostrou-se muito útil e viável economicamente, mas tinha como um de seus principais defeitos a dificuldade de reprogramação. Isto é, um programa para o ENIAC estava intrinsecamente relacionado a sua parte física, em especial a fios e interruptores. O EDVAC (*Electronic Discrete Variable Automatic Computer*) foi projetado em 1946 pela equipe de John Mauchly e John Eckert, que agregou o matemático John von Neumann[32] e tinha como principais características a possibilidade de armazenar um programa em sua memória e de ser um computador baseado no sistema binário. John von Neumann publicou um trabalho[33] descrevendo uma arquitetura de computadores em que os dados e o programa são mapeados no mesmo espaço de endereços. Essa arquitetura, conhecida como *arquitetura de von Neumann* é ainda utilizada nos processadores atuais, como por exemplo no Pentium IV.

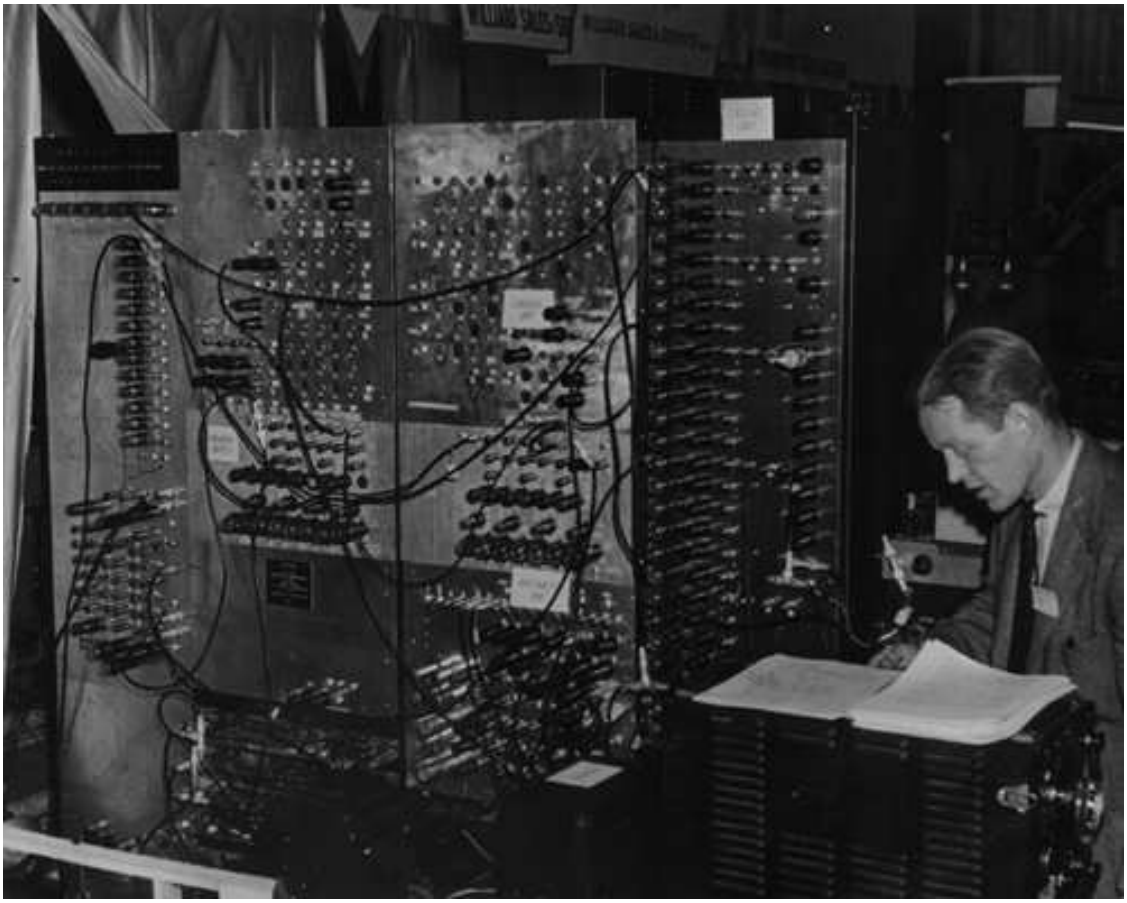


Figura 1.8: O EDVAC.

Nos anos 50, os computadores eram nada populares e pertenciam a algumas poucas universidades e ao governo norte-americano. No início da década, John Eckert e John Mauchly deixam a Universidade de Pensilvânia por problemas de patenteamento de suas invenções e

[32] Margittai Neumann János Lajos (1903 – 1957), nascido na Áustria-Hungria, matemático e erudito. John von Neumann tem muitas contribuições em diversas áreas. Na Ciência da Computação, além da arquitetura de von Neumann, propôs os autômatos celulares. Mas, infelizmente, também foi um dos cientistas a trabalhar no Projeto Manhatan, responsável pelo desenvolvimento de armas nucleares, tendo sido responsável pela escolha dos alvos de Hiroshima e Nagazaki no Japão na Segunda Guerra Mundial, onde explodiriam as primeiras bombas nucleares da história e pelo cálculo da melhor altura de explosão para que uma maior destruição fosse obtida.

[33] *First Draft of a Report on the EDVAC*.

fundam sua própria empresa de computadores, a ECKERT-MAUCHLY COMPUTER CORPORATION. Em 1951 projetam o UNIVAC, de *UNIVersal Automatic Computer*, o primeiro computador eletrônico comercial produzido em larga escala e que utilizava fita magnética. O primeiro UNIVAC foi vendido para o escritório do censo populacional dos Estados Unidos. Mas devido ao domínio da IBM, que em 1955 vendia mais computadores que o UNIVAC, a empresa de John Eckert e John Mauchly passa por muitas dificuldades até ser fechada e vendida.

Em 1955 a BELL LABORATORIES, uma empresa de tecnologia fundada em 1925, produz o primeiro computador à base de transistores. Os transistores eram menores, mais rápidos e aqueciam muito menos que as válvulas, o que tornava computadores à base de transistores muito mais eficientes e confiáveis. Em 1957 a IBM anuncia que não usaria mais válvulas e produz seu primeiro computador contendo 2.000 transistores. Em 1958, descobertas experimentais que mostravam que dispositivos semicondutores podiam substituir as válvulas e a possibilidade de produzir tais dispositivo em larga escala, possibilitaram o surgimento do primeiro circuito integrado, ou microchip, desenvolvido simultaneamente por Jack Kilby[34] da TEXAS INSTRUMENTS e por Robert Noyce[35] da FAIRCHILD SEMICONDUCTOR. Um circuito integrado é um circuito eletrônico miniaturizado, consistindo basicamente de dispositivos semicondutores, produzido na superfície de um fino substrato de material semicondutor.

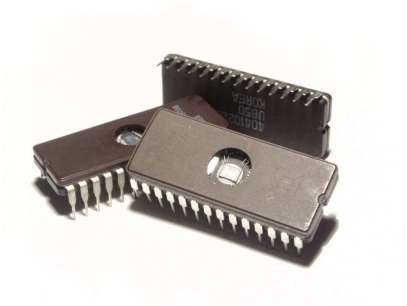


Figura 1.9: Um circuito integrado.

**De 1960 a 1980**

Figura 1.10: O primeiro *mouse*.

Nos anos 60, as empresas produziam computadores de grande porte ou *mainframes*. Esses computadores eram usados por grandes organizações para aplicações massivas, tipicamente para processamento de enormes quantidades de dados tais como censo populacional, estatísticas industriais ou comerciais e processamento de transações financeiras. Faz parte desse conjunto a série IBM 1400 da IBM e a série UNIVAC 1100 da SPERRY-RAND. Naqueles anos, juntamente com a produção de computadores, dispositivos, periféricos e linguagens de programação foram também desenvolvidos, além de propostas de padronização desses novos recursos. Em 1963, Douglas Engelbart[36] inventou e patenteou o primeiro *mouse* de computador. No mesmo ano é desenvolvido o código padrão norte-americano para troca de informações[37] para padronizar a troca de dados entre computadores. Em 1967, a IBM cria o primeiro disco flexível[38] para armazenamento de da-

[34] Jack St. Clair Kilby (1923 – 2005), nascido nos Estados Unidos, engenheiro elétrico. Ganhador do Prêmio Nobel de Física em 2000 por sua contribuição na invenção do circuito integrado.

[35] Robert Noyce (1927 – 1990), nascido nos Estados Unidos, físico. Também conhecido como "o Mestre do Vale do Silício".

[36] Douglas C. Engelbart (1925–), nascido nos Estados Unidos, engenheiro elétrico e inventor.

[37] Do inglês *American Standard Code for Information Interchange* – ASCII.

[38] Disquete ou disco flexível, do inglês *floppy disk*.

dos. Além disso, as linguagens de programação BASIC, FORTRAN e COBOL foram propostas nessa década. Com o desenvolvimento da indústria de computadores fervilhando, Gordon Moore[39], através de observações empíricas, afirma em 1965 que o número de transistores em um circuito integrado duplicaria a cada 24 meses. Essa afirmação fica conhecida depois como "Lei de Moore". Em 1968, Gordon Moore e Robert Noyce fundam a INTEL CORPORATION, uma empresa fabricante de processadores. Em 1969, um grupo de ex-executivos da FAIRCHILD SEMICONDUCTOR funda uma empresa de circuitos integrados chamada ADVANCED MICRO DEVICES, INC., – AMD. Em 1969, a AT&T BELL LABORATORIES desenvolve o Unix, um excelente sistema operacional utilizado ainda hoje especialmente em servidores. Entre 1969 e 1970, a primeira impressora matricial e a primeira impressora à laser são produzidas.

Figura 1.11: Máquinas perfuradoras de cartões IBM 26.

Como já comentamos, os computadores dos anos 60 e 70 eram computadores de grande porte estabelecidos em órgãos governamentais, universidades e algumas grandes empresas. Esses computadores podiam ser programados de duas maneiras distintas. A primeira maneira é conhecida como compartilhamento de tempo[40] onde o computador dá a cada usuário uma fatia de tempo de processamento, comportando, naquela época, 100 usuários ativos simultaneamente. A segunda maneira é chamada de modo de processamento em lote[41] onde o computador dá atenção total a um programa, mas não há usuários ativos. O programa, no momento em que o computador se dispõe a processá-lo, é lido a partir de cartões perfurados que foram previamente preparados em uma máquina perfuradora de cartões.

A década de 70 também foi muito prolífica no desenvolvimento da computação. Lingua-

[39] Gordon Earle Moore (1929−), nascido nos Estados Unidos, químico e físico.
[40] Do inglês *time sharing processing*.
[41] Do inglês *batch processing*.

gens de programação usadas até hoje foram projetadas nessa época como Pascal, concebida em 1971 por Niklaus Wirth[42], e C, proposta em 1972 por Dennis Ritchie[43]. Avanços no projeto de dispositivos também foram obtidos. Datam dessa época o CD[44], criado em 1972, o primeiro leitor de discos flexíveis, introduzido pela TOSHIBA em 1974, e o primeiro disco flexível de 5 1/4" é produzido em 1978. Ainda, o ano de 1975 marca o nascimento de uma das maiores e mais bem sucedidas empresas do mundo, a MICROSOFT, criada por Bill Gates[45] e Paul Allen[46]. No mesmo ano, Steve Wozniak[47] e Steve Jobs[48] fundam a APPLE COMPUTERS, também uma das maiores empresas fabricantes de computadores do mundo.

Mas nenhum dos avanços dessa época foi tão significativo como aquele obtido pela fabricante de microprocessadores INTEL, que conseguiu colocar um "computador inteiro" em uma única pastilha de silício logo no início da década. Em 1970 a INTEL colocou em produção o microprocessador INTEL 4004, com barramento de 4 bits, velocidade de 108 KHz e 2.300 transistores. A encomenda inicial partiu da empresa japonesa BUSICOM, que tinha a intenção de utilizar o INTEL 4004 em uma linha de calculadoras científicas. Porém, o processador mais notável que foi disponibilizado em seguida pela INTEL foi o INTEL 8080 em 1974, com barramento de 8 bits e velocidade de processamento de 2 MHz. A INTEL vendia esse processador por 360 dólares como uma brincadeira com a IBM, já que seu computador de grande porte IBM S/360 custava milhões de dólares. Em 1975, o INTEL 8080 foi empregado no Altair 8800 da MICRO INSTRUMENTATION AND TELEMETRY SYSTEMS, considerado o primeiro computador pessoal[49], tinha 1 Kbyte de memória principal e tinha de ser construído a partir de um kit de peças que chegavam pelo correio. Bill Gates, que acabara de ingressar na Universidade de Harvard, largou seus estudos para concentrar esforços na escrita de programas para esse computador.

Figura 1.12: Altair 8800.

Em 1976, três computadores pessoais concorrentes são colocados à venda. O primeiro computador pessoal que já vinha montado pela fábrica, o Apple I, é lançado pela APPLE COMPUTERS. Logo em seguida, em 1977, a APPLE inicia a produção do Apple II, o primeiro computador pessoal com monitor colorido, projetado com o processador 6502 da MOS TECHNOLOGY, de 8 bits de barramento, 1 MHz de velocidade, 4 Mbytes de memória principal e interface de

[42] Niklaus E. Wirth (1934−), nascido na Suíça, engenheiro eletrônico e cientista da computação.
[43] Dennis MacAlistair Ritchie (1941−), nascido nos Estados Unidos, físico e matemático.
[44] Do inglês *compact disk*.
[45] William Henry Gates III (1955−), nascido nos Estados Unidos, empresário.
[46] Paul Gardner Allen (1953−), nascido nos Estados Unidos, empresário.
[47] Stephan Gary Wozniak (1950−), nascido nos Estados Unidos, engenheiro da computação e filantropo.
[48] Steven Paul Jobs (1955−), nascido nos Estados Unidos, empresário.
[49] Do inglês *personal computer* – PC.

áudio-cassete para armazenamento e recuperação de dados em fita cassete. Era o computador padrão empregado na rede de ensino dos Estados Unidos nos anos 80 e 90.


Figura 1.13: O computador pessoal Apple I.

Além do Apple I, um outro computador pessoal que entrou no mercado de computadores pessoais a partir de 1976 foi o TRS-80 Model I da TANDY CORPORATION. Esse computador continha um microprocessador de 8 bits, o Zilog Z80, de 1,77 MHz de velocidade e tinha memória de 4 Kbytes.

Também em 1976, a INTEL lança o processador INTEL 8086, ainda conhecido como P1, com barramento de 16 bits, velocidade de 5 MHz e 29.000 transistores, que dá origem à arquitetura de processadores x86. O INTEL 8088 lançado em 1979 é baseado no INTEL 8086, mas tem barramento de dados de 8 bits, permitindo compatibilidade com os processadores anteriores. Esse processador foi utilizado como o processador padrão da linha de computadores pessoais da IBM, os IBM PCs, a partir de 1981. Já em 1980, a IBM contrata Bill Gates e Paul Allen para desenvolver um sistema operacional para o IBM PC, que eles denominaram *Disk Operating System* – DOS. O DOS, ou MS-DOS, era um sistema operacional com interface de linha de comandos.

### De 1980 a 2000

Com o mercado de computadores em contínuo crescimento, os anos 80 também foram de muitas inovações tecnológicas na área. O crescimento do número de computadores pessoais nos Estados Unidos nessa época revela o significado da palavra "explosão" dos computadores. Para se ter uma idéia, em 1983 o número de computadores pessoais em uso nos Estados Unidos era de 10 milhões, subindo para 30 milhões em 1986 e chegando a mais de 45 milhões em 1988. Logo em 1982, a APPLE COMPUTERS é a primeira empresa fabricante de computadores pessoais a atingir a marca de 1 bilhão de dólares em vendas anuais.

Figura 1.14: O computador pessoal IBM PC.

Processadores, e conseqüentemente os computadores pessoais, foram atualizados a passos largos. A EPSON CORPORATE HEADQUARTERS, em 1982, introduz no mercado o primeiro computador pessoal portátil[50]. Os processadores da família x86 286, 386 e 486 foram lançados gradativamente pela INTEL na década de 80 e foram incorporados em computadores pessoais de diversas marcas. O INTEL 80486, por exemplo, era um processador de 32 bits, 50 MHz e 1,2 milhões de transistores. Processadores compatíveis com a família x86 da INTEL foram produzidos por outras empresas como IBM, TEXAS INSTRUMENTS, AMD, CYRIX e CHIPS AND TECHNOLOGIES. Em 1984, a APPLE lança com sucesso estrondoso o primeiro MacIntosh, o primeiro computador pessoal a usar uma interface gráfica para interação entre o usuário e a máquina, conhecido como MacOS. O MacIntosh era equipado com o processador 68000 da MOTOROLA de 8 MHz e 128 Kbytes de memória. Uma atualização com expansão de memória de 1 Mbytes foi feita em 1986 no MacIntosh Plus.

Os programas também evoluíram na década de 80. O sistema operacional MS-DOS partiu gradativamente da versão 1.0 em 1981 e atingiu a versão 4.01 em 1988. Em 1985, o sistema operacional Windows 1.0 é vendido por 100 dólares pela MICROSOFT em resposta à tendência crescente de uso das interfaces gráficas de usuários popularizadas pelo MacIntosh. Em 1987 o Windows 2.0 é disponibilizado. Nessa época a APPLE trava uma batalha judicial por cópia de direitos autorais do sistema operacional do MacIntosh contra a MICROSOFT, pelo sistema operacional Windows 2.03 de 1988, e a HEWLETT-PACKARD, pelo sistema operacional NewWave de 1989.

Em 1990 Tim Berners-Lee[51] propõe um sistema de hipertexto que é o primeiro impulso da Rede Mundial de Computadores[52]. O primeiro provedor comercial de linha discada da Internet torna-se ativo em 1991, quando a WWW é disponibilizada para o público em geral como uma ferramenta de busca.

[50] Também conhecido como *notebook*.
[51] Sir Timothy John Berners-Lee (1955−), nascido na Inglaterra, físico.
[52] Do inglês *World Wide Web* – WWW.

Figura 1.15: O símbolo do Linux.

Richard Stallman[53], em 1985, escreve o Manifesto GNU que apresenta sua motivação para desenvolver o sistema operacional GNU, o primeiro projeto de software livre proposto. Desde meados dos anos 90, Stallman tem gastado muito de seu tempo como um ativista político defendendo o software livre, bem como fazendo campanha contra patentes de software e a expansão das leis de direitos autorais. Os mais destacados programas desenvolvidos por Stallman são o GNU Emacs, o GNU Compiler Collection (`gcc`) e o GNU Debugger (`gdb`). Inspirado pelo Minix, um sistema operacional baseado no Unix voltado para o ensino, Linus Torvalds[54] projetou em 1991 um sistema operacional para computadores pessoais chamado Linux. O Linux é um exemplo de destaque do que chamamos de software livre e de desenvolvimento de código aberto. Seu código fonte, escrito na linguagem C, é disponibilizado para qualquer pessoa usar, modificar e redistribuir livremente.

O sistema operacional MS-DOS teve seu fim na década de 90, em sua última versão comercial 6.22, tendo sido completamente substituído pelo bem sucedido Windows. A versão 3.0 do Windows vendeu 3 milhões de cópias em 1990. Em 1992, a versão 3.1 vendeu 1 milhão de cópias em apenas 2 meses depois de seu lançamento. Já em 1997, após o lançamento do Windows 95 em 1995, Bill Gates é reconhecido como o homem mais rico do mundo. Nessa época de explosão do uso dos sistemas operacionais da MICROSOFT, os vírus de computador passaram a infestar cada vez mais computadores e a impor perdas cada vez maiores de tempo, de recursos e de dinheiro às pessoas e empresas que utilizavam tais sistemas operacionais. Um dos primeiros e mais famosos vírus de computador é o Monkey Virus, um vírus de setor de *boot* descoberto no Canadá em 1991, que se espalhou muito rapidamente pelos Estados Unidos, Inglaterra e Austrália. Seguiu-se ao Windows 95 o Windows 98 em 1998 e o Windows ME em 2000.

A Rede Mundial de Computadores e a Internet também mostram um desenvolvimento destacado nessa época. O ano de 1993 registra um crescimento espantoso da Internet e 50 servidores WWW já são conhecidos até aquele momento. Em 1994, os estudantes de doutorado em engenharia elétrica da Universidade de Stanford Jerry Yang[55] e David Filo[56] fundam a Yahoo!, uma empresa de serviços de Internet que engloba um portal de Internet, uma ferramenta de busca na Internet, serviço de e-mail, entre outros. A Yahoo! obtém grande sucesso entre usuários e em outubro de 2005 sua rede de serviços espalhada pelo mundo recebe em média 3,4 bilhões de visitas por dia. Em 1994, o *World Wide Web Consortium* – W3C é fundado por Tim Berners-Lee para auxílio no desenvolvimento de protocolos comuns para avaliação da Rede Mundial de Computadores. O Wiki foi criado em 1995 pelo Repositório Padrão de Portland, nos Estados Unidos, e é um banco de dados aberto à edição, permitindo que qualquer usuário possa atualizar e adicionar informação, criar novas páginas, etc., na Internet. Nesse mesmo ano, a SUN MICROSYSTEMS lança a linguagem de programação orientada a objetos Java, amplamente utilizada hoje em dia para criar aplicações para a Internet. Os rudimentos

[53] Richard Matthew Stallman (1953−), nascido nos Estados Unidos, físico, ativista político e ativista de software.
[54] Linus Benedict Torvalds (1969−), nascido na Finlândia, cientista da computação.
[55] Jerry Chih-Yuan Yang (1968−), nascido em Taiwan, engenheiro elétrico e empresário.
[56] David Filo (?−), nascido nos Estados Unidos, engenheiro da computação e empresário.

da ferramenta de busca Google são desenvolvidos em 1996 como um projeto de pesquisa dos alunos de doutorado Larry Page[57] e Sergey Brin[58] da Universidade de Stanford. Em 1998 a página do Google é disponibilizada na Internet e, atualmente, é a ferramenta de busca mais utilizada na rede. A WebTV também é disponibilizada em 1996 possibilitando aos usuários navegar pela Internet a partir de sua TV.

Com relação aos dispositivos eletrônicos, o Barramento Serial Universal – USB[59] é padronizado em 1995 pela INTEL, COMPAQ, MICROSOFT, entre outras. O USB é um barramento externo padronizado que permite transferência de dados e é capaz de suportar até 127 dispositivos periféricos. Em 1997 o mercado começa a vender uma nova mídia de armazenamento ótico de dados, o Disco Versátil Digital – DVD[60], com as mesmas dimensões do CD, mas codificado em um formato diferente com densidade muito mais alta, o que permite maior capacidade de armazenamento. Os DVDs são muito utilizados para armazenar filmes com alta qualidade de som e vídeo. Em 1998 o primeiro tocador de MP3, chamado de MPMan, é vendido no Japão pela empresa SAEHAN.

Os processadores também foram vendidos como nunca nos anos 90. A INTEL lança em 1993 o sucessor do INTEL 486, conhecido como Pentium, ou Pentium I, de 32 bits, 60 MHz e 3,1 milhões de transistores em sua versão básica. A partir de 1997, a INTEL lança seus processadores seguidamente, ano após ano. Em 1997 anuncia a venda dos processadores Pentium MMX e Pentium II. O Celeron é produzido a partir de 1998 e em 1999, a INTEL anuncia o início da produção do Pentium III. O Pentium IV é produzido a partir de 2000 e é um processador de 32 bits, 1,4 GHz e 42 milhões de transistores. A AMD já produzia microprocessadores desde a década de 70, mas entrou em forte concorrência com a INTEL a partir dessa época, com o lançamento do K5 em 1995, concorrente do Pentium I, de 32 bits, 75 MHz e 4,3 milhões de transistores em sua primeira versão. Seguiu-se ao K5 o K6 em 1997, de 32 bits, 66 MHZ e 8,8 milhões de transistores, e a série K7 de 1999, que engloba os processadores Athlon e Duron. A partir da década de 90, uma fatia considerável das fábricas produz computadores pessoais com processadores da INTEL ou da AMD. No início da década de 90, APPLE, IBM e MOTOROLA se unem para projetar o processador PowerPC de 32 bits que equipou o PowerMac da APPLE a partir de 1994. A aceitação desse novo computador não foi como o esperado e, após um período de baixas, a APPLE ressurge em 1998 com o iMac, um computador pessoal com projeto visual arrojado e com a filosofia do "tudo-em-um-só". A década de 90 é marcada pela facilidade de acesso aos computadores pessoais, já que muitos deles passam a ser vendidos por menos que 1.000 dólares.

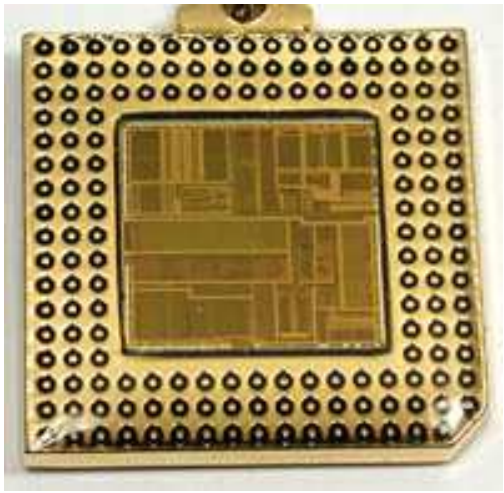


Figura 1.16: Processador Pentium.

[57] Lawrence Edward Page (1973–), nascido nos Estados Unidos, engenheiro da computação e empresário.
[58] Sergey Brin (1973–), nascido na Rússia, matemático, cientista da computação e empresário.
[59] Do inglês *Universal Serial Bus*.
[60] Do inglês *Digital Versatile Disc*.

### De 2000 até hoje

O ano de 2000 inicia receoso para governos e grandes empresas pelo medo das conseqüências potencialmente desastrosas da virada do século nos sistemas computacionais, o que se chamou na época de "Bug do Milênio". Além disso, o juiz Thomas Penfield anuncia a divisão do império MICROSOFT em duas companhias e Bill Gates deixa o cargo de Chefe-Executivo da empresa.

A MICROSOFT lança o Windows XP em 2001 e recentemente o Windows Vista, em janeiro de 2007. A APPLE disponibiliza o MacOS X 10.0 em 2001 e seguidas novas atualizações até 2006, com o MAC OS X 10.5. O Linux tem maior penetração e aceitação dos usuários de computadores pessoais e diversas distribuições são disponibilizadas como Ubuntu, Mandriva Linux, Fedora Core e SUSE Linux. Estas distribuições e os projetos comunitários Debian e Gentoo, montam e testam os programas antes de disponibilizar sua distribuição. Atualmente, existem centenas de projetos de distribuição Linux em ativo desenvolvimento, constantemente revisando e melhorando suas respectivas distribuições.

Em 2001 a DELL passa a ser a maior fabricante de computadores pessoais do mundo. Em 2002, uma empresa de consultoria norte-americana faz uma estimativa que até aquele ano aproximadamente 1 bilhão de computadores pessoais tinham sido vendidos no mundo inteiro desde sua consolidação.

A INTEL tem investido mais recentemente sua produção em processadores de 64 bits com núcleos múltiplos, como por exemplo o Xeon de 2004, o Pentium D de 2005 e o INTEL Core Duo de 2006. A AMD produz os processadores de 64 bits Duron em 2000 e o Sempron em 2004, ambos da série K7, e o Opteron em 2003 da série K8, este último um processador de dois núcleos, para competir com os processadores de múltiplos núcleos, como o Xeon da INTEL. A APPLE, que vinha utilizando o processador PowerPC em sua linha de iMacs, anuncia que a partir de 2006 começará a usar os processadores da INTEL. A versão de 2006 do iMac usa o processador INTEL Core 2 Duo. Na lista dos 500 supercomputadores mais rápidos do mundo, a TOP500, de novembro de 2006, 22,6% dos supercomputadores eram baseados em processadores Opteron da AMD e 21,6% em processadores Xeon da INTEL.

Figura 1.17: O processador Opteron da AMD.

Alguns modelos de computadores experimentais e inovadores têm sido propostos, com implementações. O computador de DNA, o computador molecular, o computador químico e o computador quântico são exemplos de propostas recentes e promissoras.

## Exercícios

1.1 Visite a página do *Computer History Museum*, que contém uma enorme quantidade de informações.

1.2 Escolha um ou dois personagens da pré-história da computação e leia os verbetes sobre esses personagens na Wikipedia.

1.3 Leia os verbetes associados às duas mulheres Augusta Ada Byron e Grace Murray Hopper, que deram origem a muito do que conhecemos como computação.

1.4 Leia o verbete sobre Alan Mathison Turing, o "Pai da Ciência da Computação", na Wikipedia.

1.5 Visite a página *GNU Operating System* e leia o verbete GNU na Wikipedia.

1.6 Visite as página *Free Software Foundation* e Fundação Software Livre América Latina.

1.7 Leia o verbete Linux na Wikipedia.

AULA 2

# COMO FUNCIONA UM COMPUTADOR

Na aula 1 vimos um pequeno histórico sobre os computadores. Nesta aula de hoje, vamos mencionar brevemente como essas máquinas são organizadas internamente. Como veremos, John von Neumann foi o responsável, em 1945, pelo desenvolvimento de um modelo[1] de computador que sobrevive até hoje.

Este texto é baseado principalmente nas referências [10, 12].

## 2.1 Contextualização

As primeiras máquinas computacionais projetadas tinham programas fixos. Assim, realizavam apenas aquelas tarefas para as quais foram concebidas. Modificar o programa de uma dessas máquinas significava reestruturar toda a máquina internamente. Mesmo para computadores como o ENIAC, programar e reprogramar era um processo geralmente laborioso, com uma quantidade enorme de rearranjo de fiação e de cabos de conexão. Dessa forma, podemos destacar que a programação dos computadores que foram projetados antes do EDVAC tinha uma característica curiosa e comum: a programação por meios externos. Ou seja, os programadores, para que essas máquinas executassem seus programas, deviam usar cartões perfurados, fitas perfuradas, cabos de conexão, entre outros. A pequena disponibilidade de memória para armazenamento de dados e resultados intermediários de processamento também representava uma outra dificuldade complicadora importante.

Em 1936, Alan Turing publicou um trabalho [16][2] que descrevia um computador universal teórico, conhecido hoje como máquina universal de Turing. Esse computador tinha capacidade infinita de armazenamento, onde se poderia armazenar dados e instruções. O termo usado foi o de computador com armazenamento de programas[3]. Sobre esse mesmo conceito, o engenheiro alemão Konrad Zuse, que projetou a família Z de computadores, escreveu um trabalho [19][4] independentemente em 1936. Também de forma independente, John Presper Eckert e John Mauchly, que projetaram o ENIAC na Universidade da Pensilvânia, propuseram um trabalho em 1943, conforme menção em [11], sobre o conceito de computadores com armazenamento de programas. Em 1944, no projeto do novo computador EDVAC, John Eckert deixou registrado que sua equipe estava propondo uma forma inovadora de armazenamento de dados e programas em um dispositivo de memória endereçável, feito de mercúrio, conhecido como linhas de atraso. John von Neumann, tendo trabalhado com a equipe que construiu o ENIAC, juntou-

[1] Veja o verbete *von Neumann architecture* na Wikipedia.
[2] Artigo disponível na seção "Bibliografia complementar" na página da disciplina no Moodle.
[3] Do inglês *stored-program computer*.
[4] Veja mais em *The Life and Work of Konrad Zuse*, especialmente a parte 10.

se também às discussões do projeto desse novo computador e ficou responsável por redigir o documento com a descrição dessas idéias. Com a publicação desse trabalho [17][5], divulgado contendo apenas seu nome, essa organização interna de um computador passou a ser conhecida como "arquitetura de von Neumann", apesar de Turing, Zuse, Eckert e Mauchly terem contribuído ou já apresentado idéias semelhantes. Esse trabalho obteve grande circulação no meio acadêmico e seu sucesso como proposta de implementação de um novo computador foi imediato. Até nossos dias, essa organização ainda é usada nos projetos de computadores mais conhecidos.

## 2.2 Arquitetura de von Neumann

As idéias publicadas por John von Neumann foram revolucionárias a ponto de estabelecer um novo paradigma de concepção de computadores para a época e que permanece válido até os nossos dias. Informalmente, a maneira de organizar os componentes que constituem um computador é chamada arquitetura do computador. Assim, como a maioria dos computadores atuais segue o modelo proposto por von Neumann, veremos aqui uma breve descrição da assim chamada arquitetura de von Neumann.

Nesse modelo, um computador é constituído por três componentes principais: (*i*) a unidade central de processamento ou UCP[6], (*ii*) a memória e (*iii*) os dispositivos de entrada e saída. A UCP, por sua vez, é composta pela unidade lógico-aritmética ou ULA e pela unidade de controle ou UC. Esses três componentes principais estão conectados e se comunicam através de linhas de comunicação conhecidas como barramento do computador.

Podemos ilustrar a arquitetura de von Neumann como na figura 2.1.

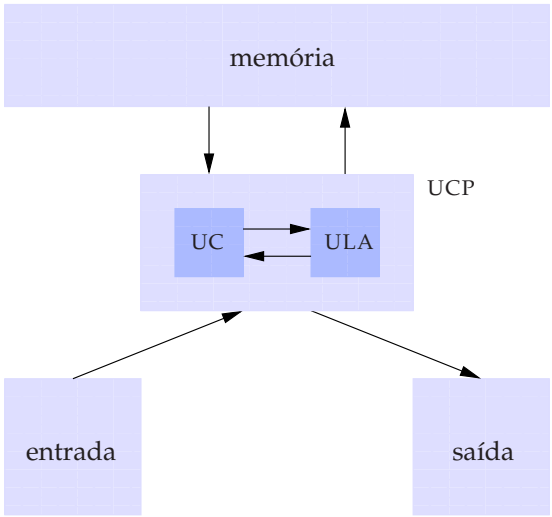



Figura 2.1: Arquitetura de von Neumann.

A unidade central de processamento é responsável pela execução das instruções armazenadas, também chamadas de programa. Atualmente, o termo UCP é quase um sinônimo de microprocessador. Um microprocessador é a implementação de uma UCP através de um único,

[5] Artigo disponível na seção "Bibliografia complementar" na página da disciplina no Moodle.
[6] Do inglês *central processing unit – CPU*.

ou de muito poucos, circuitos integrados. Diversos fabricantes disponibilizam microprocessadores no mercado, como os Core 2 Duo e Core 2 Quad da Intel e os Phenom X3 triple-core e Phenom X4 quad-core da AMD.

A unidade central de processamento é composta também pela unidade lógico-aritmética e pela unidade de controle. A ULA executa dois tipos de operações: (*i*) operações aritméticas, como adições e subtrações, e (*ii*) operações lógicas ou de comparação, como por exemplo a verificação se um número é maior ou igual a outro número.

A memória de um computador pode ser vista como uma lista linear de compartimentos ou células. Cada compartimento tem um identificador ou endereço numérico e pode armazenar uma certa quantidade pequena de informação. A informação armazenada em um compartimento de memória pode ser uma instrução de um programa ou um dado, uma informação que deve ser processada através das instruções. Há dois tipos distintos de memória: (*i*) memórias voláteis e (*ii*) memórias não voláteis. As memórias do primeiro tipo necessitam de uma fonte de energia para que seu conteúdo seja mantido. As memórias que permitem acesso aleatório[7] aos seus compartimentos são, em geral, memórias voláteis. Esses dispositivos permitem acesso rápido à informação armazenada, mas são muito mais caros que os dispositivos não voláteis de memória. O que em geral chamamos de memória principal de um computador é um dispositivo de memória volátil, pois quando desligamos o computador, todas as informações armazenadas em seus compartimentos são perdidas. Os dispositivos de armazenamento de informações não voláteis são aqueles que, ao contrário dos primeiros, podem reter a informação armazenada mesmo sem uma fonte de energia conectada. Como exemplo desses dispositivos, podemos listar os discos rígidos, discos óticos, fitas magnéticas, memórias do tipo *flash*, memórias holográficas, entre outros. Pela forma como em geral são implementados, esses dispositivos não permitem que as informações sejam acessadas rapidamente em um compartimento qualquer, o que acarreta um tempo maior para busca e recuperação das informações. No entanto, o custo de produção de tais dispositivos é inferior ao custo dos dispositivos de memórias voláteis.

Se enxergarmos um computador como uma caixa-preta que alimentamos com informações e colhemos resultados dessas informações processadas, podemos associar os seus dispositivos de entrada e saída como aqueles que fazem a comunicação do computador com o mundo externo. O mundo externo pode ser composto por pessoas ou mesmo outros computadores. Teclados, *mouse*s, microfones, câmeras, entre outros, são dispositivos comuns de entrada. Monitores e impressoras são dispositivos de saída. Placas de redes são dispositivos tanto de entrada como de saída de informações.

Um computador funciona então como uma máquina que, a cada passo, carrega uma instrução e dados da memória, executa essa instrução sobre esses dados e armazena os resultados desse processamento em sua memória. Então o processo se repete, isto é, o computador novamente carrega uma próxima instrução e novos dados da memória, executa essa instrução sobre os dados e grava os resultados desse processamento em sua memória. Enquanto existirem instruções a serem executadas em um programa, esse processo se repete. Todo computador tem um sinal de relógio[8] que marca esse passo mencionado, que é mais conhecido como ciclo do relógio. Dessa forma, o sinal de relógio de um computador é usado para coordenar e sincronizar as suas ações.

[7] Do inglês *random acess memory – RAM*.
[8] Do inglês *clock signal*.

Uma outra representação a arquitetura de von Neumann, mais próxima ao que conhecemos como um computador pessoal, pode ser vista na figura 2.2.

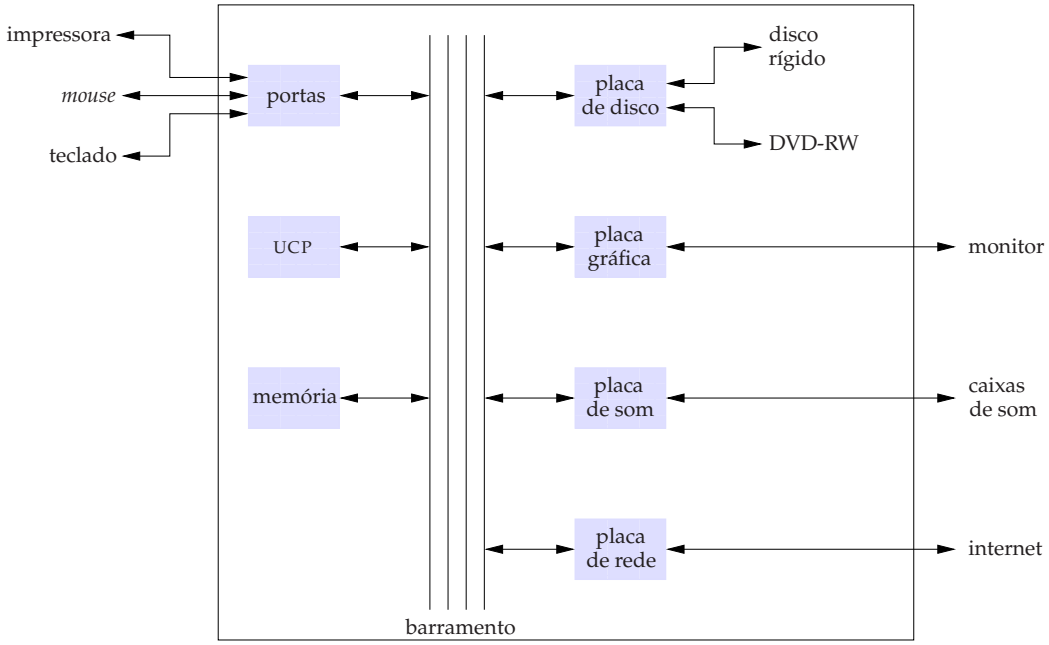



Figura 2.2: Uma outra ilustração da arquitetura de von Neumann.

## 2.3 Algoritmos e programas


Figura 2.3: Al-Khwārizmī.

Há uma certa controvérsia sobre como definir de maneira formal um algoritmo. Informalmente, podemos dizer que um algoritmo é uma seqüência precisa, sistemática e finita de passos ou instruções para solução de algum problema. Uma tentativa de formalização do termo teve início com a tentativa de solução do problema *Entscheidungsproblem*, ou o problema da decisão, proposto por David Hilbert em 1928. O termo "algoritmo" tem sua origem no cientista persa Al-Khwārizmī[9]. A latinização de seu nome para "Algoritmi" deu origem ao termo algoritmo em português e em outras línguas. O termo "algarismo", que significa dígito numérico, também tem origem em seu nome. Suas principais contribuições incluem inovações importantes na matemática, em especial em álgebra e trigonometria. Um de seus mais importantes livros, Sobre o Cálculo com Números Hindus, do ano de 825, foi responsável pela disseminação do sistema hindu de numeração no Oriente Médio e na Europa.

[9] Mohammad ebne Mūsā Khwārazmī, (780 − 850), nascido na Pérsia, matemático, astrônomo e geógrafo.

Um algoritmo é então um conjunto de idéias abstratas para solução de um problema. Lidamos com algoritmos desde nossa infância em diversas situações. Por exemplo, aprendemos no ensino primário o algoritmo de Euclides para obtenção do máximo divisor comum entre dois números inteiros. Nesse algoritmo, tomamos dois números inteiros $a$ e $b$ e se $b = 0$ então o máximo divisor comum entre $a$ e $b$ é o número $a$. Caso contrário, o máximo divisor comum entre $a$ e $b$ é dado pelo máximo divisor comum de $b$ e do resto da divisão de $a$ por $b$. O processo então se repete até que $b$ seja igual a 0 (zero). É importante destacar que esse algoritmo sempre obtém um $b = 0$ em algum dos passos e, portanto, o algoritmo sempre termina. Além disso, o algoritmo é correto, já que sempre produz uma resposta correta para um par de números inteiros fornecidos como entrada.

Após o desenvolvimento de um algoritmo para solução de algum problema, o próximo passo é informá-lo a um computador. No entanto, os computadores não compreendem, e portanto são incapazes de obedecerem e executarem, instruções em uma linguagem natural como o português. Dessa forma, precisamos traduzir nosso algoritmo para um programa em uma linguagem de programação escolhida. Uma linguagem de programação é uma linguagem artificial projetada para expressar computações que podem ser executadas por um computador. Dessa forma, um programa projetado em uma linguagem de programação possui uma sintaxe e semântica precisas. Além disso, um programa escrito em uma linguagem de programação pode ser facilmente traduzido para um programa em linguagem de máquina, que nada mais é que uma seqüência de dígitos binários (bits) que representam dados e instruções e que produzem o comportamento desejado do computador, em sua arquitetura específica.

Em geral, trabalhamos com linguagens de programação chamadas "de alto nível", o que significa que são linguagens de programação mais facilmente compreensíveis para os seres humanos. As linguagens C, C++, Java, PHP, entre outras, são linguagens de programação de alto nível. Em contraposição, existem linguagens de programação "de baixo nível", que são muito próximas da arquitetura do computador que usamos. A linguagem *assembly* é uma linguagem de programação de baixo nível. Infelizmente, programas escritos em uma linguagem programação qualquer não estão prontos para serem executados por um computador. Ainda há processos de tradução intermediários que levam o programa de um ponto compreensível por um ser humano e incompreensível para o computador para o outro extremo, isto é, basicamente incompreensível pelo ser humano e completamente compreensível pelo computador e sua arquitetura.

Assim, dado um programa em uma linguagem de programação de alto nível, usamos um programa especial, chamado compilador, para efetuar a tradução deste programa para um programa em linguagem *assembly*, de baixo nível e associada ao processador do computador que irá executar as instruções. Um segundo e último processo de tradução ainda é realizado, chamado de montagem, onde há a codificação do programa na linguagem *assembly* para um programa em formato binário, composto por 0s e 1s, e que pode enfim ser executado pela máquina. Um esquema dos passos desde a concepção de um algoritmo até a obtenção de um programa executável equivalente em um computador é mostrado na figura 2.4.

Modelos conceituais inovadores e alternativos à arquitetura de von Neumann têm sido propostos e alguns deles implementados. O computador de DNA, o computador molecular, o computador químico e o computador quântico são exemplos de propostas recentes e promissoras.

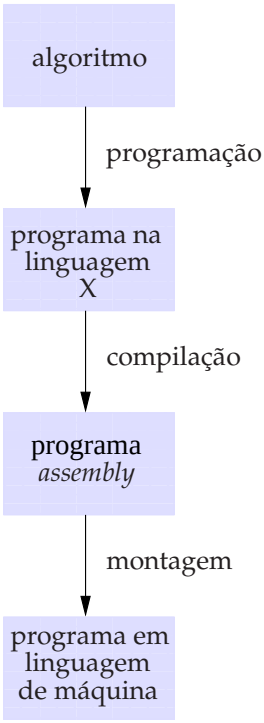



Figura 2.4: Passos desde a concepção de um algoritmo até o programa executável por um computador.

## Exercícios

2.1 Leia o verbete *Computer* na Wikipedia.

2.2 Leia o verbete *von Neumann architecture* na Wikipedia.

2.3 Dê uma olhada no trabalho de John von Neumann na referência [17][10], sobre a "arquitetura de von Neumann".

2.4 Dê uma olhada no trabalho de Alan Turing na referência [16][11] com a descrição da máquina de Turing.

2.5 Visite a página sobre a vida e o trabalho de Konrad Zuse.

2.6 Visite as páginas *C History* e *C Programming Language* e leia o verbete *C (programming language)* na Wikipedia.

[10] Artigo disponível na seção "Bibliografia complementar" na página da disciplina no Moodle.
[11] Artigo disponível na seção "Bibliografia complementar" na página da disciplina no Moodle.

AULA 3

# DICAS INICIAIS

Nesta aula aprenderemos dicas importantes sobre as ferramentas que mais usaremos durante o desenvolvimento da disciplina. Em particular, veremos a definição de uma interface do sistema operacional (*shell* ou Unix *shell*) e aprenderemos a usar alguns dentre os tantos comandos disponíveis para essa interface, aprenderemos alguns comandos e funções do ambiente de trabalho GNU/Emacs e aprenderemos a usar o compilador da linguagem C da coleção de compiladores GNU, o `gcc`.

## 3.1 Interface do sistema operacional

Podemos definir uma interface de um sistema como um programa que permite a interação entre o sistema e seus usuários. Dessa forma, diversos sistemas contêm interfaces desse tipo. No entanto, esse termo é quase que automaticamente associado aos sistemas operacionais, onde uma interface é fornecida aos usuários para acesso aos serviços do núcleo do sistema operacional. As interfaces dos sistemas operacionais são freqüentemente usadas com o propósito principal de invocar outros programas, mas também possuem outras habilidades adicionais.

Há em geral dois tipos de interfaces de sistemas operacionais: as interfaces de linhas de comando e as interfaces gráficas. Neste curso, usaremos as interfaces de linhas de comando para solicitar serviços e executar certos programas nativos do sistema operacional Linux. A seguir listamos alguns desses comandos.

Antes de mais nada, é necessário saber duas coisas sobre o sistema que temos instalado em nossos laboratórios. Primeiro, devemos saber que o sistema operacional instalado nesses computadores é o Linux. A distribuição escolhida é a Debian, versão Lenny. Essa e uma distribuição não comercial e livre do GNU/Linux, isto é, uma versão gratuita e de código fonte aberto. O nome 'Debian' vem dos nomes dos seus fundadores, *Ian* Murdock e de sua esposa, *Deb*ra. Diversas distribuições comerciais baseiam-se, ou basearam-se, na distribuição Debian, como Linspire, Xandros, Kurumin, Debian-BR-CDD, Ubuntu e Libranet. A segunda informação importante é sobre a interface gráfica do Linux usada nos laboratórios. A interface gráfica completa do ambiente de trabalho é a plataforma GNOME. Essa plataforma é um projeto colaborativo internacional inteiramente baseado em software livre que inclui o desenvolvimento de arcabouços, de seleção de aplicações para o ambiente de trabalho e o desenvolvimento de programas que gerenciam o disparo de aplicações, manipulação de arquivos, o gerenciamento de janelas e de tarefas. O GNOME faz parte do Projeto GNU e pode ser usado com diversos sistemas operacionais do tipo Unix, especialmente aqueles que possuem um núcleo Linux.

Isso posto, um dos programas mais comuns disponíveis para os usuários do sistema operacional GNU/Linux é chamado de 'terminal', como uma abreviação de 'terminal de linhas de comando', que nada mais é que uma interface entre o usuário e o sistema operacional. Para disparar esse programa no gerenciador de janelas GNOME, selecione a opção 'Aplicações' do menu principal, depois a opção 'Acessórios' no sub-menu e, em seguida, a opção 'Terminal'. Um exemplo de uma janela de terminal de linhas de comando do GNOME é apresentado na figura 3.1.

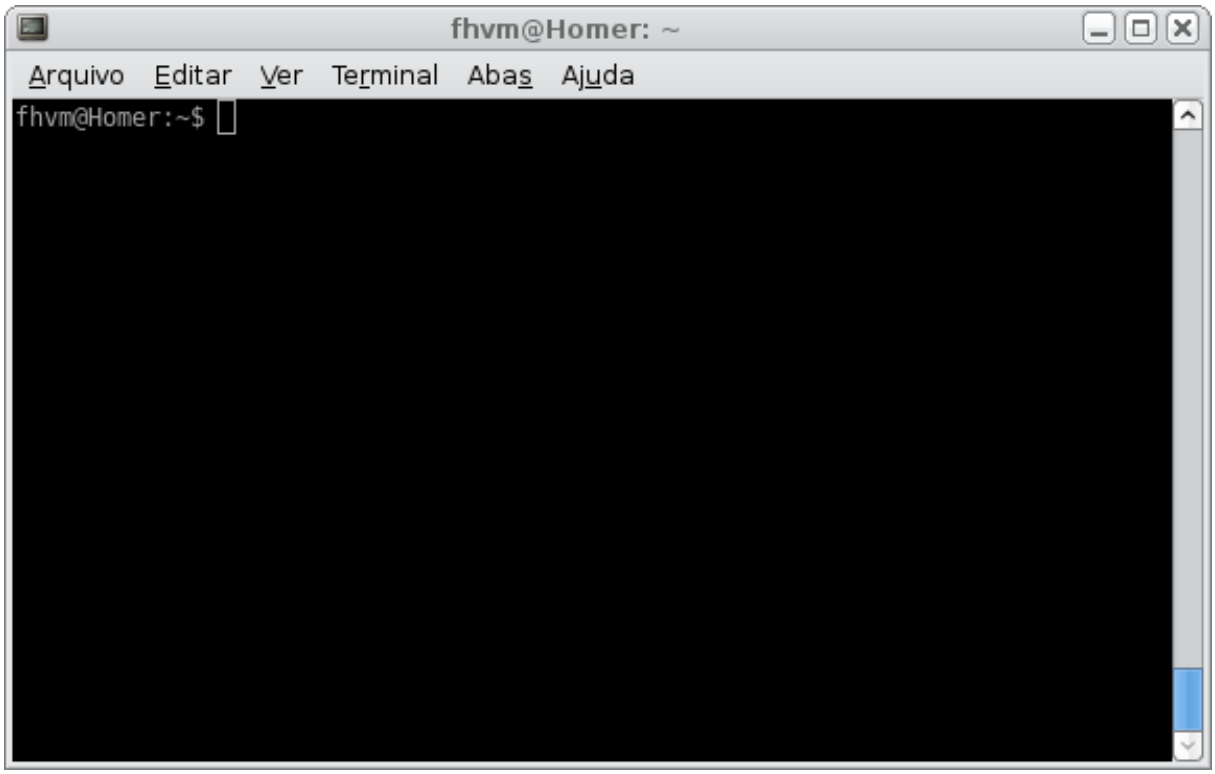



Figura 3.1: Uma interface, ou janela, de linhas de comandos do sistema de janelas GNOME, chamada no computador `Homer` pelo usuário `fhvm`.

Uma janela de linha de comandos possibilita a interação entre o usuário e o computador, intermediada pelo sistema operacional, onde o primeiro solicita tarefas diretamente ao outro e, depois de executadas, recebe seus resultados. Daqui por diante, veremos alguns dos comandos e utilitários que serão mais usados durante o decorrer da disciplina. Alguns deles são apresentam, até certo ponto, algum grau de dificuldade de compreensão neste momento. Todavia, é importante ressaltar que todos esses comandos e utilitários serão muito úteis a partir de momentos oportunos, como veremos.

A seguir, apresentamos a tabela 3.1 contendo os principais comandos e utilitários do sistema operacional, com suas descrições breves, os quais mais usaremos no desenvolvimento da disciplina de Programação de Computadores I. Esta tabela é apenas um guia de referência rápida e, apesar da maioria dos comandos e utilitários serem intuitivos e muito fáceis de usar, a descrição breve contida na tabela infelizmente não nos ensina de fato como fazer uso correto de cada um deles. Também não mostra todas as opções de execução que em geral todo comando ou utilitário possui.

Dessa forma, para aprender gradativa e corretamente a usar os comandos e utilitários apresentados na tabela 3.1 é necessário: (*i*) fazer os exercícios ao final desta aula; (*ii*) usar com freqüência o utilitário `man` ou `info`, que são interfaces para o manual de referência dos comandos e ferramentas do sistema[1]. Por exemplo, podemos digitar:

```
prompt$ man comando
```
ou
```
prompt$ info comando
```

onde `prompt$` é o sinal de pronto do sistema, `man` e `info` são utilitários do sistema e *`comando`* é o nome do comando/ferramenta do sistema para o qual informações do manual serão exibidas; e (*iii*) usar a opção `--help` na chamada de um comando/utilitário. Essa opção mostra como usá-lo corretamente, listando as opções disponíveis. Por exemplo,

```
prompt$ man --help
```

| **Comando** | **Descrição** |
|---|---|
| `cat` *arq* | mostra o conteúdo de *arq* ou da entrada padrão |
| `cd` *dir* | troca o diretório atual para o diretório *dir* |
| `cmp` *arq1 arq2* | compara *arq1* e *arq2* *byte* a *byte* |
| `cp` *arq1 arq2* | copia o *arq1* para o *arq2* |
| `cp` *arq(s) dir* | copia *arq(s)* para o diretório *dir* |
| `date` | mostra a data e a hora do sistema |
| `diff` *arq1 arq2* | compara *arq1* e *arq2* linha a linha |
| `echo` *args* | imprime *args* |
| `enscript` *arq* | converte arquivos-texto para PostScript, HTML, RTF, ANSI e outros |
| `fmt` *arq* | formata *arq* |
| `gprof` *arq* | fornece um perfil de execução do programa *arq* |
| `indent` *arq* | faz a indentação correta de um programa na linguagem C em *arq* |
| `info` *cmd* | mostra a página (mais completa) de manual de *cmd* |
| `ls` *arq(s)* | lista *arq(s)* |
| `ls` *dir* | lista os arquivos no diretório *dir* ou no diretório atual |
| `man` *cmd* | mostra a página de manual de *cmd* |
| `mkdir` *dir(s)* | cria diretório(s) *dir(s)* |
| `mv` *arq1 arq2* | move/renomeia o *arq1* para o *arq2* |
| `mv` *arq(s) dir* | move *arq(s)* para o diretório *dir* |
| `pwd` | mostra o nome do diretório atual |
| `pr` *arq* | converte *arq*, do tipo texto, para impressão |
| `rm` *arq(s)* | remove *arq(s)* |
| `rmdir` *dir(s)* | remove diretórios vazios *dir(s)* |
| `sort` *arq(s)* | ordena as linhas de *arq(s)* ou da entrada padrão |
| `wc` *arq(s)* | conta o número de linhas, palavras e caracteres em *arq(s)* ou na entrada padrão |
| `who` | imprime a identificação do usuário do computador |

Tabela 3.1: Tabela de comandos e utilitários mais usados.

[1] Esses manuais também podem ser encontrados em páginas da internet, como Manpages e SS64.com.

Para aumentar a flexibilidade e facilidade na especificação de argumentos de comandos do sistema operacional, podemos usar caracteres especiais de substituição, chamados de coringas. Como veremos, os coringas são muito usados quando os argumentos dos comandos são nomes de arquivos.

O caractere `*` (asterisco) é um coringa que substitui zero ou mais caracteres. No comando apresentado a seguir:

```
prompt$ ls *.c
```

`prompt$` é o sinal de pronto do sistema, `ls` é um comando que permite listar os arquivos de um diretório e `*.c` é um argumento desse comando que representa todos os arquivos do diretório cujos nomes terminam com os caracteres `.c`.

O caractere `?` (ponto de interrogação) é um outro coringa muito usado como parte de argumentos de comandos, representando exatamente um caractere e que pode ser um caractere qualquer. Por exemplo, no comando abaixo:

```
prompt$ ls ???.c
```

`prompt$` é o sinal de pronto do sistema, `ls` é um comando que permite listar os arquivos de um diretório e `???.c` é um argumento desse comando que representa todos os arquivos do diretório cujos nomes contêm exatamente 3 caracteres iniciais quaisquer seguidos pelos caracteres `.c`.

Os colchetes `[ ]` também são caracteres coringas muito usados nos argumentos de comandos e indicam a substituição de um único caractere especificado no interior dos colchetes. No exemplo abaixo:

```
prompt$ ls *[abc]
```

o argumento `*[abc]` representa todos os arquivos do diretório cujos nomes terminam com o caractere `a`, `b` ou `c`.

Um intervalo de caracteres pode ser obtido no interior dos colchetes se o caractere `-` é usado. Por exemplo:

```
prompt$ ls *[a-z]
```

onde o argumento representa os nomes dos arquivos que terminam com uma das 26 letras do alfabeto.

Por fim, é importante destacar que muitos comandos do sistema operacional recebem informações de entrada a partir do terminal e também enviam as informações resultantes de saída para o terminal de comandos. Um comando em geral lê as informações de entrada de um local chamado de entrada padrão, que é pré-definido como o terminal de linha de comandos, e escreve sua saída em um local chamado de saída padrão, que também é pré-definido como o terminal de linhas de comando.

Por exemplo, o comando `sort` pode ordenar nomes fornecidos linha a linha por um usuário no terminal de comandos, ordenar esses nomes lexicograficamente e apresentar o resultado no terminal de linha de comandos:

```
prompt$ sort
Zenaide
Carlos
Giovana
Amanda
Ctrl-d
Amanda
Carlos
Giovana
Zenaide
prompt$
```

Se dispomos os caracteres `> arq` ao final de qualquer comando ou utilitário que normalmente envia sua saída para a saída padrão, ou seja, para o terminal de linha de comandos, o saída desse comando será escrita para o arquivo com nome *arq* ao invés do terminal. Por exemplo, a seqüência a seguir:

```
prompt$ sort > nomes.txt
Zenaide
Carlos
Giovana
Amanda
Ctrl-d
prompt$
```

faz com que quatro nomes digitados pelo usuário sejam ordenados e, em seguida, armazenados no arquivo `nomes.txt`.

Assim como a saída dos comandos e utilitários pode ser redirecionada para um arquivo, a entrada também pode ser redirecionada a partir de um arquivo. Se, por exemplo, existe um arquivo `temp.txt` contendo diversos nomes apresentados linha a linha, então o comando a seguir:

```
prompt$ sort < temp.txt
```

faz com que todos os nomes contidos no arquivo `temp.txt` sejam ordenados lexicograficamente e posteriormente apresentados seguida na saída padrão, isto é, na janela do terminal de linha de comandos.

Podemos ainda redirecionar a entrada e a saída de um comando concomitantemente. Por exemplo, o comando:

```
prompt$ sort < temp.txt > nomes.txt
```

recebe, pelo redirecionamento da entrada, os nomes contidos no arquivo `temp.txt`, ordena-os lexicograficamente e os armazena em um novo arquivo chamado `nomes.txt`, devido ao redirecionamento da saída.

Dessa forma, temos que os símbolos `<` e `>` são os símbolos de redirecionamento de entrada e saída, respectivamente.

## 3.2 Compilador

A coleção de compiladores GNU (GCC) é, na verdade, um arcabouço de compiladores para diversas linguagens de programação. Esse sistema foi desenvolvido pelo Projeto GNU e, por ter sido adotado como o compilador oficial do sistema operacional GNU, também foi adotado como compilador padrão por muitos dos sistemas operacionais derivados do Unix, tais como o GNU/Linux, a família BSD e o Mac OS X. Além disso, o GCC tem sido traduzido para uma grande variedade de arquiteturas.

Richard Stallman desenvolveu a primeira versão do GCC em 1987, como uma extensão de um compilador já existente da linguagem C. Por isso, naquela época o compilador era conhecido como *GNU C Compiler*. No mesmo ano, o compilador foi estendido para também compilar programas na linguagem C++. Depois disso, outras extensões foram incorporadas, como Fortran, Pascal, Objective C, Java e Ada, entre outras. A Fundação para o Software Livre distribui o GCC sob a Licença Geral Pública (GNU GPL – GNU *General Public License*). O GCC é um software livre.

Um compilador faz o trabalho de traduzir um programa na linguagem C, ou de outra linguagem de alto nível qualquer, para um programa em uma linguagem intermediária, que ainda será traduzido, ou ligado, para um programa em uma linguagem de máquina. Dessa forma, um argumento óbvio que devemos fornecer ao GCC é o nome do arquivo que contém um programa na linguagem C. Além disso, podemos informar ao compilador um nome que será atribuído ao programa executável resultante da compilação e ligação. Há também diversas opções disponíveis para realizar uma compilação personalizada de um programa e que podem ser informadas no processo de compilação.

Então, o formato geral de compilação de um programa é dado a seguir:

```
prompt$ gcc programa.c -o executável -Wall -ansi -pedantic
```

onde `gcc` é o programa compilador, *`programa.c`* é um arquivo com extensão `.c` que contém um programa fonte na linguagem C, `-o` é uma opção do compilador que permite fornecer um nome ao programa executável resultante do processo (*`executável`*), `-Wall` é uma opção que solicita que o compilador mostre qualquer mensagem de erro que ocorra durante a compilação, `-ansi` é uma opção que força o programa fonte a estar escrito de acordo com o padrão ANSI da linguagem C e `-pedantic` é uma opção que deixa o compilador muito sensível a qualquer possível erro no programa fonte.

No momento, ainda não escrevemos um programa na linguagem C e, por isso, não conseguimos testar o compilador de forma satisfatória. Esta seção é então apenas um guia de referência rápida que será muito usada daqui por diante.

## 3.3 Emacs

Em geral, usamos a palavra Emacs para mencionar um editor/processador de textos, mas Emacs é, na verdade, o nome de uma classe de editores/processadores que se caracteriza especialmente pela sua extensibilidade. O Emacs tem mais de 1.000 comandos de edição, mais do que qualquer outro editor existente, e ainda permite que um usuário combine esses comandos em macros para automatização de tarefas. A primeira versão do Emacs foi escrita em 1976

por Richard Stallman. A versão mais popular do Emacs é o GNU/Emacs, uma parte do Projeto GNU. O GNU/Emacs é um editor extensível, configurável, auto-documentado e de tempo real. A maior parte do editor é escrita na linguagem Emacs Lisp, um dialeto da linguagem de programação funcional Lisp.

O Emacs é considerado por muitos especialistas da área como o editor/processador de textos mais poderoso existente nos dias de hoje. Sua base em Lisp permite que se torne configurável a ponto de se transformar em uma ferramenta de trabalho completa para escritores, analistas e programadores.

Em modo de edição, o Emacs comporta-se como um editor de texto normal, onde o pressionamento de um caractere alfanumérico no teclado provoca a inserção do caractere correspondente no texto, as setas movimentam o ponto ou cursor de edição, a tecla `Backspace` remove um caractere, a tecla `Insert` troca o modo de inserção para substituição ou vice-versa, etc. Comandos podem ser acionados através do pressionamento de uma combinação de teclas, pressionando a tecla `Ctrl` e/ou o `Meta/Alt` juntamente com uma tecla normal. Todo comando de edição é de fato uma chamada de uma função no ambiente Emacs Lisp. Alguns comandos básicos são mostrados na tabela 3.2. Observe que a tecla `Ctrl` é representada por `C` e a tecla `Alt` por `M`. Um manual de referência rápida está disponível na bibliografia complementar na página da disciplina.

Para carregar o Emacs, podemos usar o menu principal do GNOME e escolher a opção 'Aplicações', escolhendo em seguida a opção 'Acessórios' e então a opção 'Emacs'. Ou então, de uma linha de comandos, podemos usar:

```
prompt$ emacs &
```

Vejamos a tabela 3.2, que contém um certo número de comandos[2] importantes.

| Teclas | Descrição |
|---|---|
| `C-z` | minimiza a janela do Emacs |
| `C-x C-c` | finaliza a execução do Emacs |
| `C-x C-f` | carrega um arquivo no Emacs |
| `C-x C-s` | salva o arquivo atual no disco |
| `C-x i` | insere o conteúdo de um outro arquivo no arquivo atual |
| `C-x C-w` | salva o conteúdo do arquivo em um outro arquivo especificado |
| `C-h r` | carrega o manual do Emacs |
| `C-h t` | mostra um tutorial do Emacs |
| `C-g` | aborta o comando parcialmente informado |
| `C-s` | busca uma cadeia de caracteres a partir do cursor |
| `C-r` | busca uma cadeia de caracteres antes do cursor |
| `M-%` | interativamente, substitui uma cadeia de caracteres |

Tabela 3.2: Uma pequena tabela de comandos do Emacs.

É importante observar que o Emacs, quando executado em sua interface gráfica, tem a possibilidade de executar muitos de seus comandos através de botões do menu ou através da barra de menus. No entanto, usuários avançados preferem usar os atalhos de teclado para esse tipo de execução, por terem um acesso mais rápido e mais conveniente depois da memorização dessas seqüências de teclas.

[2] Veja também o cartão de referência do Emacs na seção de bibliografia complementar da disciplina no Moodle.

## Exercícios

3.1 Abra um terminal. A partir de seu diretório inicial, chamado de `home` pelo sistema operacional, faça as seguintes tarefas:

- crie três diretórios com os seguintes nomes: `progi`, `algodadi` e `temp`;
- entre no diretório `progi`;
- crie três subdiretórios dentro do diretório `progi` com os seguintes nomes: `aulas`, `trabalhos` e `provas`;
- entre no diretório `aulas`;
- usando o comando `cat` e o símbolo de redirecionamento de saída `>`, crie um arquivo com nome `agenda.txt`, que contém linha a linha, nome e telefone de pessoas. Adicione pelo menos 10 nomes nesse arquivo;
- copie o arquivo `agenda.txt` para os diretórios `algodadi` e `temp`;
- verifique o conteúdo dos diretórios `progi/aulas`, `algodadi` e `temp`;
- use o comando `sort` para ordenar o conteúdo do arquivo `agenda.txt`. Execute o comando duas vezes: na primeira, use-o visualizando o resultado na tela do computador. Na segunda, redirecione a saída desse mesmo comando, criando um novo arquivo com nome `agenda-ordenada.txt`;
- verifique o conteúdo do diretório;
- compare os arquivos `agenda.txt` e `agenda-ordenada.txt` com os comandos `cmp` e `diff`;
- crie um arquivo com nome `frase` contendo uma frase qualquer, com pelo menos 20 caracteres. Da mesma forma como antes, use o comandos `cat` e o símbolo `>` de redirecionamento de saída;
- conte o número de palavras no arquivo `frase` usando o comando `wc`;
- neste diretório, liste todos os arquivos que começam com a letra `a`;
- neste diretório, liste todos os arquivos que começam com a letra `f`;
- neste diretório, liste todos os arquivos que terminam com a seqüência de caracteres `txt`;
- repita os três últimos comandos, usando no entanto o comando `echo`;
- liste todos os arquivos contidos neste diretório, mas redirecione a saída do comando para um arquivo de nome `lista`;
- execute o comando `ls [afl]*` e verifique a saída;
- execute o comando `ls [a-l]*` e verifique a saída;
- execute o comando `ls ?g*` e verifique a saída;
- execute o comando `ls ?[gr]*` e verifique a saída;
- execute o comando `ls ?????` e verifique a saída;
- execute o comando `ls ..` e verifique a saída;
- execute o comando `ls ~/progi` e verifique a saída;
- execute o comando `ls ~` e verifique a saída;

- mova os arquivos cujos nomes têm exatamente 5 caracteres neste diretório para o diretório `temp`;
- remova o diretório `temp` do seu sistema de arquivos;
- remova todos os arquivos criados até aqui, mantendo apenas os diretórios.

3.2 Abra o Emacs. Então, realize as seguintes tarefas:

- digite um texto que resume o que você aprendeu nas aulas de Programação de Computadores I até aqui;
- salve o arquivo no diretório `progi/aulas`;
- busque as palavras `computador`, `Turing`, `von Neumann`, `ENIAC`, `ls`, `cp` e `Emacs` no seu texto;
- troque a palavra `Emacs` no seu texto pela palavra `GNU/Emacs`.

AULA 4

# PRIMEIROS PROGRAMAS

Nesta aula aprenderemos a codificar nossos primeiros programas na linguagem C.

## 4.1 Digitando

Abra o seu Emacs, pressione `C-x C-f`, digite em seguida um nome para seu primeiro programa, como por exemplo `primeiro.c`, e então digite o seguinte código.

Programa 4.1: Primeiro programa.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     printf("Programar é bacana!\n");
5     return 0;
6  }
```

Depois de ter digitado o programa, pressione as teclas `C-x C-s` para salvá-lo em um diretório adequado da sua área de trabalho.

Na linguagem C, letras minúsculas e maiúsculas são diferentes. Além disso, em programas na linguagem C não há distinção de onde você inicia a digitação das suas linhas e, assim, usamos essa característica a nosso favor, adicionando espaços em algumas linhas do programa para facilitar sua leitura. Essa adição de espaços em uma linha é chamada **indentação** ou **tabulação**. No Emacs, você pode usar a tecla `Tab` para adicionar indentações de forma adequada. Veremos que a indentação é uma prática de programação muito importante.

## 4.2 Compilando e executando

Abra um terminal onde possa digitar comandos, solicitando ao sistema operacional que os execute. Então, compile o programa 4.1 com o compilador gcc:

```
prompt$ gcc primeiro.c
prompt$
```

Depois disso, o programa executável `a.out` estará disponível para execução no mesmo diretório. Então é só digitar `a.out` (ou `./a.out`) na linha de comando do terminal para ver o resultado da execução:

```
prompt$ a.out
Programar é bacana!
prompt$
```

O nome `a.out` é um nome padrão que o compilador `gcc` dá aos arquivos executáveis resultantes de uma compilação. Em geral, atribuímos um nome mais significativo a um programa executável, como por exemplo o mesmo nome do programa na linguagem C, mas sem a sua extensão. No exemplo acima, o programa executável associado tem o nome `primeiro`. O compilador `gcc` pode gerar um executável dessa forma como segue:

```
prompt$ gcc primeiro.c -o primeiro
prompt$
```

## 4.3 Olhando o primeiro programa mais de perto

A primeira linha do nosso primeiro programa

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

será muito provavelmente incluída em todo programa que você fará na linguagem C. Essa linha fornece informações ao compilador sobre a função de saída de dados de nome `printf`, que é usada depois no programa.

A segunda linha do programa

```
int main(void)
```

informa ao compilador onde o programa inicia de fato. Em particular, essa linha indica que `main`, do inglês *principal*, é o início do programa principal e, mais que isso, que esse trecho do programa é na verdade uma função da linguagem C que não recebe valores de entrada (`void`) e devolve um valor inteiro (`int`). Esses conceitos de função, parâmetros de entrada de uma função e valor de saída serão elucidados oportunamente. Por enquanto, devemos memorizar que essa linha indica o início de nosso programa.

A próxima linha contém o símbolo abre-chave `{` que estabelece o início do bloco de comandos do programa. Em seguida, a próxima linha contém uma chamada à função `printf`:

```
printf("Programar é bacana!\n");
```

Essa função tem um único **argumento** ou **parâmetro**, que é a seqüência de símbolos da tabela ASCII `printf("Programar é bacana!\n");`. Note que todos os símbolos dessa seqüência são mostrados na saída, a menos de `\n`, que tem um significado especial quando sua impressão é solicitada por `printf`: saltar para a próxima linha. A função `printf` é uma rotina

da biblioteca `stdio.h` da linguagem C que mostra o seu argumento na saída padrão, que geralmente é o monitor.

Depois disso, adicionamos a linha

```
return 0;
```

que termina a execução do programa. Finalmente o símbolo fecha-chave `}` indica o final do bloco de comandos do programa.

## 4.4 Próximo programa

Vamos digitar um próximo programa.

Programa 4.2: Segundo programa.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int num1, num2, soma;
5     num1 = 25;
6     num2 = 30;
7     soma = num1 + num2;
8     printf("A soma de %d e %d é %d\n", num1, num2, soma);
9     return 0;
10 }
```

Este exemplo é ligeiramente diferente do primeiro e introduz algumas novidades. Agora, a primeira linha após a linha que contém o nome da função `main` apresenta uma **declaração de variáveis**. Três variáveis do tipo inteiro são declaradas: `num1`, `num2` e `soma`. Isso significa que, durante a execução desse programa, três compartimentos de memória são reservados pelo computador para armazenamento de informações que, neste caso, são números inteiros. Além disso, a cada compartimento é associado um nome: `num1`, `num2` e `soma`. Esses compartimentos de memória são também conhecidos como **variáveis**, já que seu conteúdo pode variar durante a execução de um programa.

Depois disso, na próxima linha do nosso programa temos uma **atribuição** do valor do tipo inteiro 25 para a variável `num1`. Observe então que o **símbolo de atribuição** da linguagem C é `=`. Na linha seguinte, uma outra atribuição é realizada, do número 30 para a variável `num2`. Na próxima linha temos uma atribuição para a variável `soma`. No entanto, note que não temos mais um número no lado direito da expressão de atribuição, mas sim a **expressão aritmética** `num1 + num2`. Neste caso, durante a execução dessa linha do programa, o computador consulta o conteúdo das variáveis `num1` e `num2`, realiza a operação de adição com os dois valores obtidos dessas variáveis e, após isso, atribui o resultado à variável `soma`. No nosso exemplo, o resultado da expressão aritmética $25 + 30$, ou seja, o valor $55$, será atribuído à variável `soma`. A linha seguinte de nosso programa contém uma chamada à função `printf`, mas agora com

quatro argumentos: o primeiro é agora uma **cadeia de caracteres de formatação**, contendo não apenas caracteres a serem impressos na saída, mas símbolos especiais, iniciados com `%`, conhecidos como **conversores de tipo** da linguagem C. Neste caso, os símbolos `%d` permitem que um número inteiro seja mostrado na saída. Note que três conversores `%d` estão contidos na seqüência de símbolos de saída e, a cada um deles e na ordem como são apresentados está associado uma das variáveis `num1`, `num2` e `soma`, que são os argumentos restantes da função `printf`.

## 4.5 Documentação

Uma boa documentação de um programa, conforme [4], significa inserir comentários apropriados no código de modo a explicar *o que* cada uma das funções que compõem o programa faz. A documentação de uma função é um pequeno manual que dá instruções precisas e completas sobre o uso da função.

Comentários são então introduzidos em programas com o objetivo de documentá-los e de incrementar a sua legibilidade. O(a) programador(a) é responsável por manter seus códigos legíveis e bem documentados para que, no futuro, possa retomá-los e compreendê-los sem muito esforço e sem desperdício de tempo. Na linguagem C padrão, os comentários são envolvidos pelos símbolos `/*` e `*/`. Um comentário é completamente ignorado quando encontrado pelo compilador e, portanto, não faz diferença alguma no programa executável produzido pelo compilador da linguagem.

A seguir, mostramos nosso último programa desta aula com comentários explicativos adicionados nos pontos onde são realmente necessários.

Programa 4.3: Segundo programa com comentários explicativos.

```
1  #include <stdio.h>
2  /* Esta função faz a adição de dois números inteiros fixos
3     e mostra o resultado da operação na saída. */
4  int main(void)
5  {
6     int num1, num2, soma;
7     num1 = 25;
8     num2 = 30;
9     soma = num1 + num2;
10    printf("A soma de %d e %d é %d\n", num1, num2, soma);
11    return 0;
12 }
```

Ainda conforme [4], uma boa documentação não se preocupa em explicar *como* uma função faz o que faz, mas sim *o que* ela faz de fato, informando quais são os valores de entrada da função, quais são os valores de saída e quais as relações que esses valores que entram e saem da função e as transformações pela função realizadas.

## Exercícios

4.1 Escreva uma programa na linguagem C que escreva a seguinte mensagem na saída padrão:

```
a) Comentários na linguagem C iniciam com /* e terminam com */
b) Letras minúsculas e maiúsculas são diferentes na linguagem C
c) A palavra chave main indica o início do programa na linguagem C
d) Os símbolos { e } envolvem um bloco de comandos na linguagem C
e) Todos os comandos na linguagem C devem terminar com um ponto e vírgula
```

4.2 Qual é a saída esperada para o programa 4.4?

Programa 4.4: Programa do exercício 4.2.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     printf("Alô! ");
5     printf("Alô! ");
6     printf("Tem alguém aí?");
7     printf("\n");
8     return 0;
9  }
```

4.3 Escreva um programa na linguagem C que subtraia 14 de 73 e mostre o resultado na saída padrão com uma mensagem apropriada.

4.4 Verifique se o programa 4.5 está correto. Em caso negativo, liste os erros de digitação que você encontrou.

Programa 4.5: Programa do exercício 4.4.

```
1  include <stdio.h>
2  /* computa a soma de dois números
3  Int main(void)
4  {
5     int resultado;
6     resultado = 13 + 22 - 7
7     printf("O resultado da operação é %d\n" resultado);
8     return 0;
9  }
```

AULA 5

# ENTRADA E SAÍDA

A função usada para entrada de dados `scanf` e a função usada para saída de dados `printf` serão estudadas nesta aula. Estas são funções básicas da linguagem C que fazem a interação com usuário durante a execução de um programa, solicitando valores de entrada e mostrando valores na saída.

## 5.1 Um exemplo

Vamos ver um primeiro exemplo de programa na linguagem C que usa a função de leitura `scanf`. Abra seu Emacs, pressione `C-x C-f`, dê um nome para seu programa, como por exemplo `soma.c`, e digite o código apresentado no programa 5.1.

Programa 5.1: Programa com entrada e saída de dados.

```
1  #include <stdio.h>
2  /* Programa que lê dois números inteiros e mostra
3     o resultado da soma desses dois números */
4  int main(void)
5  {
6     int num1, num2, soma;
7     printf("Informe um número: ");
8     scanf("%d", &num1);
9     printf("Informe outro número: ");
10    scanf("%d", &num2);
11    soma = num1 + num2;
12    printf("A soma de %d mais %d é %d\n", num1, num2, soma);
13    return 0;
14 }
```

Vamos olhar mais de perto este programa. Observe que, pela primeira vez, usamos a função `scanf`. Desconsiderando o esqueleto do programa, a primeira linha a ser notada é a declaração de variáveis:

```
int num1, num2, soma;
```

Estamos, com esta linha do programa, reservando três compartimentos de memória que poderão ser utilizados para armazenamento de números inteiros, cujos nomes desses compartimentos são `num1`, `num2` e `soma`.

Em seguida, há uma função para escrita e uma função para leitura:

```
printf("Informe um número:  ");
scanf("%d", &num1);
```

A função para escrita `printf` já é bem conhecida, vista nas aulas anteriores. A função de leitura `scanf` tem dois parâmetros: uma cadeia de caracteres de formatação `"%d"` e uma variável `num1` correspondente a este formato. Diferentemente da função `printf`, a variável na função `scanf` deve vir precedida com o símbolo `&`. No comando acima, a cadeia de caracteres de formatação `"%d"` indica que o valor informado pelo usuário será convertido para um inteiro. Ainda, este valor será então armazenado na variável do tipo inteiro `num1`. Observe, mais uma vez, que esta variável é precedida por um `&`.

A próxima seqüência de instruções é equivalente e solicita ao usuário do programa a entrada de um outro número, que será armazenado na variável `num2`:

```
printf("Informe outro número:  ");
scanf("%d", &num2);
```

Depois disso, há um comando um pouco mais complexo:

```
soma = num1 + num2;
```

Este é um **comando de atribuição** da linguagem C, determinado pelo símbolo `=`. Um comando de atribuição tem uma sintaxe bem definida: do lado esquerdo do símbolo de atribuição `=` há sempre uma variável; do lado direito há uma constante, uma variável ou uma expressão do mesmo tipo da variável à esquerda. Este comando funciona da seguinte forma: avalia a expressão do lado direito do símbolo de atribuição e atribui o resultado desta avaliação para a variável do lado esquerdo. Em nosso exemplo, o lado direito é uma expressão aritmética de adição, que soma os conteúdos das variáveis do tipo inteiro `num1` e `num2`. O resultado da avaliação dessa expressão é um número inteiro que será atribuído à variável `soma`, do lado esquerdo do símbolo de atribuição.

Por fim, a chamada à função

```
printf("A soma de %d mais %d é %d\n", num1, num2, soma);
```

mostra, além dos valores das duas parcelas `num1` e `num2`, o resultado dessa soma armazenado na variável `soma`.

## 5.2 Segundo exemplo

Este segundo exemplo é muito semelhante ao exemplo anterior. A diferença é apenas a expressão aritmética: no primeiro exemplo, uma adição é realizada; neste segundo exemplo, uma multiplicação é realizada. O operador aritmético de multiplicação é o símbolo `*`. Informe, compile e execute o seguinte programa.

Programa 5.2: Segundo exemplo.

```
#include <stdio.h>
/* A função main lê dois números inteiros e mostra na saída
   o resultado da multiplicação desses dois números */
int main(void)
{
   int num1, num2, produto;
   printf("Informe um número: ");
   scanf("%d", &num1);
   printf("Informe outro número: ");
   scanf("%d", &num2);
   produto = num1 * num2;
   printf("O produto de %d por %d é %d\n", num1, num2, produto);
   return 0;
}
```

## Exercícios

5.1 Faça um programa que leia três números inteiros $a$, $b$ e $c$, calcule $a * b + c$ e mostre o resultado na saída padrão para o usuário.

Programa 5.3: Uma solução possível para o exercício 5.1.

```
#include <stdio.h>
int main(void)
{
   int a, b, c;
   printf("Informe 3 números inteiros: ");
   scanf("%d %d %d", &a, &b, &c);
   printf("A expressão %d*%d+%d tem resultado %d.\n", a, b, c, a*b+c);
   return 0;
}
```

5.2 Faça um programa que leia um número inteiro e mostre o seu quadrado e seu cubo. Por exemplo, se o número de entrada é 3, a saída deve ser 9 e 27.

5.3 Faça um programa que leia três números inteiros e mostre como resultado a soma desses três números e também a multiplicação desses três números.

5.4 Escreva um programa que leia um número inteiro e mostre o resultado da quociente (inteiro) da divisão desse número por 2 e por 3.

AULA 6

# ESTRUTURAS CONDICIONAIS

Nesta aula veremos alguns pequenos exemplos de programas que usam estruturas condicionais. Além da estrutura seqüencial, aquela que tivemos contato nas aulas anteriores, a estrutura condicional é uma ferramenta de programação importante que auxilia o programador a decidir que trechos de programa devem ser executados ou não, dependendo da avaliação da condição expressa.

Para maior compreensão das estruturas condicionais é certamente necessário o entendimento dos conceitos de constantes, variáveis, expressões aritméticas e expressões lógicas da linguagem C. Por enquanto, esses conceitos ficarão implícitos, já que vamos estudá-los com bastante detalhe na aula 7.

## 6.1 Estrutura condicional simples

Como vimos nas aulas anteriores, a linguagem C é usada para solicitar que o computador realize uma seqüência de comandos ou instruções tais como leitura de dados, impressão de dados e atribuição. Além de ter esse poder, a linguagem C também pode ser usada para tomar decisões. Uma decisão, na linguagem C, é tomada com o uso de uma estrutura condicional simples ou de uma estrutura condicional composta. Uma estrutura condicional simples tem o seguinte formato:

```
if (condição) {
   comando 1;
   comando 2;
   ...
   comando n;
}
```

para `n` $\geqslant 1$. No caso em que `n` $= 1$, os delimitadores de bloco, isto é, as chaves `{` e `}`, são opcionais.

Uma decisão é tomada de acordo com a avaliação da `condição`, que é uma expressão lógica: caso o resultado dessa avaliação seja verdadeiro, o comando ou o bloco de comandos será executado; caso contrário, será ignorado. Se o número de comandos a serem executados em um bloco interno à estrutura condicional é pelo menos dois, então esse bloco tem de vir envolvido por chaves.

Vejamos um pequeno exemplo no programa 6.1.

Programa 6.1: Um exemplo contendo uma estrutura condicional simples.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int idade;
5     printf("Quantos anos você tem? ");
6     scanf("%d", &idade);
7     if (idade < 30)
8        printf("Nossa! Você é bem jovem!\n");
9     printf("Até breve!\n");
10    return 0;
11 }
```

Neste programa 6.1, uma única variável `idade` do tipo inteiro é declarada no início do programa. Em seguida, é solicitado ao usuário do programa 6.1 que informe um número inteiro, que será armazenado na variável `idade`. *Se* o valor informado pelo usuário for um número inteiro menor que 30, então uma mensagem `Nossa! Você é bem jovem!` aparecerá no monitor. Por fim, a mensagem `Até breve!` é impressa.

## 6.2 Estrutura condicional composta

Uma estrutura condicional composta tem o seguinte formato geral:

```
if (condição) {
   comando 1;
   comando 2;
   ...
   comando m;
}
else {
   comando 1;
   comando 2;
   ...
   comando n;
}
```

para `m`, `n` $\geqslant 1$. No caso em que `m` $= 1$ ou `n` $= 1$, os delimitadores de bloco, isto é, as chaves `{` e `}`, são opcionais.

Então, vejamos um outro exemplo no programa 6.2.

Programa 6.2: Um exemplo usando uma estrutura condicional composta.

```
#include <stdio.h>
int main(void)
{
   int idade;
   printf("Quantos anos você tem? ");
   scanf("%d", &idade);
   if (idade < 30)
      printf("Nossa! Você é bem jovem!\n");
   else
      printf("Puxa! Você já é velhinho!\n");
   printf("Até breve!\n");
   return 0;
}
```

Observe agora que no programa 6.2 as mensagens que podem ser emitidas para o usuário são excludentes.

## 6.3 Troca de conteúdos

O programa 6.3 lê dois números inteiros quaisquer e os imprime em ordem não decrescente.

Programa 6.3: Um exemplo de troca de conteúdos entre variáveis do mesmo tipo.

```
#include <stdio.h>
int main(void)
{
   int x, y, aux;
   printf("Informe o valor de x: ");
   scanf("%d", &x);
   printf("Informe o valor de y: ");
   scanf("%d", &y);
   if (x > y)
   {
      aux = x;
      x = y;
      y = aux;
   }
   printf("%d é menor ou igual a %d\n", x, y);
   return 0;
}
```

Na verdade, o programa 6.3 lê dois valores do tipo inteiro e os armazena nas variáveis `x` e `y` e, em seguida, armazena o menor desses dois valores em `x` e o maior em `y`. Se houver empate, a mensagem emitida está correta. Suponha que o programa 6.3 está sendo executado e que os valores de `x` e `y` já foram lidos. Observe primeiro que se um usuário desse programa informa um valor para `x` e um valor para `y` de tal forma que o valor de `x` é menor ou igual ao valor de `y` então a avaliação da condição `x > y` tem resultado falso e o bloco de comandos interno à estrutura condicional não é executado. Agora suponha o contrário, isto é, suponha que o usuário do programa informa um valor para `x` maior que o valor informado para `y`. Então, a avaliação da condição `x > y` tem resultado verdadeiro, o bloco de comandos interno à estrutura condicional é executado e uma "troca" será realizada: o conteúdo da variável `x` é armazenado temporariamente na variável `aux`, a variável `x` recebe o conteúdo da variável `y` e, finalmente, a variável `y` recebe o conteúdo da variável `aux`. Note que a troca não poderia ser realizada sem o uso de uma variável auxiliar. A última instrução do programa é a chamada à função de impressão `printf` para mostrar o conteúdo das variáveis `x` e `y`. Por exemplo, suponha que um usuário do programa 6.3 forneça o valor 2 para `x` e 5 para `y`. Então, a troca não é realizada e a mensagem impressa na saída padrão é `2 é menor ou igual a 5`. Se, ao contrário, o usuário fornece 5 para `x` e 2 para `y`, a troca é realizada e a mensagem na saída padrão é também `2 é menor ou igual a 5`.

## Exercícios

6.1 Escreva um programa que receba a temperatura ambiente em graus Célsius e mostre uma mensagem para o usuário informando se a temperatura está muito quente. Considere como temperatura limite o valor de 30 graus Célsius.

Programa 6.4: Uma possível solução para o exercício 6.1.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int temperatura;
5     printf("Informe uma temperatura (em graus Celsius): ");
6     scanf("%d", &temperatura);
7     if (temperatura > 30)
8       printf("Hojé é um dia bem quente!\n");
9     else
10      printf("Hoje não está tão quente.\n");
11    return 0;
12 }
```

6.2 Escreva um programa que receba três valores, armazenando-os nas variáveis $x$, $y$ e $z$, e ordene esses valores de modo que, ao final, o menor valor esteja armazenado na variável $x$, o valor intermediário esteja armazenado na variável $y$ e o maior valor esteja armazenado na variável $z$.

AULA 7

# NÚMEROS INTEIROS

Nas aulas anteriores, nos programas que fizemos, tomamos contato com a estrutura de programação seqüencial, onde comandos de atribuição, chamadas de funções de entrada e saída e também declarações de variáveis são dispostos linha após linha em um bloco de comandos envolvido por `{` e `}`. Vimos o uso de funções, com seus argumentos, tais como a função para entrada de dados `scanf` e a função para saída de dados `printf`. Também vimos as estruturas condicionais da linguagem C, onde o fluxo de execução de um programa pode ser modificado de acordo com a avaliação de uma expressão lógica.

A partir da aula de hoje formalizaremos todos esses conceitos. Iniciamos essa formalização com os números inteiros, definindo constantes, variáveis e expressões que envolvem esses números. No final desta aula, explicamos também como é feita a representação de números inteiros em um computador.

## 7.1 Constantes e variáveis

Como mencionamos nas aulas 1 e 2, os primeiros programadores tinham de programar diretamente na linguagem que o computador compreende, a linguagem de máquina ou linguagem binária, já que não existia na época nenhuma linguagem de alto nível. Isso significa que instruções em código binário deviam ser escritas pelos programadores antes que fossem digitadas nesses computadores. Além disso, os programadores tinham de referenciar explicitamente um valor que representasse um endereço de memória para armazenar valores temporários. Felizmente, as linguagens de programação de alto nível permitem que um programador se concentre muito mais na solução do problema que em códigos específicos da máquina e em endereços de memória.

Uma linguagem de alto nível permite que um programador use facilmente compartimentos de memória para armazenamento de informações durante a execução de um programa. Como valores podem ser atribuídos e sobrescritos nesses compartimentos e, portanto, podem variar durante a execução do programa, esses compartimentos de memória são chamados de **variáveis** do programa. Além disso, uma linguagem de alto nível permite que um programador atribua um nome simbólico a um endereço de um compartimento de memória. Esse nome simbólico é conhecido como **identificador** ou **nome** da variável. O identificador de uma variável pode ser escolhido pelo programador de modo a refletir o seu conteúdo. Nas aulas anteriores, temos usado diversas variáveis para armazenar valores que são números inteiros, como `num1`, `soma` e `produto`, por exemplo. A linguagem C permite o uso de outros tipos de dados além de inteiros, como veremos mais adiante.

Há uma regra fundamental sobre variáveis e programação na linguagem C: *qualquer variável usada em um programa deve ser previamente declarada.* Há também regras para um programador determinar os identificadores de variáveis na linguagem C. Essas regras são as apresentadas a seguir:

- os símbolos válidos em um identificador de variável são os caracteres do alfabeto da língua inglesa, minúsculos ou maiúsculos, os dígitos numéricos e o sublinhado[1] (_);
- o identificador de uma variável deve iniciar com uma letra ou com um sublinhado (_);
- após o primeiro caracter, o identificador de uma variável é determinado por qualquer seqüência de letras minúsculas ou maiúsculas, de números ou de sublinhados.

Observe que as regras descritas acima excluem quaisquer caracteres que estejam fora do conjunto { `a` , . . . , `z` , `A` , . . . , `Z` } e isso significa que caracteres acentuados e cedilhas não podem fazer parte de identificadores de variáveis. Dessa forma, os seguintes identificadores são nomes válidos de variáveis:

```
soma
num1
i
soma_total
_sistema
A3x3
```

Por outro lado, os identificadores listados a seguir não são nomes válidos de variáveis na linguagem C:

```
preço$
soma total
4quant
int
```

No primeiro caso, o identificador `preço$` contém dois símbolos não válidos: ç e $. O segundo identificador também tem o mesmo problema, já que um identificador não pode conter um caracter espaço. O terceiro exemplo tem um identificador que inicia com um caracteres não válido, um dígio numérico. O último exemplo é um identificador não válido já que `int` é uma palavra reservada da linguagem C.

Também é importante destacar que letras minúsculas e maiúsculas na linguagem C são diferentes. Assim, as variáveis `soma` , `Soma` e `SOMA` são diferentes.

A declaração de variáveis do tipo inteiro na linguagem C pode ser realizada com um único comando `int` e mais uma lista de identificadores de variáveis, separados por um espaço e

[1] Do inglês *underscore* ou *underline*.

uma vírgula, e finalizada com um ponto e vírgula; ou ainda, pode ser realizada uma a uma, com um comando `int` para cada variável:

```
int i, num1, num2, soma, produto, aux;
```

ou

```
int i;
int num1;
int num2;
int soma;
int produto;
int aux;
```

Os identificadores das variáveis podem ser tão longos quanto se queira, apesar de apenas os 63 primeiros caracteres serem significativos para o compilador da linguagem C. Um programa escrito com identificadores de variáveis muito longos pode ser difícil de ser escrito. Por exemplo,

```
TotalDeTodoDinheiroQueTenhoGuardado = TotalDoDinheiroQueGuardeiNoAnoPassado +
          TotalDeDinheiroQueGuardeiEm2009 - TotalDeImpostosEmReais;
```

é bem menos significativo que

```
TotalGeral = TotalPassado + Total2009 - Impostos;
```

Em um programa na linguagem C, os números inteiros que ocorrem em expressões são conhecidos como **constantes**. Por exemplo, o número 37 representa um valor inteiro constante. Expressões aritméticas que se constituem apenas de valores constantes são chamadas de **expressões aritméticas constantes**. Por exemplo,

```
781 + 553 - 12 * 44
```

é uma expressão aritmética constante.

## 7.2 Expressões aritméticas com números inteiros

A linguagem C possui cinco operadores aritméticos binários que podemos usar com números inteiros: `+` para adição, `-` para subtração, `*` para multiplicação, `/` para divisão e `%` para resto da divisão. Devemos ressaltar que todos esses operadores são **binários**, isto é, necessitam de dois operandos para execução da operação aritmética correspondente. Esses operandos podem ser constantes, variáveis ou expressões aritméticas do tipo inteiro. O programa 7.1 é um exemplo de programa que usa todos esses operadores em expressões aritméticas com números inteiros.

Programa 7.1: Programa contendo expressões aritméticas com números inteiros.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int x, y, z, resultado;
5     x = 25;
6     y = 10;
7     resultado = x + y;
8     printf("A soma de %d e %d é %d\n", x, y, resultado);
9     x = 38;
10    y = 29;
11    resultado = x - y;
12    printf("A substração de %d e %d é %d\n", x, y, resultado);
13    x = 51;
14    y = 17;
15    resultado = x * y;
16    printf("A multiplicação de %d por %d é %d\n", x, y, resultado);
17    x = 100;
18    y = 25;
19    resultado = x / y;
20    printf("A divisão de %d por %d é %d\n", x, y, resultado);
21    x = 17;
22    y = 3;
23    resultado = x % y;
24    printf("O resto da divisão de %d por %d é %d\n", x, y, resultado);
25    x = -7;
26    resultado = -x;
27    printf("%d com sinal trocado é %d\n", x, resultado);
28    x = 10;
29    y = 4;
30    z = 15;
31    resultado = x + y * z;
32    printf("A expressão %d + %d * %d tem resultado %d\n", x, y, z, resultado);
33    return 0;
34 }
```

Há ainda um operador **unário** `-` na linguagem C, que trabalha sobre um único valor e que troca o seu sinal, isto é, troca o sinal de uma constante, uma variável ou uma expressão aritmética do tipo inteiro. No exemplo acima, o comando de atribuição `x = -7;` na linha 25 e o comando de atribuição `resultado = -x;` na linha 26 fazem com que o valor armazenado na variável `resultado` seja `7`.

Observe finalmente que o resultado da avaliação da expressão aritmética `10 + 4 * 15` não é $14 \times 15 = 210$, mas sim $10 + 60 = 70$. Os operadores aritméticos na linguagem C têm precedências uns sobre os outros. O operador de menos unário precede todos os outros. Multiplicações, divisões e restos de divisões têm precedência sobre a adição e subtração e, por isso, o resultado da operação

```
resultado = x + y * z;
```

é dado primeiro pela avaliação da multiplicação e depois pela avaliação da adição. Parênteses são utilizados para modificar a ordem das operações. Por exemplo,

```
resultado = (x + y) * z;
```

teria como resultado da avaliação o valor 210.

Segue um resumo das precedências dos operadores aritméticos binários sobre números inteiros:

| Operador | Descrição | Precedência |
|---|---|---|
| `*` `/` `%` | Multiplicação, divisão e resto da divisão | 1 (máxima) |
| `+` `-` | Adição e subtração | 2 (mínima) |

## 7.3 Representação de números inteiros

Esta seção é completamente inspirada no apêndice C do livro [4].

A memória do computador é uma seqüência de bytes. Um **byte** consiste em 8 bits, onde **bit** significa a menor unidade de informação que pode ser armazenada na memória e pode assumir apenas dois valores possíveis: 0 ou 1. Assim, um byte pode assumir 256 valores possíveis: `00000000`, `00000001`, ..., `11111111`. De fato, o conjunto de todas as seqüências de $k$ bits representa os números naturais de 0 a $2^k - 1$.

Uma seqüência de bits é chamada de um **número binário** ou **número na base 2**. Por exemplo, `010110` é um número binário representado por uma seqüência de 6 bits. Podemos converter um número binário em um **número decimal**, ou **número na base 10**, que nada mais é que um número natural. Por exemplo, o número binário `010110` representa o número decimal 22, pois $0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$.

Na linguagem C, os números inteiros positivos e negativos são conhecidos como inteiros com sinal. Para declarar uma variável `i` do tipo inteiro com sinal, temos de digitar

```
int i;
```

Cada variável do tipo `int` é armazenada em $b$ bytes consecutivos na memória, onde $b$ é dependente da arquitetura do computador. A maioria dos computadores pessoais atuais têm $b = 4$ e assim cada variável do tipo `int` é representada por uma seqüência de $8 \times 4 = 32$ bits, perfazendo um total de $2^{32}$ valores possíveis, contidos no conjunto

$$-2^{31}, \ldots, -1, 0, 1, \ldots, 2^{31} - 1$$

ou

$$-2147483648, \ldots, -1, 0, 1, \ldots, 2147483647 \; .$$

Cada seqüência de 32 bits que começa com `0` representa um `int` não-negativo em notação binária. Cada seqüência de 32 bits que começa com `1` representa o inteiro negativo $k - 2^{32}$, onde $k$ é o valor da seqüência em notação binária. Esta representação de números inteiros negativos é chamada de **complemento-de-dois**. Números inteiros que estejam fora do intevalo $\left[-2^{31}, 2^{31} - 1\right]$ são representados módulo $2^{32}$.

Ressaltamos ainda que todos os valores acima podem ser obtidos usando fórmulas gerais em função do número de bytes consecutivos usados para representar um `int`.

## Exercícios

7.1 Escreva um programa que receba um número inteiro $x$ e avalie o polinômio

$$p(x) = 3x^3 - 5x^2 + 2x - 1 \; .$$

Programa 7.2: Possível solução para o exercício 7.1.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int x, p;
5     printf("Informe x: ");
6     scanf("%d", &x);
7     p = 3 * x * x * x - 5 * x * x + 2 * x - 1;
8     printf("p(%d) = %d.\n", x, p);
9     return 0;
10 }
```

7.2 Para transformar um número inteiro $i$ no menor inteiro $m$ maior que $i$ e múltiplo de um número inteiro $j$, a seguinte fórmula pode ser utilizada:

$$m = i + j - i \bmod j \; ,$$

onde o operador mod é o operador de resto de divisão inteira na notação matemática usual, que corresponde ao operador `%` na linguagem C.

Por exemplo, suponha que usamos $i = 256$ dias para alguma atividade e queremos saber qual o total de dias $m$ que devemos ter de forma que esse número seja divisível por $j = 7$, para termos uma idéia do número de semanas que usaremos na atividade. Então, pela fórmula acima, temos que

$$\begin{aligned} m &= 256 + 7 - 256 \bmod 7 \\ &= 256 + 7 - 4 \\ &= 259 \, . \end{aligned}$$

Escreva um programa que receba dois números inteiros positivos $i$ e $j$ e devolva o menor inteiro $m$ maior que $i$ e múltiplo de $j$.

AULA 8

# ESTRUTURA DE REPETIÇÃO `while`

Estruturas de repetição são, juntamente com a estrutura seqüencial e as estruturas condicionais, elementos fundamentais em algoritmos e programação. São essas as estruturas básicas que permitem que um programador consiga resolver centenas e centenas de problemas de maneira efetiva. Nesta aula motivaremos o estudo das estruturas de repetição e apresentaremos a estrutura de repetição `while` da linguagem C.

## 8.1 Motivação

Suponha que alguém queira escrever um programa para imprimir os dez primeiros números inteiros positivos. Este problema, apesar de muito simples e pouco útil à primeira vista, motiva a introdução das estruturas de repetição, como veremos adiante. Com o que aprendemos até o momento, é bem fácil escrever um programa como esse.

Programa 8.1: Um programa para imprimir os 10 primeiros inteiros positivos.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     printf("1\n");
5     printf("2\n");
6     printf("3\n");
7     printf("4\n");
8     printf("5\n");
9     printf("6\n");
10    printf("7\n");
11    printf("8\n");
12    printf("9\n");
13    printf("10\n");
14    return 0;
15 }
```

Além de bem simples, o programa 8.1 parece estar realmente correto, isto é, parece realizar o propósito de imprimir os dez primeiros números inteiros positivos de forma correta. Mas e se quiséssemos imprimir os 1.000 primeiros números inteiros positivos? Um programa seme-

lhante ao programa acima ficaria um pouco mais complicado e monótono de escrever, já que teria muitas e muitas linhas de código.

Uma das propriedades fundamentais dos computadores é que eles possuem a capacidade de executar repetitivamente um conjunto de comandos. Essa capacidade de repetição permite que um programador escreva programas mais concisos que contenham processos repetitivos que poderiam necessitar de centenas ou milhares de linhas se não fossem assim descritos. A linguagem de programação C tem três estruturas de repetição diferentes. A seguir veremos a estrutura `while`.

## 8.2 Estrutura de repetição `while`

Vamos fazer um programa que mostra os primeiros 100 números na saída padrão, linha após linha, usando uma estrutura de repetição. Veja o programa 8.2.

Programa 8.2: Um programa para imprimir os inteiros de 1 a 100.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int numero;
5     numero = 1;
6     while (numero <= 100) {
7       printf("%d\n", numero);
8       numero = numero + 1;
9     }
10    return 0;
11 }
```

A novidade neste programa é a estrutura de repetição `while`. O programa tem início com a declaração da variável `numero` que pode armazenar números inteiros. A linha 5 contém um comando de atribuição que faz com que a constante numérica `1` seja armazenada na variável `numero`. Depois disso, na linha 6 temos a estrutura de repetição `while`.

A estrutura de repetição `while` é composta por um conjunto de elementos: (*i*) *inicialização* da variável que controla a estrutura de repetição; (*ii*) *condição*, ou *expressão lógica*, que controla a repetição do bloco de comandos da estrutura; (*iii*) *seqüência de comandos*, que formam o bloco de comandos, a menos da última linha, da estrutura de repetição e que serão executados; e (*iv*) *passo*, que é, em geral, a última linha do bloco de comandos da estrutura de repetição e determina a freqüência com que o bloco de comandos será executado. No programa 8.2, a inicialização é realizada antes da estrutura de repetição `while`, como mostrado na linha 5, com o comando de atribuição `numero = 1;`. A condição, ou expressão lógica, que sempre vem envolvida por parênteses e aparece logo após o comando `while`, é dada `numero <= 100`. A seqüência de comandos é composta apenas pelo comando `printf("%d\n", numero);`. E o passo vem, em geral, após a seqüência de comandos. No caso do nosso programa o passo é um incremento, `numero = numero + 1`.

O funcionamento da estrutura de repetição `while` é dado da seguinte forma. Sempre que um programa encontra o comando `while` a expressão lógica envolvida por parênteses que vem logo a seguir é avaliada. Se o resultado da avaliação for verdadeiro, o bloco de comandos interno a essa estrutura será executado. Ao final da seqüência de comandos, o fluxo de execução volta ao comando `while` para nova avaliação da sua expressão lógica. Se novamente o resultado da avaliação dessa expressão for verdadeiro, seu bloco de comandos é repetido. Esse processo termina quando o resultado da avaliação da expressão lógica é falso. Quando isso acontece, o fluxo de execução do programa salta para a próxima linha após o bloco de comandos da estrutura de repetição `while`. No caso do nosso programa, a linha alcançada por esse salto é a linha 10.

Vamos fazer agora um programa que soma os 100 primeiros números inteiros positivos. Veja o programa 8.3.

Programa 8.3: Programa que soma os 100 primeiros números inteiros positivos.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int numero, soma;
5     soma = 0;
6     numero = 1;
7     while (numero <= 100) {
8       soma = soma + numero;
9       numero = numero + 1;
10    }
11    printf("A soma dos 100 primeiros inteiros positivos é %d\n", soma);
12    return 0;
13 }
```

## Exercícios

8.1 Dado um inteiro positivo $n$, somar os $n$ primeiros inteiros positivos.

8.2 Dado $n$, imprimir os $n$ primeiros naturais ímpares.

Exemplo:

Para $n = 4$ a saída deverá ser $1, 3, 5, 7$.

8.3 Dado um inteiro positivo $n$, calcular $n!$.

8.4 Dado $n$, imprimir as $n$ primeiras potências de 2.

Exemplo:

Para $n = 5$ a saída deverá ser $1, 2, 4, 8, 16$.

8.5 Dados $x$ inteiro e $n$ um natural, calcular $x^n$.

Programa 8.4: Programa para o exercício 8.1.

```
#include <stdio.h>
int main(void)
{
   int n, numero, soma;
   printf("Informe n: ");
   scanf("%d", &n);
   soma = 0;
   numero = 1;
   while (numero <= n) {
     soma = soma + numero;
     numero = numero + 1;
   }
   printf("O resultado da soma dos %d primeiros inteiros é %d\n", n, soma);
   return 0;
}
```

8.6 Dado um inteiro positivo $n$ e uma seqüência de $n$ inteiros, somar esses $n$ números.

AULA 9

# EXERCÍCIOS

Nesta aula faremos exercícios para treinar a estrutura de repetição `while`.

## Enunciados

9.1 Dado um inteiro positivo $n$ e uma seqüência de $n$ números inteiros, determinar a soma dos números inteiros positivos da seqüência.

Exemplo:

> Se $n = 7$ e a seqüência de números inteiros é $6, -2, 7, 0, -5, 8, 4$ a saída deve ser $19$.

Programa 9.1: Solução para o exercício 9.1.

```
#include <stdio.h>
int main(void)
{
  int n, conta, num, soma;
  printf("Informe n: ");
  scanf("%d", &n);
  soma = 0;
  conta = 1;
  while (conta <= n) {
    printf("Informe um número: ");
    scanf("%d", &num);
    if (num > 0)
      soma = soma + num;
    conta = conta + 1;
  }
  printf("A soma dos números positivos da seqüência é %d.\n", soma);
  return 0;
}
```

9.2 Dado um inteiro positivo $n$ e uma seqüência de $n$ inteiros, somar os números pares e os números ímpares.

9.3 Durante os 31 dias do mês de março foram tomadas as temperaturas médias diárias de Campo Grande, MS. Determinar o número de dias desse mês com temperaturas abaixo de zero.

9.4 Dado um inteiro positivo $n$ e uma seqüência de $n$ inteiros, determinar quantos números da seqüência são positivos e quantos são não-positivos. Um número é não-positivo se é negativo ou se é igual a 0 (zero).

9.5 Dado um inteiro positivo $n$ e uma seqüência de $n$ inteiros, determinar quantos números da seqüência são pares e quantos são ímpares.

9.6 Uma loja de discos anota diariamente durante o mês de abril a quantidade de discos vendidos. Determinar em que dia desse mês ocorreu a maior venda e qual foi a quantidade de discos vendida nesse dia.

9.7 Dados o número $n$ de estudantes de uma turma de Programação de Computadores I e suas notas de primeira prova, determinar a maior e a menor nota obtidas por essa turma, onde a nota mínima é $0$ e a nota máxima é $100$.

AULA 10

# EXERCÍCIOS

## Enunciados

10.1 Dado um número inteiro positivo $n$ e dois números naturais não nulos $i$ e $j$, imprimir em ordem crescente os $n$ primeiros naturais que são múltiplos de $i$ ou de $j$ ou de ambos.

Exemplo:

Para $n = 6, i = 2$ e $j = 3$ a saída deverá ser 0, 2, 3, 4, 6, 8.

Programa 10.1: Solução para o exercício 10.1.

```
#include <stdio.h>
int main(void)
{
  int n, i, j, numero, contador;
  printf("Informe os números n, i e j: ");
  scanf("%d %d %d", &n, &i, &j);
  numero = 0;
  contador = 1;
  while (contador <= n) {
    if (numero % i == 0) {
      printf("%d ", numero);
      contador = contador + 1;
    }
    else {
      if (numero % j == 0) {
        printf("%d ", numero);
        contador = contador + 1;
      }
    }
    numero = numero + 1;
  }
  printf("\n");
  return 0;
}
```

Programa 10.2: Outra solução para o exercício 10.1.

```c
#include <stdio.h>
int main(void)
{
  int n, i, j, numero, contador;
  printf("Informe o número n: ");
  scanf("%d", &n);
  printf("Informe os números i e j: ");
  scanf("%d %d", &i, &j);
  numero = 0;
  contador = 1;
  while (contador <= n) {
    if ((numero % i == 0) || (numero % j == 0)) {
      printf("%d ", numero);
      contador = contador + 1;
    }
    numero = numero + 1;
  }
  printf("\n");
  return 0;
}
```

10.2 Dizemos que um número natural é **triangular** se é produto de três números naturais consecutivos.

Exemplo:

120 é triangular, pois $4 \cdot 5 \cdot 6 = 120$.

Dado $n$ natural, verificar se $n$ é triangular.

10.3 Dados dois números inteiros positivos, determinar o máximo divisor comum entre eles utilizando o algoritmo de Euclides.

Exemplo:

| | 1 | 1 | 1 | 2 |
|---|---|---|---|---|
| 24 | 15 | 9 | 6 | 3 |
| 9 | 6 | 3 | 0 | |

= mdc(24,15)

10.4 Dados dois números inteiros positivos $a$ e $b$, representando a fração $a/b$, escreva um programa que reduz $a/b$ para uma fração irredutível.

Exemplo:

Se a entrada é $9/12$ a saída tem de ser $3/4$.

10.5 Dados a quantidade de dias de um mês e o dia da semana em que o mês começa, escreva um programa que imprima os dias do mês por semana, linha a linha. Considere o dia da semana 1 como domingo, 2 como segunda-feira, e assim por diante, até o dia 7 como sábado.

Exemplo:

Se a entrada é 31 e 3 então a saída deve ser

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | 31 | | |

AULA 11

# EXPRESSÕES COM INTEIROS

Nesta aula veremos expressões aritméticas, relacionais e lógicas com números inteiros. Em aulas anteriores já tomamos contato com essas expressões, mas pretendemos formalizá-las aqui. A compreensão destas expressões com números inteiros é fundamental para que possamos escrever programas mais eficientes e poderosos e também porque podemos estender esse entendimento a outros tipos de variáveis.

## 11.1 Expressões aritméticas

Uma **expressão aritmética** é qualquer seqüência de símbolos formada apenas por constantes numéricas, variáveis numéricas, operadores aritméticos e parênteses. Como já vimos, uma **constante numérica do tipo inteiro** é qualquer número inteiro digitado em seu programa, como $34$ ou $-7$. Uma **variável numérica do tipo inteiro** é aquela que foi declarada com um comando `int` no início do programa. Os **operadores aritméticos** são divididos em duas classes: operadores aritméticos unários e operadores aritméticos binários. Só conhecemos, até agora, um **operador aritmético unário**: o `-`, que troca o sinal da expressão aritmética que o sucede. Existe também um outro operador aritmético unário, o `+` que não faz nada além de enfatizar que uma constante numérica é positiva. Os **operadores aritméticos binários** são aqueles que realizam as operações básicas sobre dois inteiros: `+` para adição, `-` para subtração, `*` para multiplicação, `/` para divisão e `%` para o resto da divisão.

As expressões aritméticas formadas por operações binárias envolvendo operandos e operadores têm precedências umas sobre as outras: as operações de multiplicação, divisão e resto da divisão têm prioridade sobre as operações de adição e a subtração. Caso haja necessidade de modificar a prioridade de uma operação em uma expressão, parênteses devem ser utilizados. Expressões envolvidas por parênteses têm maior prioridade sobre expressões que estão fora desses parênteses.

Suponha, por exemplo, que tenhamos declarado as variáveis `x`, `y`, `a`, `soma` e `parcela` do tipo inteiro:

```
int x, y, a, soma, parcela;
```

Suponha ainda que os seguintes comandos de atribuição tenham sido executados sobre essas variáveis:

```
x = 1;
y = 2;
a = 5;
soma = 100;
parcela = 134;
```

As sentenças a seguir são exemplos de expressões aritméticas:

```
2 * x * x + 5 * -x - +4
soma + parcela % 3
4 * 1002 - 4412 % 11 * -2 + a
(((204 / (3 + x)) * y) - ((y % x) + soma))
```

De acordo com os valores das inicializações das variáveis envolvidas e o resultado da avaliação das próprias expressões, a avaliação de cada uma das linhas acima tem valores -7, 102, 4037 e 202, respectivamente.

A tabela abaixo, contendo as prioridades dos operadores aritméticos unários e binários, já foi exibida em parte na aula 7.

| Operadores | Tipo | Descrição | Precedência |
|---|---|---|---|
| + - | unários | constante positiva, troca de sinal | 1 (máxima) |
| * / % | binários | multiplicação, divisão e resto da divisão | 2 |
| + - | binários | adição e subtração | 3 (mínima) |

## 11.2 Expressões relacionais

Uma **expressão relacional**, ou simplesmente uma **relação**, é uma comparação entre dois valores do mesmo tipo básico. Esses valores são representados na relação através de constantes, variáveis ou expressões aritméticas. Note que, até o momento, conhecemos apenas o tipo básico `int`, mas essa definição é bastante abrangente e engloba todos os tipos básicos que ainda veremos.

Os operadores relacionais da linguagem C, que indicam a comparação a ser realizada entre os termos da relação, são os seguintes:

| Operador | Descrição |
|---|---|
| == | igual a |
| != | diferente de |
| < | menor que |
| > | maior que |
| <= | menor que ou igual a |
| >= | maior que ou igual a |

O resultado da avaliação de uma expressão relacional é sempre um valor lógico, isto é, verdadeiro ou falso. Assim, supondo que temos declarado três variáveis `a`, `b` e `c` do tipo inteiro como a seguir,

```
int a, b, c;
```

e as seguintes atribuições realizadas

```
a = 2;
b = 3;
c = 4;
```

então as expressões relacionais

```
a == 2
a > b + c
b + c <= 5 - a
b != 3
```

têm valores verdadeiro, falso, falso e falso, respectivamente.

Uma observação importante neste momento é que o tipo lógico ou booleano não existe na linguagem C. Isso significa que literalmente não existem constantes lógicas e variáveis lógicas. O resultado da avaliação de uma expressão relacional ou lógica é, na verdade, um valor numérico do tipo inteiro. Se este é um valor diferente de zero, então a expressão é considerada ter valor verdadeiro. Caso contrário, isto é, se este valor é igual a zero, então a expressão tem valor falso. Dessa forma, representaremos em geral o valor 1 (um) para a constante lógica com valor verdadeiro e 0 (zero) para a constante lógica com valor falso. Lembrando apenas que, na verdade, qualquer valor diferente de 0 (zero) é avaliado como verdadeiro em uma expressão lógica.

## 11.3 Expressões lógicas

Uma **proposição** é qualquer sentença que pode ser valorada com o valor verdadeiro ou falso. Traduzindo essa definição para a linguagem C, uma proposição é qualquer sentença que pode ser valorada com um valor inteiro. Com o que aprendemos até agora, uma proposição é uma relação, uma variável do tipo inteiro ou uma constante do tipo inteiro. Uma **expressão condicional** ou **lógica**, ou ainda **booleana**[1], é formada por uma ou mais proposições. Neste

[1] Devido a George Lawlor Boole (1815 – 1864), nascido na Inglaterra, matemático e filósofo.

último caso, relacionamos as proposições através de **operadores lógicos**. Os operadores lógicos da linguagem C utilizados como conectivos nas expressões lógicas são apresentados a seguir.

| Operador | Descrição |
|---|---|
| && | conjunção |
| \|\| | disjunção |
| ! | negação |

Duas proposições `p` e `q` podem ser combinadas pelo conectivo `&&` para formar uma única proposição denominada **conjunção** das proposições originais: `p && q` e lemos "`p` e `q`". O resultado da avaliação da conjunção de duas proposições é verdadeiro se e somente se ambas as proposições têm valor verdadeiro, como mostra a tabela a seguir.

| p | q | p && q |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Duas proposições `p` e `q` podem ser combinadas pelo conectivo `||` para formar uma única proposição denominada **disjunção** das proposições originais: `p || q` e lemos "`p` ou `q`", com sentido de e/ou. O resultado da avaliação da disjunção de duas proposições é verdadeiro se pelo menos uma das proposições tem valor verdadeiro, como mostra a tabela a seguir.

| p | q | p \|\| q |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

Dada uma proposição `p`, uma outra proposição, chamada **negação** de `p`, pode ser obtida através da inserção do símbolo `!` antes da proposição: `!p` e lemos "não `p`". Se a proposição `p` tem valor verdadeiro, então `!p` tem valor falso e se `p` tem valor falso, então a proposição `!p` tem valor verdadeiro, como mostra a tabela a seguir.

| p | !p |
|---|---|
| 1 | 0 |
| 0 | 1 |

Exemplos de expressões lógicas são mostrados a seguir. Inicialmente, suponha que declaramos as variáveis do tipo inteiro a seguir:

```
int a, b, c, x;
```

e as seguintes atribuições são realizadas

```
a = 2;
b = 3;
c = 4;
x = 1;
```

então as expressões lógicas a seguir

```
5                       a == 2 && a < b + c
a                       b + c >= 5 - a || b != 3
!x                      a + b > c || 2 * x == b && 4 < x
```

têm valores verdadeiro, verdadeiro, falso na primeira coluna e verdadeiro, verdadeiro e falso na segunda coluna.

Observe que avaliamos as expressões lógicas em uma ordem: primeiro avaliamos as expressões aritméticas, depois as expressões relacionais e, por fim, as expressões lógicas em si. Por exemplo,

```
a == 2 && a + x > b + c
2 == 2 && 3 > 3 + 4
2 == 2 && 3 > 7
     1 && 0
       1
```

A tabela abaixo ilustra a prioridade de todos os operadores aritméticos, relacionais e lógicos, unários e binários, vistos até aqui.

| Operador | Tipo | Precedência |
|---|---|---|
| + - | unários | 1 (máxima) |
| * / % | binários | 2 |
| + - | binários | 3 |
| == != >= <= > < | binários | 4 |
| ! | unário | 5 |
| && | binário | 6 |
| \|\| | binário | 7 (mínima) |

Observe que o operador de negação `!` é um operador unário, ou seja, necessita de apenas um operando como argumento da operação. Os outros operadores são todos binários.

Lembre-se também que para modificar a precedência de alguma expressão é necessário o uso de parênteses, como já vimos nas expressões aritméticas.

Dessa forma, se considerarmos as mesmas variáveis e atribuições acima, a expressão lógica abaixo seria avaliada da seguinte forma:

```
x + c >= a + b || 2 * x < b && a > b + x
1 + 4 >= 2 + 3 || 2 * 1 < 3 && 2 > 3 + 1
    5 >= 5     ||   2 < 3   && 2 > 4
       1       ||     1     &&   0
       1       ||           0
               1
```

Observe que, como o operador lógico de conjunção `&&` tem prioridade sobre o operador lógico de disjunção `||`, o resultado da expressão acima é verdadeiro. Se tivéssemos realizado a disjunção primeiramente, como poderíamos intuitivamente supor devido ao posicionamento mais à esquerda do operador de disjunção, o resultado da expressão seria diferente.

## Exercícios

Alguns exercícios desta aula ensinam um importante truque de programação que é, na verdade, o uso de uma variável que simula um tipo lógico e que indica que algo ocorreu durante a execução do programa. Essa variável é chamada de **indicadora de passagem**.

11.1 Dado $p$ inteiro, verificar se $p$ é primo.

11.2 Dado um número inteiro positivo $n$ e uma seqüência de $n$ números inteiros, verificar se a seqüência está em ordem crescente.

11.3 Dados um número inteiro $n > 0$ e um dígito $d$, com $0 \leqslant d \leqslant 9$, determinar quantas vezes o dígito $d$ ocorre no número $n$.

11.4 Dado um número inteiro positivo $n$, verificar se este número contém dois dígitos consecutivos iguais.

11.5 Dado um número inteiro positivo $n$, verificar se o primeiro e o último dígito deste número são iguais.

Programa 11.1: Uma solução para o exercício 11.1.

```
#include <stdio.h>
int main(void)
{
  int p, divisor;
  printf("Informe um número: ");
  scanf("%d", &p);
  divisor = 2;
  while (divisor <= p/2) {
    if (p % divisor == 0)
      divisor = p;
    else
      divisor = divisor + 1;
  }
  if (divisor == p/2 + 1)
    printf("%d é primo\n", p);
  else
    printf("%d não é primo\n", p);
  return 0;
}
```

Programa 11.2: Solução para o exercício 11.1 usando uma variável indicadora de passagem.

```
#include <stdio.h>
int main(void)
{
  int p, divisor, primo;
  printf("Informe um número: ");
  scanf("%d", &p);
  divisor = 2;
  primo = 1;
  while (divisor <= p/2 && primo == 1) {
    if (p % divisor == 0)
      primo = 0;
    else
      divisor = divisor + 1;
  }
  if (primo == 1)
    printf("%d é primo\n", p);
  else
    printf("%d não é primo\n", p);
  return 0;
}
```

Programa 11.3: Terceira solução para o exercício 11.1.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int p, divisor, primo;
5    printf("Informe um número: ");
6    scanf("%d", &p);
7    divisor = 2;
8    primo = 1;
9    while (divisor <= p/2 && primo) {
10     if (! (p % divisor))
11       primo = 0;
12     else
13       divisor = divisor + 1;
14   }
15   if (primo)
16     printf("%d é primo\n", p);
17   else
18     printf("%d não é primo\n", p);
19   return 0;
20 }
```

AULA 12

# ESTRUTURA DE REPETIÇÃO `for`

Relembrando, as estruturas de repetição são estruturas de programação fundamentais tanto para construção de algoritmos como de programas. Todas as linguagens de programação de alto nível apresentam uma ou mais estruturas de repetição. De fato, são essas as estruturas básicas que permitem que um programador resolva centenas e centenas de problemas. Nesta aula, além de uma formalização sobre as estruturas de programação, apresentaremos a estrutura de repetição `for` da linguagem C.

## 12.1 Estruturas de programação

Como já mencionamos, as estruturas básicas de programação são a estrutura seqüencial, as estruturas condicionais e as estruturas de repetição. Na **estrutura seqüencial**, a mais básica das estruturas de programação, os comandos seguem-se uns aos outros, linha após linha, cada uma delas terminada por um ponto e vírgula ( `;` ). Esses comandos devem fazer parte de um **bloco de comandos**, envolvidos pelos símbolos `{` e `}`.

Vejamos um exemplo da estrutura seqüencial de programação no programa 12.1, uma resposta ao exercício 5.2.

Programa 12.1: Um exemplo de um pequeno programa contendo apenas a estrutura seqüencial de programação.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int x, quadrado, cubo;
5    printf("Informe um número: ");
6    scanf("%d", &x);
7    quadrado = x * x;
8    cubo = quadrado * x;
9    printf("O quadrado de %d é %d\n", x, quadrado);
10   printf("O cubo de %d é %d\n", x, cubo);
11   return 0;
12 }
```

Observe que o programa 12.1 contém uma única estrutura de programação, onde as linhas são dispostas uma após outra e finalizadas com um ponto e vírgula ( `;` ). Ou seja, após a linha que determina o início do programa principal, a única estrutura de programação apresentada neste programa é a estrutura seqüencial.

Também já vimos as estruturas condicionais, simples e compostas, de programação. Uma **estrutura condicional** realiza um desvio no fluxo de execução de um programa, condicionado ao valor resultante de uma expressão lógica. Vejamos um exemplo do uso de uma estrutura condicional no programa 12.2.

Programa 12.2: Um exemplo de um programa contendo uma estrutura condicional composta. Note que a estrutura condicional está inserida na estrutura seqüencial da função `main`. E, além disso, no interior da estrutura condicional há estruturas seqüenciais.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int x, y, z;
5    printf("Informe um número: ");
6    scanf("%d", &x);
7    printf("Informe outro número: ");
8    scanf("%d", &y);
9    if (x > y) {
10     z = x * x;
11     printf("O quadrado de %d é %d\n", x, z);
12   }
13   else {
14     z = y * y * y;
15     printf("O cubo de %d é %d\n", y, z);
16   }
17   return 0;
18 }
```

Após o início do programa 12.2, seguem-se cinco linhas, na estrutura seqüencial de programação, contendo primeiro a declaração das variáveis e depois funções de entrada e saída. Depois disso, uma estrutura condicional composta é apresentada e o fluxo de execução do programa fica sujeito à expressão lógica `u > v`. Caso o resultado da avaliação dessa expressão seja verdadeiro, o bloco de comandos que vem logo a seguir será executado. Caso contrário, isto é, se o resultado for falso, o bloco de comandos após a palavra reservada `else` será executado.

Observe ainda que as estruturas de programação podem estar encaixadas umas dentro das outras, como pode ser observado no programa 12.2: o bloco de comandos após o comando `if`, contendo duas linhas, é apresentado na estrutura seqüencial, assim como o bloco de comandos após o comando `else`, que também contém duas linhas na estrutura seqüencial de programação. Após essa estrutura condicional, o programa é finalizado com o último comando ( `return 0;` ).

E, finalmente, como vimos nas aulas anteriores mais recentes, as **estruturas de repetição** fazem com que um bloco de comandos seja executado enquanto uma condição é satisfeita. Quando essa condição deixa de ser satisfeita, o fluxo de execução salta para a próxima linha após este bloco. Este é o caso da estrutura de repetição `while` que vimos nas últimas aulas. Nesta aula, aprenderemos a estrutura de repetição `for`, muito semelhante à estrutura de repetição `while`.

## 12.2 Estrutura de repetição `for`

Retomando o programa 8.2, isto é, o primeiro exemplo com uma estrutura de repetição que vimos até agora, vamos refazer o programa que mostra os primeiros cem números inteiros positivos na saída padrão, mas agora usando a estrutura de repetição `for`. Vejamos o programa 12.3 a seguir.

Programa 12.3: Programa que mostra os 100 primeiros inteiros positivos, usando a estrutura de repetição `for`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int numero;
5    for (numero = 1; numero <= 100; numero = numero + 1)
6      printf("%d\n", numero);
7    return 0;
8  }
```

A novidade neste programa é a estrutura de repetição `for`. Um esqueleto da estrutura de repetição `for` é dado a seguir:

```
for (inicialização; condição; passo) {
  seqüência de comandos
}
```

A estrutura de repetição `for` dispõe a inicialização, a condição e o passo todos na mesma linha, logo após o comando `for`, envolvidos por parênteses. O funcionamento deste comando é dado da seguinte forma. Sempre que um comando `for` é encontrado, a *inicialização* é executada. Em seguida, a *condição* é verificada. Se o resultado da avaliação desta condição é verdadeiro, então o fluxo de execução é desviado para a *seqüência de comandos* no bloco de comandos interno à estrutura `for`. Finalizada a seqüência de comandos, o *passo* é realizado e o fluxo de execução é desviado para a linha do comando `for`, onde novamente a *condição* será testada e o processo todo repetido, até que esta *condição* tenha como resultado de sua

avaliação o valor falso e o fluxo da execução é então desviado para o próximo comando após o bloco de comandos do comando `for`.

As semelhanças entre a estrutura de repetição `for` e `while` são destacadas a seguir.

```
inicialização
while (condição) {
  seqüência de comandos
  passo
}
```

```
for (inicialização; condição; passo) {
   seqüência de comandos
}
```

Vamos fazer um outro exemplo de programa, que também já fizemos antes no programa 8.3, que soma os 100 primeiros números inteiros positivos e utiliza a estrutura de repetição `for`. Vejamos o programa 12.4.

Programa 12.4: Programa que soma os 100 primeiros inteiros positivos usando a estrutura de repetição `for`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int numero, soma;
5     soma = 0;
6     for (numero = 1; numero <= 100; numero = numero + 1)
7       soma = soma + numero;
8     printf("A soma dos 100 primeiros números inteiros é %d\n", soma);
9     return 0;
10 }
```

Aqui talvez seja um bom momento para entrarmos em contato com outros dois operadores aritméticos da linguagem C. Os operadores unários de **incremento** `++` e de **decremento** `--` têm como função adicionar ou subtrair uma unidade ao valor armazenado em seu operando, respectivamente. Mas infelizmente, a compreensão das formas de uso desses operadores pode ser facilmente prejudicada. Isso porque, além de modificar os valores de seus operandos, `++` e `--` podem ser usados como operadores **prefixos** e também como operadores **posfixos**. Por exemplo, o trecho de código abaixo

```
int cont;
cont = 1;
printf("cont vale %d\n", ++cont);
printf("cont vale %d\n", cont);
```

imprime `cont vale 2` e `cont vale 2` na saída. Por outro lado, o trecho de código abaixo

```
int cont;
cont = 1;
printf("cont vale %d\n", cont++);
printf("cont vale %d\n", cont);
```

imprime `cont vale 1` e `cont vale 2` na saída.

Podemos pensar que a expressão `++cont` significa "incremente `cont` imediatamente" enquanto que a expressão `cont++` significa "use agora o valor de `cont` e depois incremente-o". O "depois" nesse caso depende de algumas questões, mas seguramente a variável `cont` será incrementada antes da próxima sentença ser executada.

O operador `--` tem as mesmas propriedades do operador `++`.

É importante observar que os operadores de incremento e decremento, se usados como operadores posfixos, têm maior prioridade que os operadores unários `+` e `-`. Se são usados como operadores prefixos, então têm a mesma prioridade dos operadores `+` e `-`.

O programa 12.4, que soma os 100 primeiros números inteiros positivos, pode ser reescrito como no programa 12.5.

Programa 12.5: Programa que realiza a mesma tarefa do programa 12.4, mas usando o operador aritmético unário de incremento.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int numero, soma;
5     soma = 0;
6     for (numero = 1; numero <= 100; ++numero)
7       soma = soma + numero;
8     printf("O resultado da soma dos 100 primeiros inteiros é %d\n", soma);
9     return 0;
10 }
```

## Exercícios

12.1 Dado um número natural na base binária, transformá-lo para a base decimal.

Exemplo:

Dado 10010 a saída será 18, pois $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 18$.

Programa 12.6: Uma solução do exercício 12.1.

```
#include <stdio.h>
int main(void)
{
  int bin, dec, pot2;
  printf("Informe um número na base binária: ");
  scanf("%d", &bin);
  pot2 = 1;
  dec = 0;
  for ( ; bin != 0; bin = bin/10) {
    dec = dec + bin % 10 * pot2;
    pot2 = pot2 * 2;
  }
  printf("%d\n", dec);
  return 0;
}
```

12.2 Dado um número natural na base decimal, transformá-lo para a base binária.

Exemplo:

Dado 18 a saída deverá ser 10010.

12.3 Dado um número inteiro positivo $n$ que não contém um dígito $0$, imprimi-lo na ordem inversa de seus dígitos.

Exemplo:

Dado 26578 a saída deverá ser 87562.

12.4 Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos.

Exemplos:

1297: 12 e 97.
5314: 53 e 14.

Escreva um programa que imprima todos os números de quatro algarismos cuja raiz quadrada seja a soma das dezenas formadas pela divisão acima.

Exemplo:

$\sqrt{9801} = 99 = 98 + 01$.
Portanto, 9801 é um dos números a ser impresso.

AULA 13

# ESTRUTURA DE REPETIÇÃO `do-while`

Nesta aula veremos uma terceira, e última, estrutura de repetição da linguagem C, a estrutura `do-while`. Diferentemente das estruturas de repetição `while` e `for` que estabelecem seus critérios de repetição logo no início, a estrutura de repetição `do-while` tem um critério de repetição ao final do bloco de comandos associado a elas. Isso significa, entre outras coisas, que o bloco de comandos associado à estrutura de repetição `do-while` é executado pelo menos uma vez, diferentemente das outras duas estruturas de repetição, que podem ter seus blocos de comandos nunca executados.

## 13.1 Definição e exemplo de uso

Suponha, por exemplo, que seja necessário resolver um problema simples, muito semelhante aos problemas iniciais que resolvemos usando uma estrutura de repetição. O enunciado do problema é o seguinte: dada uma seqüência de números inteiros terminada com 0, calcular a soma desses números.

Com o que aprendemos sobre programação até o momento, este problema parece bem simples de ser resolvido. Vejamos agora uma solução um pouco diferente, dada no programa 13.1, contendo a estrutura de repetição `do-while` que ainda não conhecíamos.

Programa 13.1: Um programa que usa a estrutura de repetição `do-while`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int numero, soma;
5    soma = 0;
6    do {
7      printf("Informe um número: ");
8      scanf("%d", &numero);
9      soma = soma + numero;
10   } while (numero != 0);
11   printf("A soma dos números informados é %d\n", soma);
12   return 0;
13 }
```

O programa 13.1 é muito semelhante aos demais programas que vimos construindo até o momento. A única diferença significativa é o uso da estrutura de repetição `do-while`. Quando o programa encontra essa estrutura de programação pela primeira vez, com uma linha contendo a palavra reservada `do` seguida de um símbolo de início de bloco de comandos `{`, o fluxo de programação segue para o primeiro comando deste bloco. Os comandos desse bloco são então executados um a um, até que o símbolo de fim do bloco de comandos `}` seja encontrado. Logo em seguida, encontramos a palavra reservada `while` e, depois, uma expressão lógica envolvida por parênteses. Se o resultado da avaliação dessa expressão lógica for verdadeiro, então o fluxo de execução do programa é desviado para o primeiro comando do bloco de comandos desta estrutura de repetição e todos os comandos são novamente executados. Caso contrário, o fluxo de execução é desviado para o próximo comando após a linha contendo o final desta estrutura de repetição, isto é, a linha que contém a palavra reservada `while`.

Um esqueleto da estrutura de repetição `do-while` é mostrado a seguir, onde a *seqüência de comandos* é composta por quaisquer comandos da linguagem C ou qualquer outra estrutura de programação:

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

Observe que, assim como em qualquer outra estrutura de programação, o bloco de comandos da estrutura de repetição `do-while` pode conter um único comando e, por isso, os símbolos de delimitação de um bloco `{` e `}` são dispensáveis neste caso.

Note ainda que o teste de continuidade desta estrutura de repetição é realizado sempre no final, após todos os comandos de seu bloco interno de comandos terem sido executados. Isso significa que esse bloco de comandos será executado ao menos uma vez. Neste sentido, esta estrutura de repetição `do-while` difere bastante das estruturas de repetição `while` e `for` que vimos anteriormente, já que nesses dois últimos casos o resultado da avaliação da expressão condicional associada a essas estruturas pode fazer com que seus blocos de comandos respectivos não sejam executados nem mesmo uma única vez.

## Exercícios

13.1 Dado um número inteiro não-negativo $n$, escreva um programa que determine quantos dígitos o número $n$ possui.

13.2 Solucione o exercício 12.3 usando a estrutura de repetição `do-while`.

13.3 Dizemos que um número natural $n$ com pelo menos 2 algarismos é **palíndromo** se

> o primeiro algarismo de $n$ é igual ao seu último algarismo;
> o segundo algarismo de $n$ é igual ao se penúltimo algarismo;
> e assim sucessivamente.

Exemplos:

Programa 13.2: Uma solução para o exercício 13.1.

```
#include <stdio.h>
int main(void)
{
  int n, digitos;
  printf("Informe n: ");
  scanf("%d", &n);
  digitos = 0;
  do {
    n = n / 10;
    digitos++;
  } while (n > 0);
  printf("O número tem %d dígitos\n", digitos);
  return 0;
}
```

567765 é palíndromo;
32423 é palíndromo;
567675 não é palíndromo.

Dado um número natural $n$, $n \geqslant 10$, verificar se $n$ é palíndromo.

13.4 Dados um número inteiro $n > 0$ e uma seqüência de $n$ números inteiros, determinar quantos segmentos de números iguais consecutivos compõem essa seqüência.

Exemplo:

Para $n = 9$, a seqüência $\overbrace{5}, \overbrace{-2,-2}, \overbrace{4,4,4,4}, \overbrace{1,1}$ é formada por 4 segmentos de números iguais.

13.5 Dados um número inteiro $n > 0$ e uma seqüência de $n$ números inteiros, determinar o comprimento de um segmento crescente de comprimento máximo.

Exemplos:

Na seqüência $5, 10, 6, \overbrace{2, 4, 7, 9}, 8, -3$ o comprimento do segmento crescente máximo é 4.
Na seqüência $10, 8, 7, 5, 2$ o comprimento do segmento crescente máximo é 1.

AULA 14

# NÚMEROS COM PONTO FLUTUANTE

Dadas as dificuldades inerentes da representação de números reais nos computadores, as linguagens de programação de alto nível procuram superá-las abstraindo essa representação através do uso de números com ponto flutuante. Na linguagem C, números com ponto flutuante podem ser manipulados como constantes ou variáveis do tipo ponto flutuante. O armazenamento destes números em memória se dá de forma distinta do armazenamento de números inteiros e necessita de mais espaço, como poderíamos supor.

Nesta aula veremos como manipular números de ponto flutuante, formalizando as definições de constantes e variáveis envolvidas, além de aprender as regras de uso desses elementos em expressões aritméticas.

## 14.1 Constantes e variáveis do tipo ponto flutuante

Números inteiros não são suficientes ou adequados para solucionar todos os problemas. Muitas vezes são necessárias variáveis que possam armazenar números com dígitos após a vírgula, números muito grandes ou números muito pequenos. A vírgula, especialmente em países de língua inglesa, é substituída pelo ponto decimal. A linguagem C permite que variáveis sejam declaradas de forma a poderem armazenar números de ponto flutuante, com os tipos básicos `float` e `double`.

Uma **constante do tipo ponto flutuante** se distingue de uma constante do tipo inteiro pelo uso do ponto decimal. Dessa forma, `3.0` é uma constante do tipo ponto flutuante, assim como `1.125` e `-765.567`. Podemos omitir os dígitos antes do ponto decimal ou depois do ponto decimal, mas obviamente não ambos. Assim, `3.` e `-.1115` são constantes do tipo ponto flutuante válidas.

Na linguagem C, uma **variável do tipo ponto flutuante** pode armazenar valores do tipo ponto flutuante e deve ser declarada com a palavra reservada `float` ou `double`. Por exemplo, a declaração

```
float x, y;
double arco;
```

realiza a declaração das variáveis `x` e `y` como variáveis do tipo ponto flutuante.

A distinção entre os tipos de ponto flutuante `float` e `double` se dá pela quantidade de precisão necessária. Quando a precisão não é crítica, o tipo `float` é adequado. O tipo `double` fornece precisão maior.

A tabela a seguir mostra as características dos tipos de ponto flutuante quando implementados de acordo com o padrão IEEE[1]. Em computadores que não seguem o padrão IEEE, esta tabela pode não ser válida.

| Tipo | Menor valor (positivo) | Maior valor | Precisão |
|---|---|---|---|
| `float` | $1.17549 \times 10^{-38}$ | $3.40282 \times 10^{38}$ | 6 dígitos |
| `double` | $2.22507 \times 10^{-308}$ | $1.79769 \times 10^{308}$ | 15 dígitos |

O padrão IEEE 754 estabelece dois formatos primários para números de ponto flutuante: o formato de precisão simples, com 32 bits, e o formato de precisão dupla, com 64 bits. Os números de ponto flutuante são armazenados no formato de notação científica, composto por três partes: um sinal, um expoente e uma fração. O número de bits reservado para representar o expoente determina quão grande o número pode ser, enquanto que o número de bits da fração determina sua precisão.

Um exemplo de um programa que usa constantes e variáveis do tipo `float` é apresentado no programa 14.1. Neste programa, uma seqüência de cem números do tipo ponto flutuante é informada pelo usuário. Em seguida, a média aritmética desses números é calculada e, por fim, mostrada na saída padrão.

Programa 14.1: Calcula a média de 100 números do tipo ponto flutuante.

```
1   #include <stdio.h>
2   int main(void)
3   {
4      int i;
5      float numero, soma, media;
6      soma = 0.0;
7      for (i = 1; i <= 100; i++) {
8        printf("Informe um número: ");
9        scanf("%f", &numero);
10       soma = soma + numero;
11     }
12     media = soma / 100;
13     printf("A média dos 100 números é %f\n", media);
14     return 0;
15  }
```

Observe que a cadeia de caracteres de formatação contém um especificador de conversão para um número de ponto flutuante `%f`, diferentemente do especificador de conversão para números inteiros (`%d`) que vimos anteriormente.

[1] Instituto de Engenheiros Elétricos e Eletrônicos, do inglês *Institute of Electrical and Eletronics Engineers* (IEEE).

## 14.2 Expressões aritméticas

Na linguagem C existem regras de conversão implícita de valores do tipo inteiro e do tipo ponto flutuante. A regra principal diz que a avaliação de uma expressão aritmética, isto é, seu resultado, será do tipo ponto flutuante caso algum de seus operandos – resultante da avaliação de uma expressão aritmética – seja também do tipo ponto flutuante. Caso contrário, isto é, se todos os operandos são valores do tipo inteiro, o resultado da expressão aritmética será um valor do tipo inteiro. Vejamos um exemplo no programa 14.2.

Programa 14.2: Programa que apresenta valores do tipo inteiro e do tipo ponto flutuante.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int i1, i2;
5     float f1, f2;
6     i1 = 190;
7     f1 = 100.5;
8     i2 = i1 / 100;
9     printf("i2 = %d\n", i2);
10    f2 = i1 / 100;
11    printf("f2 = %f\n", f2);
12    f2 = i1 / 100.0;
13    printf("f2 = %f\n", f2);
14    f2 = f1 / 100;
15    printf("f2 = %f\n", f2);
16    return 0;
17 }
```

O resultado da execução desse programa é apresentado a seguir.

```
i2 = 1
f2 = 1.000000
f2 = 1.900000
f2 = 1.005000
```

O primeiro valor é um número do tipo inteiro que representa o quociente da divisão de dois números do tipo inteiro `190` e `100`. Já tínhamos trabalhado com expressões aritméticas semelhantes em aulas anteriores. No segundo valor temos de olhar para a expressão aritmética à direita do comando de atribuição. Esta expressão é uma expressão aritmética que só contém operandos do tipo inteiro. Por isso, o resultado é o número do tipo inteiro `1`, que é atribuído à variável `f2`. O comando `printf` mostra o conteúdo da variável `f2` como um número do tipo ponto flutuante e assim a saída é `f2 = 1.000000`. Em seguida, nas próximas duas impressões, as expressões aritméticas correspondentes contêm operandos do tipo ponto

flutuante: a constante do tipo ponto flutuante `100.0` na primeira e a variável do tipo ponto flutuante `f1` na segunda. Por isso, as expressões aritméticas têm como resultados números do tipo ponto flutuante e a saída mostra os resultados esperados.

Suponha, no entanto, que um programa semelhante, o programa 14.3, tenha sido digitado, salvo, compilado e executado.

Programa 14.3: Outro programa que apresenta valores do tipo inteiro e de ponto flutuante.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int i1, i2;
5     float f1;
6     i1 = 3;
7     i2 = 2;
8     f1 = i1 / i2;
9     printf("f1 = %f\n", f1);
10    return 0;
11 }
```

O resultado da execução deste programa é:

```
f1 = 1.000000
```

Mas e se quiséssemos que a divisão `i1/ i2` tenha como resultado um valor do tipo ponto flutuante? Neste caso, devemos usar um operador unário chamado **operador conversor de tipo**, do inglês *type cast operator*, sobre alguma das variáveis da expressão. No caso acima, poderíamos usar qualquer uma das atribuições abaixo:

```
f1 = (float) i1 / (float) i2;
f1 = (float) i1 / i2;
f1 = i1 / (float) i2;
```

e o resultado e a saída do programa seria então:

```
f1 = 1.500000
```

O operador `(float)` é assim chamado de **operador conversor do tipo ponto flutuante**.

Um outro operador unário, que faz o inverso do operador conversor do tipo ponto flutuante, é o **operador conversor do tipo inteiro**, denotado por `(int)`. Esse operador conversor do tipo inteiro converte o valor de uma expressão aritmética para um valor do tipo inteiro.

Vejamos o programa 14.4.

Programa 14.4: Mais um programa que apresenta valores do tipo inteiro e de ponto flutuante.

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int i1, i2, i3, i4;
5     float f1, f2;
6     f1 = 10.8;
7     f2 = 1.5;
8     i1 = f1 / f2;
9     printf("i1 = %d\n", i1);
10    i2 = (int) f1 / f2;
11    printf("i2 = %d\n", i2);
12    i3 = f1 / (int) f2;
13    printf("i3 = %d\n", i3);
14    i4 = (int) f1 / (int) f2;
15    printf("i4 = %d\n", i4);
16    return 0;
17 }
```

A saída do programa 14.4 é

```
i1 = 7
i2 = 6
i3 = 10
i4 = 10
```

As primeiras duas atribuições fazem com que as variáveis `f1` e `f2` recebam as constantes de ponto flutuante `10.8` e `1.5`. Em seguida, a atribuição `i1 = f1 / f2;` faz com que o resultado da expressão aritmética à direita do comando de atribuição seja atribuído à variável `i1` do tipo inteiro. A expressão aritmética devolve um valor de ponto flutuante: `10.8 / 1.5` = `7.2`. Porém, como do lado esquerdo da atribuição temos uma variável do tipo inteiro, a parte fracionária deste resultado é descartada e o valor `7` é então armazenado na variável `i1` e mostrado com o comando `printf`. Na próxima linha, a expressão aritmética do lado direito da atribuição é `(int) f1 / f2` que é avaliada como `(int) 10.8 / 1.5` = `10 / 1.5` = `6.666666` e, como do lado direito do comando de atribuição temos uma variável do tipo inteiro `i2`, o valor `6` é armazenado nesta variável e mostrado na saída através do comando `printf` na linha seguinte. Na próxima linha, a expressão aritmética `f1 / (int) f2` é avaliada como `10.8 / (int) 1.5` = `10.8 / 1` = `10.8` e, de novo, como do lado direito do

comando de atribuição temos uma variável do tipo inteiro `i3`, o valor armazenado nesta variável é `10`, que também é mostrado na saída através do comando `printf` logo a seguir. E, por fim, a expressão aritmética seguinte é `(int) f1 / (int) f2` e sua avaliação é dada por `(int) 10.8 / (int) 1.5` = `10 / 1` = `10` e portanto este valor é atribuído à variável `i4` e apresentado na saída pelo comando `printf`.

Um compilador da linguagem C considera qualquer constante com ponto flutuante como sendo uma constante do tipo `double`, a menos que o programador explicitamente diga o contrário, colocando o símbolo `f` no final da constante. Por exemplo, a constante `3.1415f` é uma constante com ponto flutuante do tipo `float`, ao contrário da constante `55.726`, que é do tipo `double`.

Uma constante de ponto flutuante também pode ser expressa em notação científica. O valor `1.342e-3` é um valor de ponto flutuante e representa o valor $1{,}342 \times 10^{-3}$ ou $0{,}001324$. O valor antes do símbolo `e` é chamado **mantissa** e o valor após esse símbolo é chamado **expoente** do número de ponto flutuante.

Para mostrar um número com ponto flutuante do tipo `double` na saída padrão podemos usar a mesmo caracter de conversão de tipo `%f` que usamos para mostrar um número com ponto flutuante do tipo `float`. Para realizar a leitura de um número de ponto flutuante que será armazenado em uma variável do tipo `double` usamos os caracteres `%lf` como caracteres de conversão de tipo.

Um outro exemplo do uso de constantes e variáveis do tipo ponto flutuante (`float` e `double`) é mostrado no programa 14.5, que computa a área de um círculo cujo valor do raio é informado pelo usuário em radianos.

Programa 14.5: Programa para cálculo da área do círculo.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    float pi;
5    double raio, area;
6    pi = 3.141592f;
7    printf("Digite o valor do raio: ");
8    scanf("%lf", &raio);
9    area = (double)pi * raio * raio;
10   printf("A área do círculo é %f\n", area);
11   return 0;
12 }
```

## Exercícios

14.1 Uma pessoa aplicou um capital de $x$ reais a juros mensais de $y\%$ durante 1 ano. Determinar o montante de cada mês durante este período.

Programa 14.6: Uma solução para o exercício 14.1.

```
#include <stdio.h>
int main(void)
{
  int mes;
  float x, y;
  printf("Informe o capital inicial: ");
  scanf("%d", &x);
  printf("Informe a taxa de juros: ");
  scanf("%d", &y);
  for (mes = 1; mes <= 12; mes++) {
    x = x * (1 + y / 100);
    printf("Mês: %d  Montante: %f\n", mes, x);
  }
  return 0;
}
```

14.2 Os pontos $(x, y)$ que pertencem à figura $H$ (veja a figura 14.1) são tais que $x \geqslant 0, y \geqslant 0$ e $x^2 + y^2 \leqslant 1$. Dados $n$ pontos reais $(x, y)$, verifique se cada ponto pertence ou não a $H$.

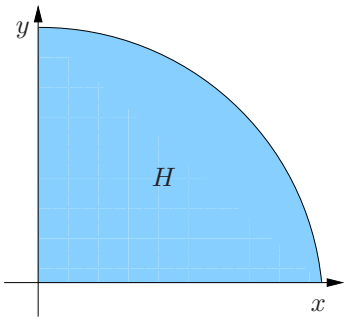



Figura 14.1: Área $H$ de um quarto de um círculo.

14.3 Considere o conjunto $H = H_1 \cup H_2$ de pontos reais, onde

$$H_1 = \{(x, y) | x \leqslant 0, y \leqslant 0, y + x^2 + 2x - 3 \leqslant 0\}$$
$$H_2 = \{(x, y) | x \geqslant 0, y + x^2 - 2x - 3 \leqslant 0\}$$

Dado um número inteiro $n > 0$, leia uma seqüência de $n$ pontos reais $(x, y)$ e verifique se cada ponto pertence ou não ao conjunto $H$, contando o número de pontos da seqüência que pertencem a $H$.

14.4 Dado um natural $n$, determine o número harmônico $H_n$ definido por

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

14.5 Para $n > 0$ alunos de uma determinada turma são dadas 3 notas de provas. Calcular a média aritmética das provas de cada aluno, a média da classe, o número de aprovados e o número de reprovados, onde o critério de aprovação é média $\geqslant 5.0$.

14.6 Dados números reais $a, b$ e $c$, calcular as raízes de uma equação do 2º grau da forma $ax^2 + bx + c = 0$. Imprimir a solução em uma das seguintes formas:

| a. | DUPLA | b. | REAIS DISTINTAS | c. | COMPLEXAS |
|---|---|---|---|---|---|
| | raiz | | raiz 1 | | parte real |
| | | | raiz 2 | | parte imaginária |

AULA 15

# EXERCÍCIOS

Nesta aula faremos mais exercícios usando números com ponto flutuante.

## Enunciados

15.1 Dado um inteiro positivo $n$, calcular e imprimir o valor da seguinte soma

$$\frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + \ldots + \frac{n}{1}\,.$$

Programa 15.1: Uma solução para o exercício 15.1.

```
#include <stdio.h>
int main(void)
{
  int n, numerador, denominador;
  float soma;
  printf("Informe o valor de n: ");
  scanf("%d", &n);
  numerador = 1;
  denomidador = n;
  soma = 0.0;
  for (i = 1; i <= n; i++) {
    soma = soma + (float) numerador / denominador;
    numerador++;
    denominador--;
  }
  printf("Soma = %f", soma);
  return 0;
}
```

15.2 Faça um programa que calcula a soma

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{9999} - \frac{1}{10000}$$

pelas seguintes maneiras:

(a) adição dos termos da direita para a esquerda;

(b) adição dos termos da esquerda para a direita;

(c) adição separada dos termos positivos e dos termos negativos da esquerda para a direita;

(d) adição separada dos termos positivos e dos termos negativos da direita para a esquerda;

(e) fórmula de recorrência;

(f) fórmula do termo geral.

15.3 Uma maneira de calcular o valor do número $\pi$ é utilizar a seguinte série:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Fazer um programa para calcular e imprimir o valor de $\pi$ através da série acima, com precisão de 4 casas decimais. Para obter a precisão desejada, adicionar apenas os termos cujo valor absoluto seja maior ou igual a 0,0001.

15.4 Dado um número real $x$ tal que $0 \leqslant x \leqslant 1$, calcular uma aproximação do arco tangente de $x$ em radianos através da série infinita:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$$

incluindo todos os termos da série até $|\frac{x^{2k+1}}{2k+1}| < 0.0001$, com $k \geqslant 0$.

15.5 Uma maneira de calcular o valor de $\mathrm{e}^x$ é utilizar a seguinte série:

$$\mathrm{e}^x = x^0 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Dados dois números reais $x$ e $\varepsilon$, calcular o valor de $\mathrm{e}^x$ incluindo todos os termos $T_k = \frac{x^k}{k!}$, com $k \geqslant 0$, até que $T_k < \varepsilon$. Mostre na saída o valor computado e o número total de termos usados na série.

15.6 Dados $x$ real e $n$ natural, calcular uma aproximação para $\cos x$, onde $x$ é dado em radianos, através dos $n$ primeiros termos da seguinte série:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^k \frac{x^{2k}}{(2k)!} + \ldots$$

com $k \geqslant 0$. Compare com os resultados de sua calculadora.

15.7 Dados $x$ e $\varepsilon$ reais, $\varepsilon > 0$, calcular uma aproximação para $\operatorname{sen} x$, onde $x$ é dado em radianos, através da seguinte série infinita

$$\operatorname{sen} x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \ldots$$

incluindo todos os termos $T_k = \frac{x^{2k+1}}{(2k+1)!}$, com $k \geqslant 0$, até que $T_k < \varepsilon$. Compare com os resultados de sua calculadora.

AULA 16

# EXERCÍCIOS

Nesta aula faremos exercícios com estruturas de repetição encaixadas.

## Enunciados

16.1 Dados um número inteiro $n$ e $n$ seqüências de números inteiros, cada qual terminada por 0, determinar a soma dos números pares de cada seqüência.

Programa 16.1: Uma solução para o exercício 16.1.

```
1  #include <stdio.n>
2  int main(void)
3  {
4    int n, num, soma;
5    printf("Informe o valor de n: ");
6    scanf("%d", &n);
7    for (i = 1; i <= n; i++) {
8      soma = 0;
9      do {
10       printf("(Sequência %d) Informe um número: ", i);
11       scanf("%d", &num);
12       if (!(num % 2))
13         soma = soma + num;
14     } while (num != 0);
15     printf("(Sequência %d) Soma = %d\n", i, soma);
16   }
17   return 0;
18 }
```

16.2 Dados um número inteiro $n > 0$ e uma seqüência de $n$ números inteiros positivos determinar o fatorial de cada número da seqüência.

16.3 Dados $n$ números inteiros positivos, calcular a soma dos que são primos.

16.4 Dado um número inteiro positivo $n$, determinar todos os inteiros entre 1 e $n$ que são comprimento de hipotenusa de um triângulo retângulo com catetos inteiros.

16.5 Dados dois naturais $m$ e $n$, determinar, entre todos os pares de números naturais $(x, y)$ tais que $x \leqslant m$ e $y \leqslant n$, um par para o qual o valor da expressão $xy - x^2 + y$ seja máximo e calcular também esse máximo.

16.6 Sabe-se que um número da forma $n^3$ é igual à soma de $n$ números ímpares consecutivos.

Exemplo:

$1^3 = 1$
$2^3 = 3 + 5$
$3^3 = 7 + 9 + 11$
$4^3 = 13 + 15 + 17 + 19$
$\vdots$

Dado $m$, determine os ímpares consecutivos cuja soma é igual a $n^3$ para $n$ assumindo valores de 1 a $m$.

16.7 Dado um número inteiro positivo, determine a sua decomposição em fatores primos, calculando também a multiplicidade de cada fator.

Exemplo:

Se $n = 600$ a saída deve ser
fator 2 multiplicidade 3
fator 3 multiplicidade 1
fator 5 multiplicidade 2

16.8 Dados $n$ inteiros positivos, determinar o máximo divisor comum entre eles.

AULA 17

# CARACTERES

Nesta aula vamos estudar um novo tipo básico de dados, o tipo caracter. Na linguagem C existem dois tipos de caracteres: caracter com sinal e caracter sem sinal. Esses dois tipos são apenas interpretações diferentes do conjunto de todas as seqüências de 8 bits, sendo que essa diferença é irrelevante na prática. Esta aula é baseada especialmente no livro [4] e também no livro [7].

## 17.1 Representação gráfica

Na prática, em todas as linguagens de programação de alto nível, incluindo a linguagem C, cada caracter é armazenado em um único byte na memória do computador, ou seja, em 8 bits. Conseqüentemente, na linguagem C um caracter sem sinal é um número do conjunto $\{0, 1, \ldots, 254, 255\}$ e um caracter com sinal é um número do conjunto $\{-128, \ldots, -1, 0, 1, \ldots, 127\}$. Ou seja, um caracter é uma seqüência de 8 bits, dentre as 256 seqüências de 8 bits possíveis.

A impressão de um caracter na saída padrão é a sua representação como um símbolo gráfico. Por exemplo, o símbolo gráfico do caracter `97` é `a`. Alguns caracteres têm representações gráficas especiais, como o caracter `10` que é representado por uma mudança de linha.

Os símbolos gráficos dos caracteres `0` a `127`, de 7 bits, foram codificados e padronizados pelo Código Padrão Americano para Troca de Informações (do inglês *American Standard Code for Information Interchange*). Por conta disso, essa representação gráfica dos caracteres é conhecida como tabela ASCII. Essa codificação foi desenvolvida em 1960 e tem como base o alfabeto da língua inglesa. Muitas das codificações de caracteres mais modernas herdaram como base a tabela ASCII.

As limitações do conjunto de caracteres da tabela ASCII, e também da tabela EBCDIC, logo mostraram-se aparentes e outros métodos foram desenvolvidos para estendê-los. A necessidade de incorporar múltiplos sistemas de escrita, incluindo a família dos caracteres do leste asiático, exige suporte a um número bem maior de caracteres. Por exemplo, o repertório completo do UNICODE compreende mais de cem mil caracteres. Outros repertórios comuns incluem ISO 8859-1 bastante usado nos alfabetos latinos e, por isso, também conhecido como ISO Latin1.

Dos 128 caracteres padronizados da tabela ASCII e de seus símbolos gráficos correspondentes, 33 deles são não-imprimíveis e os restantes 95 são imprimíveis. Os caracteres não-imprimíveis são caracteres de controle, criados nos primórdios da computação, quando se usavam máquinas teletipo e fitas de papel perfurado, sendo que atualmente grande parte deles estão obsoletos.

Na tabela a seguir, apresentamos alguns desses caracteres não-imprimíveis. Muitos deles não são mostrados por opção ou por terem caído em desuso.

| Bin | Dec | Significado |
|---|---|---|
| 0000 0000 | 00 | Nulo |
| 0000 0111 | 07 | Campainha |
| 0000 1001 | 09 | Tabulação horizontal |
| 0000 1010 | 10 | Mudança de linha |
| 0000 1011 | 11 | Tabulação vertical |
| 0000 1100 | 12 | Quebra de página |
| 0000 1101 | 13 | Retorno do carro/cursor |
| 0111 1111 | 127 | *Delete* |

A tabela a seguir mostra os 95 caracteres imprimíveis da tabela ASCII.

| Bin | Dec | Sim | Bin | Dec | Sim | Bin | Dec | Sim |
|---|---|---|---|---|---|---|---|---|
| 0010 0000 | 32 | | 0100 0000 | 64 | @ | 0110 0000 | 96 | ‘ |
| 0010 0001 | 33 | ! | 0100 0001 | 65 | A | 0110 0001 | 97 | a |
| 0010 0010 | 34 | " | 0100 0010 | 66 | B | 0110 0010 | 98 | b |
| 0010 0011 | 35 | # | 0100 0011 | 67 | C | 0110 0011 | 99 | c |
| 0010 0100 | 36 | $ | 0100 0100 | 68 | D | 0110 0100 | 100 | d |
| 0010 0101 | 37 | % | 0100 0101 | 69 | E | 0110 0101 | 101 | e |
| 0010 0110 | 38 | & | 0100 0110 | 70 | F | 0110 0110 | 102 | f |
| 0010 0111 | 39 | ’ | 0100 0111 | 71 | G | 0110 0111 | 103 | g |
| 0010 1000 | 40 | ( | 0100 1000 | 72 | H | 0110 1000 | 104 | h |
| 0010 1001 | 41 | ) | 0100 1001 | 73 | I | 0110 1001 | 105 | i |
| 0010 1010 | 42 | * | 0100 1010 | 74 | J | 0110 1010 | 106 | j |
| 0010 1011 | 43 | + | 0100 1011 | 75 | K | 0110 1011 | 107 | k |
| 0010 1100 | 44 | , | 0100 1100 | 76 | L | 0110 1100 | 108 | l |
| 0010 1101 | 45 | - | 0100 1101 | 77 | M | 0110 1101 | 109 | m |
| 0010 1110 | 46 | . | 0100 1110 | 78 | N | 0110 1110 | 110 | n |
| 0010 1111 | 47 | / | 0100 1111 | 79 | O | 0110 1111 | 111 | o |
| 0011 0000 | 48 | 0 | 0101 0000 | 80 | P | 0111 0000 | 112 | p |
| 0011 0001 | 49 | 1 | 0101 0001 | 81 | Q | 0111 0001 | 113 | q |
| 0011 0010 | 50 | 2 | 0101 0010 | 82 | R | 0111 0010 | 114 | r |
| 0011 0011 | 51 | 3 | 0101 0011 | 83 | S | 0111 0011 | 115 | s |
| 0011 0100 | 52 | 4 | 0101 0100 | 84 | T | 0111 0100 | 116 | t |
| 0011 0101 | 53 | 5 | 0101 0101 | 85 | U | 0111 0101 | 117 | u |
| 0011 0110 | 54 | 6 | 0101 0110 | 86 | V | 0111 0110 | 118 | v |
| 0011 0111 | 55 | 7 | 0101 0111 | 87 | W | 0111 0111 | 119 | w |
| 0011 1000 | 56 | 8 | 0101 1000 | 88 | X | 0111 1000 | 120 | x |
| 0011 1001 | 57 | 9 | 0101 1001 | 89 | Y | 0111 1001 | 121 | y |
| 0011 1010 | 58 | : | 0101 1010 | 90 | Z | 0111 1010 | 122 | z |
| 0011 1011 | 59 | ; | 0101 1011 | 91 | [ | 0111 1011 | 123 | { |
| 0011 1100 | 60 | < | 0101 1100 | 92 | \ | 0111 1100 | 124 | \| |
| 0011 1101 | 61 | = | 0101 1101 | 93 | ] | 0111 1101 | 125 | } |
| 0011 1110 | 62 | > | 0101 1110 | 94 | ^ | 0111 1110 | 126 | ~ |
| 0011 1111 | 63 | ? | 0101 1111 | 95 | _ | | | |

Os caracteres com sinal, quando dispostos em ordem crescente, apresentam-se de tal forma que as letras acentuadas precedem as não-acentuadas. Os caracteres sem sinal, ao contrário, em ordem crescente apresentam-se de forma que as letras não-acentuadas precedem as acentuadas. Essa é a única diferença entre caracteres com e sem sinal.

## 17.2 Constantes e variáveis

Uma variável do tipo caracter com sinal pode ser declarada na linguagem C com a palavra reservada `char`. Uma variável do tipo caracter sem sinal pode ser declarada com a mesma palavra reservada `char`, mas com o especificador de tipo `unsigned` a precedendo. Especificadores de tipos básicos serão estudados em detalhes na aula 18. Dessa forma,

```
char c, d, e;
unsigned char f, g, h;
```

são declarações válidas de variáveis do tipo caracter.

Uma constante do tipo caracter é um número no intervalo de $0$ a $255$ ou de $-128$ a $127$. Por exemplo, para as variáveis `c`, `d` e `e` declarada acima, podemos fazer

```
c = 122;
d = 59;
e = 51;
```

Mais comum e confortavelmente, podemos especificar uma constante do tipo caracter através da representação gráfica de um caracter, envolvendo-o por aspas simples. Por exemplo, `'z'`, `';'` e `'3'` são exemplos de constantes do tipo caracter. Assim, é bem mais cômodo fazer as atribuições

```
c = 'z';
d = ';';
e = '3';
```

que, na prática, são idênticas às anteriores.

Alguns caracteres produzem efeitos especiais tais como acionar um som de campainha ou realizar uma tabulação horizontal. Para representar um caracter como esse na linguagem C usamos uma seqüência de dois caracteres consecutivos iniciada por uma barra invertida. Por exemplo, `'\n'` é o mesmo que `10` e representa uma mudança de linha. A tabela a seguir mostra algumas constantes do tipo caracter.

| caracter | constante | símbolo gráfico |
|---|---|---|
| 0 | `'\0'` | caracter nulo |
| 9 | `'\t'` | tabulação horizontal |
| 10 | `'\n'` | mudança de linha |
| 11 | `'\v'` | tabulação vertical |
| 12 | `'\f'` | quebra de página |
| 13 | `'\r'` | retorno do carro/cursor |
| 32 | `' '` | espaço |
| 55 | `'7'` | 7 |
| 92 | `'\\'` | \ |
| 97 | `'a'` | a |

Na linguagem C, um **branco** (do inglês *whitespace*) é definido como sendo um caracter que é uma tabulação horizontal, uma mudança de linha, uma tabulação vertical, uma quebra de página, um retorno de carro/cursor e um espaço. Ou seja, os caracteres 9, 10, 11, 12, 13 e 32 são brancos e as constantes correspondentes são `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'` e `' '`. A função `scanf` trata todos os brancos como se fossem `' '`, assim como outras funções da linguagem C também o fazem.

Para imprimir um caracter na saída padrão com a função `printf` devemos usar o especificador de conversão de tipo caracter `%c` na seqüência de caracteres de formatação. Para ler um caracter a partir da entrada padrão com a função `scanf` também devemos usar o mesmo especificador de conversão de tipo caracter `%c` na seqüência de caracteres de formatação. Um exemplo de uso do tipo básico caracter é mostrado no programa 17.1.

Programa 17.1: Um programa usando o tipo `char`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    char c;
5    c = 'a';
6    printf("%c\n", c);
7    c = 97;
8    printf("%c\n", c);
9    printf("Informe um caracter: ");
10   scanf("%c", &c);
11   printf("%c = %d\n", c, c);
12   return 0;
13 }
```

Como vimos até aqui nesta seção, por enquanto não há muita utilidade para constantes e variáveis do tipo caracter. Mas em breve iremos usá-las para construir um tipo de dados muito importante chamado de cadeia de caracteres.

## 17.3 Expressões com caracteres

Como caracteres são implementados como números inteiros em um byte, as expressões envolvendo caracteres herdam todas as propriedades das expressões envolvendo números inteiros. Em particular, as operações aritméticas envolvendo variáveis do tipo `unsigned char` e `char` são executadas em aritmética `int`.

Dessa forma, considere o trecho de código a seguir:

```
unsigned char x, y, z;
x = 240;
y = 65;
z = x + y;
```

Na última linha o resultado da expressão `x + y` é `305`. No entanto, atribuições de números inteiros a variáveis do tipo caracter são feitas módulo 256. Por isso, a variável `z` acima receberá o valor `49`.

Do mesmo modo, após a execução do trecho de código a seguir:

```
unsigned char x;
char c;
x = 256;
c = 136;
```

a variável `x` conterá o valor `0` e a variável `c` conterá o valor `-120`.

Expressões relacionais podem naturalmente envolver caracteres. No trecho de código a seguir, algumas expressões lógicas envolvendo caracteres são mostradas:

```
char c;
c = 'a';
if ('a' <= c && c <= 'z')
  c = c - ' ';
printf("%c = %d\n", c, c);
if (65 <= c && c <= 122)
  c = c + 'a';
printf("%c = %d\n", c, c);
```

Qual a saída gerada pelas duas chamadas da função `printf` no trecho de código acima?

## Exercícios

17.1 Verifique se o programa 17.2 está correto.

Programa 17.2: O que faz este programa?

```
#include <stdio.h>
int main(void)
{
  char c;
  for (c = 0; c < 128; c++)
    printf(".");
  printf("\ntchau!\n");
  return 0;
}
```

17.2 Escreva um programa que imprima todas as letras minúsculas e todas as letras maiúsculas do alfabeto.

17.3 Escreva um programa que traduz um número de telefone alfabético de 8 dígitos em um número de telefone na forma numérica. Suponha que a entrada é sempre dada em caracteres maiúsculos.

Exemplo:

> Se a entrada é `URGENCIA` a saída deve ser `87436242`. Se a entrada é `1111FOGO` a saída deve ser `11113646`.

Se você não possui um telefone, então as letras que correspondem às teclas são as seguintes: 2=ABC, 3=DEF, 4=GHI, 5=JKL, 6=MNO, 7=PQRS, 8=TUV e 9=WXYZ.

17.4 *Scrabble* é um jogo de palavras em que os jogadores formam palavras usando pequenos quadrados que contêm uma letra e um valor. O valor varia de uma letra para outra, baseado na sua raridade. Os valores são os seguintes: AEILNORSTU=1, DG=2, BCMP=3, FHVWY=4, K=5, JX=8 e QZ=10.

Escreva um programa que receba uma palavra e compute o seu valor, somando os valores de suas letras. Seu programa não deve fazer distinção entre letras maiúsculas e minúsculas.

Exemplo:

> Se a palavra de entrada é `programa` a saída tem de ser `13`.

17.5 Escreva um programa que receba dois números inteiros $a$ e $b$ e um caracter $op$, tal que $op$ pode ser um dos cinco operadores aritméticos disponíveis na linguagem C (`+`, `-`, `*`, `/`, `%`), realize a operação $a \ op \ b$ e mostre o resultado na saída.

AULA 18

# TIPOS DE DADOS BÁSICOS

Como vimos até esta aula, a linguagem C suporta fundamentalmente dois tipos de dados numéricos: tipos inteiros e tipos com ponto flutuante. Valores do tipo inteiro são números inteiros em um dado intervalo finito e valores do tipo ponto flutuante são também dados em um intervalo finito e podem ter uma parte fracionária. Além desses, o tipo caracter também é suportado pela linguagem e um valor do tipo caracter também é um número inteiro em um intervalo finito mais restrito que o dos tipos inteiros.

Nesta aula faremos uma revisão dos tipos de dados básicos da linguagem C e veremos os especificadores desses tipos, seus especificadores de conversão para entrada e saída de dados, as conversões entre valores de tipos distintos, além de definir nossos próprios tipos de dados. Esta aula é uma revisão das aulas 7, 14 e 17 e também uma extensão desses conceitos, podendo ser usada como uma referência para tipos de dados básicos. As referências usadas nesta aula são [4, 7].

## 18.1 Tipos inteiros

Valores do tipo inteiro na linguagem C são números inteiros em um intervalo bem definido. Os tipos de dados inteiros na linguagem C têm tamanhos diferentes. O tipo `int` tem geralmente 32 bits: o bit mais significativo é reservado para o sinal do número inteiro. Por isso, o tipo `int` é chamado de tipo inteiro com sinal. A palavra reservada `signed` pode ser usada em conjunto com `int` para declarar uma variável do tipo inteiro com sinal, embora `signed` seja completamente dispensável. Uma variável do tipo inteiro sem sinal pode ser declarada com a palavra `unsigned` precedendo a palavra reservada `int`.

Além disso, programas podem necessitar de armazenar números inteiros grandes e podemos declarar variáveis do tipo inteiro para armazenar tais números com a palavra reservada `long`. Do mesmo modo, quando há necessidade de economia de espaço em memória, podemos declarar uma variável para armazenar números inteiros menores com a palavra reservada `short` precedendo a palavra reservada `int`.

Dessa forma, podemos construir um tipo de dados inteiro que melhor represente nossa necessidade usando as palavras reservadas `signed`, `unsigned`, `short` e `long`. Essas palavras reservadas são chamadas de **especificadores de tipos**, do inglês *type specifiers*, da linguagem C.

Podemos combinar os especificadores de tipos descritos acima na declaração de uma variável do tipo inteiro. No entanto, apenas seis combinações desses especificadores e da palavra reservada `int` produzem tipos de dados diferentes:

```
short int
unsigned short int
int
unsigned int
long int
unsigned long int
```

Qualquer outra combinação de especificadores na declaração de uma variável é equivalente a uma das combinações acima. A ordem que os especificadores ocorrem não tem influência sobre o resultado final: declarações feitas com `unsigned short int` ou com `short unsigned int` têm o mesmo resultado. Quando não estritamente necessário, a linguagem C permite que a palavra reservada `int` seja omitida, como por exemplo em `unsigned short` e `long`.

O intervalo de valores representado por cada um dos seis tipos inteiros varia de uma máquina para outra. Os compiladores, entretanto, devem obedecer algumas regras fundamentais. Em especial, o padrão especifica que o tipo `int` não seja menor que o tipo `short int` e que `long int` não seja menor que `int`.

As máquinas atuais, em sua grande maioria, são de 32 bits e a tabela abaixo mostra os intervalos de cada um dos possíveis tipos inteiros. Observe que os tipos `int` e `long int` têm intervalos idênticos nas máquinas de 32 bits. Em máquinas de 64 bits, há diferenciação entre esses dois tipos.

| Tipo | Menor valor | Maior valor |
|---|---:|---:|
| `short int` | −32.768 | 32.767 |
| `unsigned short int` | 0 | 65.535 |
| `int` | −2.147.483.648 | 2.147.483.647 |
| `unsigned int` | 0 | 4.294.967.294 |
| `long int` | −2.147.483.648 | 2.147.483.647 |
| `unsigned long int` | 0 | 4.294.967.294 |

Constantes numéricas, como vimos, são números que ocorrem no código dos programas, em especial em expressões aritméticas. As constantes, assim como as variáveis, podem ser números inteiros ou números com ponto flutuante.

Na linguagem C, as constantes numéricas do tipo inteiro podem ser descritas em decimal ou base 10, octal ou base 8 ou ainda hexadecimal ou base 16. Constantes decimais contêm dígitos de 0 (zero) a 9 (nove), mas não devem iniciar com 0 (zero). Exemplos de constantes decimais são mostrados a seguir:

```
75   -264   32767
```

Constantes octais contêm somente dígitos entre 0 (zero) e 7 (sete) e devem começar necessariamente com o dígito 0 (zero). Exemplos de constantes octais são mostrados a seguir:

```
075   0367   07777
```

E constante hexadecimais contêm dígitos entre 0 (zero) e 9 (nove) e letras entre `a` e `f`, e sempre devem iniciar com os caracteres `0x`. As letras podem ser maiúsculas ou minúsculas. Exemplos de constantes hexadecimais são mostrados a seguir:

```
0xf   0x8aff   0X12Acf
```

É importante destacar que a descrição de constantes como números na base octal e hexadecimal é apenas uma forma alternativa de escrever números que não tem efeito na forma como são armazenados na memória, já que sempre são armazenados na base binária. Além disso, essas notações são mais usadas quando trabalhamos com programas de baixo nível.

Não há necessidade de se convencionar uma única notação para escrever constantes e podemos trocar de uma notação para outra a qualquer momento. Na verdade, podemos inclusive misturar essas notações em expressões aritméticas como a seguir:

```
117 + 077 + 0xf0
```

e o resultado dessa expressão é 420 na base decimal.

O tipo de uma constante que é um número inteiro na base decimal é normalmente `int`. Se o valor da constante for muito grande para armazená-la como um `int`, a constante será armazenada como um `long int`. Se mesmo nesse caso ainda não for possível armazenar tal constante, o compilador tenta, como um último recurso, armazenar o valor como um `unsigned long int`. Constantes na base octal ou hexadecimal são armazenadas de maneira semelhante, com o compilador tentando armazená-la progressivamente como um `int`, `unsigned int`, `long int` ou `unsigned long int`.

Para forçar que o compilador trate uma constante como um número inteiro grande, devemos adicionar a letra `L` ou `l` ao final da constante. Como exemplo, as constantes abaixo são do tipo `long int`:

```
22L   055l   0xffffL
```

Para indicar que uma constante não tem sinal, devemos adicionar a letra `U` ou `u` ao final da constante. Por exemplo, as constantes abaixo são do tipo `unsigned int`:

```
22u   072U   0xa09DU
```

Podemos indicar que uma constante é um número inteiro grande e sem sinal usando uma combinação das letras acima, em qualquer ordem. Por exemplo, as constantes abaixo são do tipo `unsigned long int`:

```
22ul   07777UL   0xFFFFAALU
```

A função `printf` é usada para imprimir, entre outros, números inteiros. Relembrando, a função `printf` tem o seguinte formato:

```
printf(cadeia, expressão 1, expressão 2, ...);
```

onde *`cadeia`* é a **cadeia de caracteres de formatação**. Quando a função `printf` é chamada, a impressão da cadeia de caracteres de formatação ocorre na saída. Essa cadeia pode conter caracteres usuais, que serão impressos diretamente, e também **especificações de conversão**, cada uma iniciando com o caracter `%` e representando um valor a ser preenchido durante a impressão. A informação que sucede o caracter `%` 'especifica' como o valor deve ser 'convertido' a partir de sua representação interna binária para uma representação imprimível. Caracteres usuais na cadeia de caracteres de formatação são impressos na saída exatamente como ocorrem na cadeia enquanto que as especificações de conversão são trocadas por valores a serem impressos.

A tabela abaixo mostra tipos e especificadores de tipos e também especificadores de conversão para cadeias de caracteres de formatação da função `printf`, especificamente para tratamento de números inteiros.

| Especificador e tipo | Especificador de conversão |
|---|---|
| `short int` | `%hd` ou `%hi` |
| `unsigned short int` | `%hu` |
| `int` | `%d` ou `%i` |
| `unsigned int` | `%u` |
| `long int` | `%ld` ou `%li` |
| `unsigned long int` | `%lu` |

A forma geral de um especificador de conversão para números inteiros em uma cadeia de caracteres de formatação da função `printf` é a seguinte:

```
%[flags][comprimento][.precisão][hl]conversor
```

Os campos opcionais são mostrados entre colchetes. Ou seja, apenas `conversor` é obrigatório, que chamamos de especificador de conversão. A tabela a seguir mostra os possíveis *flags* para um especificador de conversão de uma cadeia de caracteres de formatação da função `printf`.

| Flag | Significado |
|---|---|
| `-` | Valor alinhado à esquerda |
| `+` | Valor precedido por + ou - |
| *espaços* | Valor positivo precedido por espaços |
| `0` | Número preenchido com zeros à esquerda |

Os especificadores de conversão `o` e `x` (ou `X`) permitem ainda que um número inteiro a ser mostrado na saída o seja na base octal ou hexadecimal, respectivamente. Se o número inteiro é pequeno, grande, com ou sem sinal, essas opções também podem ser descritas no especificador.

O programa 18.1 usa os tipos básicos conhecidos para números inteiros e seus especificadores, juntamente com caracteres especificadores de conversão nas cadeias de caracteres de formatação das chamadas da função `printf`.

Programa 18.1: Exemplo de tipos e especificadores de tipos inteiros e também de formatação de impressão com especificadores de conversão.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    short int i1;
5    unsigned short int i2;
6    int i3;
7    unsigned int i4;
8    long int i5;
9    unsigned long int i6;
10   i1 = 159;
11   i2 = 630u;
12   i3 = -991023;
13   i4 = 98979U;
14   i5 = 8393202L;
15   i6 = 34268298UL;
16   printf("%hd  %d  %ho\n", i1, i1, i1);
17   printf("%hu  %i  %hx\n", i2, i2, i2);
18   printf("%+d  % d  %09d\n", i3, i3, i3);
19   printf("%X  %d  %u\n", i4, i4, i4);
20   printf("%+ld  %ld  %10ld\n", i5, i5, i5);
21   printf("%lu  %10.8lu  %-lu\n", i6, i6, i6);
22   return 0;
23 }
```

Assim como para a função `printf`, outras opções de formatação também estão disponíveis para a função `scanf`, mais que aquelas vistas até esta aula. Neste caso também vale o conceito de especificadores de conversão. Quando uma função `scanf` é executada, especificadores de conversão são procurados na cadeia de caracteres de formatação (de leitura), após o símbolo usual `%`.

Os especificadores de conversão `h` e `l` podem ser usados em uma cadeia de caracteres de formatação de leitura, na função `scanf`, com o mesmo sentido que na cadeia de caracteres de formatação de escrita, na função `printf`, isto é, para indicar que a leitura será feita como um número inteiro curto ou longo, respectivamente. Um comprimento também pode ser usado, indicando o comprimento máximo do valor a ser armazenado na variável correspondente.

| Especificador | Significado |
|---|---|
| `d` | valor a ser lido é expresso em notação decimal; o argumento correspondente é do tipo `int`, a menos que os modificadores de conversão `h` ou `l` tenham sido usados, casos em que o valor será `short int` ou `long int`, respectivamente |
| `i` | como `%d`, exceto números na base octal (começando com `0`) ou hexadecimal (começando com `0x` ou `0X`,) que também podem ser lidos |
| `u` | valor a ser lido é um inteiro e o argumento correspondente é do tipo `unsigned int` |
| `o` | valor a ser lido é expresso na notação octal e pode opcionalmente ser precedido por `0`; o argumento correspondente é um `int`, a menos que os modificadores de conversão `h` ou `l` tenham sido usados, casos em que é `short int` ou `long int`, respectivamente |
| `x` | valor a ser lido é expresso na notação hexadecimal e pode opcionalmente ser precedido por `0x` ou `0X`; o argumento correspondente é um `unsigned int`, a menos que os modificadores de conversão `h` ou `l` tenham sido usados, casos em que é `short int` ou `long int`, respectivamente |

A função `scanf` é executada primeiro esperando por um valor a ser informado pelo usuário e, depois, formatando este valor através do uso da cadeia de caracteres de formatação e do especificador de conversão.

A função `scanf` finaliza a leitura de dados sempre que o usuário informa um **branco**, que é o caracter 32 (espaço ou `' '`), o caracter 9 (tabulação horizontal ou `'\t'`), o caracter 10 (tabulação vertical ou `'\v'`), o caracter 11 (retorno de carro/cursor ou `'\r'`), o caracter 12 (mudança de linha ou `'\n'`) ou o caracter 13 (avanço de página ou `'\f'`).

Nesse sentido, considere uma entrada como no trecho de código abaixo:

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

Neste caso, se o usuário informar

```
prompt$ ./a.out
4 7
```

ou

```
prompt$ ./a.out
4
7
```

ou ainda

```
prompt$ ./a.out
4                 7
```

o resultado será o mesmo, ou seja, os valores `4` e `7` serão armazenados nas variáveis `a` e `b`, respectivamente.

## 18.2 Números com ponto flutuante

Como apenas números inteiros são incapazes de auxiliar na soluções de diversos problemas, valores reais são necessários mais especificamente para descrever números imensamente grandes ou pequenos. Números reais são armazenados na linguagem C como números com ponto flutuante.

A linguagem C fornece três tipos de dados numéricos com ponto flutuante:

| | |
|---|---|
| `float` | Ponto flutuante de precisão simples |
| `double` | Ponto flutuante de precisão dupla |
| `long double` | Ponto flutuante de precisão estendida |

O tipo `float` é usado quando a necessidade de precisão não é crítica. O tipo `double` fornece maior precisão, suficiente para a maioria dos problemas. E o tipo `long double` fornece a maior precisão possível e raramente é usado.

A linguagem C não estabelece padrão para a precisão dos tipos com ponto flutuante, já que diferentes computadores podem armazenar números com ponto flutuante de formas diferentes. Os computadores mais recentes seguem, em geral, as especificações do Padrão 754 da IEEE. A tabela abaixo, mostrada na aula 14, mostra as características dos tipos numéricos com ponto flutuante quando implementados sob esse padrão.

| Tipo | Menor valor (positivo) | Maior valor | Precisão |
|---|---|---|---|
| `float` | $1.17549 \times 10^{-38}$ | $3.40282 \times 10^{38}$ | 6 dígitos |
| `double` | $2.22507 \times 10^{-308}$ | $1.79769 \times 10^{308}$ | 15 dígitos |

Constantes com ponto flutuante podem ser descritas de diversas formas. Por exemplo, as constantes abaixo são formas válidas de escrever o número 391,0:

```
391.0   391.   391.0e0   391E0   3.91e+2   .391e3  3910.e-1
```

Uma constante com ponto flutuante deve conter um ponto decimal e/ou um expoente, que é uma potência de 10 pela qual o número é multiplicado. Se um expoente é descrito, ele deve vir precedido da letra `e` ou `E`. Opcionalmente, um sinal `+` ou `-` pode aparecer logo após a letra `e` ou `E`.

Por padrão, constantes com ponto flutuante são armazenadas como números de ponto flutuante de precisão dupla. Isso significa que, quando um compilador C encontra em um programa uma constante `391.0`, por exemplo, ele armazena esse número na memória no mesmo formato de uma variável do tipo `double`. Se for necessário, podemos forçar o compilador a armazenar uma constante com ponto flutuante no formato `float` ou `long double`. Para indicar precisão simples basta adicionar a letra `f` ou `F` ao final da constante, como por exemplo `391.0f`. Para indicar precisão estendida é necessário adicionar a letra `l` ou `L` ao final da constante, como por exemplo `391.0L`.

Os especificadores de conversão para números com ponto flutuante de precisão simples são `%e`, `%f` e `%g`, tanto para escrita como para leitura. O formato geral de um especificador de conversão para números com ponto flutuante é parecido com aquele descrito na seção anterior, como podemos observar abaixo:

```
%[flags][comprimento][.precisão][lL]conversor
```

O especificador de conversão `%e` mostra/solicita um número com ponto flutuante no formato exponencial ou notação científica. A precisão indica quantos dígitos após o ponto serão mostrados/solicitados ao usuário, onde o padrão é 6. O especificador de conversão `%f` mostra/solicita um número com ponto flutuante no formato com casas decimais fixas, sem expoente. A precisão indica o mesmo que para `%e`. E o especificador de conversão `%g` mostra o número com ponto flutuante no formato exponencial ou com casas decimais fixas, dependendo de seu tamanho. Diferentemente dos especificadores anteriores, a precisão neste caso indica o número máximo de dígitos significativos a ser mostrado/solicitado.

Atenção deve ser dispensada com pequenas diferenças na escrita e na leitura de números com ponto flutuante de precisão dupla e estendida. Quando da leitura de um valor do tipo `double` é necessário colocar a letra `l` precedendo `e`, `f` ou `g`. Esse procedimento é necessário apenas na leitura e não na escrita. Quando da leitura de um valor do tipo `long double` é necessário colocar a letra `L` precedendo `e`, `f` ou `g`.

## 18.3 Caracteres

Como mencionamos na aula 17, na linguagem C cada caracter é armazenado em um único byte na memória do computador. Assim, um caracter sem sinal é um número do conjunto $\{0, \ldots, 255\}$ e um caracter com sinal é um número do conjunto $\{-128, \ldots, -1, 0, 1, \ldots, 127\}$. Ou seja, um caracter é uma seqüência de 8 bits, dentre as 256 seqüências de 8 bits possíveis. A impressão de um caracter na saída padrão é a sua representação como um símbolo gráfico. Por exemplo, o símbolo gráfico do caracter `97` é `a`. Alguns caracteres têm representações gráficas especiais, como o caracter `10` que é representado por uma mudança de linha.

Uma variável do tipo caracter (com sinal) pode ser declarada na linguagem C com a palavra reservada `char`. Uma variável do tipo caracter sem sinal pode ser declarada com a mesma palavra reservada `char`, mas com o especificador de tipo `unsigned` a precedendo. Assim,

```
char c, d, e;
unsigned char f, g, h;
```

são declarações válidas de variáveis do tipo caracter.

Uma constante do tipo caracter sem sinal é um número no intervalo de $0$ a $255$ e uma constante do tipo caractere com sinal é uma constante no intervalo de $-128$ a $127$. Por exemplo, para as variáveis `c`, `d` e `e` declaradas acima, podemos fazer

```
c = 122;
d = 59;
e = 51;
```

Mais comum e confortavelmente, podemos especificar uma constante do tipo caracter através da representação gráfica de um caracter, envolvendo-o por aspas simples. Por exemplo, `'z'`, `';'` e `'3'` são exemplos de constantes do tipo caracter. Assim, é bem mais cômodo fazer as atribuições

```
c = 'z';
d = ';';
e = '3';
```

que, na prática, são idênticas.

Alguns caracteres produzem efeitos especiais tais como acionar um som de campainha ou realizar uma tabulação horizontal. Para representar um caracter como esse na linguagem C usamos uma seqüência de dois caracteres consecutivos iniciada por uma barra invertida. Por exemplo, `'\n'` é o mesmo que `10` e representa uma mudança de linha. A tabela a seguir mostra algumas constantes do tipo caracter.

| **caracter** | **constante** | **símbolo gráfico** |
|---|---|---|
| 0 | `'\0'` | caracter nulo |
| 9 | `'\t'` | tabulação horizontal |
| 10 | `'\n'` | mudança de linha |
| 11 | `'\v'` | tabulação vertical |
| 12 | `'\f'` | quebra de página |
| 13 | `'\r'` | retorno do carro/cursor |
| 32 | `' '` | espaço |
| 55 | `'7'` | 7 |
| 92 | `'\\'` | \ |
| 97 | `'a'` | a |

Na linguagem C, um **branco** (do inglês *whitespace*) é definido como sendo um caracter que é uma tabulação horizontal, uma mudança de linha, uma tabulação vertical, uma quebra de página, um retorno de carro/cursor ou um espaço. Ou seja, os caracteres 9, 10, 11, 12, 13 e 32 são brancos e as constantes correspondentes são `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'` e `' '`. A função `scanf` trata todos os brancos como se fossem `' '`, assim como outras funções também o fazem.

O especificador de conversão `%c` permite que as funções `scanf` e `printf` de entrada e saída de dados, respectivamente, possam ler ou escrever um único caracter. É importante reiterar que a função `scanf` não salta caracteres brancos antes de ler um caracter. Se o próximo caracter a ser lido foi digitado como um espaço ou uma mudança de linha, então a variável correspondente conterá um espaço ou uma mudança de linha. Para forçar a função `scanf` desconsiderar espaços em branco antes da leitura de um caracter, é necessário adicionar um espaço na cadeia de caracteres de formatação de leitura antes do especificador de formatação `%c`, como a seguir:

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

Um espaço na cadeia de caracteres de formatação de leitura significa que haverá um salto de zero ou mais caracteres brancos.

O programa 18.2 mostra um exemplo de uso da função `scanf` e de uma cadeia de caracteres de formatação contendo um especificador de tipo `%c`.

Programa 18.2: Conta o número de vogais minúsculas na frase digitada pelo usuário.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    char c;
5    int conta;
6    printf("Digite uma frase: ");
7    conta = 0;
8    do {
9      scanf("%c", &c);
10     if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
11       conta++;
12   } while (c != '\n');
13   printf("A frase tem %d vogais minúsculas\n", conta);
14   return 0;
15 }
```

A função `scanf` tem compreensão dificultada em alguns casos, especialmente quando queremos ralizar a entrada de caracteres ou ainda quando misturamos a entrada de números e caracteres em um programa. A função `scanf` é essencialmente uma função de "casamento

de cadeias de caracteres" que tenta emparelharar grupos de caracteres com especificações de conversão. Essa função é também controlada por uma cadeia de caracteres de formatação e, quando chamada, começa o processamento da informação nessa cadeia a partir do caracter mais a esquerda. Para cada especificação de conversão na cadeia de caracteres de formatação, a função `scanf` tenta localizar um item do tipo apropriado na cadeia de entrada, saltando brancos se necessário. `scanf` lê então esse item, parando quando encontra um caracter que que não pode pertencer àquele item. Se o item foi lido corretamente, a função `scanf` continua processando o restante da cadeia de caracteres de formatação. Se qualquer item não pode ser lido com sucesso, a função `scanf` pára imediatamente sem olhar para o resto da cadeia de caracteres de formatação e o restante da cadeia de entrada.

Considere, por exemplo, que temos duas variáveis do tipo `int` com identificadores `a` e `b` e duas variáveis do tipo `float` com identificadores `x` e `y`. Considere a seguinte chamada à função `scanf`:

```
scanf("%d%d%f%f", &a, &b, &x, &y);
```

Suponha que o usuário informou os valores correspondentes a essas variáveis da seguinte forma:

```
prompt$ ./a.out
       -10
2   0.33
 27.8e3
```

Como, nesse caso, a leitura busca por números, a função ignora os brancos. Dessa forma, os números acima podem ser colocados todos em uma linha, separados por espaços, ou em várias linhas, como mostramos acima. A função `scanf` enxerga a entrada acima como uma cadeia de caracteres como abaixo:

```
␣␣␣␣␣␣-10•2␣␣␣0.33•␣27.8e3•
```

onde `␣` representa o caracter espaço e `•` representa o caracter de mudança de linha (ou a tecla `Enter`). Como a função salta os brancos, então os números são lidos de forma correta.

Se a cadeia de caracteres de formatação de leitura contém caracteres usuais além dos especificadores de tipo, então o casamento dessa cadeia de caracteres e da cadeia de caracteres de entrada se dá de forma ligeiramente diferente. O processamento de um caracter usual na cadeia de caracteres de formatação de uma função `scanf` depende se o caracter é um branco ou não. Ou seja,

- um ou mais brancos consecutivos na cadeia de caracteres de formatação faz com que a função `scanf` leia repetidamente brancos na entrada até que um caracter não branco seja lido; o número de brancos na cadeia de caracteres de formatação é irrelevante;

- outros caracteres na cadeia de caracteres de formatação, não brancos, faz com que a função `scanf` o compare com o próximo caracter da entrada; se os dois caracteres são idênticos, a função `scanf` descarta o caracter de entrada e continua o processamento da cadeia de caracteres de formatação; senão, a função `scanf` aborta sua execução não processando o restante da cadeia de caracteres de formatação e os caracteres de entrada.

Por exemplo, suponha que temos uma leitura como abaixo:

```
scanf("%d/%d", &dia, &mes);
```

Suponha que a entrada seja

`␣2/␣33•`

então, a função `scanf` salta o primeiro espaço, associa `%d` com `2`, casa `/` e `/`, salta o espaço e associa `%d` com `33`. Por outro lado, se a entrada é

`␣2␣/␣33•`

a função `scanf` salta o espaço, associa `%d` com `2`, tenta casar o caracter `/` da cadeia de caracteres de formatação com um espaço `␣` da entrada e como esses caracteres não casam, a função termina e o restante `␣/␣33•` da entrada permanecerá armazenado na memória até que uma próxima chamada à função `scanf` seja realizada.

Por conveniência e simplicidade, muitas vezes preferimos usar funções mais simples de entrada e saída de caracteres. As funções `getchar` e `putchar` são funções da biblioteca padrão de entrada e saída da linguagem C e são usadas exclusivamente com caracteres. A função `getchar` lê um caracter da entrada e devolve esse caracter. A função `putchar` toma um caracter como parâmetro e o exibe na saída. Dessa forma, se `c` é uma variável do tipo `char` as linhas a seguir:

```
scanf("%c", &c);
printf("%c", c);
```

são equivalentes às linhas abaixo:

```
c = getchar();
putchar(c);
```

Observe que a função `getchar` não tem parâmetros de entrada, mas, por ser uma função, carrega os parênteses como sufixo. Além disso, essa função devolve um caracter, que pode ser usado em uma expressão. Nas linhas acima, `getchar` é usada em uma atribuição. Por outro lado, a função `putchar` tem como parâmetro uma expressão do tipo caracter que, após avaliada, seu valor será exibido na saída. A função `putchar` não devolve qualquer valor.

## 18.4 Conversão de tipos

As arquiteturas dos computadores em geral restringem as operações em expressões para que sejam realizadas apenas com operandos de mesmo comprimento, isto é, mesmo número de bytes. A linguagem C, por outro lado, é menos restritiva nesse sentido e permite que valores de tipos básicos sejam misturados em expressões. Isso acarreta um maior trabalho para o compilador, já que é necessária a conversão de alguns operandos da expressão. Como essas conversões são realizadas pelo compilador, elas são chamadas de **conversões implícitas**. A linguagem C também permite que o programador faça suas próprias conversões, chamadas de **conversões explícitas**, usando operadores de conversão de tipo, como vimos na aula 14 e revisaremos adiante.

Conversões implícitas são realizadas nas seguintes situações:

- quando os operandos em uma expressão não são do mesmo tipo;
- quando o tipo do valor resultante da avaliação de uma expressão ao lado direito de um comando de atribuição não é compatível com o tipo da variável do lado esquerdo de um comando de atribuição;
- quando o tipo de um argumento em uma chamada de uma função não é compatível com o tipo do parâmetro correspondente;
- quando o tipo do valor resultante de uma expressão de devolução de uma função, junto da palavra reservada `return`, não é compatível com o tipo da função.

Nesta aula, veremos os dois primeiros casos, sendo que os dois últimos serão vistos quando tratarmos de funções, a partir da aula 33.

As conversões usuais em expressões são aplicadas aos operandos de todos os operadores binários aritméticos, relacionais e lógicos. A linguagem C faz a conversão dos operandos de maneira que os fiquem mais seguramente acomodados. Isso significa que o menor número de bits que mais seguramente acomodar os operandos será usado nessa conversão, se uma conversão for de fato necessária. As regras de conversão de tipos podem então ser divididas em dois tipos:

**o tipo de pelo menos um dos operandos é de ponto flutuante:** se um dos operandos é do tipo `long double`, então o outro operando será convertido para o tipo `long double`. Caso contrário, se um dos operandos é do tipo `double`, então o outro operando será convertido para o tipo `double`. Senão, um dos operandos é do tipo `float` e o outro será convertido para esse mesmo tipo;

**nenhum dos operandos é do tipo ponto flutuante:** primeiro, se qualquer um dos operandos é do tipo `char` ou `short int`, será convertido para o tipo `int`; depois, se um dos operandos é do tipo `unsigned long int`, então o outro operando será convertido para o tipo `unsigned long int`. Caso contrário, se um dos operandos é do tipo `long int`, então o outro operando será convertido para o tipo `long int`. Ainda, se um dos operandos é do tipo `unsigned int`, então o outro operando será convertido para o tipo `unsigned int`. Por fim, um dos operandos é do tipo `int` e o outro será convertido para esse mesmo tipo.

É importante alertar para um caso em que uma operação envolve um dos operandos com sinal e o outro sem sinal. Pelas regras acima, o operando com sinal é convertido para um operando sem sinal. Um exemplo simples dessa regra é comentado a seguir. Se um dos operandos é do tipo `int` e contém um número negativo e o outro operando é do tipo `unsigned int` e contém um número positivo, então uma operação envolvendo esses dois operandos deve ser realizada cuidadosamente. Nesse caso, o valor do tipo `int` será promovido para o tipo `unsigned int`, mas a conversão será feita usando a fórmula $k + 2^{32}$, onde $k$ é o valor com sinal. Esse problema é uma das causas de erro mais difíceis de depurar. Veja o programa 18.3.

Programa 18.3: Dificuldade na conversão de valores em expressões.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int i;
5    unsigned int j;
6    i = -1;
7    j = 1;
8    printf("i = %+d   i = %u\n", i, (unsigned int) i);
9    printf("j = %+d   j = %u\n", (int) j, j);
10   if (i < j)
11     printf("i < j\n");
12   else
13     printf("j < i\n");
14   return 0;
15 }
```

A saída do programa 18.3 é mostrada a seguir.

```
prompt$ ./a.out
i = -1   i = 4294967295
j = +1   j = 1
j < i
```

As conversões implícitas da linguagem C são bastante convenientes, como pudemos perceber até aqui. No entanto, muitas vezes é necessário que o programador tenha maior controle sobre as conversões a serem realizadas, podendo assim realizar conversões explícitas. Dessa forma, a linguagem C possui **operadores de conversão de tipo**, ou ***casts***, que já tomamos contato especialmente na aula 14. Um operador de conversão de tipo é um operador unário que tem o seguinte formato geral:

```
(nome-do-tipo) expressão
```

onde `nome-do-tipo` é o nome de qualquer tipo básico de dados que será usado para converter o valor da `expressão`, após sua avaliação.

Por exemplo, se `f` e `frac` são variáveis do tipo `float`, então o trecho de código a seguir:

```
f = 123.4567f;
frac = f - (int) f;
```

fará com que a variável `frac` contenha a parte fracionária da variável `f`.

## 18.5 Tipos de dados definidos pelo programador

A linguagem C possui poucos tipos de dados básicos, mas permite que um programador possa definir seus próprios tipos de dados. Para tanto, devemos usar a palavra reservada `typedef` no seguinte formato geral:

```
typedef tipo-básico tipo-novo;
```

onde `tipo-básico` é o nome de um dos tipos de dados básicos da linguagem C e `tipo-novo` é o nome do novo tipo de dados definido pelo programador. Um exemplo de definição e uso de um tipo é apresentado a seguir:

```
typedef int Logic;
Logic primo, crescente;
```

Na primeira linha, `typedef` é uma palavra reservada da linguagem C, `int` é o tipo básico de dados para números inteiros e `Logic` é o nome do novo tipo de dados criado pelo programador. Na segunda linha, ocorre a declaração de duas variáveis do tipo `Logic`: a variável `primo` e a variável `crescente`. O nome `Logic` desse novo tipo foi descrito com a primeira letra maiúscula, mas isso não uma obrigatoriedade. O novo tipo `Logic` criado a partir de `typedef` faz com que o compilador o adicione a sua lista de tipos conhecidos. Isso significa que podemos, a partir de então, declarar variáveis com esse tipo novo, usá-lo como operador conversor de tipo, etc. O compilador trata `Logic` como um sinônimo para `int`. Dessa forma, as variáveis `primo` e `crescente` são, na verdade, variáveis do tipo `int`.

Definições de tipos podem fazer com que o código seja mais compreensível, mais fácil de modificar e mais fácil de transportar de uma arquitetura para outra.

## 18.6 Operador sizeof

O operador unário `sizeof` permite que se determine quanto de memória é necessário para armazenar valores de um tipo qualquer. A expressão abaixo:

```
sizeof (nome-do-tipo)
```

determina um valor que é um inteiro sem sinal representando o número de bytes necessários para armazenar um valor do tipo dado por `nome-do-tipo`.

Veja o programa 18.4, que mostra as quantidades de bytes necessários para armazenamento de valores dos tipos básicos da linguagem C.

Programa 18.4: Exemplo do uso do operador unário `sizeof`.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    printf("Caracter:      %d\n", sizeof(char));
5    printf("Inteiros:\n");
6    printf("  short:       %d\n", sizeof(short int));
7    printf("  int:         %d\n", sizeof(int));
8    printf("  long int:    %d\n", sizeof(long int));
9    printf("Números de ponto flutuante:\n");
10   printf("  float:       %d\n", sizeof(float));
11   printf("  double:      %d\n", sizeof(double));
12   printf("  long double: %d\n", sizeof(long double));
13   return 0;
14 }
```

A saída do programa 18.4, executado em uma máquina de 32 bits, é apresentada abaixo:

```
Caracter:      1
Inteiros:
  short:       2
  int:         4
  long int:    4
Números de ponto flutuante:
  float:       4
  double:      8
  long double: 12
```

## 18.7 Exercícios

Exercícios para treinar os conceitos aprendidos nesta aula. Faça muitos testes com suas entradas e saídas.

18.1 Dadas $n$ triplas compostas por um símbolo de operação aritmética ($+, -, *$ ou $/$) e dois números reais, calcule o resultado ao efetuar a operação indicada para os dois números.

Faremos esse exercício usando a estrutura condicional `switch`, que compara uma expressão com uma seqüência de valores. Essa estrutura tem o seguinte formato:

```
switch (expressão) {
  case constante:
    seqüência de comandos
    break;
  case constante:
    seqüência de comandos
    break;
  ...
  default:
    seqüência de comandos
    break;
}
```

Veja o programa 18.5. Observe especialmente a leitura dos dados.

18.2 Um matemático italiano da idade média conseguiu modelar o ritmo de crescimento da população de coelhos através de uma seqüência de números naturais que passou a ser conhecida como **seqüência de Fibonacci**. A seqüência de Fibonacci é descrita pela seguinte fórmula de recorrência:

$$\begin{cases} F_1 &= 1 \\ F_2 &= 1 \\ F_i &= F_{i-1} + F_{i-2}\,, \quad \text{para } i \geqslant 3. \end{cases}$$

Escreva um programa que dado $n \geqslant 1$ calcule e exiba $F_n$.

18.3 Os babilônios descreveram a mais de 4 mil anos um método para calcular a raiz quadrada de um número. Esse método ficou posteriormente conhecido como método de Newton. Dado um número $x$, o método parte de um chute inicial $y$ para o valor da raiz quadrada de $x$ e sucessivamente encontra aproximações desse valor calculando a média aritmética de $y$ e de $x/y$. O exemplo a seguir mostra o método em funcionamento para o cálculo da raiz quadrada de 3, com chute inicial 1:

| $x$ | $y$ | $x/y$ | $(y + x/y)/2$ |
|---|---|---|---|
| 3 | 1 | 3 | 2 |
| 3 | 2 | 1.5 | 1.75 |
| 3 | 1.75 | 1.714286 | 1.732143 |
| 3 | 1.732143 | 1.731959 | 1.732051 |
| 3 | 1.732051 | 1.732051 | 1.732051 |

Programa 18.5: Solução do exercício 18.1.

```c
#include <stdio.h>
int main(void)
{
  char operador;
  int n;
  float op1, op2, result;
  printf("Informe n: ");
  scanf("%d", &n);
  for ( ; n > 0; n--) {
    printf("Informe a expressão (ex.: 5.3 * 3.1): ");
    scanf("%f %c%f", &op1, &operador, &op2);
    switch (operador) {
      case '+':
        result = op1 + op2;
        break;
      case '-':
        result = op1 - op2;
        break;
      case '*':
        result = op1 * op2;
        break;
      case '/':
        result = op1 / op2;
        break;
      default:
        break;
    }
    printf("%f %c %f = %f\n", op1, operador, op2, result);
  }
  return 0;
}
```

Escreva um programa que receba um número real positivo $x$ e um número real $\varepsilon$ e calcule a raiz quadrada de $x$ usando o método de Newton, até que o valor absoluto da diferença entre dois valores consecutivos de $y$ seja menor que $\varepsilon$. Mostre também na saída a quantidade de passos realizados para obtenção da raiz de $x$.

AULA 19

# VETORES

Até o momento, estudamos os tipos de dados básicos ou elementares da linguagem C de programação: `char`, `int`, `float` e `double`. Os tipos de dados básicos se caracterizam pelo fato que seus valores não podem ser decompostos. Por outro lado, se os valores de um tipo de dados podem ser decompostos ou subdivididos em valores mais simples, então o tipo de dados é chamado de **complexo**, **composto** ou **estruturado**. A organização desses valores e as relações estabelecidas entre eles determinam o que conhecemos como **estrutura de dados**. Estudaremos algumas dessas estruturas de dados em nossas aulas futuras, como por exemplo as listas lineares, pilhas e filas.

Nesta aula iniciaremos o estudo sobre variáveis compostas, partindo de uma variável conhecida como **variável composta homogênea unidimensional** ou simplesmente **vetor**. Características específicas da linguagem C no tratamento de vetores também serão abordadas.

## 19.1 Motivação

Como um exemplo da necessidade do uso da estrutura de dados conhecida como vetor, considere o seguinte problema:

> Dadas cinco notas de uma prova dos(as) estudantes de uma disciplina, calcular a média das notas da prova e a quantidade de estudantes que obtiveram nota maior que a média e a quantidade de estudantes que obtiveram nota menor que a média.

Uma tentativa natural e uma idéia inicial para solucionar esse problema consiste no uso de uma estrutura de repetição para acumular o valor das cinco notas informadas e o cálculo posterior da média destas cinco notas. Esses passos resolvem o problema inicial do cálculo da média das provas dos estudantes. Mas e a computação das quantidades de alunos que obtiveram nota maior e menor que a média já computada? Observe que após lidas as cinco notas e processadas para o cálculo da média em uma estrutura de repetição, a tarefa de encontrar as quantidades de estudantes que obtiveram nota superior e inferior à média é impossível, já que as notas dos estudantes não estão mais disponíveis na memória. Isto é, a menos que o(a) programador(a) peça ao usuário para informar as notas dos(as) estudantes novamente, não há como computar essas quantidades.

Apesar disso, ainda podemos resolver o problema usando uma estrutura seqüencial em que todas as cinco notas informadas pelo usuário ficam armazenadas na memória e assim podemos posteriormente computar as quantidades solicitadas consultando estes valores. Veja o programa 19.1.

Programa 19.1: Um programa para resolver o problema da seção 19.1.

```
#include <stdio.h>
int main(void)
{
  int menor, maior;
  float nota1, nota2, nota3, nota4, nota5, media;
  printf("Informe as notas dos alunos: ");
  scanf("%f%f%f%f%f%f", &nota1, &nota2, &nota3, &nota4, &nota5);
  media = (nota1 + nota2 + nota3 + nota4 + nota5) / 5;
  printf("\nMedia das provas: %2.2f\n", media);
  menor = 0;
  if (nota1 < media)
    menor++;
  if (nota2 < media)
    menor++;
  if (nota3 < media)
    menor++;
  if (nota4 < media)
    menor++;
  if (nota5 < media)
    menor++;
  maior = 0;
  if (nota1 > media)
    maior++;
  if (nota2 > media)
    maior++;
  if (nota3 > media)
    maior++;
  if (nota4 > media)
    maior++;
  if (nota5 > media)
    maior++;
  printf("\nQuantidade de estudantes com nota inferior à média: %d\n", menor);
  printf("\nQuantidade de estudantes com nota superior à média: %d\n", maior);
  return 0;
}
```

A solução que apresentamos no programa 19.1 é de fato uma solução para o problema. Isto é, dadas como entrada as cinco notas dos(as) estudantes em uma prova, o programa computa de forma correta as saídas que esperamos, ou seja, as quantidades de estudantes com nota inferior e superior à média da prova. No entanto, o que aconteceria se a sala de aula tivesse mais estudantes, como por exemplo 100? Ou 1.000 estudantes? Certamente, a estrutura seqüencial não seria apropriada para resolver esse problema, já que o programador teria de digitar centenas ou milhares de linhas repetitivas, incorrendo inclusive na possibilidade de propagação de erros e na dificuldade de encontrá-los. E assim como esse, outros problemas não podem ser resolvidos sem uma extensão na maneira de armazenar e manipular as entradas de dados.

## 19.2 Definição

Uma **variável composta homogênea unidimensional**, ou simplesmente um **vetor**, é uma estrutura de armazenamento de dados que se dispõe de forma linear na memória e é usada para armazenar valores de um mesmo tipo. Um vetor é então uma lista de células na memória de tamanho fixo cujos conteúdos são do mesmo tipo básico. Cada uma dessas células armazena um, e apenas um, valor. Cada célula do vetor tem um **endereço** ou **índice** através do qual podemos referenciá-la. O termo 'variável composta homogênea unidimensional' é bem explícito e significa que temos uma: (*i*) variável, cujos valores podem ser modificados durante a execução de um programa; (*ii*) composta, já que há um conjunto de valores armazenado na variável; (*iii*) homogênea, pois os valores armazenados na variável composta são todos de um mesmo tipo básico; e (*iv*) unidimensional, porque a estrutura de armazenamento na variável composta homogênea é linear. No entanto, pela facilidade, usamos o termo 'vetor' com o mesmo significado.

A forma geral de declaração de um vetor é dada a seguir:

```
tipo identificador[tamanho];
```

onde:

- `tipo` é um tipo de dados conhecido ou definido pelo(a) programador(a);
- `identificador` é o nome do vetor, fornecido pelo(a) programador(a); e
- `tamanho` é a quantidade de células a serem disponibilizadas para uso no vetor.

Por exemplo, a declaração a seguir

```
float nota[100];
```

faz com que 100 células contíguas de memória sejam reservadas, cada uma delas podendo armazenar números de ponto flutuante do tipo `float`. A referência a cada uma dessas células é realizada pelo identificador do vetor `nota` e por um índice. Na figura 19.1 mostramos o efeito da declaração do vetor `nota` na memória de um computador.

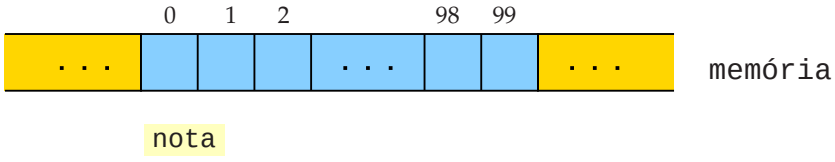



Figura 19.1: Vetor com 100 posições.

É importante observar que, na linguagem C, a primeira célula de um vetor tem índice 0, a segunda célula tem índice 1, a terceira tem índice 2 e assim por diante. Para referenciar o conteúdo da célula 0 do vetor `nota`, devemos usar o identificador do vetor e o índice 0, envolvido por colchetes, isto é, `nota[0]`. Assim, o uso de `nota[35]` em uma expressão qualquer referencia o trigésimo sexto elemento do vetor `nota`. Podemos, também, usar uma variável do tipo inteiro como índice de um vetor ou ainda uma expressão aritmética do tipo inteiro. Por exemplo, `nota[i]` acessa a $(i+1)$-ésima célula do vetor `nota`, supondo que a variável `i` é do tipo inteiro. Ou ainda, podemos fazer `nota[2*i+j]` para acessar a posição de índice $2i+j$ do vetor `nota`, supondo que `i` e `j` são variáveis do tipo inteiro é contêm valores compatíveis de tal forma que o resultado da expressão aritmética $2i+j$ seja um valor no intervalo de índices válido para o vetor `nota`. O compilador da linguagem C não verifica de antemão se os limites dos índices de um vetor estão corretos, trabalho que deve ser realizado pelo(a) programador(a).

Um erro comum, aparentemente inocente, mas que pode ter causas desastrosas, é mostrado no trecho de código abaixo:

```
int A[10], i;
for (i = 1; i <= 10; i++)
  A[i] = 0;
```

Alguns compiladores podem fazer com que a estrutura de repetição `for` acima seja executada infinitamente. Isso porque quando a variável `i` atinge o valor `10`, o programa armazena o valor `0` em `A[10]`. Observe, no entanto, que `A[10]` não existe e assim `0` é armazenado no compartimento de memória que sucede `A[9]`. Se a variável `i` ocorre na memória logo após `A[9]`, como é bem provável, então `i` receberá o valor `0` fazendo com que o laço inicie novamente.

## 19.3 Inicialização

Podemos atribuir valores iniciais a quaisquer variáveis de qualquer tipo básico no momento de suas respectivas declarações. Até o momento, não havíamos usado declarações e inicializações em conjunto. O trecho de código abaixo mostra declarações e inicializações simultâneas de variáveis de tipos básicos:

```
char c = 'a';
int num, soma = 0;
float produto = 1.0, resultado;
```

No exemplo acima, as variáveis `c`, `soma` e `produto` são inicializadas no momento da declaração, enquanto que `num` e `resultado` são apenas declaradas.

Apesar de, por alguns motivos, não termos usado declarações e inicializações simultâneas com variáveis de tipos básicos, essa característica da linguagem C é muito favorável quando tratamos de variáveis compostas. Do mesmo modo, podemos atribuir um valor inicial a um vetor no momento de sua declaração. As regras para declaração e atribuição simultâneas para vetores são um pouco mais complicadas, mas veremos as mais simples no momento. A forma mais comum de se fazer a inicialização de um vetor é através de uma lista de expressões constantes envolvidas por chaves e separadas por vírgulas, como no exemplo abaixo:

```
int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Se o inicializador tem menos elementos que a capacidade do vetor, os elementos restantes são inicializados com o valor 0 (zero), como abaixo:

```
int A[10] = {1, 2, 3, 4};
```

O resultado na memória é equivalente a termos realizado a inicialização abaixo:

```
int A[10] = {1, 2, 3, 4, 0, 0, 0, 0, 0, 0};
```

No entanto, não é permitido que o inicializador tenha mais elementos que a quantidade de compartimentos do vetor.

Podemos então facilmente inicializar um vetor todo com zeros da seguinte maneira:

```
int A[10] = {0};
```

Se um inicializador está presente em conjunto com a declaração de um vetor, então o seu tamanho pode ser omitido, como mostrado a seguir:

```
int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

O compilador então interpreta que o tamanho do vetor `A` é determinado pela quantidade de elementos do seu inicializador. Isso significa que, após a execução da linha do exemplo de código acima, o vetor `A` tem 10 compartimentos de memória, do mesmo modo como se tivéssemos especificado isso explicitamente, como no primeiro exemplo.

## 19.4 Exemplo com vetores

Vamos agora resolver o problema da seção 19.1 do cálculo das médias individuais de estudantes e da verificação da quantidade de estudantes com média inferior e também superior à média da classe. Solucionaremos esse problema construindo um programa que usa vetores e veremos como nossa solução se mostra muito mais simples, mais eficiente e facilmente extensível para qualquer quantidade de estudantes que se queira. Vejamos então o programa 19.2 a seguir.

Programa 19.2: Solução do problema proposto na seção 19.1 usando um vetor.

```
#include <stdio.h>
int main(void)
{
  int i, menor, maior;
  float nota[5], soma, media;
  for (i = 0; i < 5; i++) {
    printf("Informe a nota do(a) estudante %d: ", i+1);
    scanf("%f", &nota[i]);
  }
  soma = 0.0;
  for (i = 0; i < 5; i++)
    soma = soma + nota[i];
  media = soma / 5;
  menor = 0;
  maior = 0;
  for (i = 0; i < 5; i++) {
    if (nota[i] < media)
      menor++;
    if (nota[i] > media)
      maior++;
  }
  printf("\nMedia das provas: %2.2f\n", media);
  printf("Quantidade de alunos com nota inferior à média: %d\n", menor);
  printf("Quantidade de alunos com nota superior à média: %d\n", maior);
  return 0;
}
```

Observe que o programa 19.2 é mais conciso e de mais fácil compreensão que o programa 19.1. O programa é também mais facilmente extensível, no sentido que muito poucas modificações são necessárias para que esse programa possa solucionar problemas semelhantes com outras quantidades de estudantes. Isto é, se a turma tem 5, 10 ou 100 estudantes, ou ainda um número desconhecido $n$ que será informado pelo usuário durante a execução do programa, uma pequena quantidade de esforço será necessária para alteração de poucas linhas do programa.

## 19.5 Macros para constantes

Em geral, quando um programa contém constantes, uma boa idéia é dar nomes a essas constantes. Podemos atribuir um nome ou identificador a uma constante usando a definição de uma **macro**. Uma macro é definida através da diretiva de pré-processador `#define` e tem o seguinte formato geral:

```
#define identificador constante
```

onde `#define` é uma diretiva do pré-processador da linguagem C e `identificador` é um nome associado à `constante` que vem logo a seguir. Observe que, por ser uma diretiva do pré-processador, a linha contendo uma definição de uma macro não é finalizada por `;`. Em geral, a definição de uma macro ocorre logo no início do programa, após as diretivas `#include` para inclusão de cabeçalhos de bibliotecas de funções. Além disso, o identificador de uma macro é, preferencial mas não obrigatoriamente, descrito em letras maiúsculas.

Exemplos de macros são apresentados a seguir:

```
#define CARAC 'a'
#define NUMERADOR 4
#define MIN -10000
#define TAXA 0.01567
```

Quando um programa é compilado, o pré-processador troca cada macro definida no código pelo valor que ela representa. Depois disso, um segundo passo de compilação é executado. O programa 14.5 pode ser reescrito com o uso de uma macro, como podemos ver no programa 19.3, onde uma macro com identificador `PI` é definida e usada no código.

Programa 19.3: Cálculo da área do círculo.

```
1  #include <stdio.h>
2  #define PI 3.141592f
3  int main(void)
4  {
5    double raio, area;
6    printf("Digite o valor do raio: ");
7    scanf("%lf", &raio);
8    area = PI * raio * raio;
9    printf("A área do círculo de raio %f é %f\n", raio, area);
10   return 0;
11 }
```

O uso de macros com vetores é bastante útil porque, como já vimos, um vetor faz alocação estática da memória, o que significa que em sua declaração ocorre uma reserva prévia de um número fixo de compartimentos de memória. Por ser uma alocação estática, não há possibilidade de aumento ou diminuição dessa quantidade após a execução da linha de código contendo a declaração do vetor. O programa 19.2 pode ser ainda modificado como no programa 19.4 com a inclusão de uma macro que indica a quantidade de notas a serem processadas.

Programa 19.4: Solução do problema proposto na seção 19.1 usando uma macro e um vetor.

```
1  #include <stdio.h>
2  #define MAX 5
3  int main(void)
4  {
5    int i, menor, maior;
6    float nota[MAX], soma, media;
7    for (i = 0; i < MAX; i++) {
8      printf("Informe a nota do(a) estudante %d: ", i+1);
9      scanf("%f", &nota[i]);
10   }
11   soma = 0.0;
12   for (i = 0; i < MAX; i++)
13     soma = soma + nota[i];
14   media = soma / MAX;
15   menor = 0;
16   maior = 0;
17   for (i = 0; i < MAX; i++) {
18     if (nota[i] < media)
19       menor++;
20     if (nota[i] > media)
21       maior++;
22   }
23   printf("\nMedia das provas: %2.2f\n", media);
24   printf("Quantidade de estudantes com nota inferior à média: %d\n", menor);
25   printf("Quantidade de estudantes com nota superior à média: %d\n", maior);
26   return 0;
27 }
```

A macro `MAX` é usada quatro vezes no programa 19.4: na declaração do vetor `nota`, nas expressões relacionais das duas estruturas de repetição `for` e no cálculo da média. A vantagem de se usar uma macro é que, caso seja necessário modificar a quantidade de notas do programa por novas exigências do usuário, isso pode ser feito rápida e facilmente em uma única linha do código, onde ocorre a definição da macro.

Voltaremos a discutir mais sobre macros, sobre a diretiva `#define` e também sobre outras diretivas do pré-processador na aula 46.

## Exercícios

19.1 Dada uma seqüência de $n$ números inteiros, com $1 \leqslant n \leqslant 100$, imprimi-la em ordem inversa à de leitura.

Programa 19.5: Programa para solucionar o exercício 19.1.

```
1  #include <stdio.h>
2  #define MAX 100
3  int main(void)
4  {
5    int i, n, A[MAX];
6    printf("Informe n: ");
7    scanf("%d", &n);
8    for (i = 0; i < n; i++) {
9      printf("Informe o número %d: ", i+1);
10     scanf("%d", &A[i]);
11   }
12   printf("Números na ordem inversa da leitura:\n");
13   for (i = n-1; i >= 0; i--)
14     printf("%d  ", A[i]);
15   printf("\n");
16   return 0;
17 }
```

19.2 Uma prova consta de 30 questões, cada uma com cinco alternativas identificadas pelas letras A, B, C, D e E. Dado o cartão gabarito da prova e o cartão de respostas de $n$ estudantes, com $1 \leqslant n \leqslant 100$, computar o número de acertos de cada um dos estudantes.

19.3 Tentando descobrir se um dado era viciado, um dono de cassino o lançou $n$ vezes. Dados os $n$ resultados dos lançamentos, determinar o número de ocorrências de cada face.

19.4 Um jogador viciado de cassino deseja fazer um levantamento estatístico simples sobre uma roleta. Para isso, ele fez $n$ lançamentos nesta roleta. Sabendo que uma roleta contém 37 números (de 0 a 36), calcular a freqüência de cada número desta roleta nos $n$ lançamentos realizados.

AULA 20

# INVARIANTES

Algoritmos e programas podem conter um ou mais processos iterativos, que são controlados por estruturas de repetição. Na linguagem C, como sabemos, as estruturas de repetição são `while`, `for` e `do-while`. Processos iterativos podem ser documentados com invariantes, que nos ajudam a entender os motivos que o algoritmo e/ou programa está correto. Nesta aula definiremos invariantes, mostraremos exemplos de invariantes e, através deles, provaremos que os programas apresentados estão corretos.

## 20.1 Definição

Um **invariante** de um processo iterativo é uma relação entre os valores das variáveis que vale no início de cada iteração desse processo. Os invariantes explicam o funcionamento do processo iterativo e permitem provar por indução que ele tem o efeito desejado.

Devemos provar três elementos sobre um invariante de um processo iterativo:

**Inicialização:** é verdadeiro antes da primeira iteração da estrutura de repetição;

**Manutenção:** se é verdadeiro antes do início de uma iteração da estrutura de repetição, então permanece verdadeiro antes da próxima iteração;

**Término:** quando a estrutura de repetição termina, o invariante nos dá uma propriedade útil que nos ajuda a mostrar que o algoritmo ou programa está correto.

Quando as duas primeiras propriedades são satisfeitas, o invariante é verdadeiro antes de toda iteração da estrutura de repetição. Como usamos invariantes para mostrar a corretude de um algoritmo e/ou programa, a terceira propriedade é a mais importante, é aquela que permite mostrar de fato a sua corretude.

Dessa forma, os invariantes explicam o funcionamento dos processos iterativos e permitem provar, por indução, que esses processos têm o efeito desejado.

## 20.2 Exemplos

Nesta seção veremos exemplos do uso dos invariantes para mostrar a corretude de programas. O primeiro exemplo é bem simples e o programa contém apenas variáveis do tipo inteiro e, obviamente, uma estrutura de repetição. O segundo exemplo é um programa que usa um vetor para solução do problema.

Considere então o programa 20.1, que realiza a soma de $n$ números inteiros fornecidos pelo usuário. O programa 20.1 é simples e não usa um vetor para solucionar esse problema.

Programa 20.1: Soma $n$ inteiros fornecidos pelo usuário.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int n, i, num, soma;
5    printf("Informe n: ");
6    scanf("%d", &n);
7    soma = 0;
8    for (i = 1; i <= n; i++) {
9      /* a variável soma contém a soma dos primeiros
10        (i - 1) números fornecidos pelo usuário */
11     printf("Informe um número: ");
12     scanf("%d", &num);
13     soma = soma + num;
14   }
15   printf("Soma dos %d números é %d\n", n, soma);
16   return 0;
17 }
```

É importante destacar o comentário nas linhas 9 e 10, as primeiras do bloco de execução da estrutura de repetição `for` do programa 20.1: este é o invariante desse processo iterativo. E como Feofiloff destaca em [4], o enunciado de um invariante é, provavelmente, o único tipo de comentário que vale a pena inserir no corpo de um algoritmo, programa ou função.

Então, podemos provar a seguinte proposição.

**Proposição 20.1.** *O programa 20.1 realiza a soma de $n \geqslant 0$ números inteiros fornecidos pelo usuário.*

*Demonstração.*

Por conveniência na demonstração usaremos o modo matemático para expressar as variáveis do programa: `n` será denotada por $n$, `i` por $i$, `soma` por *soma* e `num` por *num*. Quando nos referirmos ao $i$-ésimo número inteiro fornecido pelo usuário e armazenado na variável `num`, usaremos a notação $num_i$.

Provar que o programa 20.1 está correto significa mostrar que para qualquer valor de $n$ e qualquer seqüência de $n$ números, a variável *soma* conterá, ao final do processo iterativo, o valor

$$soma = \sum_{i=1}^{n} num_i \, .$$

Vamos mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como *soma* contém o valor 0 (zero) e $i$ contém 1, é verdade que a variável *soma* contém a soma dos $i - 1$ primeiros números fornecidos pelo usuário.

Suponha agora que o invariante valha no início da $i$-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando $i$ contém o valor $n$. Por hipótese de indução, a variável *soma* contém o valor

$$\alpha = \sum_{i=1}^{n-1} num_i \, .$$

Então, no decorrer dessa $n$-ésima iteração, o usuário deve informar um número que será armazenado na variável *num* e, então, a variável *soma* conterá o valor

$$\begin{aligned} soma &= \alpha + num_n \\ &= \left( \sum_{i=1}^{n-1} num_i \right) + num_n \\ &= \sum_{i=1}^{n} num_i \, . \end{aligned}$$

Portanto, isso mostra que o programa 20.1 de fato realiza a soma dos $n$ números inteiros fornecidos pelo usuário. □

O próximo exemplo é dado da seguinte forma: dado um vetor com $n$ números inteiros fornecidos pelo usuário, encontrar o valor máximo armazenado nesse vetor. O programa 20.2 é bem simples e se propõe a solucionar esse problema.

Programa 20.2: Encontra o maior valor em um vetor com $n$ números inteiros fornecidos pelo usuário.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int n, vet[100], i, max;
5    printf("Informe n: ");
6    scanf("%d", &n);
7    for (i = 0; i < n; i++) {
8      printf("vet[%d]=", i);
9      scanf("%d", &vet[i]);
10   }
11   max = vet[0];
12   for (i = 1; i < n; i++) {
13     /* max é um elemento máximo em vet[0..i-1] */
14     if (vet[i] > max)
15       max = vet[i];
16   }
17   printf("O elemento de máximo valor no vetor é %d\n", max);
18   return 0;
19 }
```

**Proposição 20.2.** *O programa 20.2 encontra o elemento máximo de um conjunto de $n$ números fornecidos pelo usuário.*

*Demonstração.*

Novamente, por conveniência na demonstração usaremos o modo matemático para expressar as variáveis do programa: `n` será denotada por $n$, `i` por $i$, `vet` por *vet* e `max` por *max*.

Provar que o programa 20.2 está correto significa mostrar que para qualquer valor de $n$ e qualquer seqüência de $n$ números fornecidos pelo usuário e armazenados em um vetor *vet*, a variável *max* conterá, ao final do processo iterativo, o valor do elemento máximo em *vet*$[0..n-1]$.

Vamos mostrar que o invariante vale no início da primeira iteração do processo iterativo. Como *max* contém o valor armazenado em *vet*[0] e, em seguida, a variável $i$ é inicializada com o valor 1, então é verdade que a variável *max* contém o elemento máximo em *vet*$[0..i-1]$.

Suponha agora que o invariante valha no início da $i$-ésima iteração, com $1 < i < n$.

Vamos mostrar que o invariante vale no início da última iteração, quando $i$ contém o valor $n-1$. Por hipótese de indução, no início desta iteração a variável *max* contém o valor o elemento máximo de *vet*$[0..n-2]$. Então, no decorrer dessa iteração, o valor *vet*$[n-1]$ é comparado com *max* e dois casos devem ser avaliados:

(*i*) *vet*$[n-1] >$ *max*
Isso significa que o valor *vet*$[n-1]$ é maior que qualquer valor armazenado em *vet*$[0..n-2]$. Assim, na linha 15 a variável *max* é atualizada com *vet*$[n-1]$ e portanto a variável *max* conterá, ao final desta última iteração, o elemento máximo da seqüência em *vet*$[0..n-1]$.

(*ii*) *vet*$[n-1] \leqslant$ *max*
Isso significa que existe pelo menos um valor em *vet*$[0..n-2]$ que é maior ou igual a *vet*$[n-1]$. Por hipótese de indução, esse valor está armazenado em *max*. Assim, ao final desta última iteração, a variável *max* conterá o elemento máximo da seqüência em *vet*$[0..n-1]$.

Portanto, isso mostra que o programa 20.2 de fato encontra o elemento máximo em uma seqüência de $n$ números inteiros armazenados em um vetor. □

## Exercícios

20.1 O programa 20.3 recebe um número inteiro $n > 0$, uma seqüência de $n$ números inteiros, um número inteiro $x$ e verifica se $x$ pertence à seqüência de números.

Mostre que o programa 20.3 está correto.

20.2 Dado $n > 0$ e uma seqüência de $n$ números inteiros em um vetor, escreva um programa que inverta os elementos dessa seqüência armazenando-os no mesmo vetor.

Programa 20.3: Verifica se $x$ pertence à uma seqüência de $n$ números.

```
#include <stdio.h>
int main(void)
{
  int n, A[100], i, x;
  printf("Informe um valor para n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("A[%d]=", i);
    scanf("%d", &A[i]);
  }
  printf("Informe x: ");
  scanf("%d", &x);
  i = 0;
  while (i < n && A[i] != x) {
    /* x não pertence à A[0..i] */
    i++;
  }
  if (i < n)
    printf("%d é o %d-ésimo elemento do vetor\n", x, i);
  else
    printf("%d não se encontra no vetor\n", x);
  return 0;
}
```

20.3 O **piso** de um número $x$ é o único inteiro $i$ tal que $i \leqslant x < i+1$. O piso de $x$ é denotado por $\lfloor x \rfloor$. Escreva um programa que receba um número inteiro positivo $n$ e compute $\lfloor \log_2 n \rfloor$.

Segue uma amostra de valores:

| $n$ | 15 | 16 | 31 | 32 | 63 | 64 | 127 | 128 | 255 | 256 | 511 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lfloor \log_2 n \rfloor$ | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |

Documente seu programa enunciando o invariante do processo iterativo. Prove que seu programa está correto.

AULA 21

# EXERCÍCIOS

Faremos mais exercícios sobre vetores nesta aula.

## Enunciados

21.1 Dados dois vetores $x$ e $y$, ambos com $n$ elementos, $1 \leqslant n \leqslant 100$, determinar o produto escalar desses vetores.

Programa 21.1: Programa para solucionar o exercício 21.1.

```
#include <stdio.h>
int main(void)
{
  int i, n;
  float prod, x[100], y[100];
  printf("Informe n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Informe x[%d]: ", i+1);
    scanf("%f", &x[i]);
  }
  for (i = 0; i < n; i++) {
    printf("Informe y[%d]: ", i+1);
    scanf("%f", &y[i]);
  }
  prod = 0.0;
  for (i = 0; i < n; i++)
    prod = prod + x[i] * y[i];
  printf("Produto escalar dos vetores x e y é %f\n", prod);
  return 0;
}
```

21.2 São dadas as coordenadas reais $x$ e $y$ de um ponto, um número natural $n$ e as coordenadas reais de $n$ pontos, com $1 \leqslant n \leqslant 100$. Deseja-se calcular e imprimir sem repetição os raios das circunferências centradas no ponto $(x, y)$ que passem por pelo menos um dos $n$ pontos dados.

Exemplo:

$$\begin{cases} (x, y) = (1.0, 1.0) \\ n = 5 \\ \text{Pontos: } (-1.0, 1.2), (1.5, 2.0), (0.0, -2.0), (0.0, 0.5), (4.0, 2.0) \end{cases}$$

Nesse caso há três circunferências de raios $1.12, 2.01$ e $3.162$.

Observações:

(a) A distância entre os pontos $(a, b)$ e $(c, d)$ é $\sqrt{(a-c)^2 + (b-d)^2}$.

(b) Dois pontos estão na mesma circunferência se estão à mesma distância do centro.

21.3 Dadas duas seqüências de caracteres (uma contendo uma frase e outra contendo uma palavra), determine o número de vezes que a palavra ocorre na frase. Considere que essas seqüências têm no máximo 100 caracteres cada uma.

Exemplo:

Para a palavra ANA e a frase:

ANA E MARIANA GOSTAM DE BANANA.

Temos que a palavra ocorre 4 vezes na frase.

21.4 Calcule o valor do polinômio $p(x) = a_0 + a_1 x + \ldots + a_n x^n$ em $k$ pontos distintos. São dados os valores de $n$ (grau do polinômio), com $1 \leqslant n \leqslant 100$, de $a_0, a_1, \ldots a_n$ (coeficientes reais do polinômio), de $k$ e dos pontos $x_1, x_2, \ldots, x_k$.

21.5 Dado o polinômio $p(x) = a_0 + a_1 x + \ldots a_n x^n$, isto é, os valores de $n$ e de $a_0, a_1, \ldots, a_n$, com $1 \leqslant n \leqslant 100$ determine os coeficientes reais da primeira derivada de $p(x)$.

21.6 Dados dois polinômios reais

$$p(x) = a_0 + a_1 x + \ldots + a_n x^n \quad \text{e} \quad q(x) = b_0 + b_1 x + \ldots + b_m x^m$$

determinar o produto desses dois polinômios. Suponha que $1 \leqslant m, n \leqslant 100$.

21.7 Dadas duas seqüências com $n$ números inteiros entre 0 e 9, interpretadas como dois números inteiros de $n$ algarismos, $1 \leqslant n \leqslant 100$, calcular a seqüência de números que representa a soma dos dois inteiros.

Exemplo:

$n = 8$,

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1ª seqüência | | 8 | 2 | 4 | 3 | 4 | 2 | 5 | 1 |
| 2ª seqüência | + | 3 | 3 | 7 | 5 | 2 | 3 | 3 | 7 |
| | 1 | 1 | 6 | 1 | 8 | 6 | 5 | 8 | 8 |

AULA 22

# EXERCÍCIOS

Mais exercícios sobre vetores.

## Enunciados

22.1 Dada uma seqüência de $n$ números inteiros, com $1 \leqslant n \leqslant 100$, imprimi-la em ordem não decrescente de seus valores. Escreva três soluções usando o método da bolha, o método da inserção e o método da seleção em cada uma delas.

Programa 22.1: Programa para solucionar o exercício 22.1 usando o método da bolha.

```
#include <stdio.h>
int main(void)
{
  int i, j, n, aux, A[100];
  printf("Informe n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Informe A[%d]: ", i+1);
    scanf("%d", &A[i]);
  }
  for (i = 0; i < n - 1; i++)
    for (j = 0; j < n - 1 - i; j++)
      if (A[j] > A[j + 1]) {
        aux = A[j];
        A[j] = A[j + 1];
        A[j + 1] = aux;
      }
  for (i = 0; i < n; i++)
    printf("%d  ", A[i]);
  printf("\n");
  return 0;
}
```

Escreva os programas para solução deste exercício usando o método da inserção e o método da seleção.

22.2 Dizemos que uma seqüência de $n$ elementos, com $n$ par, é **balanceada** se as seguintes somas são todas iguais:

a soma do maior elemento com o menor elemento;
a soma do segundo maior elemento com o segundo menor elemento;
a soma do terceiro maior elemento com o terceiro menor elemento;
e assim por diante ...

Exemplo:

2 12 3 6 16 15 é uma seqüência balanceada, pois $16 + 2 = 15 + 3 = 12 + 6$.

Dados $n$ ($n$ par e $0 \leqslant n \leqslant 100$) e uma seqüência de $n$ números inteiros, verificar se essa seqüência é balanceada.

22.3 Dados dois números naturais $m$ e $n$, com $1 \leqslant m, n \leqslant 100$, e duas seqüências ordenadas com $m$ e $n$ números inteiros, obter uma única seqüência ordenada contendo todos os elementos das seqüências originais sem repetição.

22.4 Dada uma seqüência $x_1, x_2, \ldots, x_k$ de números inteiros, com $1 \leqslant k \leqslant 100$, verifique se existem dois segmentos consecutivos iguais nesta seqüência, isto é, se existem $i$ e $m$ tais que

$$x_i, x_{i+1}, \ldots, x_{i+m-1} = x_{i+m}, x_{i+m+1}, \ldots, x_{i+2m-1}.$$

Imprima, caso existam, os valores de $i$ e $m$.

Exemplo:

Na seqüência $7, 9, \underline{5, 4}, \underline{5, 4}, 8, 6$ existem $i = 3$ e $m = 2$.

AULA 23

# CADEIAS DE CARACTERES

Veremos nesta aula uma estrutura que possui tratamento especial na linguagem C: a cadeia de caracteres. Esta estrutura é similar a um vetor de caracteres, diferenciando-se apenas por conter um caracter especial no final, após o último caracter válido. Essa característica evita, em muitos casos, que tenhamos de manter uma variável que contenha o comprimento do vetor para saber o número de caracteres contidos nele. Outras características e diferenças importantes serão vistas a seguir.

Na aula 45 veremos funções da biblioteca `string.h`. Essa biblioteca da linguagem C contém diversas funções úteis para manipulação de cadeias de caracteres.

## 23.1 Literais

Vimos tomando contato com cadeias de caracteres desde quando escrevemos nosso primeiro programa na linguagem C. Por exemplo, na sentença abaixo:

```
printf("Programar é bacana!\n");
```

o único argumento passado para a função `printf` é a cadeia de caracteres (de formatação) `"Programar é bacana!\n"`. As aspas duplas são usadas para delimitar uma constante do tipo cadeia de caracteres, que pode conter qualquer combinação de letras, números ou caracteres especiais que não sejam as aspas duplas. Mesmo assim, é possível inserir as aspas duplas no interior de uma constante cadeia de caracteres, inserindo a seqüência `\"` nessa cadeia. Na linguagem C, uma constante do tipo cadeia de caracter é chamada de **literal**.

Quando estudamos o tipo de dados `char`, aprendemos que uma variável deste tipo pode conter apenas um único caracter. Para atribuir um caracter a uma variável, o caracter deve ser envolvido por aspas simples. Dessa forma, o trecho de código a seguir:

```
char sinal;
sinal = '+';
```

tem o efeito de atribuir o caractere cuja constante é `'+'` para a variável `sinal`. Além disso, aprendemos que existe uma distinção entre as aspas simples e as aspas duplas, sendo que no primeiro caso elas servem para definir constantes do tipo caracter e no segundo para definir constantes do tipo cadeia de caracteres. Assim, o seguinte trecho de código:

```
char sinal;
sinal = "+";
```

não está correto, já que que a variável `sinal` foi declarada do tipo `char`, podendo conter um único caracter. Lembre-se que na linguagem C as aspas simples e as aspas duplas são usadas para definir dois tipos de constantes diferentes.

Usamos literais especialmente quando chamamos as funções `printf` e `scanf` em um programa, ou seja, quando descrevemos cadeias de caracteres de formatação. Essencialmente, a linguagem C trata as literais como cadeias de caracteres. Quando o compilador da linguagem C encontra uma literal com $n$ caracteres em um programa, ele reserva $n+1$ compartimentos de memória para armazenar a cadeia de caracteres correspondente. Essa área na memória conterá os caracteres da cadeia mais um caractere extra, o caractere nulo, que registra o final da cadeia. O caractere nulo é um byte cujos bits são todos 0 (zeros) e é representado pela seqüência `'\0'`.

É importante destacar a diferença entre o caractere nulo e o caractere zero: o primeiro é um caractere não-imprimível, tem valor decimal 0 e constante `'\0'`; o segundo é um caractere imprimível, tem valor 48, símbolo gráfico `0` e constante `'0'`.

A literal `"abc"` é armazenada como um vetor de quatro caracteres na memória, como mostra a figura 23.1.

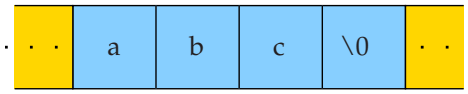



Figura 23.1: Armazenamento de uma literal na memória.

Por outro lado, uma literal também pode ser vazia. A literal `""` é uma literal vazia, representada na memória como na figura 23.2.

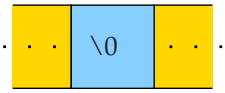



Figura 23.2: Literal vazia na memória.

## 23.2 Vetores de caracteres

Se queremos trabalhar com variáveis que comportam mais que um único caracter, temos de trabalhar com vetores de caracteres. No exercício 21.3, definimos dois vetores `palavra` e `frase` do tipo caracter, da seguinte forma:

```
char palavra[MAX+1], frase[MAX+1];
```

Para ler, por exemplo, o conteúdo da variável `palavra`, produzimos o seguinte trecho de programa:

```
printf("Informe a palavra: ");
i = 0;
do {
  scanf("%c", &palavra[i]);
  i++;
} while(palavra[i-1] != '\n');
m = i-1;
```

Para imprimir o conteúdo dessa mesma variável `palavra`, devemos escrever um trecho de programa da seguinte forma:

```
printf("Palavra: ");
for (i = 0; i < m; i++)
  printf("%c", palavra[i]);
printf("\n");
```

Note que sempre necessitamos de uma variável adicional para controlar o comprimento de um vetor de caracteres quando da sua leitura. A variável `m` faz esse papel no primeiro trecho de programa acima. Além disso, tanto para leitura como para escrita de um vetor de caracteres, precisamos de uma estrutura de repetição para processar os caracteres um a um. Com as cadeias de caracteres, evitamos esta sobrecarga de trabalho para o programador, como veremos daqui por diante.

## 23.3 Cadeias de caracteres

Algumas linguagens de programação de alto nível oferecem ao(à) programador(a) um tipo de dados especial para que variáveis do tipo cadeia de caracteres possam ser declaradas. Por outro lado, na linguagem C não há um tipo de dados como esse e, assim, qualquer vetor de caracteres pode ser usado para armazenar uma cadeia de caracteres. A diferença, nesse caso, é que uma cadeia de caracteres é sempre terminada por um caractere nulo. Uma vantagem dessa estratégia é que não há necessidade de se manter o comprimento da cadeia de caracteres associado a ela. Por outro lado, esse mesmo comprimento, se necessário, só pode ser encontrado através de uma varredura no vetor.

Suponha que necessitamos de uma variável capaz de armazenar uma cadeia de caracteres de até 50 caracteres. Como a cadeia de caracteres necessitará de um caractere nulo no final, então a cadeia de caracteres têm de ser declarada com 51 compartimentos para armazenar valores do tipo `char`, como mostrado a seguir:

```
#define MAX 50
char cadeia[MAX+1];
```

Na declaração de uma variável que pode armazenar uma cadeia de caracteres, sempre devemos reservar um compartimento a mais para que o caractere nulo possa ser armazenado. Como diversas funções da linguagem C supõem que as cadeias de caracteres são terminadas com o caractere nulo, se isso não ocorre em algum caso, o comportamento do programa passa a ser imprevisível.

A declaração de um vetor de caracteres com dimensão `MAX+1` não quer dizer que ele sempre conterá uma cadeia de caracteres com `MAX` caracteres. O comprimento de uma cadeia de caracteres depende da posição do caractere nulo na cadeia, não do comprimento do vetor onde a cadeia está armazenada.

Como em qualquer vetor, uma cadeia de caracteres também pode ser declarada e inicializada simultaneamente. Por exemplo,

```
char cidade[13] = "Campo Grande";
```

faz com que o compilador insira seqüencialmente os caracteres da cadeia de caracteres `"Campo Grande"` no vetor `cidade` e então adicione o caractere nulo ao final da cadeia. Apesar de `"Campo Grande"` parecer uma literal, na realidade, a linguagem C a enxerga como uma abreviação para um inicializador de um vetor, que poderia ter sido escrito equivalentemente como abaixo:

```
char cidade[13] = {'C','a','m','p','o',' ','G','r','a','n','d','e','\0'};
```

Se um inicializador tem menor comprimento que o comprimento do vetor, o compilador preencherá os caracteres restantes do vetor com o caractere nulo. Se, por outro lado, o inicializador tem maior comprimento que a capacidade de armazenamento do vetor associado, os caracteres iniciais do inicializador serão armazenados no vetor, sem que o último deles seja o caractere nulo, impedindo assim que essa variável seja usada como uma legítima cadeia de caracteres.

Vejamos agora o programa 23.1 que determina o comprimento de uma variável que é uma cadeia de caracteres. Qual será a saída deste programa?

Programa 23.1: Programa que determina o comprimento de uma cadeia de caracteres.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    char palavra[10] = "Ola!";
5    int n;
6    n = 0;
7    while(palavra[n] != '\0')
8      n++;
9    printf("O comprimento da palavra é %d\n", n);
10   return 0;
11 }
```

Há uma forma de ler e escrever cadeias de caracteres na linguagem C que facilita o trabalho de um(a) programador(a), evitando que sempre lance mão de uma estrutura de repetição para realizar uma dessas duas tarefas. O especificador de conversão `%s` no interior de uma cadeia de caracteres de formatação de entrada pode ser usado para mostrar um vetor de caracteres que é terminado por um caracter nulo, isto é, uma cadeia de caracteres. Assim se `palavra` é um vetor de caracteres terminado com o caractere nulo, a chamada da função abaixo:

```
printf("%s\n", palavra);
```

pode ser usada para mostrar o conteúdo completo da cadeia de caracteres `palavra` na saída padrão. Quando a função `printf` encontra o especificador de conversão `%s`, supõe que o argumento correspondente é uma cadeia de caracteres, isto é, um vetor de caracteres terminado por um caractere nulo.

Podemos também usar a mesma cadeia de caracteres de formatação `"%s"` para leitura de uma cadeia de caracteres. A função `scanf` pode ser usada com o especificador de conversão `%s` para ler uma cadeia de caracteres até que a leitura de um branco seja realizada. Assim, a chamada da função `scanf` abaixo:

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

tem o efeito de ler uma cadeia de caracteres digitada pelo usuário e de armazená-la no vetor de caracteres `palavra`.

É muito importante ressaltar que, ao contrário das chamadas anteriores da função `scanf`, no caso de leitura de cadeias de caracteres, o símbolo `&` *não* é adicionado como prefixo do identificador da variável. Veremos o porquê disto a partir da aula 49, quando aprenderemos apontadores.

Se na execução da função `scanf` anterior um(a) usuário(a) digita os caracteres `abcdefg`, a cadeia de caracteres `"abcdefg"` é armazenada no vetor `palavra`. Se, diferentemente, um(a) usuário(a) digita os caracteres `Campo Grande`, então apenas a cadeia de caracteres `"Campo"` é armazenada no vetor `palavra`, devido ao branco (␣). Os caracteres restantes da cadeia digitada ficarão disponíveis no *buffer* de entrada até que uma próxima chamada à função `scanf` seja realizada.

Para evitar os brancos na leitura de uma cadeia de caracteres, usamos o especificador de conversão `%[...]`, que também é usado na leitura de cadeias de caracteres, delimitando, dentro dos colchetes, quais são os caracteres permitidos em uma leitura. Qualquer outro caractere diferente dos especificados dentro dos colchetes finalizam a leitura. Além disso, podemos inverter essas permissões, indicando o caractere `^` como o primeiro caractere dentro dos colchetes. Por exemplo,

```
scanf("%[^\n]", palavra);
```

realiza a leitura de uma cadeia de caracteres, armazenando seu conteúdo no vetor de caracteres `palavra`. O caracter que finaliza a leitura é o `\n`. Qualquer outro caracter será lido e armazenado no vetor `palavra`.

É muito importante destacar que a função `scanf` termina automaticamente uma cadeia de caracteres que é lida com o especificador de conversão `"%s"` ou `"%[...]"` com um caracter nulo, fazendo assim que o vetor de caracteres se torne de fato uma cadeia de caracteres após sua leitura.

Veja um exemplo simples do uso dos conceitos de entrada e saída de cadeias de caracteres no programa 23.2.

Programa 23.2: Entrada e saída de uma cadeia de caracteres.

```
1  #include <stdio.h>
2  #define MAX 20
3  int main(void)
4  {
5    char palavra[MAX];
6    int n;
7    printf("Informe uma palavra (com até %d caracteres): ", MAX);
8    scanf("%s", palavra);
9    n = 0;
10   while (palavra[n] != '\0')
11     n++;
12   printf("A palavra [%s] tem %d caracteres\n", palavra, n);
13   return 0;
14 }
```

## Exercícios

23.1 Dada uma frase com no máximo 100 caracteres, determinar quantos caracteres espaço a frase contém.

Programa 23.3: Solução do exercícios 23.1.

```
1  #include <stdio.h>
2  #define MAX 100
3  int main(void)
4  {
5    char frase[MAX+1];
6    int esp, i;
7    printf("Informe uma frase: ");
8    scanf("%[^\n]", frase);
9    esp = 0;
10   for (i = 0; frase[i] != '\0'; i++)
11     if (frase[i] == ' ')
12       esp++;
13   printf("Frase tem %d espaços\n", esp);
14   return 0;
15 }
```

23.2 Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui.

23.3 Dadas duas cadeias de caracteres `cadeia1` e `cadeia2`, concatenar `cadeia2` no final de `cadeia1`, colocando o caracter nulo no final da cadeia resultante. A cadeia resultante a ser mostrada deve estar armazenada em `cadeia1`. Suponha que as cadeias sejam informadas com no máximo 100 caracteres.

23.4 Dada uma cadeia de caracter `cadeia` com no máximo 100 caracteres e um caractere `c`, buscar a primeira ocorrência de `c` em `cadeia`. Se `c` ocorre em `cadeia`, mostrar a posição da primeira ocorrência; caso contrário, mostrar o valor `-1`.

23.5 Dadas duas cadeias de caracteres `cadeia1` e `cadeia2`, cada uma com no máximo 100 caracteres, compará-las e devolver um valor menor que zero se `cadeia1` é lexicograficamente menor que `cadeia2`, o valor zero se `cadeia1` é igual ou tem o mesmo conteúdo que `cadeia2`, ou um valor maior que zero se `cadeia1` é lexicograficamente maior que `cadeia2`.

AULA 24

# MATRIZES

Nas aulas 19, 20, 21 e 22, tivemos contato com vetores ou variáveis compostas homogêneas unidimensionais. No entanto, uma variável composta homogênea pode ter qualquer número de dimensões. A partir desta aula, aprenderemos a trabalhar com as estruturas denominadas de matrizes ou variáveis compostas homogêneas bidimensionais.

Do mesmo modo, as matrizes são variáveis compostas porque representam a composição de um conjunto de valores indivisíveis, homogêneas porque esses valores são de um mesmo tipo de dados e bidimensionais porque, ao contrário dos vetores que são lineares ou têm uma única dimensão, estas estruturas têm duas dimensões. Nesta aula aprenderemos a declarar matrizes, a declarar e inicializar simultaneamente as matrizes e também a usar matrizes para solucionar problemas.

## 24.1 Definição, declaração e uso

Em matemática, uma **matriz** é uma tabela ou um quadro contendo $m$ linhas e $n$ colunas e usada, entre outros usos, para a resolução de sistemas de equações lineares e transformações lineares.

Uma matriz com $m$ linhas e $n$ colunas é chamada de uma **matriz** $m$ **por** $n$ e denota-se $m \times n$. Os valores $m$ e $n$ são chamados de **dimensões**, **tipo** ou **ordem** da matriz.

Um elemento de uma matriz $A$ que está na $i$-ésima linha e na $j$-ésima coluna é chamado de **elemento** $i, j$ ou **(**$i, j$**)-ésimo** elemento de $A$. Este elemento é denotado por $A_{i,j}$ ou $A[i, j]$. Observe que estas definições são matemáticas e que, na linguagem C, temos um deslocamento dos elementos da matriz devido aos índices das linhas e colunas iniciarem, como acontece nos vetores, pelo 0 (zero).

Uma matriz onde uma de suas dimensões é igual a 1 é geralmente chamada de **vetor**. Uma matriz de dimensões $1 \times n$, contendo uma linha e $n$ colunas, é chamada de **vetor linha** ou **matriz linha**, e uma matriz de dimensões $m \times 1$, contendo $m$ linhas e uma coluna, é chamada de **vetor coluna** ou **matriz coluna**.

Na linguagem C, as matrizes são declaradas similarmente aos vetores. Um tipo de dados é usado para a declaração, em seguida um identificador ou nome da variável matriz e, ainda, dois números inteiros envolvidos individualmente por colchetes, indicando as dimensões da matriz, isto é, seu número de linhas e seu número de colunas.

A forma geral da declaração de uma matriz é dada a seguir:

```
tipo identificador[dimensão1][dimensão2];
```

onde `tipo` é um dos tipos de dados da linguagem C ou um tipo definido pelo(a) programador(a), `identificador` é o nome da variável matriz fornecido pelo(a) programador(a) e `dimensão1` e `dimensão2` determinam a quantidade de linhas e colunas, respectivamente, a serem disponibilizadas para uso na matriz. Por exemplo, a declaração a seguir

```
int A[20][30];
```

faz com que 600 células de memória sejam reservadas, cada uma delas podendo armazenar valores do tipo `int`. A referência a cada uma dessas células é realizada pelo identificador da matriz `A` e por dois índices, o primeiro que determina a linha e o segundo que determina a coluna da matriz. Na figura 24.1 mostramos um esquema de como a matriz `A` é disposta quando declarada desta forma.

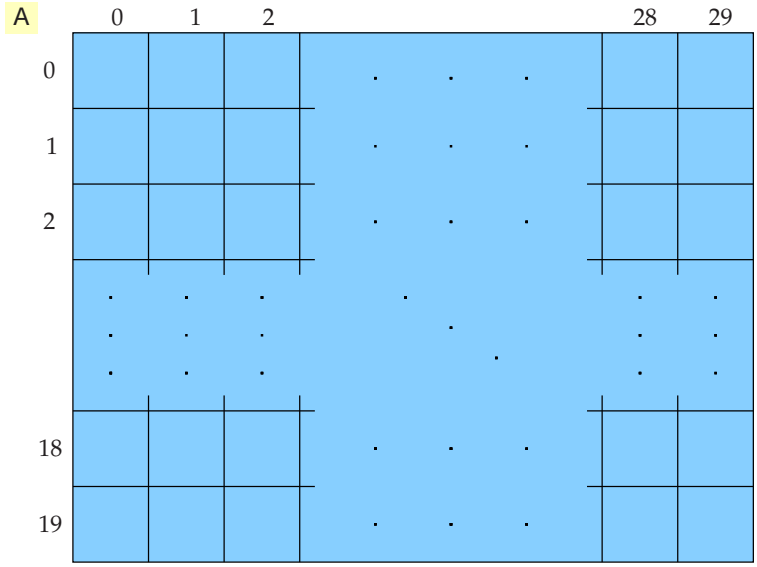



Figura 24.1: Matriz `A` com 20 linhas e 30 colunas.

Na linguagem C, a primeira linha de uma matriz tem índice 0, a segunda linha tem índice 1, e assim por diante. Do mesmo modo, a primeira coluna da matriz tem índice 0, a segunda tem índice 1 e assim por diante. Para referenciar o valor da célula da linha 0 e da coluna 3 da matriz `A`, devemos usar o identificador da variável e os índices 0 e 3 envolvidos por colchetes, ou seja, `A[0][3]`.

Apesar de visualizarmos uma matriz na forma de uma tabela bidimensional, essa não é a forma de armazenamento dessa variável na memória. A linguagem C armazena uma matriz na memória linha a linha, como mostrado na figura 24.2 para a matriz `A`.

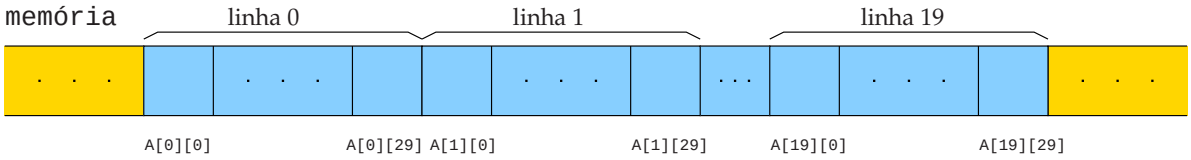



Figura 24.2: Matriz `A` com 20 linhas e 30 colunas disposta na memória.

## 24.2 Declaração e inicialização simultâneas

Da mesma forma como com os vetores, uma matriz pode ser inicializada no momento de sua declaração. Na verdade, variáveis compostas de qualquer dimensão podem ser inicializadas seguindo as mesmas regras. Para uma matriz, a declaração e inicialização simultâneas deve ser realizada agrupando os inicializadores de uma dimensão como abaixo:

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

Na declaração e inicialização acima, cada inicializador fornece valores para uma linha da matriz. A linguagem C possui algumas pequenas regras para declarar e inicializar matrizes ou variáveis compostas homogêneas de qualquer dimensão:

- se um inicializador não é grande o suficiente para inicializar um variável composta homogênea, então o restante dos elementos serão inicializados com 0 (zero). Por exemplo,

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

  inicializa as duas primeiras linhas da matriz `A`. As duas últimas linhas serão inicializadas com 0 (zero);

- se um inicializador mais interno não é longo o suficiente para inicializar uma linha, então o restante dos elementos na linha é inicializado com 0 (zero). Por exemplo,

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

- as chaves internas, que determinam as inicializações das linhas, podem ser omitidas. Neste caso, uma vez que o compilador tenha lido elementos suficientes para preencher uma linha, ele o faz e inicia o preenchimento da próxima linha. Por exemplo,

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

## 24.3 Exemplo

Apresentamos a seguir o programa 24.1, que resolve o seguinte problema:

> Dada uma matriz real $B$, de 5 linhas e 10 colunas, escrever um programa que calcule o somatório dos elementos da oitava coluna e que calcule o somatório da terceira linha.

Programa 24.1: Um programa que exemplifica o uso de matrizes.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int i, j;
5    float soma_c8, soma_l3, B[5][10];
6    for (i = 0; i < 5; i++) {
7      for (j = 0; j < 10; j++) {
8        printf("Informe B[%d][%d] = ", i, j);
9        scanf("%f", &B[i][j]);
10     }
11   }
12   soma_c8 = 0.0;
13   for (i = 0; i < 5; i++)
14     soma_c8 = soma_c8 + B[i][7];
15   printf("Valor da soma da oitava coluna é %4.4f\n", soma_c8);
16   soma_l3 = 0.0;
17   for (j = 0; j < 10; j++)
18     soma_l3 = soma_l3 + B[2][j];
19   printf("Valor da soma da terceira linha é %4.4f\n", soma_l3);
20   return 0;
21 }
```

## Exercícios

24.1 Dadas duas matrizes de números inteiros $A$ e $B$, de dimensões $m \times n$, com $1 \leqslant m, n \leqslant 100$, fazer um programa que calcule a matriz $C_{m \times n} = A + B$.

Programa 24.2: Solução do exercício 24.1.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  int m, n, i, j,
    A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
  printf("Informe as dimensões (m,n) das matrizes: ");
  scanf("%d%d", &m, &n);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("Informe A[%2d][%2d]: ", i, j);
      scanf("%d", &A[i][j]);
    }
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("Informe B[%2d][%2d]: ", i, j);
      scanf("%d", &B[i][j]);
    }
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      C[i][j] = A[i][j] + B[i][j];
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++)
      printf("%2d ", C[i][j]);
    printf("\n");
  }
  return 0;
}
```

24.2 Fazer um programa que dada uma matriz de números reais $A_{m\times n}$, determine $A^t$. Suponha que $1 \leqslant m, n \leqslant 100$.

24.3 Dada uma matriz de números reais $A$ com $m$ linhas e $n$ colunas, $1 \leqslant m, n \leqslant 100$, e um vetor de números reais $v$ com $n$ elementos, determinar o produto de $A$ por $v$.

24.4 Um vetor de números reais $x$ com $n$ elementos é apresentado como resultado de um sistema de equações lineares $Ax = b$, cujos coeficientes são representados em uma matriz de números reais $A_{m\times n}$ e o lados direitos das equações em um vetor de números reais $b$ de $m$ elementos. Dados $A, x$ e $b$, verificar se o vetor $x$ é realmente solução do sistema $Ax = b$, supondo que $1 \leqslant m, n \leqslant 100$.

AULA 25

# EXERCÍCIOS

## Enunciados

25.1 Dadas duas matrizes de números reais $A_{m\times n}$ e $B_{n\times p}$, com $1 \leqslant m, n, p \leqslant 100$, calcular o produto de $A$ por $B$.

25.2 Dada uma matriz de números inteiros $A_{m\times n}$, com $1 \leqslant m, n \leqslant 100$, imprimir o número de linhas e o número de colunas nulas da matriz.

Exemplo:

Se a matriz $A$ tem $m = 4$ linhas, $n = 4$ colunas e conteúdo

$$\begin{pmatrix} 1 & 0 & 2 & 3 \\ 4 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

então $A$ tem 2 linhas nulas e 1 coluna nula.

25.3 Dizemos que uma matriz de números inteiros $A_{n\times n}$ é uma **matriz de permutação** se em cada linha e em cada coluna houver $n - 1$ elementos nulos e um único elemento 1.

Exemplo:

A matriz abaixo é de permutação

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Observe que

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

não é de permutação.

Dada uma matriz de números inteiros $A_{n\times n}$, , com $1 \leqslant n \leqslant 100$, verificar se $A$ é de permutação.

25.4 Dada uma matriz de números reais $A_{m\times n}$, com $1 \leqslant m, n \leqslant 100$, verificar se existem elementos repetidos em $A$.

Programa 25.1: Solução do exercício 25.1.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  int m, n, p, i, j, k;
  float A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
  printf("Informe as dimensões (m,n) da matriz A: ");
  scanf("%d%d", &m, &n);
  printf("Informe o número de colunas da matriz B: ");
  scanf("%d", &p);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("Informe A[%2d][%2d]: ", i, j);
      scanf("%f", &A[i][j]);
    }
  for (i = 0; i < n; i++)
    for (j = 0; j < p; j++) {
      printf("Informe B[%2d][%2d]: ", i, j);
      scanf("%f", &B[i][j]);
    }
  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < n; k++)
        C[i][j] = C[i][j] + A[i][k] + B[k][j];
    }
  for (i = 0; i < m; i++) {
    for (j = 0; j < p; j++)
      printf("%2.2f ", C[i][j]);
    printf("\n");
  }
  return 0;
}
```

AULA 26

# EXERCÍCIOS

## Enunciados

26.1 Dizemos que uma matriz quadrada de números inteiros distintos é um **quadrado mágico**[1] se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

Exemplo:

A matriz

$$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

é um quadrado mágico.

Dada uma matriz quadrada de números inteiros $A_{n\times n}$, com $1 \leqslant n \leqslant 100$, verificar se $A$ é um quadrado mágico.

26.2 (a) Imprimir as $n$ primeiras linhas do triângulo de Pascal[2], com $1 \leqslant n \leqslant 100$.

$$\begin{array}{cccccc} 1 & & & & & \\ 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ 1 & 3 & 3 & 1 & & \\ 1 & 4 & 6 & 4 & 1 & \\ 1 & 5 & 10 & 10 & 5 & 1 \\ \vdots & & & & & \end{array}$$

(b) Imprimir as $n$ primeiras linhas do triângulo de Pascal usando apenas um vetor.

26.3 Um jogo de palavras cruzadas pode ser representado por uma matriz $A_{m\times n}$ onde cada posição da matriz corresponde a um quadrado do jogo, sendo que 0 indica um quadrado branco e $-1$ indica um quadrado preto. Indicar em uma dada matriz $A_{m\times n}$ de 0 e $-1$, com $1 \leqslant m, n \leqslant 100$, as posições que são início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras. Para isso, numere consecutivamente tais posições.

Exemplo:

[1] O primeiro registro conhecido de um quadrado mágico vem da China e data do ano de 650 a.c.
[2] Descoberto em 1654 pelo matemático francês Blaise Pascal.

Programa 26.1: Solução do exercício 26.1.

```c
#include <stdio.h>
#define MAX 100
int main(void)
{
  int n, i, j, A[MAX][MAX], soma, soma_aux, magico;
  printf("Informe a dimensão n da matriz: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      printf("Informe A[%2d][%2d]: ", i, j);
      scanf("%d", &A[i][j]);
    }
  soma = 0;
  for (j = 0; j < n; j++)
    soma = soma + A[0][j];
  magico = 1;
  for (i = 1; i < n && magico; i++) {
    soma_aux = 0;
    for (j = 0; j < n; j++)
      soma_aux = soma_aux + A[i][j];
    if (soma_aux != soma)
      magico = 0;
  }
  for (j = 0; j < n && magico; j++) {
    soma_aux = 0;
    for (i = 0; i < n; i++)
      soma_aux = soma_aux + A[i][j];
    if (soma_aux != soma)
      magico = 0;
  }
  soma_aux = 0;
  for (i = 0; i < n && magico; i++)
    soma_aux = soma_aux + A[i][i];
  if (soma_aux != soma)
    magico = 0;
  soma_aux = 0;
  for (i = 0; i < n; i++)
    soma_aux = soma_aux + A[i][n-i-1];
  if (soma_aux != soma)
    magico = 0;
  if (magico)
    printf("Quadrado mágico de ordem %d\n", n);
  else
    printf("Matriz não é quadrado mágico\n");
  return 0;
}
```

Dada a matriz

$$\begin{pmatrix} 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 \end{pmatrix}$$

a saída deverá ser

$$\begin{pmatrix} 1 & -1 & 2 & -1 & -1 & 3 & -1 & 4 \\ 5 & 6 & 0 & 0 & -1 & 7 & 0 & 0 \\ 8 & 0 & -1 & -1 & 9 & 0 & -1 & 0 \\ -1 & 10 & 0 & 11 & 0 & -1 & 12 & 0 \\ 13 & 0 & -1 & 14 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

26.4 Os elementos $a_{ij}$ de uma matriz de números inteiros $A_{n\times n}$ representam os custos de transporte da cidade $i$ para a cidade $j$. Dados uma matriz $A_{n\times n}$, com $1 \leqslant n \leqslant 100$, um número inteiro $m > 0$ representando a quantidade de itinerários, um número $k$ representando o número de cidades em cada um dos itinerários, calcular o custo total para cada itinerário.

Exemplo:

Dado

$$A = \begin{pmatrix} 4 & 1 & 2 & 3 \\ 5 & 2 & 1 & 400 \\ 2 & 1 & 3 & 8 \\ 7 & 1 & 2 & 5 \end{pmatrix}$$

$m = 1$, $k = 8$ e o itinerário 0 3 1 3 3 2 1 0, o custo total desse itinerário é

$$a_{03} + a_{31} + a_{13} + a_{33} + a_{32} + a_{21} + a_{10} = 3 + 1 + 400 + 5 + 2 + 1 + 5 = 417\,.$$

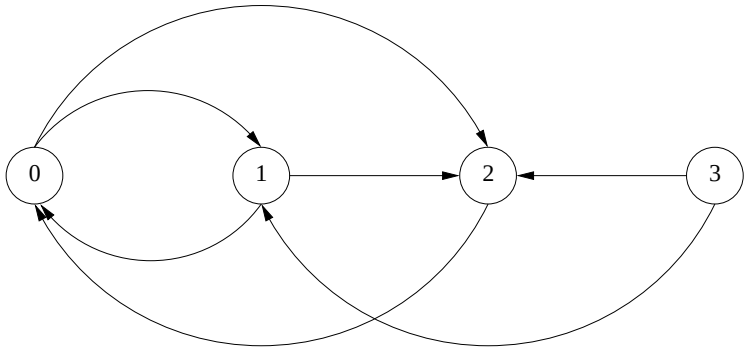
26.5 Dados um caça-palavra, representado por uma matriz $A$ de letras de dimensão $m \times n$, com $1 \leqslant m, n \leqslant 50$, e uma lista de $k > 0$ palavras, encontrar a localização (linha e coluna) no caça-palavras em que cada uma das palavras pode ser encontrada.

```
a c a t g u d k h
q y u s z j l l c
b m a m o r a w e
p f a x v n e t q
o c g j u a t d k
h j o a q y s c z   amora
z l c a d b m o r
o b g j h c t a w
t s y z l o k e u
q b r s o a e p h
r o d o m a y s l
m u l q c k a z g
```

Uma palavra é encontrada no caça-palavras se uma seqüência ininterrupta de letras na tabela coincide com a palavra. Considere que as letras são apenas as minúsculas. A busca por uma palavra deve ser feita em qualquer das oito direções no caça-palavras.

26.6 Considere $n$ cidades numeradas de 0 a $n-1$ que estão interligadas por uma série de estradas de mão única, com $1 \leqslant n \leqslant 100$. As ligações entre as cidades são representadas pelos elementos de uma matriz quadrada $A_{n \times n}$, cujos elementos $a_{ij}$ assumem o valor 1 ou 0, conforme exista ou não estrada direta que saia da cidade $i$ e chegue à cidade $j$. Assim, os elementos da coluna $j$ indicam as estradas que chegam à cidade $j$. Por convenção $a_{ii} = 1$.

A figura abaixo mostra um exemplo para $n = 4$.




$A$

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 1 |

(a) Dado $k$, determinar quantas estradas saem e quantas chegam à cidade $k$.

(b) A qual das cidades chega o maior número de estradas?

(c) Dado $k$, verificar se todas as ligações diretas entre a cidade $k$ e outras são de mão dupla.

(d) Relacionar as cidades que possuem saídas diretas para a cidade $k$.

(e) Relacionar, se existirem:

i. As cidades isoladas, isto é, as que não têm ligação com nenhuma outra;

ii. As cidades das quais não há saída, apesar de haver entrada;

iii. As cidades das quais há saída sem haver entrada.

(f) Dada uma seqüência de $m$ inteiros cujos valores estão entre 0 e $n-1$, verificar se é possível realizar o roteiro correspondente. No exemplo dado, o roteiro representado pela seqüência 2 0 1 2 3, com $m = 5$, é impossível.

(g) Dados $k$ e $p$, determinar se é possível ir da cidade $k$ para a cidade $p$ pelas estradas existentes. Você consegue encontrar o menor caminho entre as duas cidades?

AULA 27

# REGISTROS

Nas aulas 19 a 26 aprendemos a trabalhar com variáveis compostas homogêneas unidimensionais e bidimensionais – ou os vetores e as matrizes –, que permitem que um programador agrupe valores de um mesmo tipo em uma única entidade lógica. Como mencionamos antes, a dimensão de uma variável composta homogênea não fica necessariamente restrita a uma ou duas. Isto é, também podemos declarar e usar variáveis compostas homogêneas com três ou mais dimensões. Importante relembrar também que, para referenciar um elemento em uma variável composta homogênea são necessários o seu identificador e um ou mais índices. A linguagem C dispõe também de uma outra forma para agrupamento de dados, chamada variável composta heterogênea, registro ou estrutura[1]. Nelas, diferentemente do que nas variáveis compostas homogêneas, podemos armazenar sob uma mesma entidade lógica valores de tipos diferentes. Além disso, ao invés de índices ou endereços usados nas variáveis compostas homogêneas para acesso a um valor, especificamos o nome de um campo para selecionar um campo particular do registro. Os registros são os objetos de estudo desta aula.

## 27.1 Definição

Uma **variável composta heterogênea** ou **registro** é uma estrutura onde podemos armazenar valores de tipos diferentes sob uma mesma entidade lógica. Cada um desses possíveis valores é armazenado em um compartimento do registro denominado **campo do registro**, ou simplesmente **campo**. Um registro é composto pelo seu identificador e pelos seus campos. Suponha que queremos trabalhar com um agrupamento de valores que representam uma determinada mercadoria de uma loja, cujas informações relevantes são o código do produto e seu valor. Na linguagem C, podemos então declarar um registro com identificador `produto` da seguinte forma:

```
struct {
  int codigo;
  int quant;
  float valor;
} produto;
```

[1] *struct* e *member (of a structure)* são jargões comuns na linguagem C. A tradução literal dessas palavras pode nos confundir com outros termos já usados durante o curso. Por isso, escolhemos 'registro' e 'campo (do registro)' como traduções para o português.

A figura 27.1 mostra a disposição do registro `produto` e de seus campos na memória do computador.

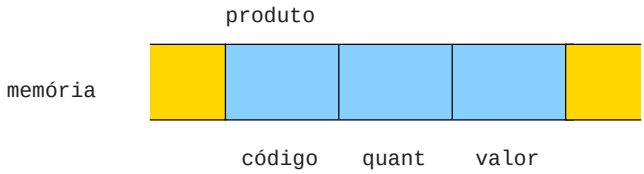



Figura 27.1: Representação do registro `produto` na memória.

A declaração de um registro sempre inicia com a palavra reservada `struct`. Em seguida, um bloco de declarações é iniciado com o símbolo `{`. Depois disso, podemos declarar os campos do registro, linha após linha. Cada linha deve conter o tipo e o identificador do campo do registro. Finalizamos então as declarações dos campos do registro com o símbolo `}` e, depois dele, realizamos de fato a declaração do registro, digitando o seu identificador. A variável do tipo registro com identificador `horario` declarada acima contém três campos, dois do tipo inteiro com identificadores `horas` e `quant`, e outro do tipo ponto flutuante `valor`.

Podemos declarar outras variáveis do tipo registro com os mesmos campos da variável `produto` da seguinte forma:

```
struct {
  int codigo;
  int quant;
  float valor;
} produto, estoque, baixa;
```

Na linguagem C, uma forma alternativa de declaração dos campos de um registro é declarar campos de um mesmo tipo em uma única linha de código. Por exemplo,

```
struct {
  char sala, turma;
  int horas_inicio, minutos_inicio, horas_fim, minutos_fim;
  float dimensoes_sala;
} aula;
```

declara um registro com identificador `aula` com seis campos, dois campos do tipo `char`, quatro outros campos do tipo `int` e um campo do tipo `float`. Apesar dessa declaração estar correta e de economizar algumas linhas de código, sempre optamos pela declaração dos campos separadamente, linha a linha, para que os campos fiquem bem destacados e, assim, facilitem sua identificação rápida no código. Dessa forma, em geral optamos pela seguinte declaração do registro `aula`:

```
struct {
  char sala;
  char turma;
  int horas_inicio;
  int minutos_inicio;
  int horas_fim;
  int minutos_fim;
  float dimensoes_sala;
} aula;
```

Diferentemente da atribuição de um valor a uma variável ou a um compartimento de uma variável composta homogênea, a atribuição de um valor a um campo de uma variável do tipo registro é realizada através do acesso a esse campo, especificando o identificador do registro, um ponto e o identificador do campo.

Por exemplo, para atribuir os valores $12, 5$ e $34.5$ aos campos `codigo`, `quant` e `valor`, respectivamente, da variável do tipo registro `produto` declarada anteriormente, devemos fazer como abaixo:

```
produto.codigo = 12;
produto.quant = 5;
produto.valor = 34.5;
```

Observe acima que, quando referenciamos um campo de uma variável do tipo registro, não são permitidos espaços entre o identificador do registro, o ponto e o identificador do campo. Também podemos usar o valor de um campo de um registro em quaisquer expressões. Por exemplo, a expressão relacional abaixo é correta:

```
if (produto.valor > 150.0)
  printf("Acima do preço de mercado!\n");
```

Declarações de registros diferentes podem conter campos com mesmo identificador. Por exemplo,

```
struct {
  char tipo;
  char fatorRH;
  int idade;
  float altura;
} coleta;
```

```
struct {
  char codigo;
  int tipo;
  int idade;
} certidao;
```

são declarações válidas na linguagem C. O acesso aos campos dessas variáveis se diferencia justamente pelo identificador dos registros. Isto é, a partir das declarações dos registros `coleta` e `certidao`, as atribuições abaixo estão corretas:

```
coleta.tipo = 'O';
certidao.tipo = 0;
coleta.idade = 29;
certidao.idade = coleta.idade + 2;
```

## 27.2 Declaração e inicialização simultâneas

Declaração e inicialização simultâneas também são válidas quando tratamos com registros. As regras são idênticas às declarações e inicializações simultâneas para os vetores. Por exemplo, o registro `produto` que declaramos acima pode ter sua inicialização realizada no momento de sua declaração da seguinte forma:

```
struct {
  int codigo;
  int quant;
  float valor;
} produto = {1, 5, 34.5};
```

Neste caso, o campo `codigo` é inicializado com o valor `1`, o campo `quant` é inicializado com o valor `5` e o campo `valor` com `34.5`. Como mencionado, as regras de declaração e inicialização simultâneas para registros são equivalentes àquelas para vetores. Assim,

```
struct {
  int codigo;
  int quant;
  float valor;
} produto = {0};
```

fará a inicialização dos campos `codigo` e `quant` com `0` e do campo `valor` com `0.0`.

## 27.3 Operações sobre registros

Variáveis compostas heterogêneas possuem uma característica importante, que não é observada em variáveis compostas homogêneas.

Se queremos copiar o conteúdo de uma variável composta homogênea para outra variável composta homogênea, é necessário realizar a cópia elemento a elemento, escrevendo uma ou mais estruturas de repetição para que essa cópia seja efetuada com sucesso. Isto é, se `A` e `B` são, por exemplo, vetores de mesmo tipo de dados e mesma dimensão, é errado tentar fazer uma atribuição como abaixo:

```
A = B;
```

para tentar copiar os valores do vetor `B` no vetor `A`.

O correto, neste caso, é fazer a cópia elemento a elemento de uma variável para outra, como mostrado a seguir:

```
for (i = 0; i < n; i++)
  A[i] = B[i];
```

supondo que `n` é a dimensão dos vetores `A` e `B`.

Quando tratamos de registros, no entanto, podemos fazer uma atribuição direta e realizar a cópia de todos os seus campos nessa única atribuição. O trecho de código a seguir mostra um exemplo com a declaração de dois registros com mesmos campos, a atribuição de valores ao primeiro registro e uma cópia completa de todos os campos do primeiro registro para o segundo:

```
struct {
  char tipo;
  int codigo;
  int quant;
  float valor;
} mercadoria1, mercadoria2;
mercadoria1.tipo = 'A';
mercadoria1.codigo = 10029;
mercadoria1.quant = 62;
mercatoria1.valor = 10.32 * TAXA + 0.53;
mercadoria2 = mercadoria1;
```

## 27.4 Exemplo

O programa 27.1 recebe um horário no formato de horas, minutos e segundos, com as horas no intervalo de 0 a 23, e atualiza este horário em um segundo.

Programa 27.1: Exemplo do uso de registros.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    struct {
5      int horas;
6      int minutos;
7      int segundos;
8    } agora, prox;
9    printf("Informe o horário atual (hh:mm:ss): ");
10   scanf("%d:%d:%d", &agora.horas, &agora.minutos, &agora.segundos);
11   prox = agora;
12   prox.segundos++;
13   if (prox.segundos == 60) {
14     prox.segundos = 0;
15     prox.minutos++;
16     if (prox.minutos == 60) {
17       prox.minutos = 0;
18       prox.horas++;
19       if (prox.horas == 24)
20         prox.horas = 0;
21     }
22   }
23   printf("Próximo horário é %02d:%02d:%02d\n",
24          prox.horas, prox.minutos, prox.segundos);
25   return 0;
26 }
```

## Exercícios

27.1 Dada uma data, escreva um programa que mostre a próxima data, isto é, a data que representa o dia seguinte à data fornecida. Não esqueça dos anos bissextos. Lembre-se que um ano é bissexto se é divisível por 400 ou, em caso negativo, se é divisível por 4 mas não por 100.

27.2 Dados dois horários de um mesmo dia expressos no formato `hh:mm:ss`, calcule o tempo decorrido entre estes dois horários, apresentando o resultado no mesmo formato.

27.3 Dadas duas datas, calcule o número de dias decorridos entre estas duas datas.

Programa 27.2: Solução do exercício 27.1.

```
#include <stdio.h>
int main(void)
{
  struct {
    int dia;
    int mes;
    int ano;
  } data, prox;
  scanf("%d/%d/%d", &data.dia, &data.mes, &data.ano);
  prox = data;
  prox.dia++;
  if (prox.dia > 31 ||
      (prox.dia == 31 && (prox.mes == 4 || prox.mes == 6 ||
                          prox.mes == 9 || prox.mes == 11)) ||
      (prox.dia >= 30 && prox.mes == 2) ||
      (prox.dia == 29 && prox.mes == 2 && (prox.ano % 400 != 0 &&
                                           prox.ano % 4 != 0))    ) {
    prox.dia = 1;
    prox.mes++;
    if (prox.mes > 12) {
      prox.mes = 1;
      prox.ano++;
    }
  }
  printf("%02d/%02d/%02d\n", prox.dia, prox.mes, prox.ano);
  return 0;
}
```

Uma maneira provavelmente mais simples de computar essa diferença é usar a fórmula 27.1 para calcular um número de dias $N$ baseado em uma data:

$$N = \left\lfloor \frac{1461 \times f(\text{ano}, \text{mês})}{4} \right\rfloor + \left\lfloor \frac{153 \times g(\text{mês})}{5} \right\rfloor + \text{dia} \tag{27.1}$$

onde

$$f(\text{ano}, \text{mês}) = \begin{cases} \text{ano} - 1, & \text{se mês } \leqslant 2\,, \\ \text{ano}, & \text{caso contrário} \end{cases}$$

e

$$g(\text{mês}) = \begin{cases} \text{mês} + 13, & \text{se mês } \leqslant 2\,, \\ \text{mês} + 1, & \text{caso contrário}\,. \end{cases}$$

Podemos calcular o valor $N_1$ para a primeira data informada, o valor $N_2$ para a segunda data informada e a diferença $N_2 - N_1$ é o número de dias decorridos entre estas duas datas informadas.

27.4 Seja $N$ computado como na equação 27.1. Então, o valor

$$D = (N - 621049) \bmod 7$$

é um número entre 0 e 6 que representa os dias da semana, de domingo a sábado. Por exemplo, para a data de 21/06/2007 temos

$$\begin{aligned} N &= \left\lfloor \frac{1461 \times f(2007, 6)}{4} \right\rfloor + \left\lfloor \frac{153 \times g(6)}{5} \right\rfloor + 21 \\ &= \left\lfloor \frac{1461 \times 2007}{4} \right\rfloor + \left\lfloor \frac{153 \times 7}{5} \right\rfloor + 21 \\ &= 733056 + 214 + 21 \\ &= 733291 \end{aligned}$$

e então

$$\begin{aligned} D &= (733291 - 621049) \bmod 7 \\ &= 112242 \bmod 7 \\ &= 4\,. \end{aligned}$$

Dada uma data fornecida pelo usuário no formato `dd/mm/aaaa`, determine o dia da semana para esta data.

AULA 28

# VETORES, MATRIZES E REGISTROS

Nesta aula, vamos trabalhar com uma extensão natural do uso de registros, declarando e usando variáveis compostas homogêneas de registros como, por exemplo, vetores de registros ou matrizes de registros. Por outro lado, estudaremos também registros contendo variáveis compostas homogêneas como campos. Veremos que combinações dessas declarações também podem ser usadas de modo a representar e organizar os dados em memória e solucionar problemas.

## 28.1 Vetores de registros

Como vimos nas aulas 19 e 24, podemos declarar variáveis compostas homogêneas a partir de qualquer tipo básico ou ainda de um tipo definido pelo(a) programador(a). Ou seja, podemos declarar, por exemplo, um vetor do tipo inteiro, uma matriz do tipo ponto flutuante, uma variável composta homogênea de $k$ dimensões do tipo caracter e etc. A partir de agora, poderemos também declarar variáveis compostas homogêneas, de quaisquer dimensões, de registros. Por exemplo, podemos declarar um vetor com identificador `cronometro` como mostrado a seguir:

```
struct {
  int horas;
  int minutos;
  int segundos;
} cronometro[10];
```

ou ainda declarar uma matriz com identificador `agenda` como abaixo:

```
struct {
  int horas;
  int minutos;
  int segundos;
} agenda[10][30];
```

O vetor `cronometro` declarado anteriormente contém 10 compartimentos de memória, sendo que cada um deles é do tipo registro. Cada registro contém, por sua vez, os campos `horas`, `minutos` e `segundos`, do tipo inteiro. Já a matriz com identificador `agenda` é uma matriz contendo 10 linhas e 30 colunas, onde cada compartimento contém um registro com os mesmos campos do tipo inteiro `horas`, `minutos` e `segundos`. Essas duas variáveis compostas homogêneas também poderiam ter sido declaradas em conjunto, como apresentado a seguir:

```
struct {
  int horas;
  int minutos;
  int segundos;
} cronometro[10], agenda[10][30];
```

A declaração do vetor `cronometro` tem um efeito na memória que pode ser ilustrado como na figura 28.1.

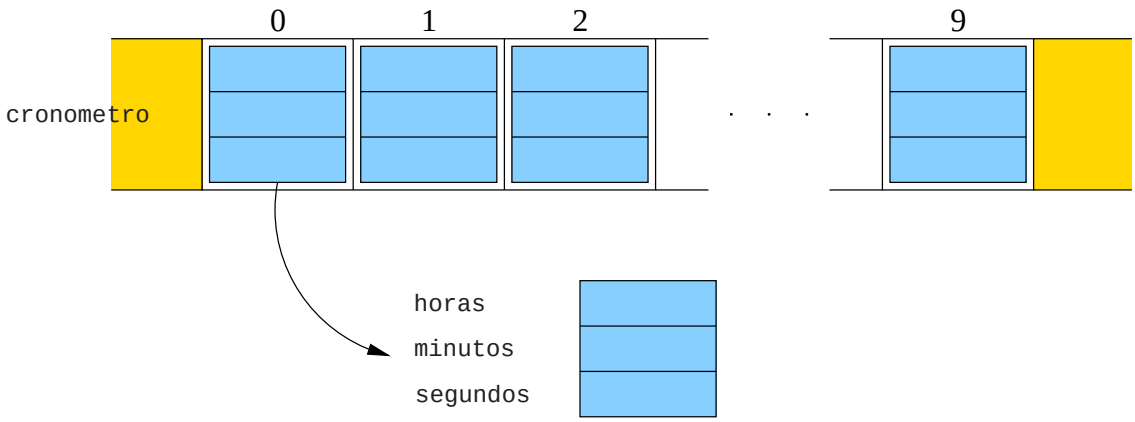



Figura 28.1: Efeito da declaração do vetor `cronometro` na memória.

Atribuições de valores do tipo inteiro aos campos do registro do primeiro compartimento deste vetor `cronometro` podem ser feitas da seguinte forma:

```
cronometro[0].horas = 20;
cronometro[0].minutos = 39;
cronometro[0].segundos = 18;
```

Além disso, como já mencionamos na aula 27, podemos fazer a atribuição direta de registros para registros. Assim, por exemplo, se declaramos as variáveis `cronometro` e `aux` como abaixo:

```
struct {
  int horas;
  int minutos;
  int segundos;
} cronometro[10], aux;
```

então as atribuições a seguir são válidas e realizam a troca dos conteúdos das posições `i` e `j` do vetor de registros `cronometro`:

```
aux = cronometro[i];
cronometro[i] = cronometro[j];
cronometro[j] = aux;
```

É importante observar novamente que todos os campos de um registro são atualizados automaticamente quando da atribuição de um registro a outro registro, não havendo a necessidade da atualização campo a campo. Ou seja, podemos fazer:

```
cronometro[i] = cronometro[j];
```

ao invés de

```
cronometro[i].horas = cronometro[j].horas;
cronometro[i].minutos = cronometro[j].minutos;
cronometro[i].segundos = cronometro[j].segundos;
```

As duas formas acima estão corretas, apesar da primeira forma ser mais prática e direta.

Nesse contexto, considere o seguinte problema:

> Dado um número inteiro $n$, com $1 \leqslant n \leqslant 100$, e $n$ medidas de tempo dadas em horas, minutos e segundos, distintas duas a duas, ordenar essas medidas de tempo em ordem crescente.

O programa 28.1 soluciona o problema acima usando o método de ordenação das trocas sucessivas ou método da bolha.

Programa 28.1: Um programa usando um vetor de registros.

```
#include <stdio.h>
int main(void)
{
  int i, j, n;
  struct {
    int horas;
    int minutos;
    int segundos;
  } cron[100], aux;
  printf("Informe a quantidade de medidas de tempo: ");
  scanf("%d", &n);
  printf("\n");
  for (i = 0; i < n; i++) {
    printf("Informe uma medida de tempo (hh:mm:ss): ");
    scanf("%d:%d:%d", &cron[i].horas, &cron[i].minutos, &cron[i].segundos);
  }
  for (i = n-1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (cron[j].horas > cron[j+1].horas) {
        aux = cron[j];
        cron[j] = cron[j+1];
        cronometro[j+1] = aux;
      }
      else if (cron[j].horas == cron[j+1].horas)
        if (cron[j].minutos > cron[j+1].minutos) {
          aux = cron[j];
          cron[j] = cron[j+1];
          cron[j+1] = aux;
        }
        else if (cron[j].minutos == cron[j+1].minutos)
          if (cron[j].segundos > cron[j+1].segundos) {
            aux = cron[j];
            cron[j] = cron[j+1];
            cron[j+1] = aux;
          }
  printf("\nHorários em ordem crescente\n");
  for (i = 0; i < n; i++)
    printf("%d:%d:%d\n", cron[i].horas, cron[i].minutos, cron[i].segundos);
  return 0;
}
```

## 28.2 Registros contendo variáveis compostas

Na aula 27 definimos registros que continham campos de tipos básicos ou de tipos definidos pelo(a) programador(a). Na seção 28.1, estudamos vetores não mais de um tipo básico, mas de registros, isto é, vetores contendo registros. Podemos também declarar registros que contêm variáveis compostas como campos. Um exemplo bastante comum é a declaração de um vetor de caracteres, ou uma cadeia de caracteres, dentro de um registro:

```
struct {
  int dias;
  char nome[3];
} mes;
```

A declaração do registro `mes` permite o armazenamento de um valor do tipo inteiro no campo `dias`, que pode representar, por exemplo, a quantidade de dias de um mês, e de três valores do tipo caracter no campo vetor `nome`, que podem representar os três primeiros caracteres do nome de um mês do ano. Assim, se declaramos as variáveis `mes` e `aux` como a seguir:

```
struct {
  int dias,
  char nome[3];
} mes, aux;
```

podemos fazer a seguinte atribuição válida ao registro `mes`:

```
mes.dias = 31;
mes.nome[0] = 'J';
mes.nome[1] = 'a';
mes.nome[2] = 'n';
```

Os efeitos da declaração da variável `mes` na memória e da atribuição de valores acima podem ser vistos na figura 28.2.

É importante salientar mais uma vez que as regras para cópias de registros permanecem as mesmas, mesmo que um campo de um desses registros seja uma variável composta. Assim, a cópia abaixo é perfeitamente válida:

```
aux = mes;
```

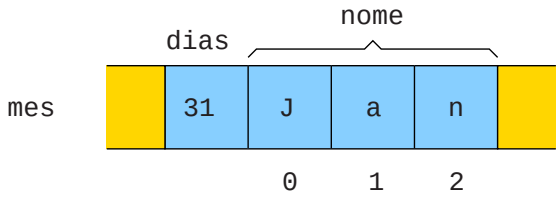



Figura 28.2: Variável `mes` na memória.

Suponha agora que temos o seguinte problema.

> Dadas duas descrições de tarefas e seus horários de início no formato `hh:mm:ss`, escreva um programa que verifica qual das duas tarefas será iniciada antes. Considere que a descrição de uma tarefa tenha no máximo 50 caracteres.

Uma solução para esse problema é apresentada no programa 28.2.

Programa 28.2: Exemplo do uso de um vetor como campo de um registro.

```
#include <stdio.h>
int main(void)
{
  int tempo1, tempo2;
  struct {
    int horas;
    int minutos;
    int segundos;
    char descricao[51];
  } t1, t2;
  printf("Informe a descrição da primeira atividade: ");
  scanf("%[^\n]", t1.descricao);
  printf("Informe o horário de início dessa atividade (hh:mm:ss): ");
  scanf("%d:%d:%d", &t1.horas, &t1.minutos, &t1.segundos);
  printf("Informe a descrição da segunda atividade: ");
  scanf(" %[^\n]", t2.descricao);
  printf("Informe o horário de início dessa atividade (hh:mm:ss): ");
  scanf("%d:%d:%d", &t2.horas, &t2.minutos, &t2.segundos);
  tempo1 = t1.horas * 3600 + t1.minutos * 60 + t1.segundos;
  tempo2 = t2.horas * 3600 + t2.minutos * 60 + t2.segundos;
  if (tempo1 <= tempo2)
    printf("%s será realizada antes de %s\n", t1.descricao, t2.descricao);
  else
    printf("%s será realizada antes de %s\n", t2.descricao, t1.descricao);
  return 0;
}
```

## Exercícios

28.1 Dadas $n$ datas, com $1 \leqslant n \leqslant 100$, e uma data de referência $d$, verifique qual das $n$ datas é mais próxima à data $d$.

Podemos usar a fórmula do exercício 27.3 para solucionar esse exercício mais facilmente.

28.2 Dadas três fichas de produtos de um supermercado, contendo as informações de seu código, sua descrição com até 50 caracteres e seu preço unitário, ordená-las em ordem alfabética de seus nomes.

28.3 Dados um número inteiro $n > 1$ e mais $n$ fichas de doadores de um banco de sangue, contendo o código do doador, seu nome, seu tipo sangüíneo e seu fator Rhesus, escreva um programa que lista os doadores do banco das seguintes formas: (*i*) em ordem crescente de códigos de doadores; (*ii*) em ordem alfabética de nomes de doadores e (*iii*) em ordem alfabética de tipos sangüíneos e fatores Rhesus.

Programa 28.3: Solução do exercício 28.1.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  int dif[MAX], menor;
  struct {
    int dia;
    int mes;
    int ano;
  } d, data[MAX];
  printf("Informe uma data de referência (dd/mm/aa): ");
  scanf("%d/%d/%d", &d.dia, &d.mes, &d.ano);
  scanf("%d", &n);
  printf("Informe a quantidade de datas: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("[%03d] Informe uma data (dd/mm/aa): ", i+1);
    scanf("%d/%d/%d", &data[i].dia, &data[i].mes, &data[i].ano);
  }
  if (d.mes <= 2)
    N1 = (1461*(d.ano-1))/4 + ((153*d.mes+13)/5) + d.dia;
  else
    N1 = (1461*d.ano)/4 + ((153*d.mes+1)/5) + d.dia;
  for (i = 0; i < n; i++) {
    if (data[i].mes <= 2)
      N2 = (1461*(data[i].ano-1))/4 + ((153*data[i].mes+13)/5) + data[i].dia;
    else
      N2 = (1461*data[i].ano)/4 + ((153*data[i].mes+1)/5) + data[i].dia;
    if (N1 >= N2)
      dif[i] = N1 - N2;
    else
      dif[i] = N2 - N1;
  }
  menor = 0;
  for (i = 1; i < n; i++)
    if (dif[i] < dif[menor])
      menor = i;
  printf("Data mais próxima de %2d/%2d/%2d é %2d/%2d/%d\n",
     d.dia, d.mes, d.ano, data[menor].dia, data[menor].mes, data[menor].ano);
  return 0;
}
```

AULA 29

# REGISTROS COM REGISTROS

Assim como na aula 28, a aula de hoje também trata da declaração de registros mais complexos, mas agora seus campos podem ser variáveis não somente de tipos básicos, de tipos definidos pelo usuário, ou ainda variáveis compostas homogêneas, mas também de variáveis compostas heterogêneas ou registros.

Uma variável que é um registro e que, por sua vez, contém como campos um ou mais registros, fornece ao(à) programador(a) uma liberdade e flexibilidade na declaração de qualquer estrutura para armazenamento de informações que lhe seja necessária na solução de um problema computacional, especialmente daqueles problemas mais complexos.

## 29.1 Registros contendo registros

É importante observar que a declaração de um registro pode conter um outro registro como um campo, em seu interior. Ou seja, uma variável composta heterogênea pode conter campos de tipos básicos, iguais ou distintos, campos de tipos definidos pelo usuário, campos que são variáveis compostas homogêneas, ou ainda campos que se constituem também como variáveis compostas heterogêneas.

Como um exemplo, podemos declarar uma variável do tipo registro com identificador `estudante` contendo um campo do tipo inteiro `rga`, um campo do tipo vetor de caracteres `nome`, com 51 posições e um campo do tipo registro `nascimento`, contendo por sua vez três campos do tipo inteiro com identificadores `dia`, `mes` e `ano`. Ou seja, o registro `estudante` pode ser declarado como a seguir:

```
struct {
  int rga;
  char nome[51];
  struct {
    int dia;
    int mes;
    int ano;
  } nascimento;
} estudante;
```

A figura 29.1 mostra o efeito da declaração da variável `estudante` na memória.

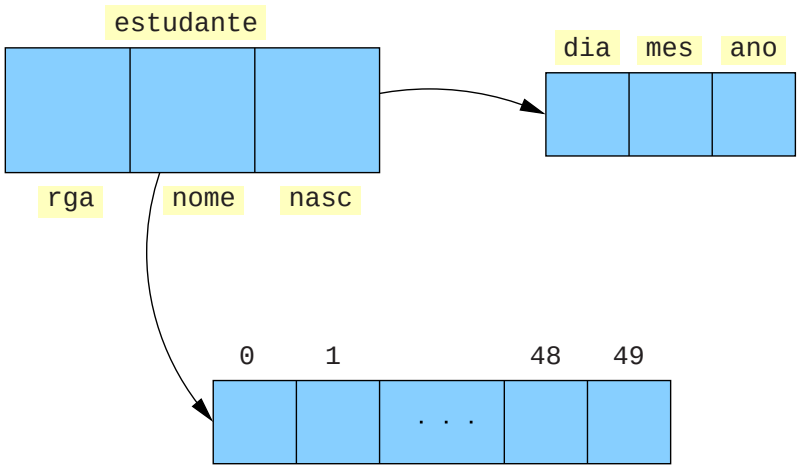



Figura 29.1: Efeitos da declaração do registro `estudante` na memória.

Observe que a variável `estudante` é um registro que mistura campos de um tipo básico – o campo `rga` que é do tipo inteiro –, um campo que é um vetor de um tipo básico – o campo `nome` do tipo vetor de caracteres – e um campo do tipo registro – o campo `nascimento` do tipo registro, contendo, por sua vez, três campos do tipo inteiro: os campos `dia`, `mes` e `ano`. Um exemplo do uso do registro `estudante` e de seus campos é dado a seguir através de atribuições de valores a cada um de seus campos:

```
estudante.rga = 200790111;
estudante.nome[0] = 'J';
estudante.nome[1] = 'o';
estudante.nome[2] = 's';
estudante.nome[3] = 'e';
estudante.nome[4] = '\0';
estudante.nasc.dia = 22;
estudante.nasc.mes = 2;
estudante.nasc.ano = 1988;
```

## 29.2 Exemplo

O programa 29.1 implementa uma agenda de telefones simples, ilustrando um exemplo de uso de um vetor que contém registros, onde cada registro contém também um registro como campo. A agenda contém três informações para cada amigo(a) incluído(a): nome, telefone e data de aniversário. Além de armazenar essa agenda na memória, em um vetor de registros, o programa ainda faz com que uma data seja fornecida pelo(a) usuário(a) e todas as pessoas que fazem aniversário nessa data são mostradas na saída.

Podemos destacar novamente, assim como fizemos nas aulas 27 e 28, que registros podem ser atribuídos automaticamente para registros, não havendo necessidade de fazê-los campo a campo. Por exemplo, se temos declarados os registros:

Programa 29.1: Um exemplo de uso de registros contendo registros.

```
#include <stdio.h>
int main(void)
{
  int i, n;
  struct {
    char nome[51];
    int telefone;
    struct {
      int dia;
      int mes;
      int ano;
    } aniver;
  } agenda[100];
  struct {
    int dia;
    int mes;
  } data;
  printf("Informe a quantidade de amigos: ");
  scanf("%d", &n);
  for(i = 0; i < n; i++) {
    printf("\nAmigo(a): %3d\n", i+1);
    printf("  Nome       : ");
    scanf(" %[^\n]", agenda[i].nome);
    printf("  Telefone   : ");
    scanf("%d", &agenda[i].telefone);
    printf("  Aniversário: ");
    scanf("%d/%d/%d", &agenda[i].aniver.dia, &agenda[i].aniver.mes,
                      &agenda[i].aniver.ano);
  }
  printf("\nInforme uma data (dd/mm): ");
  scanf("%d/%d", &data.dia, &data.mes);
  i = 0;
  while (i < n) {
    if (agenda[i].aniver.dia == data.dia && agenda[i].aniver.mes == data.mes)
      printf("%-50s  %8d\n", agenda[i].nome, agenda[i].telefone);
    i++;
  }
  return 0;
}
```

```
struct {
  int rga;
  char nome[51];
  struct {
    int dia;
    int mes;
    int ano;
  } nascimento;
} estudante1, estudante2, aux;
```

então, as atribuições a seguir são perfeitamente válidas e realizam corretamente a troca de conteúdos dos registros `estudante1` e `estudante2`:

```
aux = estudante1;
estudante1 = estudante2;
estudante2 = aux;
```

## Exercícios

29.1 Suponha que em um determinado galpão estejam armazenados os materiais de construção de uma loja que vende tais materiais. Este galpão é quadrado e mede $20 \times 20 = 400\text{m}^2$ e a cada $2 \times 2 = 4\text{m}^2$ há uma certa quantidade de um material armazenado. O encarregado do setor tem uma tabela de 10 linhas por 10 colunas, representando o galpão, contendo, em cada célula, o código do material, sua descrição e sua quantidade. O código do material é um número inteiro, a descrição o material contém no máximo 20 caracteres e a quantidade do material é um número de ponto flutuante.

Escreva um programa que receba as informações armazenadas na tabela do encarrerado e liste cada material e a sua quantidade disponível no galpão. Observe que um mesmo material pode encontrar-se em mais que um local no galpão.

29.2 Escreva um programa que receba o nome, o telefone e a data de nascimento de $n$ pessoas, com $1 \leqslant n \leqslant 100$, e implemente uma agenda telefônica com duas listagens possíveis: (*i*) uma lista dos nomes e telefones das pessoas em ordem alfabética de nomes e (*ii*) uma lista dos nomes e telefones das pessoas em ordem de datas de aniversários das pessoas.

Programa 29.2: Solução do exercício 29.1.

```
#include <stdio.h>
#define MAX 10
int main(void)
{
  int i, j, k, l, m;
  float soma;
  struct {
    int codigo;
    char descricao[21];
    float quant;
    int marca;
  } galpao[MAX][MAX], resumo[MAX*MAX];
  for(i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++) {
      scanf("%d %[\^n]%f", &galpao[i][j].codigo, galpao[i][j].descricao,
                           &galpao[i][j].quant);
      galpao[i][j].marca = 0;
    }
  for (m = 0, i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
      if (!galpao[i][j].marca) {
        if (j < MAX - 1) {
          k = i;
          l = j + 1;
        }
        else {
          k = i + 1;
          l = 0;
        }
        soma = galpao[i][j].quant;
        for ( ; k < MAX; k++, l = 0)
          for ( ; l < MAX; l++)
            if (galpao[k][l].codigo == galpao[i][j].codigo) {
              soma = soma + galpao[k][l].quant;
              galpao[k][l].marca = 1;
            }
        resumo[m] = galpao[i][j];
        resumo[m].quant = soma;
        m++;
        galpao[i][j].marca = 1;
      }
  for (i = 0; i < m; i++)
    printf("%d %s %5.2f\n", resumo[i].codigo, resumo[i].descricao,
                            resumo[i].quant);
  return 0;
}
```

AULA 30

# UNIÕES E ENUMERAÇÕES

Complementando o conteúdo apresentado até aqui sobre variáveis compostas, a linguagem C ainda oferece dois tipos delas: as uniões e as enumerações. Uma união é um registro que armazena apenas um dos valores de seus campos e, portanto, apresenta economia de espaço. Uma enumeração ajuda o(a) programador(a) a declarar e usar uma variável que representa intuitivamente uma seqüência de números naturais.

## 30.1 Uniões

Como os registros, as uniões também são compostas por um ou mais campos, possivelmente de tipos diferentes. No entanto, o compilador aloca espaço suficiente apenas para o maior dos campos de uma união. Esses campos compartilham então o mesmo espaço na memória. Dessa forma, a atribuição de um valor a um dos campos da união também altera os valores dos outros campos.

Vamos declarar a união `u` com dois campos, um do tipo caractere com sinal e outro do tipo inteiro com sinal, como abaixo:

```
union {
  char c;
  int i;
} u;
```

Observe que a declaração de uma união é idêntica à declaração de um registro que aprendemos nas aulas 27, 28 e 29, a menos da palavra reservada `union` que substitui a palavra reservada `struct`. Por exemplo, podemos declarar o registro `r` com os mesmos dois campos da união `u` como a seguir:

```
struct {
  char c;
  int i;
} r;
```

A diferença entre a união `u` e o registro `r` concentra-se apenas no fato que os campos de `u` estão localizados no mesmo endereço de memória enquanto que os campos de `r` estão localizados em endereços diferentes. Considerando que um caractere ocupa um byte na memória e um número inteiro ocupa quatro bytes, a figura 30.1 apresenta uma ilustração do que ocorre na memória na declaração das variáveis `u` e `r`.

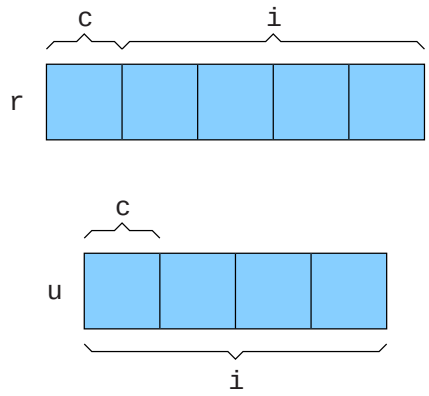



Figura 30.1: União e registro representados na memória.

No registro `r`, os campos `c` e `i` ocupam diferentes posições da memória. Assim, o total de memória necessário para armazenamento do registro `r` na memória é de cinco bytes. Na união `u`, os campos `c` e `i` compartilham a mesma posição da memória – isto é, os campos `c` e `i` de `u` têm o mesmo endereço de memória – e, portanto, o total de memória necessário para armazenamento de `u` é de quatro bytes.

Campos de uma união são acessados da mesma forma que os campos de um registro. Por exemplo, podemos fazer a atribuição a seguir:

```
u.c = 'a';
```

Também podemos fazer a atribuição de um número inteiro para o outro campo da união como apresentado abaixo:

```
u.i = 3894;
```

Como o compilador superpõe valores nos campos de uma união, a alteração de um dos campos implica na alteração de qualquer valor previamente armazenado nos outros campos. Assim, se armazenamos um valor no campo `u.c`, qualquer valor previamente armazenado no campo `u.i` será perdido. Do mesmo modo, alterar o valor do campo `u.i` altera o valor do campo `u.c`. Assim, podemos pensar na união `u` como um local na memória para armazenar um caractere em `u.c` OU ou um número em `u.i`, mas não ambos. O registro `r` pode armazenar valores em `r.c` E em `r.i`.

Como com registros, uniões podem ser copiadas diretamente usando o comando de atribuição `=`, sem a necessidade de fazê-lo campo a campo. Declarações e inicializações também são feitas similarmente. No entanto, somente o primeiro campo de uma união pode ter um valor atribuído no inicializador. Por exemplo, podemos fazer

```
union {
  char c;
  int i;
} u = {'\0'};
```

Usamos uniões freqüentemente como uma forma para economizar espaço em registros. Também, usamos uniões quando queremos criar estruturas de armazenamento de dados que contenham uma mistura de diferentes tipos de dados.

Como um exemplo do segundo caso, suponha que temos uma coleção de números inteiros e de ponto flutuante que queremos armazenar em um único vetor. Como os elementos de um vetor têm de ser do mesmo tipo, devemos usar uniões para implementar um vetor com essa característica. A declaração de um vetor como esse é dada a seguir:

```
union {
  int i;
  double d;
} vetor[100];
```

Cada compartimento do `vetor` pode armazenar um valor do tipo `int` ou um valor do tipo `double`, o que possibilita armazenar uma mistura de valores do tipo `int` e `double` no `vetor`. Por exemplo, para as atribuições abaixo atribuem números inteiros e números de ponto flutuante para as posições 0 e 1 do `vetor`:

```
vetor[0].i = 5;
vetor[1].d = 7.647;
```

Suponha que temos então o seguinte problema que precisamos construir um programa para solucioná-lo:

> Dada uma seqüência $n$ de números, inteiros e reais, com $1 \leqslant n \leqslant 100$, realizar a adição de todos os números inteiros e o produto de todos os números reais.

O programa 30.1 mostra um exemplo do uso de um vetor de registros onde um de seus campos é uma união.

Programa 30.1: Um exemplo de uso de registros e uniões.

```
#include <stdio.h>
#define TIPO_INT 0
#define TIPO_DOUBLE 1
#define MAX 100
int main(void)
{
  int i, n, soma, inteiro, real;
  double produto;
  struct {
    int tipo;
    union {
      int i;
      double d;
    } u;
  } numero[MAX];
  printf("Informe n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Informe o tipo do número (0: inteiro, 1: real): ");
    scanf("%d", &numero[i].tipo);
    printf("Informe o número: ");
    if (numero[i].tipo == TIPO_INT)
      scanf("%d", &numero[i].u.i);
    else
      scanf("%lf", &numero[i].u.d);
  }
  soma = 0;
  produto = 1.0;
  inteiro = 0;
  real = 0;
  for (i = 0; i < n; i++)
    if (numero[i].tipo == TIPO_INT) {
      soma = soma + numero[i].u.i;
      inteiro = 1;
    }
    else {
      produto = produto * numero[i].u.d;
      real = 1;
    }
  if (inteiro)
    printf("Soma dos números inteiros: %d\n", soma);
  else
    printf("Não há números inteiros na entrada\n");
  if (real)
    printf("Produto dos números reais: %g\n", produto);
  else
    printf("Não há números reais na entrada\n");
  return 0;
}
```

## 30.2 Enumerações

Muitas vezes precisamos de variáveis que conterão, durante a execução de um programa, somente um pequeno conjunto de valores. Por exemplo, variáveis lógicas ou booleanas deverão conter somente dois valores possíveis: "verdadeiro" e "falso". Uma variável que armazena os naipes das cartas de um baralho deverá conter apenas quatro valores potenciais: "paus", "copas", "espadas" e "ouros". Uma maneira natural de tratar esses valores é através da declaração de uma variável do tipo inteiro que pode armazenar valores que representam esses naipes. Por exemplo, 0 representa o naipe "paus", 1 o naipe "copas" e assim por diante. Assim, podemos por exemplo declarar uma variável naipe e atribuir valores a essa variável da seguinte forma:

```
int n;
n = 0;
```

Apesar dessa técnica funcionar, alguém que precisa compreender esse programa pode não ser capaz de saber que a variável `n` pode assumir apenas quatro valores durante a execução do programa e o significado do valor 0 não é imediatamente aparente.

Uma estratégia melhor que a anterior é o uso de macros para definir um "tipo" naipe e também para definir nomes para os diversos naipes existentes. Então, podemos fazer:

```
#define NAIPE   int
#define PAUS    0
#define COPAS   1
#define ESPADAS 2
#define OUROS   3
```

O exemplo anterior fica então bem mais fácil de ler:

```
NAIPE n;
n = PAUS;
```

Apesar de melhor, essa estratégia ainda possui restrições. Por exemplo, alguém que necessita compreender o programa não tem a visão imediata que as macros representam valores do mesmo "tipo". Além disso, se o número de valores possíveis é um pouco maior, a definição de uma macro para cada valor pode se tornar uma tarefa tediosa. Por fim, os identificadores das macros serão removidos pelo pré-processador e substituídos pelos seus valores respectivos e isso significa que não estarão disponíveis para a fase de depuração do programa. Depuração é uma atividade muito útil para programação a medida que nossos programas ficam maiores e mais complexos, como veremos na aula 31.

A linguagem C possui um tipo especial específico para variáveis que armazenam um conjunto pequeno de possíveis valores. Um **tipo enumerado** é um tipo cujos valores são listados ou enumerados pelo(a) programador(a) que deve criar um nome, chamado de uma **constante da enumeração**, para cada um dos valores. O exemplo a seguir enumera os valores `PAUS`, `COPAS`, `ESPADAS` e `OUROS` que podem ser atribuídos às variáveis `n1` e `n2`:

```
enum {PAUS, COPAS, ESPADAS, OUROS} n1, n2;
n1 = PAUS;
n2 = n1;
```

Apesar de terem pouco em comum com registros e uniões, as enumerações são declaradas de modo semelhante. Entretanto, diferentemente dos campos dos registros e uniões, os nomes das constantes da enumeração devem ser diferentes de outros nomes de outras enumerações declaradas.

A linguagem C trata variáveis que são enumerações e constantes da enumeração como números inteiros. Por padrão, o compilador atribui os inteiros $0, 1, 2, \ldots$ para as constantes da enumeração na declaração de uma variável qualquer do tipo enumeração. No exemplo acima, `PAUS`, `COPAS`, `ESPADAS` e `OUROS` representam os valores 0, 1, 2 e 3, respectivamente.

Podemos escolher valores diferentes dos acima para as constantes da enumeração. Por exemplo, podemos fazer a seguinte declaração:

```
enum {PAUS = 1, COPAS = 2, ESPADAS = 3, OUROS = 4} n;
```

Os valores das constantes da enumeração podem ser valores inteiros arbitrários, listados sem um ordem particular, como por exemplo:

```
enum {DCT = 17, DCH = 2, DEC = 21, DMT = 6} depto;
```

Se nenhum valor é especificado para uma constante da enumeração, o valor atribuído à constante é um maior que o valor da constante anterior. A primeira constante da enumeração tem o valor padrão 0 (zero). Na enumeração a seguir, `PRETO` tem o valor 0, `CINZA_CLARO` tem o valor 7, `CINZA_ESCURO` temo o valor 8 e `BRANCO` tem o valor 15:

```
enum {PRETO, CINZA_CLARO = 7, CINZA_ESCURO, BRANCO = 15} coresEGA;
```

Como as constantes de uma enumeração são, na verdade, números inteiros, a linguagem C permite que um(a) programador(a) use-as em expressões com inteiros. Por exemplo, o trecho de código a seguir é válido em um programa:

```
int i;
enum {PAUS, COPAS, ESPADAS, OUROS} n;
i = COPAS;
n = 0;
n++;
i = n + 2;
```

Apesar da conveniência de podermos usar uma constante de uma enumeração como um número inteiro, é perigoso usar um número inteiro como um valor de uma enumeração. Por exemplo, podemos acidentalmente armazenar o número 4, que não corresponde a qualquer naipe, na variável `n`.

Vejamos o programa 30.2 com um exemplo do uso de enumerações, onde contamos o número de cartas de cada naipe de um conjunto de cartas fornecido como entrada. Uma entrada é uma cadeia de caracteres, dada como "4pAp2c1o7e", onde "p" significa o naipe de paus, "c" copas, "e" espadas e "o" ouros.

## Exercícios

30.1 Escreva um programa que receba uma coleção de $n$ números, com $1 \leqslant n \leqslant 100$, que podem ser inteiros pequenos de 1 (um) byte, inteiros maiores de 4 (quatro) bytes ou números reais, e receba ainda um número $x$ e verifique se $x$ pertence a esse conjunto. Use uma forma de armazenamento que minimize a quantidade de memória utilizada. Use um vetor de registros tal que cada célula contém um campo que é uma enumeração, para indicar o tipo do número armazenado nessa célula, e um campo que é uma união, para armazenar um número.

30.2 Dados os números inteiros $m$ e $n$ e uma matriz $A$ de números inteiros, com $1 \leqslant m, n \leqslant 100$, calcular a quantidade de linhas que contêm o elemento 0 (zero) e a quantidade de colunas. Use uma variável indicadora de passagem, lógica, declarada como uma enumeração.

Programa 30.2: Uso de enumerações.

```
#include <stdio.h>
int main(void)
{
  char entrada[109];
  int i, j, conta[4] = {0};
  struct {
    char valor[2];
    enum {PAUS, COPAS, ESPADAS, OUROS} naipe;
  } mao[52];
  printf("Informe uma mão: ");
  scanf("%s", entrada);
  for (i = 0, j = 0; entrada[i]; j++) {
    mao[j].valor[0] = entrada[i];
    i++;
    if (entrada[i] == '0') {
      mao[j].valor[1] = entrada[i];
      i++;
    }
    switch (entrada[i]) {
    case 'p':
      mao[j].naipe = PAUS;
      break;
    case 'c':
      mao[j].naipe = COPAS;
      break;
    case 'e':
      mao[j].naipe = ESPADAS;
      break;
    case 'o':
      mao[j].naipe = OUROS;
      break;
    default:
      break;
    }
    i++;
  }
  for (j--; j >= 0; j--)
    conta[mao[j].naipe]++;
  printf("Cartas:\n");
  printf("  %d de paus\n", conta[PAUS]);
  printf("  %d de copas\n", conta[COPAS]);
  printf("  %d de espadas\n", conta[ESPADAS]);
  printf("  %d de ouros\n", conta[OUROS]);
  return 0;
}
```

AULA 31

# DEPURAÇÃO DE PROGRAMAS

À medida que temos resolvido problemas maiores e mais complexos, natural e conseqüentemente nossos programas têm ganhado complexidade e extensão. Uso correto de estruturas de programação, identificadores de variáveis significativos, indentação, comentários, tudo isso tem sido feito na tentativa de escrever programas corretamente. Ou seja, nosso desejo é sempre escrever programas eficientes, coesos e corretos.

Infelizmente, nem sempre isso é possível. Por falta de atenção, de experiência, de disciplina, ou mesmo pela complexidade do problema que precisamos resolver, não é incomum o desenvolvimento de um programa que contém erros. Nesta aula, aprenderemos a usar um poderoso depurador de programas interativo, chamado GDB, como aliado na construção de programas corretos. Veremos algumas de suas principais características, talvez as mais úteis. Interessados(as) em mais informações e características avançadas desta ferramenta devem consultar a página e o manual da mesma.

## 31.1 Depurador GDB

Um **depurador** é um programa que permite a um programador visualizar o que acontece no interior de um outro programa durante sua execução. Neste sentido, podemos dizer que um depurador é uma poderosa ferramenta de auxílio à programação. O depurador GDB, do Projeto GNU, projetado inicialmente por Richard Stallman[1], é usado com freqüência para depurar programas compilados com o compilador GCC, também do projeto GNU. Para auxiliar um programador na procura por erros em um programa, o GDB pode agir de uma das quatro principais formas abaixo:

- iniciar a execução de um programa, especificando qualquer coisa que possa afetar seu comportamento;
- interromper a execução de um programa sob condições estabelecidas;
- examinar o que aconteceu quando a execução do programa foi interrompida;
- modificar algo no programa, permitindo que o programador corrija um erro e possa continuar a investigar outros erros no mesmo programa.

O depurador GDB pode ser usado para depurar programas escritos nas linguagens de programação C, C++, Objective-C, Modula-2, Pascal e Fortran.

[1] Richard Matthew Stallman (1953−), nascido nos Estados Unidos, físico, ativista político e ativista de software.

GDB é um *software livre*, protegido pela Licença Pública Geral (GPL) da GNU. A GPL dá a seu usuário a liberdade para copiar ou adaptar um programa licenciado, mas qualquer pessoa que faz uma cópia também deve ter a liberdade de modificar aquela cópia – o que significa que deve ter acesso ao código fonte –, e a liberdade de distribuir essas cópias. Empresas de software típicas usam o *copyright* para limitar a liberdade dos usuários; a Fundação para o Software Livre (*Free Software Foundation*) usa a GPL para preservar essas liberdades. Fundamentalmente, a Licença Pública Geral (GPL) é uma licença que diz que todas as pessoas têm essas liberdades e que ninguém pode restringi-las.

Para executar um programa com auxílio do depurador GDB, o programa fonte na linguagem C deve ser compilado com o compilador GCC usando a diretiva de compilação ou opção `-g`. Essa diretiva de compilação faz com que o compilador adicione informações extras no programa executável que serão usadas pelo GDB.

## 31.2 Primeiro contato

De uma janela de comandos, você pode chamar o depurador GDB, como a seguir:

```
$ gdb
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(gdb)
```

Em geral, quando queremos depurar um programa, digitamos o comando `gdb programa` em uma janela de comandos, onde `programa` é um programa executável compilado e gerado pelo compilador `gcc` usando a diretiva de compilação `-g`. Dessa forma,

```
$ gdb ./consecutivos
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

faz a chamada do depurador GDB com a carga do programa executável `consecutivos`, que foi previamente gerado pelo processo de compilação, usando o compilador GCC com a diretiva de compilação `-g`.

Para sair do GDB e retornar à janela de comandos basta digitar o comando `quit`, como a seguir:

```
(gdb) quit
$
```

## 31.3 Sintaxe dos comandos do GDB

Um comando do GDB é composto por uma linha de entrada, contendo o nome do comando seguido de zero ou mais argumentos. O nome de um comando do GDB pode ser truncado, caso essa abreviação não seja ambígüa. O usuário também pode usar a tecla `Tab` para fazer com que o GDB preencha o resto do nome do comando ou para mostrar as alternativas disponíveis, caso exista mais que uma possibilidade. Uma linha em branco como entrada significa a solicitação para que o GDB repita o último comando fornecido pelo usuário.

Podemos solicitar ao GDB informações sobre os seus comandos usando o comando `help`. Através do comando `help`, que pode ser abreviado por `h`, sem argumentos, obtemos uma pequena lista das classes de comandos do GDB:

```
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Usando o nome de uma das classes gerais de ajuda como um argumento do comando `help`, podemos obter uma lista dos comandos desta classe. Por exemplo,

```
(gdb) help status
Status inquiries.

List of commands:

info -- Generic command for showing things about the program being debugged
macro -- Prefix for commands dealing with C preprocessor macros
show -- Generic command for showing things about the debugger

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Também podemos usar o comando `help` diretamente com o nome de um comando que queremos ajuda. Por exemplo,

```
(gdb) help help
Print list of commands.
(gdb)
```

Para colocar um programa executável sob execução no GDB é necessário usar o comando `run`, que pode ser abreviado por `r`. Este programa executável deve ter sido usado como um parâmetro na chamada do depurador ou então deve ser carregado no GDB usando o comando `file` ou `exec-file`.

## 31.4 Pontos de parada

Uma das principais vantagens no uso de depuradores de programas é a possibilidade de interromper a execução de um programa antes de seu término. Sob o GDB, um programa pode ser interrompido intencionalmente por diversas razões. Neste ponto de interrupção, podemos examinar e modificar o conteúdo de variáveis, criar outros pontos de parada, ou remover antigos, e continuar a execução do programa. Usualmente, as mensagens emitidas pelo GDB fornecem explicações satisfatórias sobre o estado do programa, mas podemos também solicitar essas informações explicitamente a qualquer momento, através do comando `info program`. Esse comando apresenta o estado do programa, sua identificação de processo, se está sob execução ou não e, neste último caso, os motivos pelos quais sua execução foi interrompida.

Um **ponto de parada** interrompe a execução de um programa sempre que um determinado ponto do código fonte é alcançado e/ou uma determinada condição é verificada. No

GDB existem três tipos de pontos de parada: *breakpoints*, *watchpoints* e *catchpoints*. Nesta aula aprenderemos a usar dois deles, os dois primeiros.

O GDB atribui um número para cada ponto de parada quando da sua criação. Esses números são números inteiros sucessivos começando com 1. Em diversos comandos para controle de pontos de parada, podemos usar seu número seqüencial para referenciá-lo. Assim que é criado, um ponto de parada está habilitado. No entanto, durante a execução de um programa, podemos desabilitá-lo de acordo com nossa conveniência.

Um *breakpoint* é um ponto de parada que interrompe a execução de um programa em um dado ponto especificado do código fonte. Seu uso mais freqüente é com o argumento do ponto de execução como sendo o número da linha do código fonte. Por exemplo, `breakpoint 5` estabelece um ponto de parada na linha 5 do programa. O ponto de parada interromperá o programa antes da execução do código localizado na linha especificada. O comando `breakpoint` pode ser abreviado por `br`. Outra forma de uso de um *breakpoint* é através de sua associação a uma expressão lógica, estabelecendo um ponto de parada em uma dada localização do código fonte, caso uma determinada condição seja satisfeita. Por exemplo, `breakpoint 13 if ant == prox` estabelece um ponto de parada na linha 13, caso a expressão lógica descrita após a palavra reservada `if` seja avaliada com valor verdadeiro. O ponto de parada interromperá então o programa antes da execução do código na linha 22. Por outro lado, um *breakpoint* também pode ser criado através do comando `breakpoint` *`identificador`*, onde o argumento *`identificador`* é o nome de uma função. Até o momento, projetamos programas apenas com a função com identificador `main` para solucionar problemas propostos, mas em breve projetaremos outras funções e este comando pode nos ajudar a adicionar pontos de parada no início de outras funções de um programa.

Podemos também usar um ponto de parada na execução de um programa que suspende sua execução sempre que o valor de uma determinada expressão se modifica, sem ter de predizer um ponto particular no código fonte onde isso acontece. Esse ponto de parada é chamado de *watchpoint* do GDB. Por exemplo, `watch iguais` estabelece um ponto de parada que irá suspender a execução do programa quando o valor da expressão descrita – que no caso contém apenas a variável `iguais` – for modificado. O comando `watch` pode ser abreviado por `wa`. O *watchpoint* só pode ser adicionado 'durante' a execução de um programa.

Como já mencionado, o GDB atribui um número seqüencial, iniciando com 1, para cada ponto de parada, quando da sua criação. Podemos imprimir uma tabela com informações básicas sobre os pontos de parada associados ao programa em execução com o comando `info breakpoints [`*`n`*`]`, que pode ser abreviado por `info break [`*`n`*`]`. O argumento opcional *`n`* mostra as informações apenas do ponto de parada especificado. Por exemplo,

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483f5 in main at consecutivos.c:5
        breakpoint already hit 1 time
2   breakpoint     keep y   0x0804848e in main at consecutivos.c:13
        stop only if ant == prox
3   hw watchpoint  keep y              iguais
(gdb)
```

Muitas vezes, queremos eliminar um ponto de parada que já cumpriu o seu papel na depuração do programa e não queremos mais que o programa seja interrompido naquele ponto previamente especificado. Podemos usar os comandos `clear` ou `delete` para remover um ou mais pontos de parada de um programa. Os comandos `clear` e `delete`, com um argumento numérico, removem um ponto de parada especificado. Por exemplo, os comandos `clear 1` e `delete 2` removem os pontos de parada identificados pelos números 1 e 2, respectivamente. A diferença entre esses comandos se dá quando são usados sem argumentos: o comando `clear` remove o ponto de parada da linha de código em execução, se existir algum, e o comando `delete` sem argumentos remove todos os pontos de parada existentes no programa.

Ao invés de remover definitivamente um ponto de parada, pode ser útil para o processo de depuração apenas desabilitá-lo temporariamente e, se for necessário, reabilitá-lo depois. O comando `disable` desabilita todos os pontos de parada de um programa. Podemos desabilitar um ponto de parada específico através do uso de seu número como argumento desse comando `disable`. Por exemplo, `disable 3` desabilita o ponto de parada de número 3. Podemos também reabilitar um ponto de parada através do comando `enable`, que reabilita todos os pontos de parada desabilitados anteriormente, ou `enable` *n*, que reabilita o ponto de parada de número *n*.

Após verificar o que aconteceu com o programa em um dado ponto de parada, podemos retomar a execução do programa a partir deste ponto de dois modos diferentes: através do comando `continue`, que pode ser abreviado por `c`, ou através do comando `step`, que pode ser abreviado por `s`. No primeiro comando, a execução é retomada e segue até o final do programa ou até um outro ponto de parada ser encontrado. No segundo, apenas a próxima sentença do programa – em geral, a próxima linha – é executada e o controle do programa é novamente devolvido ao GDB. O comando `step` pode vir seguido por um argumento numérico, indicando quantas sentenças do código fonte queremos progredir na sua execução. Por exemplo, `step 5` indica que queremos executar as próximas 5 sentenças do código. Caso um ponto de parada seja encontrado antes disso, o programa é interrompido e o controle repassado ao GDB.

Comumente, adicionamos um ponto de parada no início de uma função onde acreditamos que exista um problema, um erro, executamos o programa até que ele atinja este ponto de parada e então usamos o comando `step` nessa área suspeita, examinando o conteúdo das variáveis envolvidas, até que o problema se revele.

## 31.5 Programa fonte

O GDB pode imprimir partes do código fonte do programa sendo depurado. Quando um programa tem sua execução interrompida por algum evento, o GDB mostra automaticamente o conteúdo da linha do código fonte onde o programa parou. Para ver outras porções do código, podemos executar comandos específicos do GDB.

O comando `list` mostra ao(à) programador(a) algumas linhas do código fonte carregado previamente. Em geral, mostra 10 linhas em torno do ponto onde a execução do programa se encontra. Podemos explicitamente solicitar ao GDB que mostre um intervalo de linhas do código fonte, como por exemplo, `list 1,14`. Neste caso, as primeiras 14 linhas do código fonte do programa carregado no GDB serão mostradas.

```
(gdb) list 1,14
1       #include <stdio.h>
2       int main(void)
3       {
4         int num, ant, prox, iguais;
5         printf("\nInforme um número: ");
6         scanf("%d", &num);
7         prox = num % 10;
8         iguais = 0;
9         while (num != 0 && !iguais) {
10          num = num / 10;
11          ant = prox;
12          prox = num % 10;
13          if (ant == prox)
14            iguais = 1;
(gdb)
```

## 31.6 Verificação de dados

Uma forma usual de examinar informações em nosso programa é através do comando `print`, que pode ser abreviado por `p`. Esse comando avalia uma expressão e mostra seu resultado. Por exemplo, `print 2*a+3` mostra o resultado da avaliação da expressão `2*a+3`, supondo que `a` é uma variável do nosso programa. Sem argumento, esse comando mostra o último resultado apresentado pelo comando `print`.

Uma facilidade que o GDB nos oferece é mostrar o valor de uma expressão freqüentemente. Dessa forma, podemos verificar como essa expressão se comporta durante a execução do programa. Podemos criar uma **lista de impressão automática** que o GDB mostra cada vez que o programa tem sua execução interrompida. Podemos usar o comando `display` para adicionar uma expressão à lista de impressão automática de um programa. Por exemplo, `display iguais` adiciona a variável `iguais` a essa lista e faz com que o GDB imprima o conteúdo dessa veriável sempre que a execução do programa for interrompida por algum evento. Aos elementos inseridos nessa lista são atribuídos números inteiros consecutivos, iniciando com 1. Para remover uma expressão dessa lista, basta executar o comando `delete display` *n*, onde *n* é o número da expressão que desejamos remover. Para visualizar todas as expressões nesta lista é necessário executar o comando `info display`.

Todas as variáveis internas de uma função, e seus conteúdos, podem ser visualizados através do comando `info locals`.

## 31.7 Alteração de dados durante a execução

Podemos ainda alterar conteúdos de variáveis durante a execução de um programa no GDB. Em qualquer momento da execução, podemos usar o comando `set var` para alterar o con-

teúdo de uma variável. Por exemplo, `set var x = 2` modifica o conteúdo da variável `x` para 2. O lado direito após o símbolo `=` no comando `set var` pode conter uma constante, uma variável ou uma expressão válida. Essa característica do GDB permite que um(a) programador(a) possa alterar os conteúdos de variáveis durante a execução de um programa e, com isso, possa controlar seu fluxo de execução e corrigir possíveis erros.

## 31.8 Resumo dos comandos

Apresentamos agora uma relação dos comandos que aprendemos, com uma explicação breve sobre cada um. A intenção é agrupar os comandos em um lugar só, para que a relação nos sirva de referência. Vale reiterar que o GDB possui muitos comandos não cobertos aqui.

| Comando | Significado |
|---|---|
| **PROGRAMA FONTE** | |
| `list [n]` | Mostra linhas em torno da linha *n* ou as próximas 10 linhas, se não especificada |
| `list m,n` | Mostra linhas entre *m* e *n* |
| **VARIÁVEIS E EXPRESSÕES** | |
| `print expr` | Imprime *expr* |
| `display expr` | Adiciona *expr* à lista de impressão automática |
| `info display` | Mostra a lista de impressão automática |
| `delete display n` | Remove a expressão de número *n* da lista de impressão automática |
| `info locals` | Mostra o conteúdo de todas as variáveis locais na função corrente |
| `set var var = expr` | Atribui *expr* para a variável *var* |
| **PONTOS DE PARADA** | |
| `break n` | Estabelece um *breakpoint* na linha *n* |
| `break n if expr` | Estabelece um *breakpoint* na linha *n* se o valor de *expr* for verdadeiro |
| `break func` | Estabelece um *breakpoint* no início da função *func* |
| `watch expr` | Estabelece um *watchpoint* na expressão *expr* |
| `clear [n]` | Remove o ponto de parada na linha *n* ou na próxima linha, se não especificada |
| `delete [n]` | Remove o ponto de parada de número *n* ou todos os pontos de parada, se não especificado |
| `enable [n]` | Habilita todos os pontos de parada suspensos ou o ponto de parada *n* |
| `disable [n]` | Desabilita todos os pontos de parada ou o ponto de parada *n* |
| `info break` | Mostra todos os pontos de parada |
| **EXECUÇÃO DO PROGRAMA** | |
| `file [prog]` | Carrega o programa executável *prog* e sua tabela de símbolos ou libera o GDB da execução corrente, se não especificado |
| `exec-file [prog]` | Carrega o programa executável *prog* ou libera o GDB da execução corrente, se não especificado |
| `run` | Inicia a execução do programa |
| `continue` | Continua a execução do programa |
| `step [n]` | Executa a próxima sentença do programa ou as próximas *n* sentenças |
| `info program` | Mostra o estado da execução do programa |
| `quit` | Encerra a execução do GDB |
| **AJUDA** | |
| `help` | Mostra as classes de ajuda de comandos do GDB |
| `help classe` | Mostra uma ajuda sobre a *classe* de comandos |
| `help comando` | Mostra uma ajuda sobre o *comando* |

## 31.9 Exemplos de execução

Vamos usar o GDB sobre o programa 31.1 que soluciona o exercício 11.3. No exercício 11.3 temos de verificar se um número inteiro fornecido como entrada contém dois dígitos consecutivos e iguais.

Programa 31.1: Solução do exercício 11.3.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int num, ant, prox, iguais;
5    printf("Informe um número: ");
6    scanf("%d", &num);
7    prox = num % 10;
8    iguais = 0;
9    while (num != 0 && !iguais) {
10     num = num / 10;
11     ant = prox;
12     prox = num % 10;
13     if (ant == prox)
14       iguais = 1;
15   }
16   if (iguais)
17     printf("Número tem dois dígitos consecutivos iguais.\n");
18   else
19     printf("Número não tem dois dígitos consecutivos iguais.\n");
20   return 0;
21 }
```

Iniciamos o processo compilando o programa 31.1 adequadamente e carregando o programa executável gerado no GDB:

```
$ gcc consecutivos.c -o consecutivos -Wall -ansi -pedantic -g
$ gdb ./consecutivos
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb)
```

Para verificar se o programa foi de fato carregado na chamada do GDB, podemos usar o comando `list` para ver seu código fonte ou então o comando `info program`, para visualizar o estado do programa:

```
(gdb) list 1,12
1       #include <stdio.h>
2       int main(void)
3       {
4         int num, ant, prox, iguais;
5         printf("Informe um número: ");
6         scanf("%d", &num);
7         prox = num % 10;
8         iguais = 0;
9         while (num != 0 && !iguais) {
10           num = num / 10;
11           ant = prox;
12           prox = num % 10;
(gdb)
```

ou

```
(gdb) info program
The program being debugged is not being run.
(gdb)
```

O programa pode ser executado neste momento, através do comando `run`:

```
(gdb) run
Starting program: ~/ensino/disciplinas/progi/2009/programas/consecutivos

Informe um número: 12233
Número tem dois dígitos consecutivos iguais.

Program exited normally.
(gdb)
```

A execução acima foi realizada dentro do GDB, sem a adição de qualquer ponto de parada. Como o programa ainda continua carregado no GDB, podemos executá-lo quantas vezes quisermos. Antes de iniciar a próxima execução, vamos adicionar alguns pontos de parada no programa:

```
(gdb) break 5
Breakpoint 1 at 0x8048416: file consecutivos.c, line 5.
(gdb)
```

Note que não iniciamos a execução do programa ainda. Então,

```
(gdb) run
Starting program: ~/ensino/disciplinas/progi/2009/programas/consecutivos

Breakpoint 1, main () at consecutivos.c:5
5          printf("Informe um número: ");
(gdb)
```

O ponto de parada estabelecido na linha 5 do programa foi encontrado e o GDB interrompeu a execução do programa neste ponto. Podemos visualizar todas as variáveis, e seus conteúdos, dentro da função `main` usando o comando `info locals`:

```
(gdb) info locals
num = -1209114636
ant = 134518608
prox = -1075753928
iguais = 134513993
(gdb)
```

Observe que todas as variáveis declaradas constam da relação apresentada pelo GDB e que seus conteúdos são valores estranhos, já que nenhuma dessas variáveis foi inicializada até a linha 5 do programa.

Vamos adicionar mais pontos de parada neste programa, um deles um *breakpoint* associado a uma expressão lógica e o outro um *watchpoint* associado a uma variável:

```
(gdb) break 13 if ant == prox
Breakpoint 2 at 0x80484ce: file consecutivos.c, line 13.
(gdb) watch iguais
Hardware watchpoint 3: iguais
(gdb)
```

Agora, podemos visualizar todos os pontos de parada associados a este programa com o comando `info break`:

```
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x08048416 in main at consecutivos.c:5
        breakpoint already hit 1 time
2       breakpoint     keep y   0x080484ce in main at consecutivos.c:13
        stop only if ant == prox
3       hw watchpoint  keep y              iguais
(gdb)
```

Podemos continuar a execução do programa usando o comando `continue`:

```
(gdb) continue
Continuing.

Informe um número: 55123
Hardware watchpoint 3: iguais

Old value = 134513993
New value = 0
0x08048471 in main () at consecutivos.c:8
8          iguais = 0;
(gdb)
```

Observe que na linha 8 do programa a variável `iguais` é inicializada, substituindo então um valor não válido, um lixo que `iguais` continha após sua declaração, pelo valor 0 (zero). Assim, o GDB interrompe a execução do programa nessa linha, devido ao *watchpoint* de número 3, e mostra a mudança de valores que ocorreu nessa variável.

Vamos então continuar a execução do programa:

```
(gdb) continue
Continuing.

Breakpoint 2, main () at consecutivos.c:13
13            if (ant == prox)
(gdb)
```

O GDB então interrompeu a execução do programa devido ao *breakpoint* 2, na linha 13 do programa. Neste ponto, deve ter ocorrido o evento em que os conteúdos das variáveis `ant` e `prox` coincidem. Para verificar se esse evento de fato ocorreu, podemos usar o comando `print` para verificar os conteúdos dessas variáveis:

```
(gdb) print ant
$1 = 5
(gdb) print prox
$2 = 5
(gdb)
```

Vamos continuar a execução deste programa:

```
(gdb) continue
Continuing.
Hardware watchpoint 3: iguais

Old value = 0
New value = 1
main () at consecutivos.c:9
9          while (num != 0 && !iguais) {
(gdb)
```

Observe que o programa foi interrompido pelo *watchpoint*, já que o conteúdo da variável `iguais` foi modificado de 0 para 1 na linha 14 do programa. Assim, a próxima linha a ser executada é a linha 9, como mostrado pelo GDB.

Vamos dar uma olhada no conteúdo das variáveis da função `main`:

```
(gdb) info locals
num = 5
ant = 5
prox = 5
iguais = 1
(gdb)
```

O programa então está quase no fim. Vamos continuar sua execução:

```
(gdb) continue
Continuing.
Número tem dois dígitos consecutivos iguais.

Watchpoint 3 deleted because the program has left the block in
which its expression is valid.
0xb7f08250 in ?? () from /lib/ld-linux.so.2
(gdb)
```

Mais uma análise e mais alguns testes sobre o programa 31.1 nos permitem afirmar com alguma convicção que este programa está correto e que soluciona o exercício 11.3. O fato novo é que essa afirmação foi feita também com o GDB nos auxiliando. Para provar formalmente que o programa 31.1 está correto, devemos descrever algum invariante do processo iterativo das linhas 9 até 15 do programa e então provar por indução a sua validade, conforme a aula 20.

No próximo exemplo vamos executar um programa sob o controle do GDB que não soluciona o problema associado, isto é, contém algum erro que, à primeira vista, não conseguimos corrigir. O programa 31.2 se propõe a implementar a ordenação por seleção.

Programa 31.2: Uma versão de um programa que deveria realizar a ordenação por seleção.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int i, j, n, menor, indice, aux, A[10];
5    scanf("%d", &n);
6    for (i = 0; i < n; i++)
7      scanf("%d", &A[i]);
8    for (i = 0; i < n-1; i++) {
9      menor = A[i];
10     indice = i;
11     for (j = n-1; j > i; j++) {
12       if (A[j] < menor) {
13         menor = A[j];
14         indice = j;
15       }
16     }
17     aux = A[i];
18     A[i] = A[indice];
19     A[indice] = aux;
20   }
21   for (i = 0; i < n; i++)
22     printf("%d ", A[i]);
23   printf("\n");
24   return 0;
25 }
```

Em uma tentativa de execução do programa 31.2, ocorre o seguinte erro:

```
$ ./ord-selecao
8
7 5 4 1 8 6 2 3
Falha de segmentação
$
```

Esse erro não nos dá nenhuma dica de onde procurá-lo. E se já nos debruçamos sobre o código fonte e não conseguimos encontrá-lo, a melhor idéia e usar um depurador para nos ajudar. Neste exemplo, mostraremos o uso do GDB sem muitas interrupções no texto, a menos que necessárias. Então, segue uma depuração do programa 31.2.

```
$ gcc ord-selecao.c -o ord-selecao -Wall -ansi -pedantic -g
$ gdb ./ord-selecao
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) list 8,15
8         for (i = 0; i < n-1; i++) {
9           menor = A[i];
10          indice = i;
11          for (j = n-1; j > i; j++) {
12            if (A[j] < menor) {
13              menor = A[j];
14              indice = j;
15        }
(gdb) break 8
Breakpoint 1 at 0x804845c: file ord-selecao.c, line 8.
(gdb) run
Starting program: ~/ensino/disciplinas/progi/2009/programas/ord-selecao
8
7 5 4 1 8 6 2 3

Breakpoint 1, main () at ord-selecao.c:8
8         for (i = 0; i < n-1; i++) {

(gdb) step 4
12            if (A[j] < menor) {
(gdb) info locals
i = 0
j = 7
n = 8
menor = 7
indice = 0
aux = -1208630704
A = {7, 5, 4, 1, 8, 6, 2, 3, 134518484, -1075407768}
(gdb) display A[j]
1: A[j] = 3
(gdb) display j
2: j = 7
(gdb) display i
3: i = 0
```

```
(gdb) watch menor
Hardware watchpoint 2: menor
(gdb) continue
Continuing.
Hardware watchpoint 2: menor

Old value = 7
New value = 3
main () at ord-selecao.c:14
14                indice = j;
3: i = 0
2: j = 7
1: A[j] = 3

(gdb) continue
Continuing.
Hardware watchpoint 2: menor

Old value = 3
New value = -1075407768
main () at ord-selecao.c:14
14                indice = j;
3: i = 0
2: j = 9
1: A[j] = -1075407768
(gdb) c
Continuing.
Hardware watchpoint 2: menor

Old value = -1075407768
New value = -1208630704
main () at ord-selecao.c:14
14                indice = j;
3: i = 0
2: j = 15
1: A[j] = -1208630704
(gdb) step 100
Hardware watchpoint 2: menor

Old value = -1208630704
New value = -1210084267
main () at ord-selecao.c:14
14                indice = j;
3: i = 0
2: j = 18
1: A[j] = -1210084267
(gdb) set var j=-1
(gdb) set var i=n
(gdb) continue
Continuing.
```

```
7 5 4 1 8 6 2 3

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0xb7f5c251 in ?? () from /lib/ld-linux.so.2
(gdb)
```

Encontramos um erro na linha 11 do programa, no passo da estrutura de repetição `for`. Substituindo `j++` por `j--` nessa linha, temos um programa corrigido que realiza uma ordenação por seleção de $n$ números inteiros.

AULA 32

# EFICIÊNCIA DE PROGRAMAS

Medir a eficiência de um programa significa tentar predizer os recursos necessários para seu funcionamento. O recurso que temos mais interesse é o tempo embora a memória, a comunicação e as portas lógicas também podem ser de interesse. Na análise de programas alternativos para solução de um mesmo problema, aqueles mais eficientes, isto é, que usam a menor quantidade de tempo, são em geral escolhidos como melhores programas. Nesta aula faremos uma discussão inicial sobre eficiência de programas.

## 32.1 Programas

Um **programa**, como já sabemos, é uma seqüência bem definida de passos descritos em uma linguagem de programação específica que transforma um conjunto de valores, chamado de **entrada**, e produz um conjunto de valores, chamado de **saída**. Dessa forma, um programa é também uma ferramenta para solucionar um **problema computacional** bem definido.

Suponha que temos um problema computacional bem definido, o problema da busca, que já vimos antes. A descrição do problema é dada a seguir:

> Dado um número intiro $n$, com $1 \leqslant n \leqslant 100$, um conjunto $C$ de $n$ números inteiros e um número inteiro $x$, verificar se $x$ se encontra no conjunto $C$.

O problema da busca é um problema computacional básico que surge em diversas aplicações práticas. A busca é uma operação básica em computação e, por isso, vários bons programas que a realizam foram desenvolvidos. A escolha do melhor programa para uma dada aplicação depende de alguns fatores como a quantidade de elementos no conjunto $C$ e da complexidade da estrutura em que $C$ está armazenado.

Como já vimos, se para qualquer entrada um programa pára com a resposta correta, então dizemos que o mesmo está **correto**. Assim, um programa correto **soluciona** o problema computacional associado. Um programa incorreto pode sequer parar, para alguma entrada, ou pode parar mas com uma resposta indesejada.

O programa 32.1 implementa uma busca de um número inteiro $x$ em um conjunto de $n$ números inteiros $C$ armazenado na memória como um vetor. A busca se dá seqüencialmente no vetor $C$, da esquerda para direita, até que um elemento com índice $i$ no vetor $C$ contenha o valor $x$, quando o programa responde que o elemento foi encontrado, mostrando sua posição. Caso contrário, o vetor $C$ é todo percorrido e o programa informa que o elemento $x$ não se encontra no vetor $C$.

Programa 32.1: Busca de $x$ em $C$.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  int n, i, C[MAX], x;
  printf("Informe n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Informe um elemento: ");
    scanf("%d", &C[i]);
  }
  printf("Informe x: ");
  scanf("%d", &x);
  for (i = 0; i < n && C[i] != x; i++)
    ;
  if (i < n)
    printf("%d está na posição %d de C\n", x, i);
  else
    printf("%d não pertence ao conjunto C\n", x);
  return 0;
}
```

## 32.2 Análise de algoritmos e programas

Antes de analisar um programa, devemos conhecer o modelo da tecnologia de computação usada na máquina em que implementamos o programa, para estabelecer os custos associados aos recursos que o programa usa. Na análise de algoritmos e programas, consideramos regularmente um modelo de computação genérico chamado de **máquina de acesso aleatório** (do inglês *random acess machine – RAM*) com um processador. Nesse modelo, as instruções são executadas uma após outra, sem concorrência. Modelos de computação paralela e distribuída, que usam concorrência de instruções, são modelos investigativos que vêm se tornando realidade recentemente. No entanto, nosso modelo para análise de algoritmos e programas não leva em conta essas premissas.

A análise de um programa pode ser uma tarefa desafiadora envolvendo diversas ferramentas matemáticas como combinatória discreta, teoria das probabilidades, álgebra e etc. Como o comportamento de um programa pode ser diferente para cada entrada possível, precisamos de uma forma para resumir esse comportamento em fórmulas matemáticas simples e de fácil compreensão.

Mesmo com a convenção de um modelo fixo de computação para analisar um programa, ainda precisamos fazer muitas escolhas para decidir como expressar nossa análise. Um dos principais objetivos é encontrar uma forma de expressão que é simples de escrever e manipular, que mostra as características mais importantes das necessidades do programa e exclui os detalhes mais tediosos.

Não é difícil notar que a análise do programa 32.1 depende do número de elementos fornecidos na entrada. Ou seja, procurar um elemento $x$ em um conjunto $C$ com milhares de elementos gasta mais tempo que procurá-lo em um conjunto $C$ com três elementos. Além disso, o programa 32.1 gasta diferentes quantidades de tempo para buscar um elemento em conjuntos de mesmo tamanho, dependendo de como os elementos nesses conjuntos estão dispostos. Em geral, o tempo gasto por um programa cresce com o tamanho da entrada e assim é comum descrever o tempo de execução de um programa como uma função do tamanho de sua entrada.

Por **tamanho da entrada** queremos dizer, em geral, o número de itens na entrada. Por exemplo, o vetor de números com $n$ números inteiros onde a busca de um elemento será realizada. Em outros casos, como na multiplicação de dois números inteiros, a melhor medida para o tamanho da entrada é o número de bits necessários para representar essa entrada na notação binária. O **tempo de execução** de um programa sobre uma entrada particular é o número de operações primitivas ou passos executados por ele. Quanto mais independente da máquina é a definição de um passo, mais conveniente para análise de tempo dos programas.

Considere então que uma quantidade constante de tempo é necessária para executar cada linha do programa. Uma linha pode gastar uma quantidade de tempo diferente de outra linha, mas consideramos que cada execução da $i$-ésima linha gasta tempo $c_i$, onde $c_i$ é uma constante.

Iniciamos a análise do programa 32.1 destacando que o aspecto mais importante para sua análise é o tempo gasto com a busca do elemento $x$ no vetor $C$, descrita nas linhas 14 e 15. As linhas restantes contêm diretivas de pré-processador, cabeçalho da função `main`, a entrada de dados nas linhas 6 a 13, a saída nas linhas 16 a 19 e a sentença `return` que finaliza a função `main` na linha 20. Isto é, queremos saber o tempo gasto no processamento, na transformação, dos dados de entrada nos dados de saída.

| | *Custo* | *Vezes* |
|---|---|---|
| `#include <stdio.h>` | $c_1$ | 1 |
| `#define MAX 100` | $c_2$ | 1 |
| `int main(void)` | $c_3$ | 1 |
| `{` | 0 | 1 |
| `  int n, i, C[MAX], x;` | $c_4$ | 1 |
| `  printf("Informe n:  ");` | $c_5$ | 1 |
| `  scanf("%d", &n);` | $c_6$ | 1 |
| `  for (i = 0; i < n; i++) {` | $c_7$ | $n+1$ |
| `    printf("Informe um elemento:  ");` | $c_8$ | $n$ |
| `    scanf("%d", &C[i]);` | $c_9$ | $n$ |
| `  }` | 0 | $n$ |
| `  printf("Informe x:  ");` | $c_{10}$ | 1 |
| `  scanf("%d", &x);` | $c_{11}$ | 1 |
| `  for (i = 0; i < n && C[i] != x; i++)` | $c_{12}$ | $t_i$ |
| `    ;` | 0 | $t_i - 1$ |
| `  if (i < n)` | $c_{13}$ | 1 |
| `    printf("%d está na posição %d de C\n", x, i);` | $c_{14}$ | 1 |
| `  else` | $c_{15}$ | 1 |
| `    printf("%d não pertence ao conjunto C\n", x);` | $c_{16}$ | 1 |
| `  return 0;` | $c_{17}$ | 1 |
| `}` | 0 | 1 |

O tempo de execução do programa 32.1 é dado pela soma dos tempos para cada sentença executada. Uma sentença que gasta $c_i$ passos e é executada $n$ vezes contribui com $c_i n$ no tempo de execução do programa. Para computar $T(n)$, o tempo de execução do programa 32.1, devemos somar os produtos das colunas ***Custo*** e ***Vezes***, obtendo:

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ c_{10} + c_{11} + c_{12} t_i + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \, .$$

Observe que, mesmo para entradas de um mesmo tamanho, o tempo de execução de um programa pode depender de qual entrada desse tamanho é fornecida. Por exemplo, no programa 32.1, o melhor caso ocorre se o elemento $x$ encontra-se na primeira posição do vetor $C$. Assim, $t_i = 1$ e o tempo de execução do melhor caso é dado por:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ &\quad c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \\ &= (c_7 + c_8 + c_9)n + \\ &\quad (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17}) \, . \end{aligned}$$

Esse tempo de execução pode ser expresso como $an+b$ para constantes $a$ e $b$, que dependem dos custos $c_i$ das sentenças do programa. Assim, o tempo de execução é dado por uma função linear em $n$. No entanto, observe que as constantes $c_7, c_8$ e $c_9$ que multiplicam $n$ na fórmula de $T(n)$ são aquelas que tratam somente da entrada de dados. Evidentemente que para armazenar $n$ números inteiros em um vetor devemos gastar tempo $an$, para alguma constante $a$. Dessa forma, se consideramos apenas a constante $c_{12}$ na fórmula, que trata propriamente da busca, o tempo de execução de melhor caso, quando $t_i = 1$, é dado por:

$$T(n) = c_{12} t_i = c_{12} \, .$$

Por outro lado, se o elemento $x$ não se encontra no conjunto $C$, temos então o pior caso do programa. Além de comparar $i$ com $n$, comparamos também o elemento $x$ com o elemento $C[i]$ para cada $i$, $0 \leqslant i \leqslant n-1$. Uma última comparação ainda é realizada quando $i$ atinge o valor $n$. Assim, $t_i = n+1$ e o tempo de execução de pior caso é dado por:

$$T(n) = c_{12} t_i = c_{12}(n+1) = c_{12} n + c_{12} \, .$$

Esse tempo de execução de pior caso pode ser expresso como uma função linear $an+b$ para constantes $a$ e $b$ que dependem somente da constante $c_{12}$, responsável pelo trecho do programa que realiza o processamento.

Na análise do programa 32.1, estabelecemos os tempos de execução do melhor caso, quando encontramos o elemento procurado logo na primeira posição do vetor que representa o conjunto, e do pior caso, quando não encontramos o elemento no vetor. No entanto, estamos em geral interessados no **tempo de execução de pior caso** de um programa, isto é, o maior tempo de execução para qualquer entrada de tamanho $n$.

Como o tempo de execução de pior caso de um programa é um limitante superior para seu tempo de execução para qualquer entrada, temos então uma garantia que o programa nunca vai gastar mais tempo que esse estabelecido. Além disso, o pior caso ocorre muito freqüentemente nos programas em geral, como no caso do problema da busca.

### 32.2.1 Ordem de crescimento de funções matemáticas

Acabamos de usar algumas convenções que simplificam a análise do programa 32.1. A primeira abstração que fizemos foi ignorar o custo real de uma sentença do programa, usando as constantes $c_i$ para representar esses custos. Depois, observamos que mesmo essas constantes nos dão mais detalhes do que realmente precisamos: o tempo de execução de pior caso do programa 32.1 é $an + b$, para constantes $a$ e $b$ que dependem dos custos $c_i$ das sentenças. Dessa forma, ignoramos não apenas o custo real das sentenças mas também os custos abstratos $c_i$.

Na direção de realizar mais uma simplificação, estamos interessados na **taxa de crescimento**, ou **ordem de crescimento**, da função que descreve o tempo de execução de um algoritmo ou programa. Portanto, consideramos apenas o 'maior' termo da fórmula, como por exemplo $an$, já que os termos menores são relativamente insignificantes para $n$ grande. Também ignoramos o coeficiente constante do maior termo, já que fatores constantes são menos significativos que a taxa de crescimento no cálculo da eficiência computacional para entradas grandes. Dessa forma, dizemos que o programa 32.1, por exemplo, tem tempo de execução de pior caso $O(n)$.

Usualmente, consideramos um programa mais eficiente que outro se seu tempo de execução de pior caso tem ordem de crescimento menor. Essa avaliação pode ser errônea para pequenas entradas mas, entradas para suficientemente grandes, um programa que tem tempo de execução de pior caso $O(n)$ executará mais rapidamente no pior caso que um programa que tem tempo de execução de pior caso $O(n^2)$.

Quando olhamos para entradas cujos tamanhos são grandes o suficiente para fazer com que somente a taxa de crescimento da função que descreve o tempo de execução de um programa seja relevante, estamos estudando na verdade a **eficiência assintótica** de um algoritmo ou programa. Isto é, concentramo-nos em saber como o tempo de execução de um programa cresce com o tamanho da entrada *no limite*, quando o tamanho da entrada cresce ilimitadamente. Usualmente, um programa que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, excluindo talvez algumas entradas pequenas.

As notações que usamos para descrever o tempo de execução assintótico de um programa são definidas em termos de funções matemáticas cujos domínios são o conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \ldots\}$. Essas notações são convenientes para descrever o tempo de execução de pior caso $T(n)$ que é usualmente definido sobre entradas de tamanhos inteiros.

No início desta seção estabelecemos que o tempo de execução de pior caso do programa 32.1 é $T(n) = O(n)$. Vamos definir formalmente o que essa notação significa. Para uma dada função $g(n)$, denotamos por $O(g(n))$ o *conjunto de funções*

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leqslant f(n) \leqslant cg(n) \text{ para todo } n \geqslant n_0\}\,.$$

A função $f(n)$ pertence ao conjunto $O(g(n))$ se existe uma constante positiva $c$ tal que $f(n)$ "não seja maior que" $cg(n)$, para $n$ suficientemente grande. Dessa forma, usamos a notação $O$ para fornecer um limitante assintótico superior sobre uma função, dentro de um fator constante. Apesar de $O(g(n))$ ser um conjunto, escrevemos "$f(n) = O(g(n))$" para indicar que $f(n)$ é um elemento de $O(g(n))$, isto é, que $f(n) \in O(g(n))$.

A figura 32.1 mostra a intuição por trás da notação $O$. Para todos os valores de $n$ à direita de $n_0$, o valor da função $f(n)$ está sobre ou abaixo do valor da função $g(n)$.

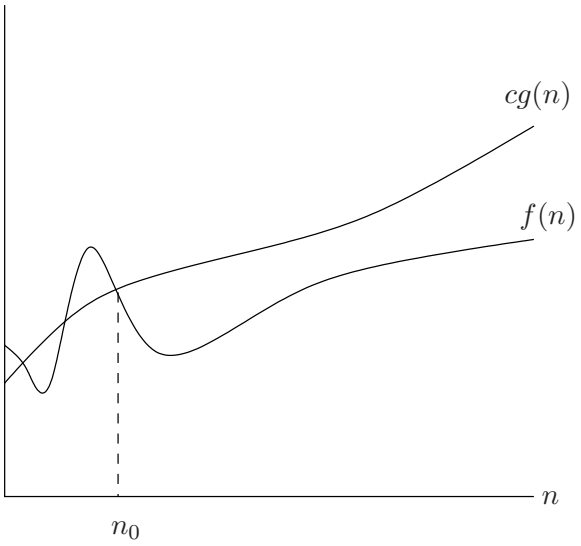



Figura 32.1: $f(n) = O(g(n))$.

Dessa forma, podemos dizer, por exemplo, que $4n + 1 = O(n)$. Isso porque existem constantes positivas $c$ e $n_0$ tais que

$$4n + 1 \leqslant cn$$

para todo $n \geqslant n_0$. Tomando $c = 5$ e $n_0 = 1$, essa desigualdade é satisfeita para todo $n \geqslant n_0$. Certamente, existem outras escolhas para as constantes $c$ e $n_0$, mas o mais importante é que existe alguma escolha. Observe que as constantes dependem da função $4n + 1$. Uma função diferente que pertence a $O(n)$ provavelmente necessita de outras constantes.

A definição de $O(g(n))$ requer que toda função pertencente a $O(g(n))$ seja assintoticamente não-negativa, isto é, que $f(n)$ seja não-negativa sempre que $n$ seja suficientemente grande. Conseqüentemente, a própria função $g(n)$ deve ser assintoticamente não-negativa, caso contrário o conjunto $O(g(n))$ é vazio. Portanto, vamos considerar que toda função usada na notação $O$ é assintoticamente não-negativa.

A ordem de crescimento do tempo de execução de um programa ou algoritmo pode ser denotada através de outras notações assintóticas, tais como as notações $\Theta$, $\Omega$, $o$ e $\omega$. Essas notações são específicas para análise do tempo de execução de um programa através de outros pontos de vista. Nesta aula, ficaremos restritos apenas à notação $O$.

## 32.3 Análise da ordenação por trocas sucessivas

Vamos fazer a análise de um outro programa que já conhecemos bem e que soluciona o problema da ordenação. O programa 32.2 faz a ordenação de um vetor de $n$ elementos fornecidos como entrada usando o método das trocas sucessivas ou o método da bolha.

O problema computacional que o programa 32.2 soluciona é descrito formalmente a seguir:

> Dado um número inteiro positivo $n$, com $1 \leqslant n \leqslant 100$, e uma seqüência de $n$ números inteiros, determinar a ordenação desses $n$ números em ordem não-decrescente.

Programa 32.2: Método de ordenação através de trocas sucessivas.

```
1  #include <stdio.h>
2  #define MAX 100
3  int main(void)
4  {
5    int i, j, n, aux, A[MAX];
6    scanf("%d", &n);
7    for (i = 0; i < n; i++)
8      scanf("%d", &A[i]);
9    for (i = n-1; i > 0; i--)
10     for (j = 0; j < i; j++)
11       if (A[j] > A[j+1]) {
12         aux = A[j];
13         A[j] = A[j+1];
14         A[j+1] = aux;
15       }
16   for (i = 0; i < n; i++)
17     printf("%d  ", A[i]);
18   printf("\n");
19   return 0;
20 }
```

Conforme já fizemos antes, a análise linha a linha do programa 32.2 é descrita abaixo.

| | *Custo* | *Vezes* |
|---|---|---|
| `#include <stdio.h>` | $c_1$ | $1$ |
| `#define MAX 100` | $c_2$ | $1$ |
| `int main(void)` | $c_3$ | $1$ |
| `{` | $0$ | $1$ |
| `  int i, j, n, aux, A[MAX];` | $c_4$ | $1$ |
| `  scanf("%d", &n);` | $c_5$ | $1$ |
| `  for (i = 0; i < n; i++)` | $c_6$ | $n+1$ |
| `    scanf("%d", &A[i]);` | $c_7$ | $n$ |
| `  for (i = n-1; i > 0; i--)` | $c_8$ | $n$ |
| `    for (j = 0; j < i; j++)` | $c_9$ | $\sum_{i=1}^{n-1}(i+1)$ |
| `      if (A[j] > A[j+1]) {` | $c_{10}$ | $\sum_{i=1}^{n-1} i$ |
| `        aux = A[j];` | $c_{11}$ | $\sum_{i=1}^{n-1} t_i$ |
| `        A[j] = A[j+1];` | $c_{12}$ | $\sum_{i=1}^{n-1} t_i$ |
| `        A[j+1] = aux;` | $c_{13}$ | $\sum_{i=1}^{n-1} t_i$ |
| `      }` | $0$ | $\sum_{i=1}^{n-1} i$ |
| `  for (i = 0; i < n; i++)` | $c_{14}$ | $n+1$ |
| `    printf("%d ", A[i]);` | $c_{15}$ | $n$ |
| `  printf("\n";` | $c_{16}$ | $1$ |
| `  return 0;` | $c_{17}$ | $1$ |
| `}` | $0$ | $1$ |

Podemos ir um passo adiante na simplificação da análise de um algoritmo ou programa se consideramos que cada linha tem custo de processamento 1. Na prática, isso não é bem verdade já que há sentenças mais custosas que outras. Por exemplo, o custo para executar uma sentença que contém a multiplicação de dois números de ponto flutuante de precisão dupla é maior que o custo de comparar dois números inteiros. No entanto, ainda assim vamos considerar que o custo para processar qualquer linha de um programa é constante e, mais ainda, que tem custo 1. Dessa forma, a coluna com rótulo ***Vezes*** nas análises acima representam então o custo de execução de cada linha do programa.

Para análise do programa 32.2 vamos considerar apenas o trecho onde ocorre o processamento, isto é, as linhas de 9 a 15. O melhor caso para o programa 32.2 ocorre quando a seqüência de entrada com $n$ números inteiros é fornecida em ordem não-decrescente. Observe que, nesse caso, as linhas 12, 13 e 14 não serão executadas sequer uma vez. Ou seja, $t_i = 0$ para todo $i$. Então, o tempo de execução no melhor caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} 0 \\
&= n + 2\,\frac{n(n-1)}{2} + n - 1 \\
&= n^2 + n - 1\,.
\end{aligned}$$

Então, escolhendo $c = 2$ e $n_0 = 1$ temos que $T(n) = O(n^2)$. Ou seja, o tempo de execução do melhor caso do programa 32.2 que implementa o método da bolha é quadrático no tamanho da entrada.

Para definir a entrada que determina o pior caso para o programa 32.2 devemos notar que quando o vetor contém os $n$ elementos em ordem decrescente, o maior número de trocas é realizado. Assim, $t_i = i$ para todo $i$ e o tempo de execução de pior caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} t_i \\
&= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} i \\
&= n + 5\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\
&= n + 5\,\frac{n(n-1)}{2} + n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{5}{2}\,n + 2n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1\,.
\end{aligned}$$

Então, escolhendo $c = 5/2$ e $n_0 = 1$ temos que $T(n) = O(n^2)$. Ou seja, o tempo de execução do pior caso do programa 32.2 é quadrático no tamanho da entrada. Note também que ambos os tempos de execução de melhor e de pior caso do programa 32.2, que implementa o método da ordenação da bolha, têm o mesmo desempenho assintótico.

## 32.4 Resumo

Um bom algoritmo ou programa é como uma faca afiada: faz o que é suposto fazer com a menor quantidade de esforço aplicada. Usar um programa errado para resolver um problema é como tentar cortar um bife com uma chave de fenda: podemos eventualmente obter um resultado digerível, mas gastaremos muito mais energia que o necessário e o resultado não deverá ser muito agradável esteticamente.

Programas alternativos projetados para resolver um mesmo problema em geral diferem dramaticamente em eficiência. Essas diferenças podem ser muito mais significativas que a diferença de tempos de execução em um supercomputador e em um computador pessoal. Como um exemplo, suponha que temos um supercomputador executando o programa que implementa o método da bolha de ordenação, com tempo de execução de pior caso $O(n^2)$, contra um pequeno computador pessoal executando o método da intercalação. Esse último método de ordenação tem tempo de execução de pior caso $O(n \log n)$ para qualquer entrada de tamanho $n$ e será visto na aula 43. Suponha que cada um desses programas deve ordenar um vetor de um milhão de números. Suponha também que o supercomputador é capaz de executar 100 milhões de operações por segundo enquanto que o computador pessoal é capaz de executar apenas um milhão de operações por segundo. Para tornar a diferença ainda mais dramática, suponha que o mais habilidoso dos programadores do mundo codificou o método da bolha na linguagem de máquina do supercomputador e o programa usa $2n^2$ operações para ordenar $n$ números inteiros. Por outro lado, o método da intercalação foi programado por um programador mediano usando uma linguagem de programação de alto nível com um compilador ineficiente e o código resultante gasta $50n \log n$ operações para ordenar os $n$ números. Assim, para ordenar um milhão de números o supercomputador gasta

$$\frac{2 \cdot (10^6)^2 \text{ operações}}{10^8 \text{ operações/segundo}} = 20.000 \text{ segundos} \approx 5{,}56 \text{ horas},$$

enquanto que o computador pessoal gasta

$$\frac{50 \cdot 10^6 \log 10^6 \text{ operações}}{10^6 \text{ operações/segundo}} \approx 1.000 \text{ segundos} \approx 16{,}67 \text{ minutos}.$$

Usando um programa cujo tempo de execução tem menor taxa de crescimento, mesmo com um compilador pior, o computador pessoal é 20 vezes mais rápido que o supercomputador!

Esse exemplo mostra que os algoritmos e os programas, assim como os computadores, são uma **tecnologia**. O desempenho total do sistema depende da escolha de algoritmos e programas eficientes tanto quanto da escolha de computadores rápidos. Assim como rápidos avanços estão sendo feitos em outras tecnologias computacionais, eles estão sendo feitos em algoritmos e programas também.

## Exercícios

32.1 Qual o menor valor de $n$ tal que um programa com tempo de execução $100n^2$ é mais rápido que um programa cujo tempo de execução é $2^n$, supondo que os programas foram implementados no mesmo computador?

32.2 Suponha que estamos comparando as implementações dos métodos de ordenação da bolha e da intercalação em um mesmo computador. Para entradas de tamanho $n$, o método da bolha gasta $8n^2$ passos enquanto que o método da intercalação gasta $64n \log n$ passos. Para quais valores de $n$ o método da bolha é melhor que o método da intercalação?

32.3 Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ na notação $O$.

32.4 Para cada função $f(n)$ e tempo $t$ na tabela abaixo determine o maior tamanho $n$ de um problema que pode ser resolvido em tempo $t$, considerando que o programa soluciona o problema em $f(n)$ microssegundos.

| | 1 segundo | 1 minuto | 1 hora | 1 dia | 1 mês | 1 ano | 1 século |
|---|---|---|---|---|---|---|---|
| $\log n$ | | | | | | | |
| $\sqrt{n}$ | | | | | | | |
| $n$ | | | | | | | |
| $n \log n$ | | | | | | | |
| $n^2$ | | | | | | | |
| $n^3$ | | | | | | | |
| $2^n$ | | | | | | | |
| $n!$ | | | | | | | |

32.5 É verdade que $2^{n+1} = O(2^n)$? E é verdade que $2^{2n} = O(2^n)$?

32.6 Considere o problema de computar o valor de um polinômio em um ponto. Dados $n$ coeficientes $a_0, a_1, \ldots, a_{n-1}$ e um número real $x$, queremos computar $\sum_{i=0}^{n-1} a_i x^i$.

(a) Escreva um programa simples com tempo de execução de pior caso $O(n^2)$ para solucionar este problema.

(b) Escreva um programa com tempo de execução de pior caso $O(n)$ para solucionar este problema usando o método chamado de regra de Horner para reescrever o polinômio:

$$\sum_{i=1}^{n-1} a_i x^i = (\cdots (a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0 \, .$$

32.7 Seja $A[0..n-1]$ um vetor de $n$ números inteiros distintos dois a dois. Se $i < j$ e $A[i] > A[j]$ então o par $(i, j)$ é chamado uma **inversão** de $A$.

(a) Liste as cinco inversões do vetor $A = \langle 2, 3, 8, 6, 1 \rangle$.

(b) Qual vetor com elementos no conjunto $\{1, 2, \ldots, n\}$ tem a maior quantidade de inversões? Quantas são?

(c) Escreva um programa que determine o número de inversões em qualquer permutação de $n$ elementos em tempo de execução de pior caso $O(n \log n)$.

AULA 33

# INTRODUÇÃO ÀS FUNÇÕES

Se concordamos que uma grande tarefa pode ser solucionada através de sua divisão em sub-tarefas e da combinação de suas soluções parciais, então podemos ter nosso trabalho facilitado focando na construção de soluções para essas sub-tarefas. Em programação essas soluções menores são chamadas de módulos de programação e fazem parte de todo processo de construção de programas para solução de problemas reais.

Os módulos são uma ferramenta da programação estruturada que fornecem ao(à) programador(a) um mecanismo para construir programas que são fáceis de escrever, ler, compreender, corrigir, modificar e manter. Na linguagem C, essa característica em um código é obtida através do uso de funções. Na verdade, todos os programas que codificamos até o momento de alguma forma já fizeram uso de funções. Por exemplo, `scanf` e `printf` são funções de entrada e saída de dados da linguagem C que já usamos muitas e muitas vezes. Além disso, temos construído nossas próprias funções `main` para solucionar todos os problemas vistos até aqui. Apesar do termo "função" vir da matemática, uma função da linguagem C nem sempre se parece com uma função matemática, já que pode não ter argumentos nem computar um valor. Cada função na linguagem C pode ser vista como um pequeno programa com suas próprias declarações de variáveis e sentenças de programação. Além de facilitar a compreensão e a modificação de um programa e evitar a duplicação de código usado mais que uma vez, as funções podem ser usadas não só no programa para o qual foram projetadas, mas também em outros programas.

## 33.1 Noções iniciais

Antes de aprender as regras formais para definir uma função, vejamos três pequenos exemplos de programas que definem funções.

Suponha que computamos freqüentemente a média de dois valores do tipo `double`. A biblioteca de funções da linguagem C não tem uma função que computa a média, mas podemos escrever facilmente a nossa própria função. Vejamos a seguir:

```
double media(double a, double b)
{
  return (a + b) / 2;
}
```

A palavra reservada `double` na primeira linha é o **tipo do valor devolvido** da função `media`, que é o tipo do valor devolvido pela função cada vez que ela é chamada. Os identificadores `a` e `b`, chamados de **parâmetros** da função, representam os dois números que serão fornecidos quando a função `media` é chamada. Assim como uma variável, um parâmetro deve ter um tipo. No exemplo, ambos os parâmetros `a` e `b` são do tipo `double`. Um parâmetro de uma função é essencialmente uma variável cujo valor inicial é fornecido quando da sua chamada.

Toda função tem um trecho executável, chamado de **corpo**, envolvido por abre e fecha chaves. O corpo de função `media` consiste de uma única sentença `return`. A execução dessa sentença faz com que a função regresse para o ponto de onde foi chamada. O valor `(a + b) / 2` será devolvido pela função.

Para chamar uma função, escrevemos o nome da função seguido por uma lista de **argumentos**. Por exemplo, `media(x, y)` é uma chamada da função `media`. Os argumentos são usados para fornecer informações para a função, como nesse caso, em que a função `media` necessita de dois valores para calcular a média. O efeito da chamada `media(x, y)` é primeiro copiar os valores de `x` e `y` nos parâmetros `a` e `b` e, em seguida, executar o corpo de `media`. Um argumento não tem de ser necessariamente uma variável. Qualquer expressão de um tipo compatível pode ser um argumento, o que nos permite escrever `media(5.1, 8.9)` ou ainda `media(x/2, y/3)`.

Devemos fazer uma chamada à função `media` no ponto do código onde usamos o seu valor devolvido. Por exemplo, para computar a média de `x` e `y`, e escrever o resultado na saída, poderíamos escrever:

```
printf("Média: %g\n", media(x, y));
```

A sentença acima tem o seguinte efeito:

1. a função `media` é chamada com argumentos `x` e `y`;
2. `x` e `y` são copiados em `a` e `b`;
3. a função `media` executa sua sentença `return`, devolvendo a média de `a` e `b`;
4. a função `printf` imprime a cadeia de caracteres e o valor que `media` devolve, sendo que esse valor devolvido torna-se um argumento da função `printf`.

Observe que o valor devolvido pela função `media` não é salvo em lugar algum. A sentença imprime o valor e então o descarta. Se há necessidade de usar o valor devolvido posteriormente, podemos capturá-lo em uma variável, como abaixo:

```
m = media(x, y);
```

A sentença acima chama a função `media` e então salva o valor devolvido na variável `m`.

Vamos usar a função `media` em um programa. O programa 33.1 tem três valores de entrada e computa a média desses valores dois a dois. Entre outras coisas, este programa mostra que uma função pode ser chamada tantas vezes quanto necessário.

Programa 33.1: Primeiro exemplo do uso de função.

```
1  #include <stdio.h>
2  double media(double a, double b)
3  {
4    return (a + b) / 2;
5  }
6  int main(void)
7  {
8    double x, y, z;
9    printf("Informe três valores: ");
10   scanf("%lf%lf%lf", &x, &y, &z);
11   printf("Média de %g e %g é %g\n", x, y, media(x, y));
12   printf("Média de %g e %g é %g\n", x, z, media(x, z));
13   printf("Média de %g e %g é %g\n", y, z, media(y, z));
14   return 0;
15 }
```

Observe que a definição da função `media` vem antes da definição da função `main`, já que teríamos problemas de compilação se fizéssemos o contrário.

Nem toda função devolve um valor como a função `media` faz. Por exemplo, uma função cujo trabalho é enviar informações para a saída muito provavelmente não devolve valor algum. Para indicar que uma função não tem valor devolvido, especificamos que seu valor devolvido é do tipo `void`, onde `void` é um tipo sem valor. Considere o seguinte exemplo, que imprime na saída a mensagem "$n$ e contando..." , onde $n$ é um valor fornecido quando a função é chamada:

```
void imprimeContador(int n)
{
  printf("%d e contando...", n);
  return;
}
```

A função `imprimeContador` tem um único parâmetro `n` do tipo `int` e não devolve valor algum. Nesse caso, especificamos `void` como o tipo do valor a ser devolvido e podemos opcionalmente omitir a sentença `return`. Dessa forma, a função `imprimeContador` também poderia ser descrita corretamente como abaixo:

```
void imprimeContador(int n)
{
  printf("%d e contando...\n", n);
}
```

Como a função `imprimeContador` não devolve um valor, não podemos chamá-la da mesma maneira que chamamos a função `media` no exemplo anterior. Ao contrário, qualquer chamada da função `imprimeContador` deve aparecer como uma única sentença, como no exemplo a seguir:

```
imprimeContador(i);
```

O programa 33.2 chama a função `imprimeContador` 10 vezes em uma estrutura de repetição `for`.

Programa 33.2: Segundo exemplo do uso de função.

```
1  #include <stdio.h>
2  void imprimeContador(int n)
3  {
4    printf("%d e contando...\n", n);
5    return;
6  }
7  int main(void)
8  {
9    int i;
10   for (i = 10; i > 0; i--)
11     imprimeContador(i);
12   return 0;
13 }
```

Inicialmente, a variável `i` tem o valor 10. Quando a função `imprimeContador` é chamada pela primeira vez, o valor em `i` é copiado no parâmetro `n` e a partir de então `n` também contém o valor 10. Como resultado, a primeira chamada da função `imprimeContador` imprimirá a seguinte mensagem:

```
10 e contando...
```

A função `imprimeContador` regressa então para o ponto em que foi chamada, o que acontece no corpo de uma estrutura de repetição `for`. A estrutura `for` continua sua execução decrementando `i` para 9 e verificando se seu valor é maior que 0. Como esse novo valor é maior que 0, `imprimeContador` é chamada novamente e imprime a mensagem abaixo:

```
9 e contando...
```

Cada vez que `imprimeContador` é chamada, o valor armazenado na variável `i` é diferente e, assim, `imprimeContador` imprimirá 10 mensagens diferentes na saída.

É importante salientar ainda que algumas funções podem também não ter parâmetros. Considere a função `imprimeMsg`, que imprime uma mensagem cada vez que é chamada:

```
void imprimeMsg(void)
{
  printf("Programar é bacana!\n");
  return;
}
```

A palavra reservada `void` entre parênteses após o identificador da função indica que `imprimeMsg` não tem argumentos. Para chamar uma função sem argumentos, devemos escrever o nome da função seguido obrigatoriamente por parênteses, como mostramos abaixo:

```
imprimeMsg();
```

O programa 33.3 mostra o uso da função `imprimeMsg`.

Programa 33.3: Terceiro exemplo do uso de função.

```
1  #include <stdio.h>
2  void imprimeMsg(void)
3  {
4    printf("Programar é bacana!\n");
5    return;
6  }
7  int main(void)
8  {
9    imprimeMsg();
10   return 0;
11 }
```

A execução do programa 33.3 começa com a primeira sentença na função `main`, que é a chamada à função `imprimeMsg`. Quando `imprimeMsg` começa sua execução, ela chama a função `printf` para mostrar uma mensagem na saída. Quando `printf` termina sua execução, `imprimeMsg` termina sua execução e volta para `main`.

## 33.2 Definição e chamada de funções

Uma **função** da linguagem C é um trecho de código que pode receber um ou mais valores de entrada armazenados em variáveis, chamados de **parâmetros (de entrada)**, que realiza algum processamento específico e que pode devolver um único valor de saída. Em geral, construímos funções para resolver pequenos problemas bem específicos e, em conjunto com outras funções, resolver problemas maiores e mais complexos. A definição de uma função na linguagem C fornece quatro informações importantes ao compilador:

1. quem pode chamá-la;
2. o tipo do valor que a função devolve;
3. seu identificador;
4. seus parâmetros de entrada.

Uma função é composta por:

- uma **interface**, que faz a comunicação entre a função e o meio exterior; a interface é definida pela primeira linha da função, onde especificamos o tipo do valor devolvido da função, seu identificador e a lista de parâmetros de entrada separados por vírgulas e envolvidos por um par de parênteses; e
- um **corpo**, que realiza o processamento sobre os valores armazenados nos parâmetros de entrada, e também nas variáveis declaradas internamente à função, e devolve um valor na saída; o corpo de uma função é composto pela declaração de variáveis locais da função e sua lista de comandos.

Depois de ter visto vários exemplos de funções na seção anterior, a forma geral de uma **definição de uma função** é dada a seguir:

```
tipo identificador(parâmetros)
{
  declaração de variáveis
  sentenças
}
```

A primeira linha apresentada é a interface da função e as linhas seguintes, envolvidas por chaves, compõem o corpo da função. A interface da função inicia com um `tipo`, que especifica

o tipo do valor a ser devolvido pela função. Funções não podem devolver variáveis compostas homogêneas, mas não há qualquer outra restrição quanto ao tipo de valor a ser devolvido. No entanto, especificar que o valor devolvido é do tipo `void` indica que a função não devolve um valor. Depois, a interface contém o `identificador` da função, que é seu nome e que a identifica e diferencia de outras funções. E, finalmente, a interface contém os `parâmetros`, que armazenam valores a serem recebidos como entrada pela função, envolvidos por parênteses. A lista de parâmetros é composta por tipos e identificadores de variáveis, separados por vírgulas. Um tipo deve ser especificado para cada parâmetro, mesmo que vários parâmetros sejam do mesmo tipo. Por exemplo, é errado escrever a função `media` como abaixo:

```
double media(double a, b)
{
  return (a + b) / 2;
}
```

Se a função não tem parâmetros, a palavra reservada `void` deve ocorrer entre os parênteses.

Após a interface, a função contém um corpo, que é uma seqüência de comandos da linguagem C envolvida por chaves. O corpo de uma função pode incluir declarações e sentenças. Por exemplo, a função `media` poderia ser escrita como abaixo:

```
double media(double a, double b)
{
  double soma;
  soma = a + b;
  return soma / 2;
}
```

A declaração de variáveis de uma função deve vir primeiro, antes de qualquer sentença do corpo da função. As variáveis declaradas no corpo de uma função pertencem exclusivamente àquela função e não podem ser examinadas ou modificadas por outras funções.

Uma **chamada de função** consiste de um identificador da função seguido por uma lista de argumentos entre parênteses. Por exemplo, abaixo temos três chamadas de funções:

```
media(x, y)
imprimeContador(i)
imprimeMsg()
```

Uma chamada de uma função com valor devolvido `void` é sempre seguida por um `;` para torná-la uma sentença. Por exemplo:

```
imprimeContador(i);
imprimeMsg();
```

Por outro lado, uma chamada de uma função com valor devolvido diferente de `void` produz um valor que pode ser armazenado em uma variável ou usado em uma expressão aritmética, relacional ou lógica. Por exemplo,

```
if (media(x, y) > 0)
  printf("Média é positiva\n");
printf("A média é %g\n", media(x, y));
```

O valor devolvido por uma função que devolve um valor diferente de `void` sempre pode ser descartado, como podemos ver na chamada a seguir:

```
media(x, y);
```

Esta chamada à função `media` é um exemplo de uma sentença que avalia uma expressão mas descarta o resultado. Apesar de estranho na chamada à função `media`, ignorar o valor devolvido pode fazer sentido em alguns casos. Por exemplo, a função `printf` sempre devolve o número de caracteres impressos na saída. Dessa forma, após a chamada abaixo, a variável `nc` terá o valor 19:

```
nc = printf("Programar é bacana\n");
```

Como em geral não estamos interessados no número de caracteres impressos, normalmente descartamos o valor devolvido pela função `printf` e fazemos:

```
printf("Programar é bacana\n");
```

Uma função que devolve um valor diferente de `void` deve usar a sentença `return` para especificar qual valor será devolvido. A sentença `return` tem a seguinte forma geral:

```
return expressão;
```

A `expressão` em geral é uma constante ou uma variável, como abaixo:

```
return 0;
return estado;
```

Expressões mais complexas são possíveis, como por exemplo:

```
return (a + b) / 2;
return a*a*a + 2*a*a - 3*a - 1;
```

Quando uma das sentenças acima é executada, primeiramente a avaliação da expressão é realizada e o valor obtido é então devolvido pela função.

Se o tipo da expressão na sentença `return` é diferente do tipo devolvido pela função, a expressão será implicitamente convertida para o tipo da função.

## 33.3 Finalização de programas

Como `main` é uma função, ela deve ter um tipo de valor de devolução. Normalmente, o tipo do valor a ser devolvido pela função `main` é `int` e é por isso que temos definido `main` da forma como fizemos até aqui.

O valor devolvido pela função `main` é um código de estado do programa que, em alguns sistemas operacionais, pode ser verificado quando o programa termina. A função `main` deve devolver 0 se o programa termina normalmente ou um valor diferente de 0 para indicar um término anormal. Na verdade, não há uma regra para o uso do valor devolvido pela função `main` e, assim, podemos usar esse valor para qualquer propósito. É uma boa prática de programação fazer com que todo programa na linguagem C devolva um código de estado, mesmo se não há planos de usá-lo, já que um(a) usuário(a) pode decidir verificá-lo.

Executar a sentença `return` na função `main` é uma forma de terminar um programa. Outra maneira é chamar a função `exit` que pertence à biblioteca `stdlib`. O argumento passado para `exit` tem o mesmo significado que o valor a ser devolvido por `main`, isto é, ambos indicam o estado do programa ao seu término. Para indicar um término normal, passamos `0` (zero) como argumento, como abaixo:

```
exit(0);
```

Como o argumento `0` não é muito significativo, a linguagem C nos permite usar o argumento `EXIT_SUCCESS`, que tem o mesmo efeito:

```
exit(EXIT_SUCCESS);
```

O argumento `EXIT_FAILURE` indica término anormal:

```
exit(EXIT_FAILURE);
```

`EXIT_SUCCESS` e `EXIT_FAILURE` são macros definidas na `stdlib.h`. Os valores associados são definidos pela arquitetura do computador em uso, mas em geral são `0` e `1`, respectivamente.

Como métodos para terminar um programa, `return` e `exit` são muito semelhantes. Ou seja, a sentença

```
return expressão;
```

na função `main` é equivalente à sentença

```
exit(expressão);
```

A diferença entre `return` e `exit` é que `exit` causa o término de um programa independente do ponto em que se encontra essa chamada, isto é, não considerando qual função chamou a função `exit`. A sentença `return` causa o término de um programa apenas se aparece na função `main`. Alguns(mas) programadores(as) usam `exit` exclusivamente para depurar programas.

## 33.4 Exemplo

Vamos resolver agora o seguinte problema:

> Construa uma função que receba dois inteiros positivos e devolva o máximo divisor comum entre eles.

Já resolvemos o problema de encontrar o máximo divisor comum de dois números inteiros positivos antes, usando o algoritmo de Euclides, mas sem o uso de uma função. Uma possível função que soluciona o problema acima é mostrada a seguir.

```
int mdc(int a, int b)
{
  int aux;
  while (b != 0) {
    aux = a % b;
    a = b;
    b = aux;
  }
  return a;
}
```

Um exemplo de um programa principal que faz uma chamada à função `mdc` descrita acima é mostrado no programa 33.4.

Programa 33.4: Exemplo de um programa com uma função `mdc`.

```
1  #include <stdio.h>
2  int mdc(int a, int b)
3  {
4    int aux;
5    while (b != 0) {
6      aux = a % b;
7      a = b;
8      b = aux;
9    }
10   return a;
11 }
12 int main(void)
13 {
14   int x, y;
15   printf("Informe dois valores: ");
16   scanf("%d%d", &x, &y);
17   printf("O mdc entre %d e %d é %d.", x, y, mdc(x, y));
18   return 0;
19 }
```

## 33.5 Declaração de funções

Nos programas que vimos nas seções 33.1 e 33.4 a definição de cada função sempre foi feita *acima* do ponto onde ocorrem suas chamadas. Ou seja, se a função `main` faz chamada a uma função escrita pelo programador, a definição dessa função tem de ocorrer antes dessa chamada, acima da própria função `main`. No entanto, a linguagem C na verdade não obriga que a definição de uma função preceda suas chamadas.

Suponha que rearranjamos o programa 33.1 colocando a definição da função `media` *depois* da definição da função `main`, como abaixo:

```
1  #include <stdio.h>
2  int main(void)
3  {
4    double x, y, z;
5    printf("Informe três valores: ");
6    scanf("%lf%lf%lf", &x, &y, &z);
7    printf("Média de %g e %g é %g\n", x, y, media(x, y));
8    printf("Média de %g e %g é %g\n", x, z, media(x, z));
9    printf("Média de %g e %g é %g\n", y, z, media(y, z));
10   return 0;
11 }
12 double media(double a, double b)
13 {
14   return (a + b) / 2;
15 }
```

Quando o compilador encontra a primeira chamada da função `media` na linha 7 da função `main`, ele não tem informação alguma sobre `media`: não conhece quantos parâmetros `media` tem, quais os tipos desses parâmetros e qual o tipo do valor que `media` devolve. Ao invés de uma mensagem de erro, o compilador faz uma tentativa de declaração, chamada de **declaração implícita**, da função `media` e, em geral, nos passos seguintes da compilação, um ou mais erros decorrem dessa declaração: um erro do tipo do valor devolvido, do número de parâmetros ou do tipo de cada parâmetro.

Uma forma de evitar erros de chamadas antes da definição de uma função é dispor o programa de maneira que a definição da função preceda todas as suas chamadas. Infelizmente, nem sempre é possível arranjar um programa dessa maneira e, mesmo quando possível, pode tornar mais difícil sua compreensão já que as definições das funções serão dispostas em uma ordem pouco natural.

Felizmente, a linguagem C oferece uma solução melhor, com a declaração de uma função antes de sua chamada. A **declaração de uma função** fornece ao compilador uma visão inicial da função cuja declaração completa será dada posteriormente. A declaração de uma função é composta exatamente pela primeira linha da definição de uma função com um ponto e vírgula adicionado no final:

```
tipo identificador(parâmetros);
```

Desnecessário dizer que a declaração de uma função tem de ser consistente com a definição da mesma função. A seguir mostramos como nosso programa ficaria com a adição da declaração de `media`:

```c
#include <stdio.h>
double media(double a, double b);    /* declaração */
int main(void)
{
  double x, y, z;
  printf("Informe três valores: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Média de %g e %g é %g\n", x, y, media(x, y));
  printf("Média de %g e %g é %g\n", x, z, media(x, z));
  printf("Média de %g e %g é %g\n", y, z, media(y, z));
  return 0;
}
double media(double a, double b)     /* definição */
{
  return (a + b) / 2;
}
```

A declaração de uma função também é conhecida como **protótipo da função**. Um protótipo de uma função fornece ao compilador uma descrição completa de como chamar a função: quantos argumentos fornecer, de quais tipos esses argumentos devem ser e qual o tipo do resultado a ser devolvido.

## Exercícios

33.1 (a) Escreva uma função com a seguinte interface:

```c
double areaTriangulo(double base, double altura)
```

que receba dois números de ponto flutuante que representam a base e a altura de um triângulo e compute e devolva a área desse triângulo.

(b) Escreva um programa que receba uma seqüência de $n$ pares de números de ponto flutuante, onde cada par representa a base e a altura de um triângulo, e calcule e escreva, para cada par, a área do triângulo correspondente. Use a função descrita no item (a).

Programa 33.5: Solução do exercício 33.1.

```c
#include <stdio.h>
double areaTriangulo(double base, double altura)
{
  return (base * altura) / 2;
}
int main(void)
{
  int i, n;
  double b, a;
  printf("Informe n: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    printf("Informe a base e a altura do triângulo: ");
    scanf("%lf%lf", &b, &a);
    printf("Área do triângulo: %g\n", areaTriangulo(b, a));
  }
  return 0;
}
```

33.2 (a) Escreva uma função com a seguinte interface:

```c
int mult(int a, int b)
```

que receba dois números inteiros positivos $a$ e $b$ e determine e devolva um valor que representa o produto desses números, usando o seguinte método de multiplicação:

i. dividir, sucessivamente, o primeiro número por 2, até que se obtenha 1 como quociente;
ii. em paralelo, dobrar, sucessivamente, o segundo número;
iii. somar os números da segunda coluna que tenham um número ímpar na primeira coluna; o total obtido é o produto procurado.

Por exemplo, para os números 9 e 6, temos que $9 \times 6$ é

| | | | |
|---|---|---|---|
| 9 | 6 | $\rightarrow$ | 6 |
| 4 | 12 | | |
| 2 | 24 | | |
| 1 | 48 | $\rightarrow$ | 48 |
| | | | 54 |

(b) Escreva um programa que leia $n \geqslant 1$ pares de números e calcule os respectivos produtos desses pares, usando a função do item (a).

33.3 Para determinar o número de lâmpadas necessárias para cada aposento de uma residência, existem normas que fornecem o mínimo de potência de iluminação exigida por metro quadrado ($m^2$) conforme o uso desse ambiente. Suponha que só temos lâmpadas de 60 watts para uso.

Seja a seguinte tabela de informações sobre possíveis aposentos de uma residência:

| Utilização | Classe | Potência/m$^2$ (W) |
|---|---|---|
| quarto | 1 | 15 |
| sala de TV | 1 | 15 |
| salas | 2 | 18 |
| cozinha | 2 | 18 |
| varandas | 2 | 18 |
| escritório | 3 | 20 |
| banheiro | 3 | 20 |

(a) Escreva uma função com a seguinte interface:

```
int numLampadas(int classe, double a, double b)
```

que receba um número inteiro representando a classe de iluminação de um aposento e dois números com ponto flutuante representando suas duas dimensões e devolva um número inteiro representando o número de lâmpadas necessárias para iluminar adequadamente o aposento.

(b) Escreva um programa que receba uma seqüência de informações contendo o nome do aposento da residência, sua classe de iluminação e as suas dimensões e, usando a função `numLampadas`, imprima a área de cada aposento, sua potência de iluminação e o número total de lâmpadas necessárias para o aposento. Além disso, seu programa deve calcular o total de lâmpadas necessárias e a potência total necessária para a residência toda.

Suponha que o término se dá quando o nome do aposento informado é `fim`.

AULA 34

# EXERCÍCIOS

34.1 (a) Escreva uma função com interface

```
int mdc(int a, int b)
```

que receba dois números inteiros positivos $a$ e $b$ e calcule e devolva o máximo divisor comum entre eles.

(b) Usando a função do item anterior, escreva um programa que receba $n \geqslant 1$ números inteiros positivos e calcule o máximo divisor comum entre todos eles.

Programa 34.1: Solução do exercício 34.1.

```
#include <stdio.h>
int mdc(int a, int b)
{
  int aux;
  while (b != 0) {
    aux = a % b;
    a = b;
    b = aux;
  }
  return a;
}
int main(void)
{
  int n, i, num, result;
  scanf("%d", &n);
  scanf("%d", &result);
  for (i = 2; i <= n; i++) {
    scanf("%d", &num);
    result = mdc(result, num);
  }
  printf("%d\n", result);
  return 0;
}
```

34.2 (a) Escreva uma função com interface

```
int verificaPrimo(int p)
```

que receba um número inteiro positivo $p$ e verifique se $p$ é primo, devolvendo $1$ em caso positivo e $0$ em caso negativo.

(b) Usando a função do item anterior, escreva um programa que receba $n \geqslant 1$ números inteiros positivos e calcule a soma dos que são primos.

34.3 Um número inteiro $a$ é dito ser **permutação** de um número inteiro $b$ se os dígitos de $a$ formam uma permutação dos dígitos de $b$.

Exemplo:

5412434 é uma permutação de 4321445, mas não é uma permutação de 4312455.

*Observação:* considere que o dígito 0 (zero) não ocorre nos números.

(a) Escreva uma função com interface

```
int contaDigitos(int n, int d)
```

que receba dois números inteiros $n$ e $d$, com $0 < d \leqslant 9$, devolva um valor que representa o número de vezes que o dígito $d$ ocorre no número $n$.

(b) Usando a função do item anterior, escreva um programa que leia dois números inteiros positivos $a$ e $b$ e responda se $a$ é permutação de $b$.

34.4 (a) Escreva uma função com interface

```
int sufixo(int a, int b)
```

que receba dois números inteiros $a$ e $b$ e verifique se $b$ é um sufixo de $a$. Em caso positivo, a função deve devolver $1$; caso contrário, a função deve devolver $0$.
Exemplo:

| $a$ | $b$ | | |
|---|---|---|---|
| 567890 | 890 | $\rightarrow$ | sufixo |
| 1234 | 1234 | $\rightarrow$ | sufixo |
| 2457 | 245 | $\rightarrow$ | não é sufixo |
| 457 | 2457 | $\rightarrow$ | não é sufixo |

(b) Usando a função do item anterior, escreva um programa que leia dois números inteiros $a$ e $b$ e verifique se o menor deles é subseqüência do outro.
Exemplo:

| $a$ | $b$ | | |
|---|---|---|---|
| 567890 | 678 | $\rightarrow$ | $b$ é subseqüência de $a$ |
| 1234 | 2212345 | $\rightarrow$ | $a$ é subseqüência de $b$ |
| 235 | 236 | $\rightarrow$ | um não é subseqüência do outro |

34.5 Uma seqüência de $n$ números inteiros não nulos é dita $m$**-alternante** se é constituída por $m$ segmentos: o primeiro com um elemento, o segundo com dois elementos e assim por diante até o $m$-ésimo, com $m$ elementos. Além disso, os elementos de um mesmo segmento devem ser todos pares ou todos ímpares e para cada segmento, se seus elementos forem todos pares (ímpares), os elementos do segmento seguinte devem ser todos ímpares (pares).

Por exemplo:

- A seqüência com $n = 10$ elementos: $\underline{8}\ \underline{3\ 7}\ \underline{2\ 10\ 4}\ \underline{5\ 13\ 9\ 11}$ é 4-alternante.
- A seqüência com $n = 3$ elementos: $\underline{7}\ \underline{2\ 8}$ é 2-alternante.
- A seqüência com $n = 8$ elementos: $\underline{1}\ \underline{12\ 4}\ \underline{3\ 13\ 5}\ \underline{8\ 6}$ não é alternante, pois o último segmento não tem tamanho 4.

(a) Escreva uma função com interface

```
int bloco(int m)
```

que receba como parâmetro um inteiro $m$ e leia $m$ números inteiros, devolvendo um dos seguintes valores:

$0$, se os $m$ números lidos forem pares;
$1$, se os $m$ números lidos forem ímpares;
$-1$, se entre os $m$ números lidos há números com paridades diferentes.

(b) Usando a função do item anterior, escreva um programa que, dados um inteiro $n$, com $n \geqslant 1$, e uma seqüência de $n$ números inteiros, verifica se a seqüência é $m$-alternante. O programa deve imprimir o valor de $m$ ou exibir uma mensagem informando que a seqüência não é alternante.

AULA 35

# ARGUMENTOS E PARÂMETROS DE FUNÇÕES

Como vimos até aqui, a interface de uma função define muito do que queremos saber sobre ela: o tipo de valor que a função devolve, seu identificador e seus parâmetros. Isso implica na conseqüente elevação do nível de abstração dos nossos programas, já que podemos entender o que um programa faz sem a necessidade de examinar os detalhes de suas funções. Caso alguns detalhes sejam de nosso interesse, também há a vantagem de que sabemos onde examiná-los. Nesta aula focaremos nos argumentos e parâmetros das funções e no escopo de seus dados.

## 35.1 Argumentos e parâmetros

Uma diferença importante entre um parâmetro e um argumento deve ser destacada aqui. Parâmetros aparecem na definição de uma função e são nomes que representam valores a serem fornecidos quando a função é chamada. Já os argumentos são expressões que aparecem nas chamadas das funções.

Conceitualmente, existem três tipos de parâmetros:

- **de entrada**, que permitem que valores sejam passados *para* a função;
- **de saída**, que permite que um valor seja devolvido *da* função;
- **de entrada e saída**, que permitem que valores sejam passados *para* a função e devolvidos *da* função.

Até o momento, entramos em contato com parâmetros de entrada e parâmetros de saída nos programas que já fizemos. Os valores dos parâmetros de entrada são passados para uma função através de um mecanismo denominado **passagem por cópia** ou **passagem por valor**. Por exemplo, na chamada

```
x = quadrado(num);
```

o valor da expressão `num` é um argumento da função `quadrado` e é atribuído ao parâmetro correspondente da função. A passagem desse valor é feita por cópia, ou seja, o valor é copiado no parâmetro correspondente da função `quadrado`.

Já os valores dos parâmetros de entrada e saída são passados/devolvidos por um mecanismo chamado **referência**. Nesse caso, temos uma variável especificada na chamada da função e um parâmetro especificado na interface da função que compartilham a mesma área de armazenamento na memória e isso significa que qualquer alteração realizada no conteúdo do parâmetro dentro da função acarreta alteração no conteúdo da variável que é o argumento da chamada. Vejamos um exemplo no programa 35.1.

Programa 35.1: Exemplo de passagem de parâmetros por referência.

```
1  #include <stdio.h>
2  void decompoe(float x, int *parte_int, float *parte_frac)
3  {
4    *parte_int = (int) x;
5    *parte_frac = x - *parte_int;
6    return;
7  }
8  int main(void)
9  {
10   float num, b;
11   int a;
12   printf("Informe um número de ponto flutuante: ");
13   scanf("%f", &num);
14   decompoe(num, &a, &b);
15   printf("Número: %f\n",  num);
16   printf("Parte inteira: %d\n", a);
17   printf("Parte fracionária: %f\n", b);
18   return 0;
19 }
```

Uma primeira observação importante é sobre a interface da função `decompoe`:

```
void decompoe(float x, int *parte_int, float *parte_frac)
```

Note que os parâmetros `parte_int` e `parte_frac` vêem precedidos do símbolo `*`. Essa é a forma que definimos parâmetros de entrada e saída na linguagem C. Observe agora o corpo da função `decompoe`:

```
{
  *parte_int = (int) x;
  *parte_frac = x - *parte_int;
  return;
}
```

Os parâmetros de entrada e saída são usados como de costume no corpo de uma função, isto é, como vimos usando os parâmetros de entrada. No entanto, os parâmetros de entrada e saída sempre devem vir precedidos pelo símbolo `*` no corpo de uma função, como mostrado acima. E por último, a chamada da função `decompoe` na função `main` é ligeiramente diferente, como destacado abaixo:

```
decompoe(num, &a, &b);
```

Observe que os argumentos `a` e `b` são precedidos pelo símbolo `&`, indicando que as variáveis `a` e `b` da função `main`, de tipos inteiro e ponto flutuante respectivamente, são passadas como parâmetros de entrada e saída para a função `decompoe`, ou seja, são passadas por referência, e qualquer alteração realizada nos parâmetros correspondentes da função refletirão nos argumentos da chamada.

## 35.2 Escopo de dados e de funções

O **escopo** de uma função e das variáveis de uma função é o trecho, ou os trechos, de código em que a função ou as variáveis pode(m) ser acessada(s). Em particular, o escopo de uma função determina quais funções podem chamá-la e quais ela pode chamar.

Variáveis definidas internamente nas funções, bem como os parâmetros de uma função, são chamadas de **variáveis locais** da mesma, ou seja, o escopo dessas variáveis circunscreve-se ao trecho de código definido pelo corpo da função. Isso significa que elas são criadas durante a execução da função, manipuladas e destruídas ao término de sua execução, quando o comando `return` é encontrado.

As funções de um programa são organizadas por níveis. No primeiro nível temos apenas a função principal `main`, aquela pela qual o programa será iniciado. As funções que são chamadas pela função `main` são ditas estarem no segundo nível. No terceiro nível estão as funções que são chamadas pelas funções do segundo nível. E assim sucessivamente. Veja a figura 35.1.

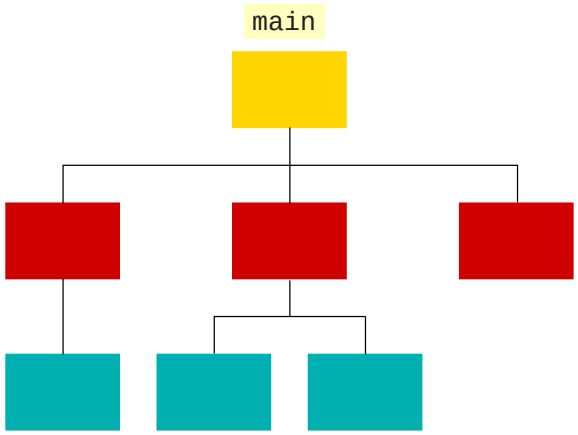



Figura 35.1: Um esquema de um programa contendo funções distribuídas em três níveis.

## Exercícios

35.1 (a) Escreva uma função com a seguinte interface:

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

que receba dois números inteiros $a$ e $b$ e troque os seus conteúdos.

(b) Usando a função `troca` definida acima, escreva um programa que leia um vetor contendo $n$ números inteiros, com $1 \leqslant n \leqslant 100$, ordene seus elementos em ordem não decrescente usando o método das trocas sucessivas e imprima o vetor resultante na saída.

(c) Repita a letra (b) implementando o método da seleção.

(d) Repita a letra (b) implementando o método da inserção.

Programa 35.2: Solução do exercício 35.1(b).

```
#include <stdio.h>
#define MAX 100
void troca(int *a, int *b)
{
  int aux;
  aux = *a;
  *a = *b;
  *b = aux;
  return;
}
int main(void)
{
  int i, j, n, A[MAX];
  scanf("%d", &n);
  for (i = 0; i < n; i++)
    scanf("%d", &A[i]);
  for (i = n-1; i > 0; i--)
    for (j = 0; j < i; j++)
      if (A[j] > A[j+1])
        troca(&A[j], &A[j+1]);
  for (i = 0; i < n; i++)
    printf("%d ", A[i]);
  printf("\n");
  return 0;
}
```

35.2 Em um dado país a moeda corrente possui apenas quatro cédulas de papel: \$1, \$5, \$10 e \$20.

(a) Escreva uma função com a seguinte interface:

```
void cedulas(int valor, int *um, int *cinco, int *dez, int *vinte)
```

que receba um valor em dinheiro e determine a menor quantidade de cédulas de 1, 5, 10 e 20 necessárias para pagar o valor especificado.

(b) Escreva um programa que dado um valor na moeda corrente determine a menor quantidade de cédulas para pagar esse valor. Use a função do item (a).

35.3 Dizemos que um número natural $n$ é **palíndromo** se lido da esquerda para direita e da direita para esquerda é o mesmo número.

Exemplos:

567765 é palíndromo.
32423 é palíndromo.
567675 não é palíndromo.

(a) Escreva uma função com a seguinte interface:

```
void quebra(int n, int *prim, int *ult, int *miolo)
```

que receba um número do tipo inteiro $n > 0$ e devolva três valores do tipo inteiro: o primeiro dígito de $n$, o último dígito de $n$ e um inteiro que represente o número $n$ sem seu primeiro e último dígitos.

Exemplo:

| valor inicial de $n$ | primeiro dígito | último dígito | miolo de $n$ |
|---|---|---|---|
| 732 | 7 | 2 | 3 |
| 14738 | 1 | 8 | 473 |
| 78 | 7 | 8 | 0 |
| 7 | 7 | 7 | 0 |

(b) Usando a função do item (a), escreva um programa que receba um inteiro positivo $n > 0$ e verifique se $n$ é palíndromo. Suponha que $n$ não contém o dígito 0.

35.4 (a) Escreva uma função com a seguinte interface:

```
int divisao(int *m, int *n, int d)
```

que receba três valores positivos do tipo inteiro $m, n$ e $d$ e devolva 1 se $d$ divide $m$, $n$ ou ambos, e 0, caso contrário. Além disso, em caso positivo, a função deve devolver um valor que representa o quociente da divisão de $m$ por $d$ e outro valor que representa o quociente da divisão de $n$ por $d$.

(b) Escreva um programa que leia dois inteiros positivos $m$ e $n$ e calcule, usando a função do item (a), o mínimo múltiplo comum entre $m$ e $n$.

35.5 (a) Escreva uma função com a seguinte interface:

```
int somabit(int b1, int b2, int *vaium)
```

que receba três valores inteiros e devolva um valor do tipo inteiro representando a soma dos três bits, e devolva também um outro valor do tipo inteiro representando o valor do vai um.

(b) Escreva um programa que leia dois números no sistema binário de numeração e, usando a função do item (a), calcule um número em binário que é a soma dos dois números dados.

AULA 36

# FUNÇÕES E VETORES

Nas aulas 33, 34 e 35 vimos funções, argumentos de funções e passagem de parâmetros para funções na linguagem C, onde trabalhamos apenas com argumentos de tipos básicos. Observe que chegamos a passar o valor de um elemento de um vetor como um argumento para uma função, por cópia e por referência. Nesta aula veremos como realizar passagem de parâmetros de tipos complexos, mais especificamente de parâmetros que são variáveis compostas homogêneas unidimensionais ou vetores.

## 36.1 Vetores como argumentos de funções

Assim como fizemos com variáveis de tipos básicos, é possível passar um vetor todo como argumento em uma chamada de uma função. O parâmetro correspondente deve ser um vetor com a mesma dimensão. Vejamos um exemplo no programa 36.1.

Programa 36.1: Um programa com uma função que tem um vetor como parâmetro.

```
1  #include <stdio.h>
2  int minimo(int A[10])
3  {
4    int i, min;
5    min = A[0];
6    for (i = 1; i < 10; i++)
7      if (A[i] < min)
8        min = A[i];
9    return min;
10 }
11 int main(void)
12 {
13   int i, vet[10];
14   for (i = 0; i < 10; i++)
15     scanf("%d", &vet[i]);
16   printf("O menor valor do conjunto é %d\n", minimo(vet));
17   return 0;
18 }
```

Observe a interface da função `minimo` do programa acima. Essa interface indica que a função devolve um valor do tipo inteiro, tem identificador `minimo` e tem como seu argumento um vetor contendo 10 elementos do tipo inteiro. Referências feitas ao parâmetro `A`, no interior da função `minimo`, acessam os elementos apropriados dentro do vetor que foi passado à função. Para passar um vetor inteiro para uma função é necessário apenas descrever o identificador do vetor, sem qualquer referência a índices, na chamada da função. Essa situação pode ser visualizada na última linha do programa principal, na chamada da função `minimo(pontos)`.

É importante destacar também que a única restrição de devolução de uma função é relativa às variáveis compostas homogêneas. De fato, um valor armazenado em uma variável de qualquer tipo pode ser devolvido por uma função, excluindo variáveis compostas homogêneas.

## 36.2 Vetores são parâmetros passados por referência

Nesta aula, o que temos visto até o momento são detalhes de como passar vetores como parâmetros para funções, detalhes na chamada e na interface de uma função que usa vetores como parâmetros. No entanto, esses elementos por si só não são a principal peculiaridade dos vetores neste contexto. No programa 36.2 a seguir iremos perceber a verdadeira diferença do uso de vetores como parâmetros em funções.

Programa 36.2: Um vetor é sempre um parâmetro passado por referência para uma função.

```
1  #include <stdio.h>
2  void dobro(int vetor[100], int n)
3  {
4    int i;
5    for (i = 0; i < n; i++)
6      vetor[i] = vetor[i] * 2;
7    return;
8  }
9  int main(void)
10 {
11   int i, n, valor[100];
12   printf("Informe a quantidade de elementos: ");
13   scanf("%d", &n);
14   for (i = 0; i < n; i++) {
15     printf("Informe o %d-ésimo valor: ", i+1);
16     scanf("%d", &valor[i]);
17   }
18   dobro(valor, n);
19   printf("Resultado: ");
20   for (i = 0; i < n; i++)
21     printf("%d ", valor[i]);
22   printf("\n");
23   return 0;
24 }
```

Perceba que a função `dobro` do programa acima modifica os valores do vetor `valor`, uma variável local do programa principal que é passada como parâmetro para a função `dobro` na variável `vetor`, uma variável local da função `dobro`. É importante observar que, quando usamos vetores como argumentos, uma função que modifica o valor de um elemento do vetor, modifica também o vetor original que foi passado como parâmetro para a função. Esta modificação tem efeito mesmo após o término da execução da função.

Nesse sentido, podemos dizer que um vetor, uma matriz ou uma variável composta homogênea de qualquer dimensão é *sempre* um parâmetro de entrada e saída, isto é, é sempre passado por referência a uma função.

No entanto, lembre-se que a modificação de elementos de um vetor em uma função se aplica somente quando o vetor completo é passado como parâmetro à função e não elementos individuais do vetor. Esse último caso já foi tratado em aulas anteriores.

## 36.3 Vetores como parâmetros com dimensões omitidas

Na definição de uma função, quando um de seus parâmetros é um vetor, seu tamanho pode ser deixado sem especificação. Essa prática de programação é freqüente e permite que a função torne-se ainda mais geral. Dessa forma, a função `dobro`, por exemplo, definida na seção anterior, poderia ser reescrita como a seguir:

```
void dobro(int vetor[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    vetor[i] = vetor[i] * 2;
  return;
}
```

Observe que a dimensão do parâmetro `vetor` da função `dobro`, que é um vetor, foi omitida. A duas definições da função `dobro`, nesta seção e na seção anterior, são equivalentes. No entanto, a definição com a dimensão omitida permite que a função `dobro` possa ser chamada com vetores de quaisquer dimensões como argumentos.

Por exemplo, se temos dois vetores de inteiros $A$ e $B$, com dimensões 50 e 200, respectivamente, a função `dobro` pode ser chamada para computar o dobro de cada coordenada do vetor $A$ e, do mesmo modo, também pode se chamada para computar o dobro de cada coordenada do vetor $B$. Ou seja, a função `dobro` pode ser chamada com vetores de dimensões diferentes como parâmetros.

```
dobro(A, 50);
dobro(B, 200);
```

## Exercícios

36.1 (a) Escreva uma função com a seguinte interface:

```
int subconjunto(int A[MAX], int m, int B[MAX], int n)
```

que receba como parâmetros um vetor $A$ de números inteiros com $m$ elementos e um vetor $B$ de números inteiros com $n$ elementos, ambos representando conjuntos, e verifica se $A$ está contido em $B$ ($A \subset B$).

(b) Escreva um programa que receba dois vetores de números inteiros $U$ e $W$, com $u$ e $w$ elementos respectivamente, $1 \leqslant u, w \leqslant 100$, e verifique se os dois conjuntos são iguais ($U = W$ se e somente se $U \subset W$ e $W \subset U$). Use a função do item (a).

Programa 36.3: Solução do exercício 36.1.

```
#include <stdio.h>
#define MAX 100
int subconjunto(int A[MAX], int m, int B[MAX], int n)
{
  int i, j, contido;
  contido = 1;
  for (i = 0; i < m && contido; i++) {
    for (j = 0; j < n && B[j] != A[i]; j++)
      ;
    if (j == n)
      contido = 0;
  }
  return contido;
}
int main(void)
{
  int tamU, i, U[MAX], tamW, j, W[MAX];
  scanf("%d", &tamU);
  for (i = 0; i < tamU; i++)
    scanf("%d", &U[i]);
  scanf("%d", &tamW);
  for (j = 0; j < tamW; j++)
    scanf("%d", &W[j]);
  if (subconjunto(U, tamU, W, tamW) && subconjunto(W, tamW, U, tamU))
    printf("U == W\n");
  else
    printf("U != W\n");
  return 0;
}
```

36.2 Em um programa na linguagem C, um conjunto pode ser representado por um vetor da seguinte forma: `V[0]` contém o número de elementos do conjunto; `V[1], V[2], ...` são os elementos do conjunto, sem repetições.

(a) Escreva uma função com a seguinte interface:

```
void intersec(int A[MAX+1], int B[MAX+1], int C[MAX+1])
```

que dados dois conjuntos de números inteiros $A$ e $B$, construa um terceiro conjunto $C$ tal que $C = A \cap B$. Lembre-se de que em `C[0]` a sua função deve colocar o tamanho da intersecção.

(b) Escreva um programa que leia um número inteiro $n \geqslant 2$ e uma seqüência de $n$ conjuntos de números inteiros, cada um com no máximo 100 elementos, e construa e imprima um vetor que representa a intersecção dos $n$ conjuntos.

*Observação:* não leia todos os conjuntos de uma só vez. Leia os dois primeiros conjuntos e calcule a primeira intersecção. Depois, leia o próximo conjunto e calcule uma nova intersecção entre esse conjunto lido e o conjunto da intersecção anterior, e assim por diante.

36.3 (a) Escreva uma função com a seguinte interface:

```
void ordena_insercao(int A[MAX], int m)
```

que receba um vetor $A$ de $m$ números inteiros distintos, com $1 \leqslant m \leqslant 100$, e ordene os elementos desse vetor em ordem crescente usando o método da inserção.

(b) Escreva uma função com a seguinte interface:

```
void intercala(int A[MAX], int m, int B[MAX], int n,
               int C[2*MAX], int *k)
```

que receba um vetor $A$ de números inteiros distintos e ordenados em ordem crescente de dimensão $m$ e um vetor $B$ de números inteiros distintos e ordenados em ordem crescente de dimensão $m$ e compute um vetor $C$ contendo os elementos de $A$ e de $B$ sem repetição e em ordem crescente.

(c) Escreva um programa que receba dois conjuntos de números inteiros e distintos $X$ e $Y$, com no máximo 100 elementos, ordene cada um deles usando a função do item (a) e intercale esses dois vetores usando a função do item (b), obtendo como resultado um vetor ordenado de números inteiros distintos dois a dois.

AULA 37

# FUNÇÕES E MATRIZES

Na aula 36 vimos funções e vetores e destacamos como estas estruturas são tratadas de forma diferenciada neste caso. Em especial, é muito importante lembrar que vetores são sempre parâmetros passados por referência às funções na linguagem C. Como veremos adiante, isso vale para qualquer variável composta homogênea, isto é, para variáveis compostas homogêneas de qualquer dimensão.

## 37.1 Matrizes

Um elemento de uma matriz pode ser passado como parâmetro para uma função assim como fizemos com uma variável de um tipo básico ou como um elemento de um vetor. Ou seja, a sentença

```
p = paridade(matriz[i][j]);
```

chama a função `paridade` passando o valor contido em `matriz[i][j]` como um argumento para a função.

Uma matriz toda pode ser passada a uma função da mesma forma que um vetor todo pode ser passado, bastando listar o identificador da matriz. Por exemplo, se a matriz `A` é declarada como uma matriz bidimensional de números inteiros, a sentença

```
multiplicaEscalar(A, escalar);
```

pode ser usada para chamar uma função que multiplica cada elemento de uma matriz `A` pelo valor armazenado na variável `escalar`. Assim como com vetores, uma modificação realizada no corpo de uma função sobre qualquer elemento de uma matriz que é um parâmetro dessa função provoca uma modificação no argumento da chamada da função, ou seja, na matriz que foi passada à função durante sua chamada.

O programa 37.1 recebe uma matriz $A_{4\times4}$ de números inteiros e um escalar e realiza o produto da matriz por esse escalar.

Programa 37.1: Uma função com um parâmetro que é uma matriz.

```
1  #include <stdio.h>
2  void multiplicaEscalar(int A[4][4], int escalar)
3  {
4    int i, j;
5    for(i = 0; i < 4; i++)
6      for(j = 0; j < 4; j++)
7        A[i][j] = A[i][j] * escalar;
8    return;
9  }
10 void imprimeMatriz(int A[4][4])
11 {
12   int i, j;
13   for(i = 0; i < 4; i++) {
14     for(j = 0; j < 4; j++)
15       printf("%3d ", A[i][j]);
16     printf("\n");
17   }
18   return;
19 }
20 int main(void)
21 {
22   int i, j, esc, matriz[4][4];
23   scanf("%d", &esc);
24   for(i = 0; i < 4; i++)
25     for(j = 0; j < 4; j++)
26       scanf("%d", &matriz[i][j]);
27   imprimeMatriz(matriz);
28   multiplicaEscalar(matriz, esc);
29   imprimeMatriz(matriz);
30   return 0;
31 }
```

## 37.2 Matrizes como parâmetros com uma dimensão omitida

Se um parâmetro de uma função é uma variável composta homogênea $k$-dimensional, com $k \geqslant 2$, então somente o tamanho da primeira dimensão pode ser omitida quando o parâmetro é declarado. Por exemplo, na função `multiplicaEscalar` da seção anterior, onde $A$ é um de seus parâmetros e é uma matriz, o número de colunas de $A$ deve ser espeficiado, embora seu número de linhas não seja necessário. Assim, na definição da função, poderíamos escrever:

```
void multiplicaEscalar(int A[][4], int escalar)
```

## Exercícios

37.1 (a) Escreva uma função com a seguinte interface:

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

que receba dois números inteiros $a$ e $b$ e troque os seus conteúdos.

(b) Usando a função anterior, escreva um programa que receba uma matriz de números inteiros $A$, de dimensão $m \times n$, com $1 \leqslant m, n \leqslant 100$, e dois números inteiros $i$ e $j$, troque os conteúdos das linhas $i$ e $j$ da matriz $A$ e imprima a matriz resultante.

Programa 37.2: Solução do exercício 37.1.

```
1  #include <stdio.h>
2  #define MAX 100
3  void troca(int *a, int *b)
4  {
5    int aux;
6    aux = *a;
7    *a = *b;
8    *b = aux;
9    return;
10 }
11 int main(void)
12 {
13   int m, n, i, j, k, A[MAX][MAX];
14   scanf("%d%d", &m, &n);
15   for(i = 0; i < m; i++)
16     for(j = 0; j < n; j++)
17       scanf("%d", &A[i][j]);
18   scanf("%d%d", &i, &j);
19   for (k = 0; k < n; k++)
20     troca(&A[i][k], &A[j][k]);
21   for (i = 0; i < m; i++) {
22     for (j = 0; j < n; j++)
23       printf("%d ", A[i][j]);
24     printf("\n");
25   }
26   return 0;
27 }
```

37.2 A $k$-ésima potência de uma matriz quadrada $A_{n\times n}$, denotada por $A^k$, é a matriz obtida a partir da matriz $A$ da seguinte forma:

$$A^k = I_n \times \overbrace{A \times A \times \cdots \times A}^{k},$$

onde $I_n$ é a matriz identidade de ordem $n$.

Para facilitar a computação de $A^k$, podemos usar a seguinte fórmula:

$$A^k = \begin{cases} I_n, & \text{se } k = 0\,, \\ A^{k-1} \times A, & \text{se } k > 0\,. \end{cases}$$

(a) Escreva uma função com a seguinte interface:

```
void identidade(int I[MAX][MAX], int n)
```

que receba uma matriz $I$ e uma dimensão $n$ e preencha essa matriz com os valores da matriz identidade de ordem $n$.

(b) Escreva uma função com a seguinte interface:

```
void multMat(int C[MAX][MAX],
             int A[MAX][MAX], int B[MAX][MAX], int n)
```

que receba as matrizes $A, B$ e $C$, todas de ordem $n$, e compute $C = A \times B$.

(c) Escreva uma função com a seguinte interface:

```
void copia(int A[MAX][MAX], int B[MAX][MAX], int n)
```

que receba as matrizes $A$ e $B$ de ordem $n$, e copie os elementos da matriz $B$ na matriz $A$.

(d) Escreva um programa que, usando as funções em (a) e (b), receba uma matriz de números inteiros $A$ de dimensão $n$ e um inteiro $k$ e compute e imprima $A^k$.

37.3 Dizemos que uma matriz $A_{n\times n}$ é um **quadrado latino de ordem** $n$ se em cada linha e em cada coluna aparecem todos os inteiros $1, 2, 3, \ldots, n$, ou seja, cada linha e coluna é permutação dos inteiros $1, 2, \ldots, n$.

Exemplo:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 4 & 1 & 2 & 3 \\ 3 & 4 & 1 & 2 \end{pmatrix}$$

A matriz acima é um quadrado latino de ordem 4.

(a) Escreva uma função com a seguinte interface:

```
int linha(int A[MAX][MAX], int n, int i)
```

que receba como parâmetros uma matriz $A_{n \times n}$ de números inteiros e um índice $i$, e verifique se na linha $i$ de $A$ ocorrem todos os números inteiros de 1 a $n$, devolvendo 1 em caso positivo e 0 caso contrário.

(b) Escreva uma função com a seguinte interface:

```
int coluna(int A[MAX][MAX], int n, int j)
```

que receba como parâmetros uma matriz $A_{n \times n}$ de números inteiros e um índice $j$, e verifique se na coluna $j$ de $A$ ocorrem todos os números inteiros de 1 a $n$, devolvendo 1 em caso positivo e 0 caso contrário.

(c) Usando as funções dos itens (a) e (b), escreva um programa que verifique se uma dada matriz $A_{n \times n}$ de números inteiros, com $1 \leqslant n \leqslant 100$, é um quadrado latino de ordem $n$.

AULA 38

# FUNÇÕES E REGISTROS

Já vimos funções com parâmetros de tipos básicos e de tipos complexos, como vetores e matrizes. Nesta aula, aprenderemos como comportam-se os registros no contexto das funções. Como veremos, um registro, diferentemente das variáveis compostas homogêneas, comporta-se como uma variável de um tipo básico qualquer quando trabalhamos com funções. Ou seja, um registro comporta-se como qualquer outra variável excluindo as compostas homogêneas, isto é, um registro é sempre um argumento passado por cópia a uma função. Se quisermos que um registro seja um argumento passado por referência a uma função, devemos explicitamente indicar essa opção usando o símbolo `&` no argumento e o símbolo `*` no parâmetro correspondente.

## 38.1 Tipo registro

Antes de estudar diretamente o funcionamento das funções com registros, precisamos aprender algo mais sobre esses últimos. Primeiro, é importante notar que apesar das aulas anteriores mostrarem como declarar variáveis que são registros, não discutimos em detalhes o que é um tipo registro. Então, suponha que um programa precisa declarar diversas variáveis que são registros com mesmos campos. Se todas as variáveis podem ser declaradas ao mesmo tempo, não há problemas. Mas se for necessário declarar essas variáveis em diferentes pontos do programa, então temos uma situação mais complicada. Para superá-la, precisamos saber definir um nome que representa um *tipo* registro, não uma variável registro particular. A linguagem C fornece duas boas maneiras para realizar essa tarefa: declarar uma etiqueta de registro ou usar `typedef` para criar um tipo registro.

Uma **etiqueta de registro** é um nome usado para identificar um tipo particular de registro. Essa etiqueta não reserva compartimentos na memória para armazenamento dos campos e do registro, mas sim adiciona informações a uma tabela usada pelo compilador para que variáveis registros possam ser declaradas a partir dessa etiqueta. A seguir temos a declaração de uma etiqueta de registro com nome `cadastro`:

```
struct cadastro {
  int codigo;
  char nome[MAX+1];
  int fone;
};
```

Observe que o caractere ponto e vírgula ( `;` ) segue o caractere fecha chaves `}`. O `;` deve estar presente neste ponto para terminar a declaração da etiqueta do registro.

Uma vez que criamos a etiqueta `cadastro`, podemos usá-la para declarar variáveis, como a seguir:

```
struct cadastro ficha1, ficha2;
```

Infelizmente, não podemos abreviar a declaração acima removendo a palavra reservada `struct`, já que `cadastro` não é o nome de um tipo. Isso significa que sem a palavra reservada `struct` a declaração acima não tem sentido.

Também, a declaração de uma etiqueta de um registro pode ser combinada com a declaração de variáveis registros, como mostrado a seguir:

```
struct cadastro {
  int codigo;
  char nome[MAX+1];
  int fone;
} ficha1, ficha2;
```

Ou seja, acima temos a declaração da etiqueta de registro `cadastro` e a declaração de duas variáveis registros `ficha1` e `ficha2`.

Todas as variáveis registros declaradas com o tipo `struct cadastro` são compatíveis entre si e, portanto, podemos atribuir uma a outra sem problemas, com mostramos abaixo:

```
struct cadastro ficha1 = {10032, "Sir Lancelot", 77165115};
struct cadastro ficha2;
ficha2 = ficha1;
```

Uma alternativa para declaração de etiquetas de registros é o uso da palavra reservada `typedef` para definir o nome de um tipo genuíno. Por exemplo, poderíamos definir um tipo com nome `Cadastro` da seguinte forma:

```
typedef struct {
  int codigo;
  char nome[MAX+1];
  int fone;
} Cadastro;
```

Observe que o nome do tipo, `Cadastro`, deve ocorrer no final, não após a palavra reservada `struct`. Além disso, como `Cadastro` é um nome de um tipo definido por `typedef`, não é permitido escrever `struct Cadastro`. Todas as variáveis do tipo `Cadastro` são compatíveis, desconsiderando onde foram declaradas. Assim, a declaração de registros com essa definição pode ser dada como abaixo:

```
Cadastro ficha1, ficha2;
```

Quando precisamos declarar um nome para um registro, podemos escolher entre declarar uma etiqueta de registro ou definir um tipo registro com a palavra reservada `typedef`. Entretanto, em geral preferimos usar etiquetas de registros por diversos motivos que serão justificados posteriormente.

## 38.2 Registros e passagem por cópia

Um campo de um registro pode ser passado como parâmetro para uma função assim como fizemos com uma variável de um tipo básico ou com uma célula de um vetor ou matriz. Ou seja, a sentença

```
x = raizQuadrada(poligono.diagonal);
```

chama a função `raizQuadrada` com argumento `poligono.diagonal`, isto é, com o valor armazenado no campo `diagonal` do registro `poligono`.

Um registro todo pode ser argumentos de funções da mesma forma que um vetor ou uma matriz podem ser, bastando listar o identificador do registro. Por exemplo, se temos um registro `tempo`, contendo os campos `hora`, `minutos` e `segundos`, então a sentença

```
s = totalSegundos(tempo);
```

pode ser usada para chamar a função `totalSegundos` que provavelmente calcula o total de segundos de uma medida de tempo armazenada no registro `tempo`.

Diferentemente das variáveis compostas homogêneas, uma modificação realizada em qualquer campo de um registro que é um parâmetro de uma função não provoca uma modificação no conteúdo do campo correspondente que é um argumento da função. Nesse caso, a princípio, um argumento que é um registro em uma chamada de uma função tem o mesmo comportamento de um argumento que é uma variável de um tipo básico qualquer, ou seja, uma expressão que é um argumento de uma chamada de uma função e que, na verdade é um registro, é de fato um parâmetro de entrada da função, passado por cópia.

Um exemplo é mostrado no programa 38.1.

Programa 38.1: Um programa com uma função com parâmetro do tipo registro.

```
1  #include <stdio.h>
2  struct tipo_reg {
3    int hh;
4    int mm;
5    int ss;
6  };
7  int converteSegundos(struct tipo_reg tempo)
8  {
9    return tempo.hh * 3600 + tempo.mm * 60 + tempo.ss;
10 }
11 int main(void)
12 {
13   int segundos;
14   struct tipo_reg horario;
15   printf("Informe um horario (hh:mm:ss): ");
16   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
17   segundos = converteSegundos(horario);
18   printf("Passaram-se %d segundo(s) neste dia.\n", segundos);
19   return 0;
20 }
```

Devemos destacar pontos importantes neste programa. O primeiro, e talvez mais importante, é a definição da etiqueta de registro `tipo_reg`. Em nenhuma outra oportunidade anterior havíamos nos desenvolvido um programa que continha uma declaração de uma etiqueta de registro. Observe ainda que a declaração da etiqueta `tipo_reg` do programa 38.1 foi feita *fora* dos corpos das funções `converteSegundos` e `main`. Em todos os programas descritos até aqui, não temos um exemplo de uma declaração de uma etiqueta de registro, ou mesmo uma declaração de uma variável de qualquer tipo, fora do corpo de qualquer função. A declaração da etiqueta `tipo_reg` realizada dessa maneira se deve ao fato que a mesma é usada nas funções `converteSegundos` e `main`, obrigando que essa declaração seja realizada *globalmente*, ou seja, fora do corpo de qualquer função.

Os outros pontos de destaque do programa 38.1 são todos relativos ao argumento da chamada da função `converteSegundos` e do parâmetro da mesma, descrito em sua interface. Note que essa função tem como parâmetro o registro `tempo` do tipo `struct tipo_reg`. O argumento da função, na chamada dentro da função `main` é o registro `horario` de mesmo tipo `struct tipo_reg`.

Finalmente, é importante observar mais uma vez que a passagem de um registro para uma função é feita por cópia, ou seja, o parâmetro correspondente é um parâmetro de entrada e, por isso, qualquer alteração realizada em um ou mais campos do registro dentro da função não afeta o conteúdo dos campos do registro que é um argumento na sua chamada. No exemplo acima não há alterações nos campos do registro `tempo` no corpo da função

`converteSegundos`, mas, caso houvesse, isso não afetaria qualquer um dos campos do registro `horario`, o argumento da chamada da função `converteSegundos` no corpo da função `main`.

## 38.3 Registros e passagem por referência

Se precisamos de uma função com um parâmetro de entrada e saída, isto é, um parâmetro passado por referência, devemos fazer como usualmente fazemos com qualquer variável de um tipo básico e usar os símbolos usuais `&` e `*`. O tratamento dos campos do registro em questão tem uma peculiaridade que precisamos observar com cuidado e se refere ao uso do símbolo `*` no parâmetro passado por referência no corpo da função. Vejamos um exemplo no programa 38.2.

Programa 38.2: Função com um registro como parâmetro de entrada e saída.

```
1  #include <stdio.h>
2  struct tipo_reg {
3    int hh;
4    int mm;
5    int ss;
6  };
7  void adicionaSegundo(struct tipo_reg *tempo)
8  {
9    (*tempo).ss++;
10   if ((*tempo).ss == 60) {
11     (*tempo).ss = 0;
12     (*tempo).mm++;
13     if ((*tempo).mm == 60) {
14       (*tempo).mm = 0;
15       (*tempo).hh++;
16       if ((*tempo).hh == 24)
17         (*tempo).hh = 0;
18     }
19   }
20   return;
21 }
22 int main(void)
23 {
24   struct tipo_reg horario;
25   printf("Informe um horario (hh:mm:ss): ");
26   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
27   adicionaSegundo(&horario);
28   printf("Novo horário %02d:%02d:%02d.\n", horario.hh, horario.mm, horario.ss
29   return 0;
30 }
```

Uma observação importante sobre o programa 38.2 é a forma como um registro que é um parâmetro passado por referência é apresentado no corpo da função. Como o símbolo `.` que seleciona um campo de um registro tem prioridade sobre o símbolo `*`, que indica que o parâmetro foi passado por referência, os parênteses envolvendo o o símbolo `*` e o identificador do registro são essenciais para evitar erros. Dessa forma, uma expressão envolvendo, por exemplo, o campo `ss` do registro `tempo`, escrita como `*tempo.ss`, está incorreta e não realiza a seleção do campo do registro que tencionamos.

Alternativamente, podemos usar os símbolos `->` para indicar o acesso a um campo de um registro que foi passado por referência a uma função. O uso desse símbolo simplifica bastante a escrita do corpo das funções que têm parâmetros formais que são registros passados por referência. Dessa forma, o programa 38.2 pode ser reescrito como o programa 38.3. Observe que esses dois programas têm a mesma finalidade, isto é, dada uma entrada, realizam o mesmo processamento e devolvem as mesmas saídas.

Programa 38.3: Uso do símbolo `->` para registros passados por referência.

```
1  #include <stdio.h>
2  struct tipo_reg {
3    int hh;
4    int mm;
5    int ss;
6  };
7  void adicionaSegundo(struct tipo_reg *tempo)
8  {
9    tempo->ss++;
10   if (tempo->ss == 60) {
11     tempo->ss = 0;
12     tempo->mm++;
13     if (tempo->mm == 60) {
14       tempo->mm = 0;
15       tempo->hh++;
16       if (tempo->hh == 24)
17         tempo->hh = 0;
18     }
19   }
20   return;
21 }
22 int main(void)
23 {
24   struct tipo_reg horario;
25   printf("Informe um horario (hh:mm:ss): ");
26   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
27   adicionaSegundo(&horario);
28   printf("Novo horário %02d:%02d:%02d.\n", horario.hh, horario.mm, horario.ss
29   return 0;
30 }
```

Observe que o argumento passado por referência para a função `adicionaSegundo` é idêntico nos programas 38.2 e 38.3. Mais que isso, a interface da função `adicionaSegundo` é idêntica nos dois casos e o parâmetro é declarado como `struct tipo_reg *tempo`. No corpo desta função do programa 38.3 é que usamos o símbolo `->` como uma abreviação, ou uma simplificação, da notação usada no corpo da função `adicionaSegundo` do programa 38.2. Assim, por exemplo, a expressão `(*tempo).ss` no programa 38.2 é reescrita como `tempo->ss` no programa 38.3.

## 38.4 Funções que devolvem registros

Uma função pode devolver valores de tipos básicos, mas também pode devolver um valor de um tipo complexo como um registro. Observe que variáveis compostas homogêneas, pelo fato de sempre poderem ser argumentos que correspondem a parâmetros de entrada e saída, isto é, de sempre poderem ser passadas por referência, nunca são devolvidas explicitamente por uma função, através do comando `return`. O único valor de um tipo complexo que podemos devolver é um valor do tipo registro.

No problema da seção anterior, em que é dado um horário e queremos atualizá-lo em um segundo, usamos um registro como parâmetro de entrada e saída na função `adicionaSegundo`. Podemos fazer com que a função apenas receba uma variável do tipo registro contendo um horário como parâmetro de entrada e devolva um registro com o horário atualizado como saída. Assim, os programas 38.2 e 38.3 poderiam ser reescritos como no programa 38.4.

Observe que a interface da função `adicionaSegundo` é diferente nesse caso. O tipo do valor devolvido é o tipo `struct tipo_reg`, indicando que a função devolverá um registro. O corpo dessa função também é diferente das anteriores e há a declaração de uma variável `atualizado` do tipo `struct tipo_reg` que ao final conterá o horário do registro `tempo`, parâmetro de entrada da função, atualizado em um segundo. Na função principal, a chamada da função `adicionaSegundo` também é diferente, já que é descrita do lado direito de um comando de atribuição, indicando que o registro devolvido pela função será armazenado no registro descrito do lado esquerdo da atribuição: `novo = adicionaSegundo(agora);`.

Programa 38.4: Um programa com uma função que devolve um registro.

```
#include <stdio.h>
struct tipo_reg {
  int hh;
  int mm;
  int ss;
};
struct tipo_reg adicionaSegundo(struct tipo_reg tempo)
{
  struct tipo_reg atualizado;
  atualizado.ss = tempo.ss + 1;
  if (atualizado.ss == 60) {
    atualizado.ss = 0;
    atualizado.mm = tempo.mm + 1;
    if (atualizado.mm == 60) {
      atualizado.mm = 0;
      atualizado.hh = tempo.hh + 1;
      if (atualizado.hh == 24)
        atualizado.hh = 0;
    }
    else
      atualizado.hh = tempo.hh;
  }
  else {
    atualizado.mm = tempo.mm;
    atualizado.hh = tempo.hh;
  }
  return atualizado;
}
int main(void)
{
  struct tipo_reg agora, novo;
  printf("Informe um horario (hh:mm:ss): ");
  scanf("%d:%d:%d", &agora.hh, &agora.mm, &agora.ss);
  novo = adicionaSegundo(agora);
  printf("Novo horário %02d:%02d:%02d.\n", novo.hh, novo.mm, novo.ss);
  return 0;
}
```

## Exercícios

38.1 Este exercício tem o mesmo objetivo do exercício 27.1.

(a) Escreva uma função com a seguinte interface:

```
int bissexto(struct data d)
```

que receba uma data e verifique se o ano é bissexto, devolvendo 1 em caso positivo e 0 caso contrário. Um ano é bissexto se é divisível por 4 e não por 100 ou é divisível por 400.

(b) Escreva uma função com a seguinte interface:

```
int diasMes(struct data d)
```

que receba uma data e devolva o número de dias do mês em questão.

Exemplo:

Se a função recebe a data `10/04/1992` deve devolver `30` e se recebe a data `20/02/2004` deve devolver `29`.

(c) Escreva uma função com a seguinte interface:

```
struct data diaSeguinte(struct data d)
```

que receba uma data e devolva a data que representa o dia seguinte. Use as funções dos itens (a) e (b).

(d) Escreva um programa que receba uma data e imprima a data que representa o dia seguinte. Use as funções dos itens anteriores.

38.2 Reescreva a função do item (c) do exercício 38.1, fazendo com que a função `diaSeguinte` tenha o registro `d` como um parâmetro de entrada e saída. Isto é, a interface da função deve ser:

```
void diaSeguinte(struct data *d)
```

38.3 (a) Escreva uma função com a seguinte interface:

```
struct tempo tempoDecorr(struct tempo t1, struct tempo t2)
```

receba duas medidas de tempo no formato `hh:mm:ss` e calcule o tempo decorrido entre essas duas medidas de tempo. Tenha cuidado com um par de medidas de tempo que cruzam a meia noite.

(b) Escreva um programa que receba dois horários e calcule o tempo decorrido entre esses dois horários. Use a função anterior.

AULA 39

# RECURSÃO

O conceito de recursão tem importância fundamental em computação. Na linguagem C, uma função recursiva é aquela que contém em seu corpo uma ou mais chamadas a si mesma. Naturalmente, uma função deve possuir pelo menos uma chamada proveniente de uma outra função externa. Uma função não recursiva, em contrapartida, é aquela para qual todas as suas chamadas são externas. Nesta aula construiremos funções recursivas para soluções de problemas.

## 39.1 Definição

Em muitos problemas computacionais encontramos a seguinte propriedade: cada entrada do problema contém uma entrada menor do mesmo problema. Dessa forma, dizemos que esses problemas têm uma **estrutura recursiva**. Para resolver um problema como esse, usamos em geral a seguinte estratégia:

```
se a entrada do problema é pequena então
   resolva-a diretamente;
senão,
   reduza-a a uma entrada menor do mesmo problema,
   aplique este método à entrada menor
   e volte à entrada original.
```

A aplicação dessa estratégia produz um **algoritmo recursivo**, que implementado na linguagem C torna-se um **programa recursivo** ou um programa que contém uma ou mais **funções recursivas**.

Uma **função recursiva** é aquela que possui, em seu corpo, uma ou mais chamadas a si mesma. Uma chamada de uma função a si mesma é dita uma **chamada recursiva**. Como sabemos, uma função deve possuir ao menos uma chamada proveniente de uma outra função, externa a ela. Se a função só possui chamadas externas a si, então é chamada de função não recursiva. Até a aula de hoje construimos apenas funções não recursivas.

Em geral, a toda função recursiva corresponde uma outra não recursiva que executa exatamente a mesma computação. Como alguns problemas possuem estrutura recursiva natural, funções recursivas são facilmente construídas a partir de suas definições. Além disso, a demonstração da corretude de um programa recursivo é facilitada pela relação direta entre sua estrutura e a indução matemática. Outras vezes, no entanto, a implementação de um programa recursivo demanda um gasto maior de memória, já que durante seu processo de execução muitas informações devem ser guardadas na sua pilha de execução.

## 39.2 Exemplos

Um exemplo clássico de uma função recursiva é aquela que computa o fatorial de um número inteiro $n \geqslant 0$. A idéia da solução através de uma recursiva é baseada na fórmula mostrada a seguir:

$$n! = \begin{cases} 1\,, & \text{se } n \leqslant 1\,, \\ n \times (n-1)!\,, & \text{caso contrário}\,. \end{cases}$$

A função recursiva `fat`, que calcula o fatorial de um dado número inteiro não negativo $n$, é mostrada a seguir:

```
int fat(int n)
{
  int result;
  if (n <= 1)
    result = 1;
  else
    result = n * fat(n-1);
  return result;
}
```

A sentença `return` pode aparecer em qualquer ponto do corpo de uma função e por isso podemos escrever a função `fat` como a seguir:

```
int fat(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fat(n-1);
}
```

Geralmente, preferimos a primeira versão implementada acima, onde existe apenas uma sentença com a palavra reservada `return` posicionada no final do corpo da função `fat`. Essa maneira de escrever funções recursivas evita confusão e nos permite seguir o fluxo de execução dessas funções mais naturalmente. No entanto, a segunda versão da função `fat` apresentada acima é equivalente à primeira e isso significa que também é válida. Além disso, essa segunda solução é mais compacta e usa menos memória, já que evita o uso de uma variável. Por tudo isso, essa segunda forma de escrever funções recursivas é muito usada por programadores(as) mais experientes.

A execução da função `fat` se dá da seguinte forma. Imagine que uma chamada `fat(3)` foi realizada. Então, temos ilustrativamente a seguinte situação:

`fat(3)`
- `fat(2)`
  - `fat(1)`
    - devolve 1
  - devolve $2 \times 1$
- devolve $3 \times 2$

Repare nas indentações que ilustram as chamadas recursivas.

Vamos ver um próximo exemplo. Considere o problema de determinar um valor máximo de um vetor `v` com $n$ elementos. O tamanho de uma entrada do problema é $n \geqslant 1$. Se $n = 1$ então `v[0]` é o único elemento do vetor e portanto `v[0]` é máximo. Se $n > 1$ então o valor que procuramos é o maior dentre o máximo do vetor `v[0..n-2]` e o valor armazenado em `v[n-1]`. Dessa forma, a entrada `v[0..n-1]` do problema fica reduzida à entrada `v[0..n-2]`. A função `maximo` a seguir implementa essa idéia.

```
/* esta função recebe um vetor de números inteiros v e um
   número inteiro n >= 1 e devolve um elemento máximo no vetor v[0..n-1] */
int maximo(int v[MAX], int n)
{
  int aux;
  if (n == 1)
    return v[0];
  else {
    aux = maximo(v, n-1);
    if (aux > v[n-1])
      return aux;
    else
      return v[n-1];
  }
}
```

Para verificar que uma função recursiva está correta, devemos seguir o seguinte roteiro:

**Passo 1:** escreva *o que* a função deve fazer;

**Passo 2:** verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

**Passo 3:** imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

## Exercícios

39.1 (a) Escreva uma função não recursiva com a seguinte interface:

```
int pot(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.

(b) Escreva uma função recursiva com a seguinte interface:

```
int potR(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.
Como no primeiro exemplo da seção anterior quando computamos o fatorial de um número, o valor de $x^n$ pode ser computado recursivamente basicamente da mesma forma, observando a seguinte fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0\,, \\ x \cdot x^{n-1}\,, & \text{se } n > 1\,. \end{cases}$$

(c) Escreva um programa que receba dois números inteiros $x$ e $n$, com $n \geqslant 0$, e devolva $x^n$. Use as funções em (a) e (b) para mostrar os dois resultados.

39.2 O que faz a função abaixo?

```
void ib(int n)
{
  if (n != 0) {
    ib(n / 2);
    printf("%c", '0' + n % 2);
  }
  return;
}
```

Escreva um programa para testar a função `ib`.

39.3 (a) Escreva uma função recursiva que implementa o algoritmo de Euclides. Veja o exercício 10.3.

(b) Escreva um programa que receba dois números inteiros e calcule o máximo divisor comum entre eles. Use a função do item (a).

Programa 39.1: Solução do exercício 39.1.

```
#include <stdio.h>
int pot(int x, int n)
{
  int i, result;
  result = 1;
  for (i = 1; i <= n; i++)
    result = result * x;
  return result;
}
int potR(int x, int n)
{
  if (n == 0)
    return 1;
  else
    return x * potR(x, n-1);
}
int main(void) {
  int x, n;
  scanf("%d%d", &x, &n);
  printf("Não resursiva: %d^%d = %d\n", x, n, pot(x, n));
  printf("Resursiva    : %d^%d = %d\n", x, n, potR(x, n));
  return 0;
}
```

AULA 40

# EXERCÍCIOS

40.1 (a) Escreva uma função recursiva com a seguinte interface:

```
float soma(float v[MAX], int n)
```

que receba um vetor $v$ e um número inteiro $n$, tal que $v$ contém $n \geqslant 1$ números com ponto flutuante, e calcule e devolva a soma desses $n$ números.

(b) Usando a função do item anterior, escreva um programa que receba um número inteiro $n$, com $n \geqslant 1$, e mais $n$ números reais e calcule a soma desses números.

Programa 40.1: Solução do exercício 40.1.

```
1  #include <stdio.h>
2  #define MAX 100
3  float soma(float v[MAX], int n)
4  {
5    if (n == 1)
6      return v[0];
7    else
8      return soma(v, n-1) + v[n-1];
9  }
10 int main(void)
11 {
12   int n, i;
13   float V[MAX];
14   scanf("%d", &n);
15   for (i = 0; i < n; i++)
16     scanf("%f", &V[i]);
17   printf("%g\n", soma(V, n));
18   return 0;
19 }
```

40.2 (a) Escreva uma função recursiva com a seguinte interface:

```
int somaDigitos(int n)
```

que receba um número inteiro positivo $n$ e devolva a soma de seus dígitos.

(b) Escreva um programa que receba um número inteiro $n$ e imprima a soma de seus dígitos. Use a função do item (a).

40.3 Como no exercício 18.2, neste exercício temos de imprimir a seqüência de Fibonacci. Essa seqüência é dada pela seguinte fórmula:

$$\begin{cases} F_1 &= 1 \\ F_2 &= 1 \\ F_i &= F_{i-1} + F_{i-2}\,, \quad \text{para } i \geqslant 3. \end{cases}$$

(a) Escreva uma função recursiva com a seguinte interface:

```
int Fib(int i)
```

que receba um número inteiro positivo $i$ e devolva o $i$-ésimo termo da seqüência de Fibonacci, isto é, $F_i$.

(b) Escreva um programa que receba um número inteiro $i \geqslant 1$ e imprima o termo $F_i$ da seqüência de Fibonacci. Use a função do item (a).

40.4 Como no exercício 20.3, neste exercício temos computar a função $\lfloor \log_2 n \rfloor$. Lembre-se que o **piso** de um número $x$ é o único inteiro $i$ tal que $i \leqslant x < i + 1$. O piso de $x$ é denotado por $\lfloor x \rfloor$.

Segue uma amostra de valores da função $\lfloor \log_2 n \rfloor$:

| $n$ | 15 | 16 | 31 | 32 | 63 | 64 | 127 | 128 | 255 | 256 | 511 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lfloor \log_2 n \rfloor$ | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |

(a) Escreva uma função recursiva com a seguinte interface:

```
int log2(int n)
```

que receba um número inteiro positivo $n$ e devolva $\lfloor \log_2 n \rfloor$.

(b) Escreva um programa que receba um número inteiro $n \geqslant 1$ e imprima $\lfloor \log_2 n \rfloor$. Use a função do item (a).

40.5 Considere o seguinte processo para gerar uma seqüência de números. Comece com um inteiro $n$. Se $n$ é par, divida por 2. Se $n$ é ímpar, multiplique por 3 e some 1. Repita esse processo com o novo valor de $n$, terminando quando $n = 1$. Por exemplo, a seqüência de números a seguir é gerada para $n = 22$:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

É conjecturado que esse processo termina com $n = 1$ para todo inteiro $n > 0$. Para uma entrada $n$, o **comprimento do ciclo de** $n$ é o número de elementos gerados na seqüência. No exemplo acima, o comprimento do ciclo de 22 é 16.

(a) Escreva uma função não recursiva com a seguinte interface:

```
int ciclo(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(b) Escreva uma versão recursiva da função do item (a) com a seguinte interface:

```
int cicloR(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(c) Escreva um programa que receba um número inteiro $n \geqslant 1$ e determine a seqüência gerada por esse processo e também o comprimento do ciclo de $n$. Use as funções em (a) e (b) para testar.

40.6 Podemos calcular a potência $x^n$ de uma maneira mais eficiente. Observe primeiro que se $n$ é uma potência de 2 então $x^n$ pode ser computada usando seqüências de quadrados. Por exemplo, $x^4$ é o quadrado de $x^2$ e assim $x^4$ pode ser computado usando somente duas multiplicações ao invés de três. Esta técnica pode ser usada mesmo quando $n$ não é uma potência de 2, usando a seguinte fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0, \\ (x^{n/2})^2\,, & \text{se } n \text{ é par}, \\ x \cdot x^{n-1}\,, & \text{se } n \text{ é ímpar}. \end{cases} \tag{40.1}$$

(a) Escreva uma função com interface

```
int potencia(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$ usando a fórmula (40.1).

(b) Escreva um programa que receba dois números inteiros $a$ e $b$ e imprima o valor de $a^b$.

AULA 41

# BUSCA

O problema da busca de um elemento em um conjunto é uma operação básica em computação. A maneira como esse conjunto é armazenado na memória do computador permite que algumas estratégias possam ser usadas para realizar a tarefa da busca. Na aula de hoje revemos os processos de busca que vimos usando corriqueiramente até o momento como uma sub-tarefa realizada na solução de diversos problemas práticos. A busca será fixada, como antes, com os dados envolvidos como sendo números inteiros. O conjunto de números inteiros onde a busca se dará é armazenado em um vetor. Além de rever essa estratégia, chamada de busca seqüencial, vemos ainda uma estratégia nova e muito eficiente de busca em um vetor ordenado, chamada de busca binária.

## 41.1 Busca seqüencial

Vamos fixar o elemento a ser buscado e o conjunto a ser varrido como sendo constituídos de números inteiros, observando que o problema da busca não se modifica essencialmente se esse tipo de dados for alterado. Assim, dado um número inteiro $x$ e um vetor de números inteiros $v[0..n-1]$, considere o problema de encontrar um índice $k$ tal que $v[k] = x$. O problema faz sentido com qualquer $n \geqslant 0$. Se $n = 0$, o vetor é vazio e portanto essa entrada do problema não tem solução.

É preciso começar com uma decisão de projeto: que fazer se $x$ não estiver no vetor? Adotaremos a convenção de devolver $-1$ nesse caso. A convenção é razoável pois $-1$ não pertence ao conjunto $\{0, \ldots, n-1\}$ de índices válidos. Para implementar essa convenção, convém varrer o vetor do fim para o começo, como mostra a função `busca` abaixo:

```
/* Recebe um vetor v[0..n-1], com n >= 0, e um número x
   e devolve k no intervalo [0, n-1] tal que v[k] == x.
   Se tal k não existe, devolve -1. */
int busca(int v[MAX], int n, int x)
{
  int k;
  for (k = n - 1; k >= 0 && v[k] != x; k--)
    ;
  return k;
}
```

Observe como a função é eficiente e elegante, funcionando corretamente mesmo quando o vetor está vazio, isto é, quando $n$ vale 0.

Um exemplo de uma chamada à função `busca` é apresentado abaixo:

```
if (busca(v, n, x) >= 0)
  printf("Encontrei!\n");
```

A função `busca` pode ser escrita em versão recursiva. A idéia do código é simples: se $n = 0$ então o vetor é vazio e portanto $x$ não está em $v[0..n-1]$; se $n > 0$ então $x$ está em $v[0..n-1]$ se e somente se $x = v[n-1]$ ou $x$ está no vetor $v[0..n-2]$. A versão recursiva é então mostrada abaixo:

```
/* Recebe um vetor v[0..n-1], com n >= 0, e um número x e devolve k
   tal que 0 <= k < n e v[k] == x. Se tal k não existe, devolve -1. */
int buscaR(int v[MAX], int n, int x)
{
  if (n == 0)
    return -1;
  else
    if (x == v[n - 1])
      return n - 1;
    else
      return buscaR(v, n - 1, x);
}
```

A corretude da função `busca` foi mostrada no exercício 20.1. Seu tempo de execução é linear no tamanho da entrada, isto é, $O(n)$, como mostrado na aula 32. A função `buscaR` tem corretude e análise de eficiência equivalentes.

## 41.2 Busca em um vetor ordenado

Dizemos que um vetor de números inteiros $v[0..n-1]$ é **crescente** se $v[0] \leqslant v[1] \leqslant \cdots \leqslant v[n-1]$ e **decrescente** se $v[0] \geqslant v[1] \geqslant \cdots \geqslant v[n-1]$. Dizemos ainda que o vetor é **ordenado** se é crescente ou decrescente. Nesta seção, vamos focar no problema da busca de um elemento $x$ em um vetor ordenado $v[0..n-1]$.

Agora, em lugar de perguntar em que posição $x$ está no vetor $v[0..n-1]$, é mais útil e conveniente perguntar em que posição $x$ *deveria* estar. Assim, o problema pode ser reformulado da seguinte maneira: dado um número inteiro $x$ e um vetor de números inteiros crescente $v[0..n-1]$, encontrar um índice $k$ tal que

$$v[k-1] < x \leqslant v[k] \,. \tag{41.1}$$

De posse de tal índice $k$, é muito fácil resolver o problema da busca, bastando comparar $x$ com $v[k]$.

Observe ainda que qualquer valor de $k$ no intervalo $[0, n]$ pode ser uma solução do problema da busca. Nos dois extremos do intervalo, a condição (41.1) deve ser interpretada adequadamente. Isto é, se $k = 0$, a condição se reduz apenas a $x \leqslant v[0]$, pois $v[-1]$ não faz sentido. Se $k = n$, a condição se reduz somente a $v[n-1] < x$, pois $v[n]$ não faz sentido. Tudo se passa como se o vetor $v$ tivesse um componente imaginário $v[-1]$ com valor $-\infty$ e um componente imaginário $v[n]$ com valor $+\infty$. Também, para simplificar um pouco o raciocínio, suporemos que $n \geqslant 1$.

Isso posto, observe então que uma busca seqüencial, recursiva ou não, pode ser realizada para resolver o problema, conforme apresentado na seção anterior. Vejamos então a função `buscaOrd`.

```
/* Recebe um vetor crescente v[0..n-1] com n >= 1 e um inteiro x e
   devolve um índice k em [0, n] tal que v[k-1] < x <= v[k]. */
int buscaOrd(int v[MAX], int n, int x)
{
  int k;
  for (k = 0; k < n && v[k] < x; k++)
    ;
  return k;
}
```

Uma chamada à função `buscaOrd` é mostrada a seguir:

```
ind = buscaOrd(v, n, x);
```

Se aplicamos a estratégia da busca seqüencial, isto é, se aplicamos qualquer das duas funções apresentadas na seção anterior, para solução do problema da busca sobre um vetor de entrada que se encontra ordenado, então certamente obtemos uma resposta correta, porém ineficiente do ponto de vista de seu consumo de tempo. Isso porque, no pior caso, a busca seqüencial realiza a comparação do elemento $x$ com cada um dos elementos do vetor $v$ de entrada. Ou seja, o problema da busca é resolvido em tempo proporcional ao número de elementos do vetor de entrada $v$, deixando de explorar sua propriedade especial de se encontrar ordenado.

Com uma busca binária, podemos fazer o mesmo trabalho de forma bem mais eficiente. A busca binária se baseia no método que usamos às vezes para encontrar uma palavra no dicionário: abrimos o dicionário ao meio e comparamos a primeira palavra desta página com a palavra buscada. Se a primeira palavra é "menor" que a palavra buscada, jogamos fora a primeira metade do dicionário e repetimos a mesma estratégia considerando apenas a metade restante. Se, ao contrário, a primeira palavra é "maior" que a palavra buscada, jogamos fora a segunda metade do dicionário e repetimos o processo. A função `buscaBin` abaixo implementa essa idéia.

```
/* Recebe um vetor crescente v[0..n-1] com n >= 1 e um inteiro x e
   devolve um índice k em [0, n] tal que v[k-1] < x <= v[k]. */
int buscaBin(int v[MAX], int n, int x)
{
  int e, d, m;
  e = -1;
  d = n;
  while (e < d - 1) {
    m = (e + d) / 2;
    if (v[m] < x)
      e = m;
    else
      d = m;
  }
  return d;
}
```

Um exemplo de chamada à função `buscaBin` é mostrado abaixo:

```
k = buscaBin(v, n, x);
```

Para provar a correção da função `buscaBin`, basta verificar o seguinte invariante:

> no início de cada repetição `while`, imediatamente antes da comparação de `e` com `d - 1`, vale a relação `v[e] < x <= v[d]`.

Com esse invariante em mãos, podemos usar a estratégia que aprendemos na aula 20 para mostrar finalmente que essa função está de fato correta.

Quantas iterações a função `buscaBin` executa? Essa conta revela o valor aproximado que representa o consumo de tempo dessa função em um dado vetor de entrada. Observe que em cada iteração, o tamanho do vetor `v` é dado por `d - e - 1`. No início da primeira iteração, o tamanho do vetor é $n$. No início da segunda iteração, o tamanho do vetor é aproximadamente $n/2$. No início da terceira, aproximadamente $n/4$. No início da $(k+1)$-ésima, aproximadamente $n/2^k$. Quando $k > \log_2 n$, temos $n/2^k < 1$ e a execução da função termina. Assim, o número de iterações é aproximadamente $\log_2 n$. O consumo de tempo da função é proporcional ao número de iterações e portanto proporcional a $\log_2 n$. Esse consumo cresce com $n$ muito mais lentamente que o consumo da busca seqüencial.

Uma solução recursiva para o problema da busca em um vetor ordenado é apresentada a seguir. Antes, é necessário reformular o problema ligeiramente. A função recursiva `buscaBinR` procura o elemento $x$ no vetor crescente $v[e..d]$, supondo que o valor $x$ está entre os extremos $v[e]$ e $v[d]$.

```
/* Recebe um vetor crescente v[e..d] e um inteiro x tais que
   v[e] < x <= v[d] e devolve um índice k em [e+1, d]
   tal que v[k-1] < x <= v[k]. */
int buscaBinR(int v[MAX], int e, int d, int x)
{
  int m;
  if (e == d - 1)
    return d;
  else {
    m = (e + d) / 2;
    if (v[m] < x)
      return BuscaBinR(v, m, d, x);
    else
      return BuscaBinR(v, e, m, v);
  }
}
```

Uma chamada da função `buscaBinR` pode ser realizada da seguinte forma:

```
k = buscaBinR(v, -1, n);
```

Quando a função `buscaBinR` é chamada com argumentos $(v, -1, n, x)$, ela chama a si mesma cerca de $\lfloor \log_2 n \rfloor$ vezes. Este número de chamadas é a profundidade da recursão.

## Exercícios

41.1 Tome uma decisão de projeto diferente daquela da seção 41.1: se $x$ não estiver em $v[0..n-1]$, a função deve devolver $n$. Escreva a versão correspondente da função `busca`. Para evitar o grande número de comparações de $k$ com $n$, coloque uma "sentinela" em $v[n]$.

```
1   /* Recebe um vetor v[0..n-1], com n >= 0, e um número x
2      e devolve k no intervalo [0, n-1] tal que v[k] == x.
3      Se tal k não existe, devolve n. */
4   int buscaS(int v[MAX], int n, int x)
5   {
6     int k;
7     v[n] = x;
8     for (k = 0; v[k] != x; k++)
9       ;
10    return k;
11  }
```

41.2 Considere o problema de determinar o valor de um elemento máximo de um vetor $v[0..n-1]$. Considere a função `maximo` abaixo.

```
1  int maximo(int v[MAX], int n)
2  {
3    int i, x;
4    x = v[0];
5    for (i = 1; i < n; i++)
6      if (x < v[i])
7        x = v[i];
8    return x;
9  }
```

(a) A função `maximo` acima resolve o problema?

(b) Faz sentido trocar `x = v[0]` pode `x = 0`?

(c) Faz sentido trocar `x = v[0]` pode `x = INT_MIN`[1]?

(d) Faz sentido trocar `x < v[i]` por `x <= v[i]`?

41.3 O autor da função abaixo afirma que ela decide se $x$ está no vetor $v[0..n-1]$. Critique seu código.

```
1  int busc(int v[MAX], int n, int x)
2  {
3    if (v[n-1] == x)
4      return 1;
5    else
6      return busc(v, n-1, x);
7  }
```

41.4 A operação de remoção consiste de retirar do vetor $v[0..n-1]$ o elemento que tem índice $k$ e fazer com que o vetor resultante tenha índices $0, 1, \ldots, n-2$. Essa operação só faz sentido se $0 \leqslant k < n$.

(a) Escreva uma função iterativa com a seguinte interface:

```
int remove(int v[MAX], int n, int k)
```

que remove o elemento de índice $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k < n$.

(b) Escreva uma função recursiva para a remoção com a seguinte interface:

[1] `INT_MIN` é o valor do menor número do tipo `int`.

```
int removeR(int v[MAX], int n, int k)
```

41.5 A operação de inserção consiste em introduzir um novo elemento $y$ entre a posição de índice $k-1$ e a posição de índice $k$ no vetor $v[0..n-1]$, com $0 \leqslant k \leqslant n$.

(a) Escreva uma função iterativa com a seguinte interface:

```
int insere(int v[MAX], int n, int k, int y)
```

que insere o elemento $y$ entre as posições $k-1$ e $k$ do vetor $v[0..n-1]$ e devolve o novo valor de $n$, supondo que $0 \leqslant k \leqslant n$.

(b) Escreva uma função recursiva para a inserção com a seguinte interface:

```
int insereR(int v[MAX], int n, int k, int x)
```

41.6 Na busca binária, suponha que $v[i] = i$ para todo $i$.

(a) Execute a função `buscaBin` com $n = 9$ e $x = 3$;

(b) Execute a função `buscaBin` com $n = 14$ e $x = 7$;

(c) Execute a função `buscaBin` com $n = 15$ e $x = 7$.

41.7 Execute a função `buscaBin` com $n = 16$. Quais os possíveis valores de $m$ na primeira iteração? Quais os possíveis valores de $m$ na segunda iteração? Na terceira? Na quarta?

41.8 Confira a validade da seguinte afirmação: quando $n+1$ é uma potência de 2, o valor da expressão $(e+d)$ é divisível por 2 em todas as iterações da função `buscaBin`, quaisquer que sejam $v$ e $x$.

41.9 Responda as seguintes perguntas sobre a função `buscaBin`.

(a) Que acontece se `while (e < d - 1)` for substituído por `while (e < d)`?

(b) Que acontece se `if (v[m] < x)` for substituído por `if (v[m] <= x)`?

(c) Que acontece se `e = m` for substituído por `e = m+1` ou por `e = m - 1`?

(d) Que acontece se `d = m` for substituído por `d = m + 1` ou por `d = m - 1`?

41.10 Se $t$ segundos são necessários para fazer uma busca binária em um vetor com $n$ elementos, quantos segundos serão necessários para fazer uma busca em $n^2$ elementos?

41.11 Escreva uma versão da busca binária para resolver o seguinte problema: dado um inteiro $x$ e um vetor decrescente $v[0..n-1]$, encontrar $k$ tal que $v[k-1] > x \geqslant v[k]$.

41.12 Suponha que cada elemento do vetor $v[0..n-1]$ é uma cadeia de caracteres (ou seja, temos uma matriz de caracteres). Suponha também que o vetor está em ordem lexicográfica. Escreva uma função eficiente, baseada na busca binária, que receba uma cadeia de caracteres $x$ e devolva um índice $k$ tal que $x$ é igual a $v[k]$. Se tal $k$ não existe, a função deve devolver $-1$.

41.13 Suponha que cada elemento do vetor $v[0..n-1]$ é um registro com dois campos: o nome do(a) estudante e o número do(a) estudante. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um(a) estudante e devolva seu nome. Se o número não estiver no vetor, a função deve devolver a cadeia de caracteres vazia.

41.14 Escreva uma função que receba um vetor estritamente crescente $v[0..n-1]$ de números inteiros e devolva um índice $i$ entre 0 e $n-1$ tal que $v[i] = i$. Se tal $i$ não existe, a função deve devolver $-1$. A sua função não deve fazer mais que $\lfloor \log_2 n \rfloor$ comparações envolvendo os elementos de $v$.

AULA 42

# ORDENAÇÃO: MÉTODOS ELEMENTARES

Além da busca, a ordenação é outra operação elementar em computação. Nesta aula revisaremos os métodos de ordenação elementares que já tivemos contato em aulas anteriores: o método das trocas sucessivas, da seleção e da inserção. Esses métodos foram projetados a partir de idéias simples e têm, como característica comum, tempo de execução de pior caso quadrático no tamanho da entrada.

## 42.1 Problema da ordenação

Um vetor $v[0..n-1]$ é **crescente** se $v[0] \leqslant v[1] \leqslant \cdots \leqslant v[n-1]$. O problema da ordenação de um vetor consiste em rearranjar, ou permutar, os elementos de um vetor $v[0..n-1]$ de tal forma que se torne crescente. Nesta aula nós discutimos três funções simples para solução desse problema. Nas aulas 43 e 44 examinaremos funções mais sofisticadas e eficientes.

## 42.2 Método das trocas sucessivas

O método das trocas sucessivas, popularmente conhecido como método da bolha, é um método simples de ordenação que, a cada passo, posiciona o maior elemento de um subconjunto de elementos do vetor de entrada na sua localização correta neste vetor. A função `ordTrocas` implementa esse método.

```
1  void ordTrocas(int v[], int n)
2  {
3    int i, j;
4    for (i = n - 1; i > 0; i--)
5      for (j = 0; j < i; j++)
6        if (v[j] > v[j+1])
7          troca(&v[j], &v[j+1]);
8  }
```

Um exemplo de chamada da função `ordTrocas` é dado a seguir:

```
ordTrocas(v, n);
```

Para entender a função `ordTrocas` basta observar que no início de cada repetição do `for` externo vale que:

- o vetor $v[0..n-1]$ é uma permutação do vetor original,
- o vetor $v[i+1..n-1]$ é crescente e
- $v[j] \leqslant v[i+1]$ para $j = 0, 1, \ldots, i$.

Além disso, o consumo de tempo da função `ordTrocas` é proporcional ao número de execuções da comparação "$v[j] > v[j+1]$", que é proporcional a $n^2$ no pior caso.

## 42.3 Método da seleção

O método de ordenação por seleção é baseado na idéia de escolher um menor elemento do vetor, depois um segundo menor elemento e assim por diante. A função `ordSelecao` implementa esse método.

```
1  void ordSelecao(int v[], int n)
2  {
3    int i, j, min;
4    for (i = 0; i < n-1; i++) {
5      min = i;
6      for (j = i+1; j < n; j++)
7        if (v[j] < v[min])
8          min = j;
9      troca(&v[i], &v[min]);
10   }
11 }
```

Um exemplo de chamada da função `ordSelecao` é dado a seguir:

```
ordSelecao(v, n);
```

Para entender como e por que o a função `ordSelecao` funciona, basta observar que no início de cada repetição do `for` externo valem os seguintes invariantes:

- o vetor $v[0..n-1]$ é uma permutação do vetor original,

- o vetor $v[0..i-1]$ está em ordem crescente e
- $v[i-1] \leqslant v[j]$ para $j = i, i+1, \ldots, n-1$.

O terceiro invariante pode ser assim interpretado: $v[0..i-1]$ contém todos os elementos "pequenos" do vetor original e $v[i..n-1]$ contém todos os elementos "grandes". Os três invariantes garantem que no início de cada iteração os elementos $v[0], \ldots, v[i-1]$ já estão em suas posições definitivas.

Uma análise semelhante à que fizemos para a função `ordTrocas` mostra que o método da inserção implementado pela função `ordSelecao` consome $n^2$ unidades de tempo no pior caso.

## 42.4 Método da inserção

O método da ordenação por inserção é muito popular. É freqüentemente usado para colocar em ordem um baralho de cartas. A função `ordInsercao` implementa esse método.

```
1  void ordInsercao(int v[], int n)
2  {
3    int i, j, x;;
4    for (i = 1; i < n; i++) {
5      x = v[i];
6      for (j = i-1; j >= 0 && v[j] > x; j--)
7        v[j+1] = v[j];
8      v[j+1] = x;
9    }
10 }
```

Um exemplo de chamada da função `ordInsercao` é dado a seguir:

```
ordInsercao(v, n);
```

Para entender a função `ordInsercao` basta observar que no início de cada repetição do `for` externo, valem os seguintes invariantes:

- o vetor $v[0..n-1]$ é uma permutação do vetor original e
- o vetor $v[0..i-1]$ é crescente.

O consumo de tempo da função `ordInsercao` é proporcional ao número de execuções da comparação "$v[j] > x$", que é proporcional a $n^2$.

## Exercícios

42.1 Escreva uma função que verifique se um dado vetor $v[0..n-1]$ é crescente.

```
int verifOrd(int v[], int n)
{
  for (i = 0; i < n-1; i++)
    if (v[i] > v[i+1])
      return 0;
  return 1;
}
```

42.2 Que acontece se trocarmos `i > 0` por `i >= 0` no código da função `ordTrocas`? Que acontece se trocarmos `j < i` por `j <= i`?

42.3 Troque `v[j] > v[j+1]` por `v[j] >= v[j+1]` no código da função `ordTrocas`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

42.4 Escreva uma versão recursiva do método de ordenação por trocas sucessivas.

42.5 Que acontece se trocarmos `i = 0` por `i = 1` no código da função `ordSelecao`? Que acontece se trocarmos `i < n-1` por `i < n`?

42.6 Troque `v[j] < v[min]` por `v[j] <= v[min]` no código da função `ordSelecao`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

42.7 Escreva uma versão recursiva do método de ordenação por seleção.

42.8 No código da função `ordInsercao`, troque `v[j] > x` por `v[j] >= x`. A nova função continua produzindo uma ordenação crescente de $v[0..n-1]$?

42.9 No código da função `ordInsercao`, que acontece se trocarmos `i = 1` por `i = 0`? Que acontece se trocarmos `v[j+1] = x` por `v[j] = x`?

42.10 Escreva uma versão recursiva do método de ordenação por inserção.

42.11 Escreva uma função que rearranje um vetor $v[0..n-1]$ de modo que ele fique em ordem estritamente crescente.

42.12 Escreva uma função que permute os elementos de um vetor $v[0..n-1]$ de modo que eles fiquem em ordem decrescente.

AULA 43

# ORDENAÇÃO POR INTERCALAÇÃO

Na aula 42 revimos os métodos de ordenação mais básicos, que são todos iterativos, simples e têm tempo de execução de pior caso proporcional a $n^2$, onde $n$ é o tamanho da entrada. Métodos mais eficientes de ordenação são baseados em recursão, técnica introduzida na aula 39. Nesta aula, estudamos o método da ordenação por intercalação, conhecido como *mergesort*.

A ordenação por intercalação, que veremos nesta aula, e a ordenação por separação, que veremos na aula 44, são métodos eficientes baseados na técnica recursiva chamada **dividir para conquistar**, onde quebramos o problema em vários sub-problemas de menor tamanho que são similares ao problema original, resolvemos esses sub-problemas recursivamente e então combinamos essas soluções para produzir uma solução para o problema original.

## 43.1 Dividir para conquistar

A técnica de dividir para conquistar é uma técnica geral de construção de algoritmos e programas, tendo a recursão como base, que envolve três passos em cada nível da recursão:

**Dividir** o problema em um número de sub-problemas;

**Conquistar** os sub-problemas solucionando-os recursivamente. No entanto, se os tamanhos dos sub-problemas são suficientemente pequenos, resolva os sub-problemas de uma maneira simples;

**Combinar** as soluções dos sub-problemas na solução do problema original.

Como mencionamos na aula 41, o algoritmo da busca binária é um método de busca que usa a técnica de dividir para conquistar na solução do problema da busca. O método de ordenação por intercalação, que veremos nesta aula, e o método da ordenação por separação, que veremos na aula 44, também são algoritmos baseados nessa técnica.

## 43.2 Problema da intercalação

Antes de apresentar o método da ordenação por intercalação, precisamos resolver um problema anterior, que auxilia esse método, chamado de problema da intercalação. O problema da intercalação pode ser descrito de forma mais geral como a seguir: dados dois conjuntos crescentes $A$ e $B$, com $m$ e $n$ elementos respectivamente, obter um conjunto crescente $C$ a partir de $A$ e $B$. Variantes sutis desse problema geral podem ser descritas como no caso em que se

permite ou não elementos iguais nos dois conjuntos de entrada, isto é, conjuntos de entrada $A$ e $B$ tais que $A \cap B \neq \emptyset$ ou $A \cap B = \emptyset$.

O problema da intercalação que queremos resolver aqui é mais específico e pode ser assim descrito: dados dois vetores crescentes $v[p..q-1]$ e $v[q..r-1]$, rearranjar $v[p..r-1]$ em ordem crescente. Isso significa que queremos de alguma forma intercalar os vetores $v[0..q-1]$ e $v[q..r-1]$. Nesse caso, a primeira vista parece que os vetores de entrada pode ter elementos em comum. Entretanto, este não é o caso, já que estamos considerando o mesmo conjunto inicial de elementos armazenados no vetor $v$.

Uma maneira fácil de resolver o problema da intercalação é usar um dos métodos de ordenação da aula 42 tendo como entrada o vetor $v[p..r-1]$. Essa solução, no entanto, tem consumo de tempo de pior caso proporcional ao quadrado do número de elementos do vetor e é ineficiente por desconsiderar as características dos vetores $v[p..q-1]$ e $v[q..r-1]$. Uma solução mais eficiente, que usa um vetor auxiliar, é mostrada a seguir.

```
1   /* Recebe os vetores crescentes v[p..q-1] e v[q..r-1] e
2      rearranja v[p..r-1] em ordem crescente */
3   void intercala(int v[MAX], int p, int q, int r)
4   {
5     int w[MAX], i, j, k;
6     i = p;
7     j = q;
8     k = 0;
9     while (i < q && j < r) {
10      if (v[i] < v[j]) {
11        w[k] = v[i];
12        i++;
13      }
14      else {
15        w[k] = v[j];
16        j++;
17      }
18      k++;
19    }
20    while (i < q) {
21      w[k] = v[i];
22      i++;
23      k++;
24    }
25    while (j < r) {
26      w[k] = v[j];
27      j++;
28      k++;
29    }
30    for (i = p; i < r; i++)
31      v[i] = w[i-p];
32  }
```

A função `intercala` tem tempo de execução de pior caso proporcional ao número de comparações entre os elementos do vetor, isto é, $r - p$. Assim, podemos dizer que o consumo de tempo no pior caso da função `intercala` é proporcional ao número de elementos do vetor de entrada.

## 43.3 Ordenação por intercalação

Com o problema da intercalação resolvido, podemos agora descrever uma função que implementa o método da ordenação por intercalação. Nesse método, dividimos ao meio um vetor $v$ com $r - p$ elementos, ordenamos recursivamente essas duas metades de $v$ e então as intercalamos. A função `mergesort` a seguir é recursiva e a base da recursão ocorre quando $p \geqslant r - 1$, quando não é necessário qualquer processamento.

```
1   /* recebe um vetor v[p..r-1] e o rearranja em ordem crescente */
2   void mergesort(int v[MAX], int p, int r)
3   {
4     int q;
5     if (p < r - 1) {
6       q  = (p + r) / 2;
7       mergesort(v, p, q);
8       mergesort(v, q, r);
9       intercala(v, p, q, r);
10    }
11  }
```

Como a expressão $(p + q)/2$ da função `mergesort` é do tipo inteiro, observe que seu resultado é, na verdade, avaliado como $\lfloor \frac{p+q}{2} \rfloor$.

Observe também que para ordenar um vetor $v[0..n-1]$ basta chamar a função `mergesort` com os seguintes argumentos:

```
mergesort(v, 0, n);
```

Vejamos um exemplo de execução da função `mergesort` na figura 43.1, para um vetor de entrada $v[0..7] = \{4, 6, 7, 3, 5, 1, 2, 8\}$ e chamada

```
mergesort(v, 0, 8);
```

Observe que as chamadas recursivas são realizadas até a linha divisória imaginária ilustrada na figura, quando $p \geqslant r - 1$. A partir desse ponto, a cada volta de um nível de recursão,

uma intercalação é realizada. No final, uma última intercalação é realizada e o vetor original torna-se então um vetor crescente com os mesmos elementos de entrada.

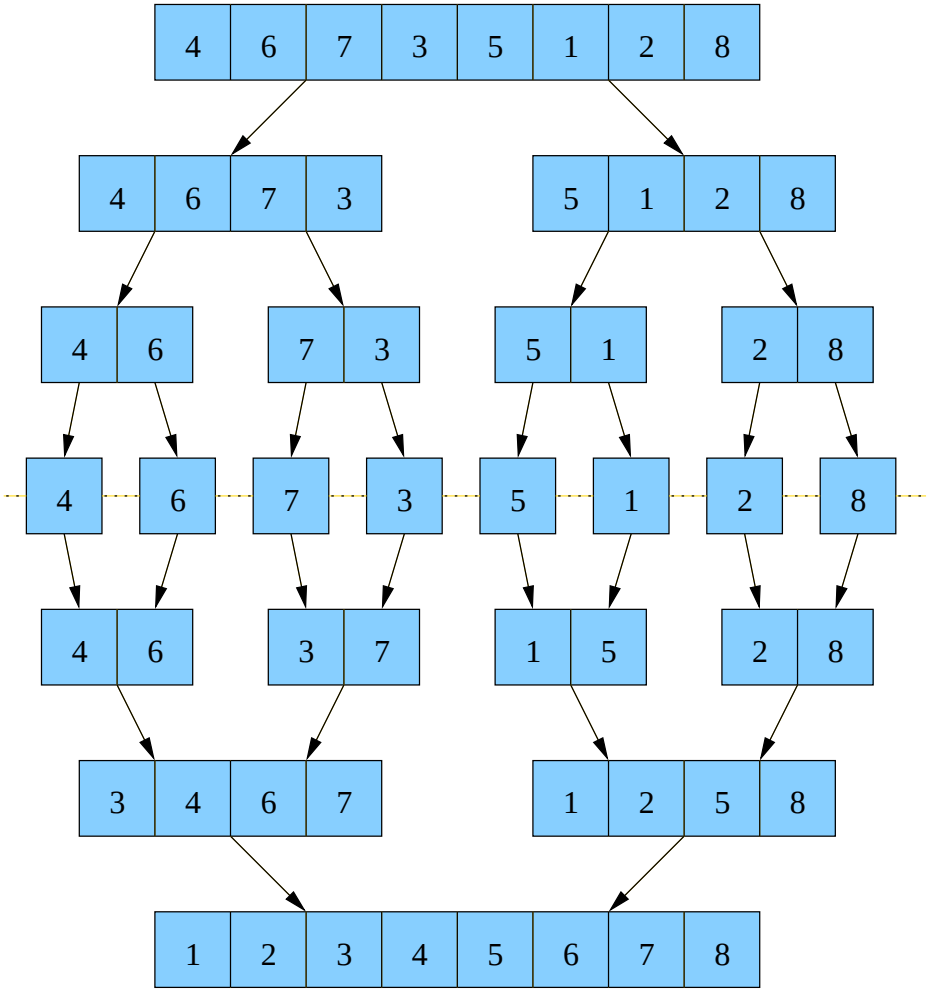



Figura 43.1: Exemplo de execução da ordenação por intercalação.

Qual o desempenho da função `mergesort` quando queremos ordenar um vetor $v[0..n-1]$? Suponha, para efeito de simplificação, que $n$ é uma potência de 2. Se esse não é o caso, podemos examinar duas potências de 2 consecutivas, justamente aquelas tais que $2^{k-1} < n \leqslant 2^k$, para algum $k \geqslant 0$. Observe então que o número de elementos do vetor é diminuído a aproximadamente metade a cada chamada da função `mergesort`. Ou seja, o número aproximado de chamadas é proporcional a $\log_2 n$. Na primeira vez, o problema original é reduzido a dois sub-problemas onde é necessário ordenar os vetores $v[0..\frac{n}{2}-1]$ e $v[\frac{n}{2}..n-1]$. Na segunda vez, cada um dos sub-problemas são ainda divididos em mais dois sub-problemas cada, gerando quatro sub-problemas no total, onde é necessário ordenar os vetores $v[0..\frac{n}{4}-1]$, $v[\frac{n}{4}..\frac{n}{2}-1]$, $v[\frac{n}{2}..\frac{3n}{4}-1]$ e $v[\frac{3n}{4}..n-1]$. E assim por diante. Além disso, como já vimos, o tempo total que a função `intercala` gasta é proporcional a $n$. Portanto, a função `mergesort` consome tempo proporcional a $n \log_2 n$.

## Exercícios

43.1 Simule detalhadamente a execução da função `mergesort` sobre o vetor de entrada $v[0..7] = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

43.2 A função `intercala` está correta nos casos extremos $p = q$ e $q = r$?

43.3 Um algoritmo de intercalação é **estável** se não altera a posição relativa dos elementos que têm um mesmo valor. Por exemplo, se o vetor tiver dois elementos de valor 222, um algoritmo de intercalação estável manterá o primeiro 222 antes do segundo. A função `intercala` é estável? Se a comparação `v[i] <= v[j]` for trocada por `v[i] < v[j]` a função fica estável?

43.4 O que acontece se trocarmos `(p + r)/2` por `(p + r - 1)/2` no código da função `mergesort`? Que acontece se trocarmos `(p + r)/2` por `(p + r + 1)/2`?

43.5 Escreva uma versão da ordenação por intercalação que rearranje um vetor $v[p..r-1]$ em ordem decrescente.

43.6 Escreva uma função eficiente que receba um conjunto $S$ de $n$ números reais e um número real $x$ e determine se existe um par de elementos em $S$ cuja soma é exatamente $X$.

43.7 Seja $A$ um vetor de $n$ números inteiros distintos. Se $i < j$ e $A[i] > A[j]$ então o par $(i, j)$ é chamado uma **inversão** de $A$.

(a) Liste as cinco inversões do vetor $\{2, 3, 8, 6, 1\}$.

(b) Qual vetor com elementos do conjunto $\{1, 2, \ldots, n\}$ tem o maior número de inversões? Quantas são?

(c) Qual a relação entre o tempo de execução da ordenação por inserção e o número de inversões em um vetor de entrada? Justifique sua resposta.

(d) Modificando a ordenação por intercalação, escreva uma função eficiente que determine o número de inversões em uma permutação de $n$ elementos.

43.8 Escreva um programa para comparar experimentalmente o desempenho da função `mergesort` com o das funções `ordTrocas`, `ordSelecao` e `ordInsercao` da aula 42. Use um vetor aleatório para fazer os testes.

43.9 Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:

- Sorting Algorithms de J. Harrison;
- Sorting Algorithms de P. Morin;
- Sorting Algorithms Animations de D. R. Martin.

AULA 44

# ORDENAÇÃO POR SEPARAÇÃO

O método de ordenação por separação resolve o problema da ordenação descrito na aula 42, rearranjando um vetor $v[0..n-1]$ de modo que se torne crescente. Em geral, este método é muito mais rápido que os métodos elementares vistos na aula 42, mas pode ser tão lento quanto aqueles para certas entradas especiais do problema.

O método da ordenação por separação é recursivo e também se baseia na estratégia de dividir para conquistar. O método é comumente conhecido como *quisksort*. É freqüentemente usado na prática para ordenação já que é destacadamente rápido "na média". Apenas para algumas entradas especiais o método é tão lento quanto os métodos elementares de ordenação.

## 44.1 Problema da separação

O problema da separação é o núcleo do método da ordenação por separação, que é descrito de forma propositalmente vaga abaixo:

> rearranjar um vetor $v[p..r]$ de modo que os elementos pequenos fiquem todos do lado esquerdo e os grandes fiquem todos do lado direito.

Gostaríamos que os dois lados tivessem aproximadamente o mesmo número de elementos, mas estamos dispostos a aceitar resultados menos equilibrados. De todo modo, é importante que a separação não resulte degenerada, deixando um dos lados vazio. A dificuldade está em construir uma função que resolva o problema de maneira rápida e não use um vetor auxiliar.

O problema da separação que estamos interessados pode ser formulado concretamente da seguinte maneira:

> rearranjar o vetor $v[p..r]$ de modo que tenhamos
>
> $$v[p..q] \leqslant v[q+1..r]$$
>
> para algum $q$ em $p..r-1$.

A função `separa` abaixo soluciona o problema da separação. No início, um pivô $x$ é escolhido e, a partir de então, o significado de *pequeno* e *grande* passa a ser associado a esse pivô: os elementos do vetor que forem maiores que $x$ serão considerados grandes e os demais serão considerados pequenos.

```
1  int separa(int v[], int p, int r)
2  {
3    int x, i, j;
4    x = v[p];
5    i = p - 1;
6    j = r + 1;
7    while (1) {
8      do
9        j--;
10     while (v[j] > x);
11     do
12       i++;
13     while (v[i] < x);
14     if (i < j)
15       troca(&v[i], &v[j]);
16     else
17       return j;
18   }
19 }
```

Um exemplo de execução da função `separa` é apresentado na figura 44.1.

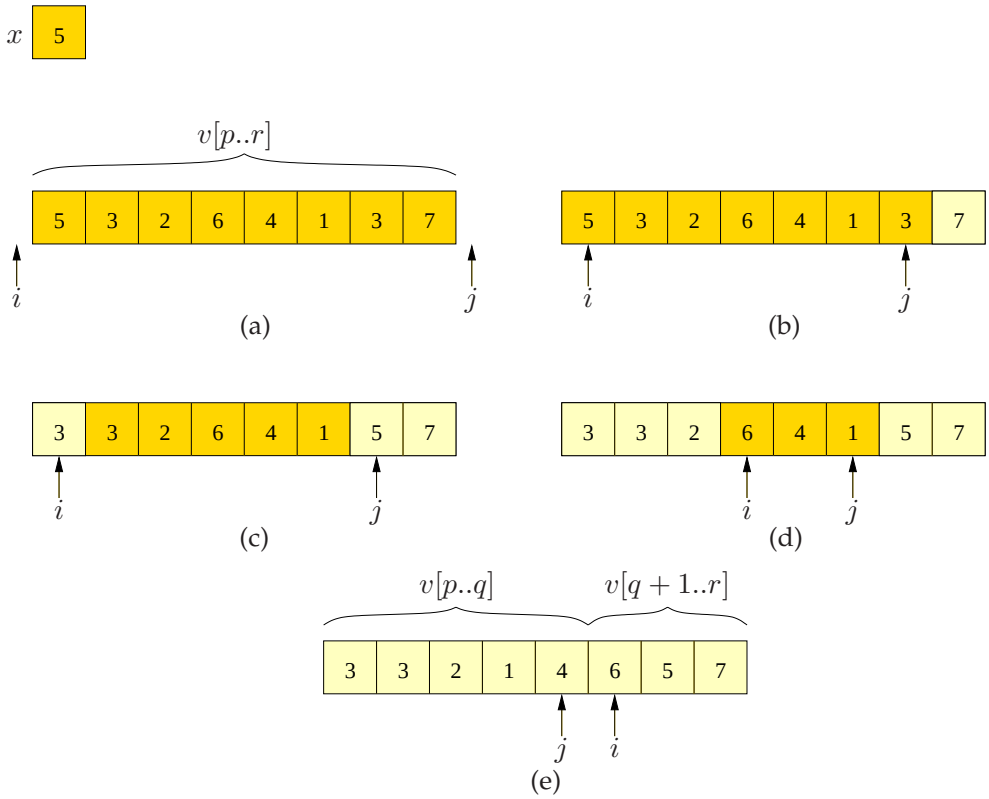



Figura 44.1: Uma execução da função `separa`.

O corpo da estrutura de repetição `while` da função `separa` é repetido até que $i \geqslant j$ e, neste ponto, o vetor $v[p..r]$ acabou de ser separado em $v[p..q]$ e $v[q+1..r]$, com $p \leqslant q < r$, tal que nenhum elemento de $v[p..q]$ é maior que um elemento de $v[q+1..r]$. O valor $q = j$ é devolvido então pela função `separa`.

Em outras palavras, podemos dizer que no início de cada iteração valem os seguintes invariantes:

- $v[p..r]$ é uma permutação do vetor original,
- $v[p..i] \leqslant x \leqslant v[j..r]$ e
- $p \leqslant i \leqslant j \leqslant r$.

O número de iterações que a função realiza é proporcional a $r - p + 1$, isto é, proporcional ao número de elementos do vetor.

## 44.2 Ordenação por separação

Com uma função que soluciona o problema da separação, podemos descrever agora o método da ordenação por separação. Esse método também usa a estratégia de dividir para conquistar e o faz de maneira semelhante ao método da ordenação por intercalação, mas com chamadas "invertidas".

```
1  void quicksort(int v[], int p, int r)
2  {
3    int q;
4    if (p < r) {
5      q = separa(v, p, r);
6      quicksort(v, p, q);
7      quicksort(v, q+1, r);
8    }
9  }
```

Uma chamada da função `quicksort` para ordenação de um vetor $v[0..n-1]$ deve ser feita como a seguir:

```
quicksort(v, 0, n-1);
```

Observe ainda que a função `quicksort` está correta mesmo quando $p > r$, isto é, quando o vetor está vazio.

O consumo de tempo do método de ordenação por separação é proporcional ao número de comparações realizadas entre os elementos do vetor. Se o índice devolvido pela função

`separa` sempre tiver valor mais ou menos médio de $p$ e $r$, então o número de comparações será aproximadamente $n \log_2 n$. Caso contrário, o número de comparações será da ordem de $n^2$. Observe que isso ocorre, por exemplo, quando o vetor já estiver ordenado ou quase-ordenado. Portanto, o consumo de tempo de pior caso da ordenação por separação não é melhor que o dos métodos elementares vistos na aula 42. Felizmente, o pior caso para a ordenação por separação é raro. Dessa forma, o consumo de tempo *médio* da função `quicksort` é proporcional a $n \log_2 n$.

## Exercícios

44.1 Ilustre a operação da função `separa` sobre o vetor $v$ que contém os elementos do conjunto $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.

44.2 Qual o valor de $q$ a função `separa` devolve quando todos os elementos no vetor $v[p..r]$ têm o mesmo valor?

44.3 A função `separa` produz o resultado correto quando $p = r$?

44.4 Escreva uma função que rearranje um vetor $v[p..r]$ de números inteiros de modo que os elementos negativos e nulos fiquem à esquerda e os positivos fiquem à direita. Em outras palavras, rearranje o vetor de modo que tenhamos $v[p..q-1] \leqslant 0$ e $v[q..r] > 0$ para algum $q$ em $p..r+1$. Procure escrever uma função eficiente que não use um vetor auxiliar.

44.5 Digamos que um vetor $v[p..r]$ está **arrumado** se existe $q$ em $p..r$ que satisfaz

$$v[p..q-1] \leqslant v[q] < v[q+1..r] \,.$$

Escreva uma função que decida se $v[p..r]$ está arrumado. Em caso afirmativo, sua função deve devolver o valor de $q$.

44.6 Que acontece se trocarmos `if (p < r)` por `if (p != r)` no corpo da função `quicksort`?

44.7 Compara as funções `quicksort` e `mergesort`. Discuta as semelhanças e diferenças.

44.8 Como você modificaria a função `quicksort` para ordenar elementos em ordem decrescente?

44.9 Os bancos freqüentemente gravam transações sobre uma conta corrente na ordem das datas das transações, mas muitas pessoas preferem receber seus extratos bancários em listagens ordenadas pelo número do cheque emitido. As pessoas em geral emitem cheques na ordem da numeração dos cheques, e comerciantes usualmente descontam estes cheques com rapidez razoável. O problema de converter uma lista ordenada por data de transação em uma lista ordenada por número de cheques é portanto o problema de ordenar uma entrada quase já ordenada. Argumente que, neste caso, o método de ordenação por inserção provavelmente se comportará melhor do que o método de ordenação por separação.

44.10 Forneça um argumento cuidadoso para mostrar que a função `separa` é correta. Prove o seguinte:

(a) Os índices $i$ e $j$ nunca referenciam um elemento de $v$ fora do intervalo $[p..r]$;

(b) O índice $j$ não é igual a $r$ quando `separa` termina (ou seja, a partição é sempre não trivial);

(c) Todo elemento de $v[p..j]$ é menor ou igual a todo elemento de $v[j+1..r]$ quando a função `separa` termina.

44.11 Escreva uma versão da função `quicksort` que coloque um vetor de cadeias de caracteres em ordem lexicográfica.

44.12 Considere a seguinte solução do problema da separação, devido a N. Lomuto. Para separar $v[p..r]$, esta versão incrementa duas regiões, $v[p..i]$ e $v[i+1..j]$, tal que todo elemento na primeira região é menor ou igual a $x = v[r]$ e todo elemento na segunda região é maior que $x$.

```
1  int separaLomuto(int v[], int p, int r)
2  {
3    int x, i, j;
4    x = v[r];
5    i = p - 1;
6    for (j = p; j <= r; j++)
7      if (v[j] <= x) {
8        i++;
9        troca(&v[i], &v[j]);
10     }
11   if (i < r)
12     return i;
13   else
14     return i - 1;
15 }
```

(a) Ilustre a operação da função `separaLomuto` sobre o vetor $v$ que contém os elementos $\{13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21\}$.

(b) Argumente que `separaLomuto` está correta.

(c) Qual o número máximo de vezes que um elemento pode ser movido pelas funções `separa` e `separaLomuto`? Justifique sua resposta.

(d) Argumente que `separaLomuto`, assim como `separa`, tem tempo de execução proporcional a $n$ sobre um vetor de $n = r - p + 1$ elementos.

(e) Como a troca da função `separa` pela função `separaLomuto` afeta o tempo de execução do método da ordenação por separação quando todos os valores de entrada são iguais?

44.13 A função `quicksort` contém duas chamadas recursivas para ela própria. Depois da chamada da `separa`, o sub-vetor esquerdo é ordenado recursivamente e então o sub-vetor direito é ordenado recursivamente. A segunda chamada recursiva no corpo da função `quicksort` não é realmente necessária; ela pode ser evitada usando uma estrutura de

controle iterativa. Essa técnica, chamada **recursão de cauda**, é fornecida automaticamente por bons compiladores. Considere a seguinte versão da ordenação por separação , que simula a recursão de cauda.

```
1  void quicksort2(int v[], int p, int r)
2  {
3    while (p < r) {
4      q = separa(v, p, r);
5      quicksort2(v, p, q);
6      p = q + 1;
7    }
8  }
```

Ilustre a operação da função `quicksort2` sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `quicksort2(v, 0, n-1)`. Argumente que a função `quicksort2` ordena corretamente o vetor $v$.

44.14 Um famoso programador propôs o seguinte método de ordenação de um vetor $v[0..n-1]$ de números inteiros:

```
1  void ordPateta(int v[], int i, int j)
2  {
3    int k;
4    if (v[i] > v[j])
5      troca(&v[i], &v[j]);
6    if (i + 1 < j) {
7      k = (j - i + 1) / 3;
8      ordPateta(v, i, j-k);
9      ordPateta(v, i+k, j);
10     ordPateta(v, i, j-k);
11   }
12 }
```

Ilustre a operação desse novo método de ordenação sobre o vetor $v$ que contém os elementos $\{21, 7, 5, 11, 6, 42, 13, 2\}$. Use a chamada `ordPateta(v, 0, n-1)`. Argumente que a função `ordPateta` ordena corretamente o vetor $v$.

44.15 Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:

- Sorting Algorithms de J. Harrison;
- Sorting Algorithms de P. Morin;
- Sorting Algorithms Animations de D. R. Martin.

44.16 Familiarize-se com a função `qsort` da biblioteca `stdlib` da linguagem C.

AULA 45

# BIBLIOTECA PADRÃO

Nas últimas aulas aprendemos a construir nossas próprias funções. Os benefícios dessa prática são muitos, como percebemos. Entre eles, podemos destacar abstração, organização e reaproveitamento de código, por exemplo. Felizmente, não é sempre que precisamos de um trecho de código que realiza um determinado processamento que temos necessariamente de programar esse trecho, construindo uma função. A linguagem C fornece uma biblioteca padrão de funções que nos ajuda em muitas tarefas importantes, desde funções de entrada e saída de dados, manipulação de cadeias de caracteres até funções matemáticas. Nesta aula, veremos algumas funções importantes, destacando que muitas outras não serão cobertas aqui. Os interessados em mais informações devem buscar a página do guia de referência da biblioteca da linguagem C. Nesta aula, cobrimos com algum detalhe os arquivos que contêm definições de macros, tipos e os protótipos das funções da biblioteca padrão de funções da linguagem C. Esses arquivos são também conhecidos como arquivos-cabeçalhos e têm extensão `.h`, do inglês h*eader*.

Iniciamos com uma visão sobre os qualificadores de tipos existentes na linguagem C, fundamentais especialmente na declaração de parâmetros sensíveis à alteração, tais como os vetores e as matrizes, que são sempre parâmetros por referência. Em seguida, definimos arquivos-cabeçalhos para, por fim, cobrir os arquivos-cabeçalhos da biblioteca padrão da linguagem C. Esta aula é um guia de referência da biblioteca padrão da linguagem C, suas constantes, tipos e funções principais. Algumas funções, por serem muito específicas, não foram listadas aqui.

## 45.1 Qualificadores de tipos

Um **qualificador de tipo** fornece informação adicional sobre um tipo de uma variável ao compilador. Na linguagem C, um qualificador de tipo informa o compilador que a variável, ou as células de memórias que são reservadas para seu uso, tem alguma propriedade especial, a saber, um valor que não deve ser modificado pelo(a) programador(a) durante a execução do programa ou ainda um valor que potencialmente se modifica durante a execução do programa sem que haja controle do(a) programador(a) sobre o processo.

Os qualificadores de tipo `const` e `volatile` são, respectivamente, os dois qualificadores da linguagem C que correspondem a essas descrições. O segundo, por tratar de programação de baixo nível, será descrito apropriadamente na aula 48.

O qualificador de tipo `const` é usado para declarar objetos que se parecem com variáveis mas são variáveis apenas "de leitura", isto é, um programa pode acessar seu valor, mas não pode modificá-lo. Por exemplo, a declaração abaixo:

```
const int n = 10;
```

cria um objeto `const` do tipo `int` com identificador `n` cujo valor é `10`. A declaração abaixo:

```
const int v[] = {71, 13, 67, 88, 4, 52};
```

cria um vetor `const` de números inteiros e identificador `v`.

Muitas vantagens podem ser destacadas quando declaramos um objeto do tipo `const`, como por exemplo auto-documentação e verificação pelo compilador de tentativas alteração.

A primeira vista, pode parecer que o qualificador de tipo `const` tem o mesmo papel da diretiva `#define`, que foi usada em aulas anteriores para definir nomes para constantes ou macros. Existem diferenças significativas entre as duas, que são listadas abaixo:

- podemos usar `#define` para criar um nome par uma constante numérica, caractere ou cadeia de caracteres; por outro lado, podemos usar `const` para criar objetos somente para leitura de qualquer tipo, incluindo vetores, estruturas, uniões, apontadores, etc;
- objetos criados com `const` estão sujeitos às mesmas regras de escopo de qualquer variável, mas constantes criadas com `#define` não;
- o valor de um objeto criado com `const` pode ser visto em um depurador de programas; o valor de uma macro não pode;
- objetos criados com `const` não podem ser usados em expressões constantes; as constantes criadas com `#define` podem;
- podemos aplicar o operador de endereçamento `&` sobre um objeto `const`, já que ele possui um endereço; uma macro não tem um endereço.

Não há uma regra bem definida que determina o uso de `#define` ou `const` na declaração de constantes. Em geral, usamos a diretiva `#define` mais especificamente para criar constantes que representam números ou caracteres.

## 45.2 Arquivo-cabeçalho

A diretiva `#include` ordena o pré-processador a abrir um arquivo especificado e a inserir seu conteúdo no arquivo atual. Assim, se queremos que vários arquivos de código fonte tenham acesso à mesma informação, devemos colocar essa informação em um arquivo e então usar a diretiva `#include` para trazer o conteúdo do arquivo em cada arquivo de código fonte. Arquivos que são incluídos dessa forma são chamados de **arquivos-cabeçalhos**, do inglês *header files* ou também *include files*. Por convenção, um arquivo como esse tem a extensão `.h`.

A diretiva `#include` pode ser usada de duas formas. A primeira forma é usada para arquivos-cabeçalhos que pertencem à biblioteca padrão da linguagem C:

```
#include <arquivo.h>
```

A segunda forma é usada para todos os outros arquivos-cabeçalhos, incluindo aqueles que são escritos por programadores(as):

```
#include "arquivo.h"
```

A diferença entre as duas formas se dá pela maneira como o compilador busca o arquivo-cabeçalho. Na primeira, o compilador busca o arquivo-cabeçalho no(s) diretório(s) em que os arquivos-cabeçalhos do sistema se encontram. Nos sistemas baseados no UNIX, como o LINUX, os arquivos-cabeçalhos são mantidos usualmente no diretório `/usr/include`. Na segunda forma, a busca é realizada no diretório corrente e, em seguida, no(s) diretório(s) em que os arquivos-cabeçalhos do sistema se encontram.

Arquivos-cabeçalhos auxiliam no compartilhamento de definições de macros, de tipos e de protótipos de funções por dois ou mais arquivos de código fonte.

Abaixo apresentamos o arquivo-cabeçalho completo `assert.h` da biblioteca padrão da linguagem C.

```
/* Copyright (C) 1991,1992,1994-2001,2003,2004,2007
   Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

/*
 *        ISO C99 Standard: 7.2 Diagnostics        <assert.h>
```

```
 */

#ifdef        _ASSERT_H

# undef        _ASSERT_H
# undef        assert
# undef __ASSERT_VOID_CAST

# ifdef        __USE_GNU
#  undef assert_perror
# endif

#endif /* assert.h        */

#define        _ASSERT_H        1
#include <features.h>

#if defined __cplusplus && __GNUC_PREREQ (2,95)
# define __ASSERT_VOID_CAST static_cast<void>
#else
# define __ASSERT_VOID_CAST (void)
#endif

/* void assert (int expression);

   If NDEBUG is defined, do nothing.
   If not, and EXPRESSION is zero, print an error message and abort.  */

#ifdef        NDEBUG

# define assert(expr)                (__ASSERT_VOID_CAST (0))

/* void assert_perror (int errnum);

   If NDEBUG is defined, do nothing.  If not, and ERRNUM is not zero, print an
   error message with the error text for ERRNUM and abort.
   (This is a GNU extension.) */

# ifdef        __USE_GNU
#  define assert_perror(errnum)        (__ASSERT_VOID_CAST (0))
# endif

#else /* Not NDEBUG.  */

#ifndef _ASSERT_H_DECLS
#define _ASSERT_H_DECLS
__BEGIN_DECLS

/* This prints an "Assertion failed" message and aborts.  */
extern void __assert_fail (__const char *__assertion, __const char *__file,
                           unsigned int __line, __const char *__function)
     __THROW __attribute__ ((__noreturn__));
```

```
/* Likewise, but prints the error text for ERRNUM.  */
extern void __assert_perror_fail (int __errnum, __const char *__file,
                                  unsigned int __line,
                                  __const char *__function)
     __THROW __attribute__ ((__noreturn__));


/* The following is not at all used here but needed for standard
   compliance.  */
extern void __assert (const char *__assertion, const char *__file, int __line
     __THROW __attribute__ ((__noreturn__));


__END_DECLS
#endif /* Not _ASSERT_H_DECLS */

# define assert(expr)                                                   `
  ((expr)                                                               \
   ? __ASSERT_VOID_CAST (0)                                              \
   : __assert_fail (__STRING(expr), __FILE__, __LINE__, __ASSERT_FUNCTION))

# ifdef        __USE_GNU
#  define assert_perror(errnum)
  (!(errnum)                                                             \
   ? __ASSERT_VOID_CAST (0)                                             \
   : __assert_perror_fail ((errnum), __FILE__, __LINE__, __ASSERT_FUNCTION))
# endif

/* Version 2.4 and later of GCC define a magical variable `__PRETTY_FUNCTION_
   which contains the name of the function currently being defined.
   This is broken in G++ before version 2.6.
   C9x has a similar variable called __func__, but prefer the GCC one since
   it demangles C++ function names.  */
# if defined __cplusplus ? __GNUC_PREREQ (2, 6) : __GNUC_PREREQ (2, 4)
#   define __ASSERT_FUNCTION        __PRETTY_FUNCTION__
# else
#  if defined __STDC_VERSION__ && __STDC_VERSION__ >= 199901L
#   define __ASSERT_FUNCTION        __func__
#  else
#   define __ASSERT_FUNCTION        ((__const char *) 0)
#  endif
# endif

#endif /* NDEBUG.  */
```

## 45.3 Arquivos-cabeçalhos da biblioteca padrão

### 45.3.1 Diagnósticos

**assert.h** é o arquivo-cabeçalho que contém informações sobre a biblioteca de diagnósticos. Contém uma única macro, `assert`, que permite aos programadores inserirem pontos de verificação nos programas. Se uma verificação falha, o programa termina sua execução.

No exemplo de trecho de código a seguir, a macro `assert` monitora o valor do índice $i$ do vetor $v$. Se esse valor não está no intervalo especificado, o programa é terminado.

```
assert(0 <= i && i <= 9);
v[i] = 0;
```

### 45.3.2 Manipulação, teste e conversão de caracteres

A biblioteca de manipulação, teste e conversão de caracteres pode ser acessada através das informações contidas no arquivo-cabeçalho **ctype.h**.

#### Funções `is...`

Uma função cujo protótipo se encontra no arquivo-cabeçalho **ctype.h** com identificador da forma **is...** verifica alguma propriedade de um caractere, argumento na chamada da função, e devolve um valor diferente de zero, indicando verdadeiro, isto é, que o caractere tem a propriedade, ou um valor igual a zero, indicando falso ou que o caractere não possui a propriedade.

Listamos abaixo as interfaces das funções da forma **is...** em **ctype.h**, acompanhadas de explicações:

**`int isalnum(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **c** é uma letra (de **'A'** a **'Z'** ou de **'a'** a **'z'**) ou um dígito (de **'0'** a **'9'**);

**`int isalpha(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **c** é uma letra (de **'A'** a **'Z'** ou de **'a'** a **'z'**);

**`int iscntrl(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **c** é um caractere de controle (de **0** a **31** ou **127**);

**`int isdigit(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **c** é um dígito (de **'0'** a **'9'**);

**`int isgraph(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é um caractere imprimível, exceto o espaço em branco, isto é, caracteres **`33`** a **`126`**);

**`int islower(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é uma letra minúscula (de **`'a'`** a **`'z'`**);

**`int isprint(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é um caractere imprimível, isto é, caracteres de **`32`** a **`126`**;

**`int ispunct(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é um caractere de pontuação;

**`int isspace(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é um branco (*whitespace*);

**`int isupper(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é uma letra maiúscula (de **`'A'`** a **`'Z'`**);

**`int isxdigit(int c)`**

Devolve verdadeiro se o conteúdo do parâmetro **`c`** é um dígito em hexadecimal (de **`'0'`** a **`'9'`**, ou de **`'a'`** a **`'f'`**, ou ainda de **`'A'`** a **`'F'`**).

#### Funções `to...`

Uma função em **`ctype.h`** com identificador da forma **`to...`** faz a conversão de um caractere. Se o caractere atende a uma condição, então é convertido. Caso contrário, o caractere é devolvido sem modificações.

Listamos abaixo as interfaces das funções da forma **`to...`** em **`ctype.h`**, acompanhadas de explicações:

**`int tolower(int c)`**

Se o parâmetro **`c`** é uma letra maiúscula (de **`'A'`** a **`'Z'`**), então o caractere é convertido para uma letra minúscula e devolvido;

**`int toupper(int c)`**

Se o parâmetro **`c`** é uma letra minúscula (de **`'a'`** a **`'z'`**), então o caractere é convertido para uma letra maiúscula e devolvido.

### 45.3.3 Erros

A biblioteca de monitoramento de erros é acessada por meio do arquivo-cabeçalho **`errno.h`**. Algumas funções da biblioteca padrão da linguagem C indicam ocorrência de uma falha através do armazenamento de um código de erro, um número inteiro positivo, em `errno`, uma variável do tipo `int` declarada em **`errno.h`**. O trecho de código a seguir mostra o uso da variável `errno`:

```
errno = 0;
y = sqrt(x);
if (errno != 0)
  exit(EXIT_FAIULURE);
```

Nesse trecho de código, observe que inicializamos a variável `errno` declarada na biblioteca **`errno.h`**. Se o valor dessa variável é diferente de zero após a execução da função `sqrt`, isso significa que um erro ocorreu em sua execução, como por exemplo, quando o valor de `x` é negativo.

O valor armazenado na variável `errno` é, em geral, `EDOM` ou `ERANGE`, ambas macros definidas em **`errno.h`**. Essas macros representam os dois tipos de erros que podem ocorrer quando uma função matemática é chamada: erro de domínio ou erro de intervalo, respectivamente. No exemplo acima, se $x$ tem valor negativo, então o valor de `EDOM` é armazenado na variável `errno`. Por outro lado, se o valor devolvido por uma função é muito grande, ultrapassando a capacidade de representação do tipo do valor a ser devolvido pela função, então o valor `ERANGE` é armazenado em `errno`. Por exemplo, se 1000 é argumento da função **`exp`**, então isso provoca um erro de intervalo já que $e^{1000}$ é muito grande para ser representado por um `double`.

### 45.3.4 Características dos tipos com ponto flutuante

Características dos tipos com ponto flutuante podem ser obtidas através da inclusão do arquivo-cabeçalho **`float.h`**. Esse arquivo fornece apenas macros que definem o intervalo e a precisão dos tipos `float`, `double` e `long double`. Não há variáveis, tipos ou funções em **`float.h`**.

Duas macros se aplicam a todos os tipos de ponto flutuante: `FLT_ROUNDS` e `FLT_RADIX`. A primeira descreve a direção do arredondamento. A segunda especifica o comprimento da representação do expoente.

As macros restantes descrevem as características dos tipos específicos de ponto flutuante.

### 45.3.5 Localização

**`locale.h`** é o arquivo-cabeçalho que contém as informações da biblioteca de funções que ajudam um programa a adaptar seu comportamento a um país ou a uma região geográfica. O comportamento específico de um local inclui a forma como os números são impressos, o formato dos valores monetários, o conjunto de caracteres e a aparência da data e da hora.

Com a mudança de um local, um programa pode adaptar seu comportamento para uma área diferente do mundo. Mas essa mudança pode afetar muitas partes da biblioteca, algumas das quais podemos preferir não alterar.

### 45.3.6 Matemática

A biblioteca matemática da linguagem C pode ser acessada através do arquivo-cabeçalho **`math.h`**, que contém os protótipos de diversas funções matemáticas úteis, agrupadas em funções trigonométricas, funções logarítmicas, de exponenciação e de potenciação, além de outras funções.

Quando incluímos o arquivo-cabeçalho **`math.h`** em um programa e usamos suas funções, é importante destacar que há necessidade de adicionar a diretiva de compilação `-lm` no processo de compilação deste programa, na chamada do `gcc`. Caso contrário, um erro de compilação será emitido pelo compilador da linguagem C.

#### Constantes

Uma única macro é definida neste arquivo-cabeçalho: `HUGE_VAL`. Essa macro indica que o valor devolvido por uma função é muito grande para ser representado como um número de precisão dupla. `HUGE_VAL` é do tipo `double` e pode ser entendido como "infinito".

#### Erros

Todas as funções da biblioteca matemática da linguagem C manipulam erros de forma similar. No caso em que o argumento passado à função excede o intervalo permitido, então a variável `errno` recebe o valor `EDOM`. O valor devolvido pela função depende da máquina. No caso do valor devolvido ser muito grande para ser representado como um número de precisão dupla, então a função devolve a macro `HUGE_VAL` e atualiza a variável `errno` para `ERANGE`. Se o valor é muito pequeno para ser representado como um número de precisão dupla, então a função devolve zero. Nesse caso, se a variável `errno` recebe `ERANGE` ou não é uma decisão dependente da máquina.

#### Funções trigonométricas

Listamos abaixo as interfaces das funções trigonométricas no arquivo-cabeçalho **`math.h`**, acompanhadas de explicações:

**`double cos(double x)`**

Devolve o cosseno de um ângulo **`x`** dado em radianos. O valor devolvido está no intervalo $[-1, +1]$;

**`double sin(double x)`**

Devolve o seno de um ângulo **`x`** dado em radianos. O valor devolvido está no intervalo $[-1, +1]$;

**`double tan(double x)`**

Devolve a tangente de um ângulo **`x`** dado em radianos;

**`double acos(double x)`**

Devolve o arco-cosseno de **`x`** em radianos. O valor de **`x`** deve estar no intervalo $[-1, +1]$. O valor devolvido está no intervalo $[0, \pi]$;

**`double asin(double x)`**

Devolve o arco-seno de **`x`** em radianos. O valor de **`x`** deve estar no intervalo $[-1, +1]$. O valor devolvido está no intervalo $[-\pi/2, +\pi/2]$;

**`double atan(double x)`**

Devolve o arco-tangente de **`x`** em radianos. O valor devolvido está no intervalo $[-\pi/2, +\pi/2]$;

**`double atan2(double y, double x)`**

Devolve o arco-tangente em radianos de **`y`**/**`x`** baseado nos sinais de ambos os valores para determinar o quadrante correto. O valor devolvido está no intervalo $[-\pi/2, \pi/2]$;

**`double cosh(double x)`**

Devolve o cosseno hiperbólico de **`x`**;

**`double sinh(double x)`**

Devolve o seno hiperbólico de **`x`**;

**`double tanh(double x)`**

Devolve a tangente hiperbólica de **`x`**. O valor devolvido está no intervalo $[-1, +1]$.

#### Funções logarítmicas, de exponenciação e de potenciação

Listamos abaixo as interfaces das funções logarítmicas, de exponenciação e de potenciação em **`math.h`**, acompanhadas de explicações:

**`double exp(double x)`**

Devolve o valor de **`e`** elevado à **`x`**-ésima potência;

**`double frexp(double x, int *expoente)`**

O número com ponto flutuante **`x`** é subdividido em uma mantissa e um expoente. O valor devolvido pela função é a mantissa e o parâmetro de entrada e saída **`expoente`** contém o expoente. Observe que

$$\texttt{x} = \texttt{matissa} \times 2^{\texttt{expoente}} .$$

A **`mantissa`** deve estar no intervalo $[0.5, 1.0]$.

**`double ldexp(double x, int expoente)`**

Devolve **`x`** multiplicado por 2 elevado à potência **`expoente`**, isto é, $\texttt{x} \times 2^{\texttt{expoente}}$.

**`double log(double x)`**

Devolve o logaritmo natural de **`x`**, isto é, o logaritmo na base **`e`**;

**`double log10(double x)`**

Devolve o logaritmo de **`x`** na base 10;

**`double modf(double x, double *inteiro)`**

Subdivide o número com ponto flutuante **`x`** nas partes inteira e fracionária. O valor devolvido pela função é a parte fracionária, depois do ponto decimal, e o parâmetro de entrada e saída **`inteiro`** contém a parte inteira de **`x`**;

**`double pow(double x, double y)`**

Devolve **`x`** elevado à potência **`y`**. O valor de **`x`** não pode ser negativo se **`y`** é um valor fracionário. **`x`** não pode ser zero se **`y`** é menor ou igual a zero.

**`double sqrt(double x)`**

Devolve a raiz quadrada de **`x`**. O valor armazenado em **`x`** não pode ser negativo.

#### Outras funções

Listamos abaixo as interfaces de outras funções matemáticas em **`math.h`**, acompanhadas de explicações:

**`double ceil(double x)`**

Devolve o menor inteiro que é maior ou igual a **`x`**;

**`double floor(double x)`**

Devolve o maior inteiro que é menor ou igual a **`x`**;

**`double fabs(double x)`**

Devolve o valor absoluto de **`x`**. Isto é, se **`x`** é um valor negativo, a função devolve o valor positivo. Caso contrário, devolve o valor de **`x`**;

**`double fmod(double x, double y)`**

Devolve o resto de **`x`** dividido por **`y`**. O valor de **`y`** deve ser diferente de zero.

### 45.3.7 Saltos não-locais

**`setjmp.h`** é o arquivo-cabeçalho que contém informações sobre saltos não-locais de baixo nível para controle de chamadas de funções e desvio de fluxo de execução das mesmas. Duas funções constam desta biblioteca, uma delas que marca um ponto no programa e outra que pode ser usada para fazer com que o programa tenha seu fluxo de execução desviado para um dos pontos marcados. Isso significa que é possível, com o uso das funções desta biblioteca, uma função saltar diretamente para uma outra função sem que tenha encontrado uma sentença `return`. Esses recursos são usados especialmente para depuração de erros.

### 45.3.8 Manipulação de sinais

Arquivo-cabeçalho **`signal.h`** que contém informações sobre a biblioteca de manipulação de sinais **`signal.h`**. Contém funções que tratam de condições excepcionais, incluindo interrupções e erros de tempo de execução. Alguns sinais são assíncronos e podem acontecer a qualquer instante durante a execução de um programa, não apenas em certos pontos do códigos que são conhecidos pelo programador.

### 45.3.9 Número variável de argumentos

Biblioteca de ferramentas para tratamento de funções que possuem um número variável de argumentos acessada pelo arquivo-cabeçalho `stdarg.h`. Funções como `printf` e `scanf` permitem um número variável de argumentos e programadores também podem construir suas próprias funções com essa característica.

### 45.3.10 Definições comuns

O arquivo-cabeçalho **`stddef.h`** contém informações sobre definições de tipos e macros que são usadas com freqüência nos programas. Não há funções declaradas neste arquivo.

### 45.3.11 Entrada e saída

A biblioteca padrão de entrada e saída da linguagem C é acessada através do arquivo-cabeçalho **`stdio.h`** e contém tipos, macros e funções para efetuar entrada e saída de dados, incluindo operações sobre arquivos. Listamos a seguir apenas alguns desses elementos.

#### Entrada e saída formatadas

Listamos abaixo as interfaces das funções de entrada e saída formatadas em **`stdio.h`**, acompanhadas de explicações:

**`int printf(const char *formato, ...)`**

Imprime informações na saída padrão, tomando uma cadeia de caracteres de formatação especificada pelo argumento **`formato`** e aplica para cada argumento subseqüente o especificador de conversão na cadeia de caracteres de formatação, da esquerda para direita. Veja a aula 18 para informações sobre especificadores de conversão.

O número de caracteres impressos na saída padrão é devolvido. Se um erro ocorreu na chamada da função, então o valor $-1$ é devolvido.

**`int scanf(const char *formato, ...)`**

Lê dados de uma forma que é especificada pelo argumento **`formato`** e e armazenada cada valor lido nos argumentos subseqüentes, da esquerda para direita. Cada entrada formatada é definida na cadeia de caracteres de formatação, através de um especificador de conversão precedido pelo símbolo `%` que especifica como uma entrada deve ser armazenada na variável respectiva. Outros caracteres listados na cadeia de caracteres de

formatação especificam caracteres que devem corresponder na entrada mas não armazenados nos argumentos da função. Um branco pode corresponder a qualquer outro branco ou ao próximo caractere incompatível. Mais informações sobre entrada formatada, veja a aula 18.

Se a execução da função obteve sucesso, o número de valores lidos, convertidos e armazenados nas variáveis é devolvido.

#### Entrada e saída de caracteres

Listamos abaixo as interfaces das funções de entrada e saída de caracteres em **`stdio.h`**, acompanhadas de explicações:

**`int getchar(void)`**

Lê um caractere (um **`unsigned char`**) da entrada padrão. Se a leitura tem sucesso, o caractere é devolvido;

**`int putchar(int caractere)`**

Imprime um caractere (um **`unsigned char`**) especificado pelo argumento **`caractere`** na saída padrão. Se a impressão tem sucesso, o caractere é devolvido.

### 45.3.12 Utilitários gerais

A biblioteca de utilitários gerais da linguagem C pode ser acessada pelo arquivo-cabeçalho **`stdlib.h`**. É uma miscelânea de definições de tipos, macros e funções que não se encaixam nos outros arquivos-cabeçalhos.

#### Constantes

Listamos abaixo as macros definidas em **`stdlib.h`**, acompanhadas de explicações:

**`NULL`**

Valor de um apontador nulo;

**`EXIT_FAILURE`** e **`EXIT_SUCCESS`**

Valores usados pela função `exit` para devolver o estado do término da execução de um programa;

**`RAND_MAX`**

Valor máximo devolvido pela função `rand`.

#### Funções sobre cadeias de caracteres

Listamos abaixo as interfaces das funções de cadeias de caracteres em **`stdlib.h`**, acompanhadas de explicações:

**`double atof(const char *cadeia)`**

A cadeia de caracteres `cadeia` é convertida e devolvida como um número de ponto flutuante do tipo `double`. Brancos iniciais são ignorados. O número pode conter um sinal opcional, uma seqüência de dígitos com um caractere de ponto decimal opcional, mais uma letra `e` ou `E` opcional seguida por um expoente opcionalmente sinalizado. A conversão pára quando o primeiro caractere não reconhecível é encontrado.

Se a conversão foi realizada com sucesso, o número é devolvido. Caso contrário, zero é devolvido;

**`int atoi(const char *cadeia)`**

A cadeia de caracteres `cadeia` é convertida e devolvida como um número inteiro do tipo `int`. Bancos iniciais são ignorados. O número pode conter um sinal opcional, mais uma seqüência de dígitos. A conversão pára quando o primeiro caractere não reconhecível é encontrado.

Se a conversão foi realizada com sucesso, o número é devolvido. Caso contrário, zero é devolvido;

**`long int atol(const char *cadeia)`**

A cadeia de caracteres `cadeia` é convertida e devolvida como um número inteiro do tipo `long int`. Brancos iniciais são ignorados. O número pode conter um sinal opcional, mais uma seqüência de dígitos. A conversão pára quando o primeiro caractere não reconhecível é encontrado.

Se a conversão foi realizada com sucesso, o número é devolvido. Caso contrário, zero é devolvido.

#### Funções de ambiente

Listamos abaixo as interfaces das funções de ambiente em **`stdlib.h`**, acompanhadas de explicações:

**`void abort(void)`**

Finaliza o programa anormalmente, sinalizando o término sem sucesso para o ambiente;

**`void exit(int estado)`**

Finaliza o programa normalmente, sinalizando para o ambiente o valor armazenado em `estado`, que pode ser `EXIT_FAILURE` ou `EXIT_SUCCESS`.

#### Funções matemáticas

Listamos abaixo as interfaces das funções matemáticas em **`stdlib.h`**, acompanhadas de explicações:

**`int abs(int x)`**

Devolve o valor absoluto de `x`;

**`long int labs(long int x)`**

Devolve o valor absoluto de `x`;

**`int rand(void)`**

Devolve um número pseudo-aleatório no intervalo $[0,$ `RAND_MAX` $]$;

**`void srand(unsigned int semente)`**

Inicializa o gerador de número pseudo-aleatórios usado pela função `rand`. Inicializar `srand` com o mesmo valor de `semente` faz com que a função `rand` devolva a mesma seqüência de números pseudo-aleatórios. Se `srand` não for executada, `rand` age como se `srand(1)` tivesse sido executada.

### 45.3.13 Manipulação de cadeias

A biblioteca de manipulação de cadeias de caracteres pode ser acessada através do arquivo-cabeçalho **`string.h`**.

#### Constantes e tipos

A macro **`NULL`** é definida em **`string.h`**, para denotar uma cadeia de caracteres vazia, assim como o tipo **`typedef size_t`**, que permite a manipulação de variáveis que contêm comprimentos de cadeias de caracteres. Ou seja, esse tipo é na verdade um tipo `int`.

#### Funções

Listamos abaixo as principais funções em **`string.h`**, acompanhadas de explicações:

**`char *strcat(char *cadeia1, const char *cadeia2)`**

Concatena a cadeia de caracteres `cadeia2` no final da cadeia de caracteres `cadeia1`. O caractere nulo `'\0'` da `cadeia1` é sobrescrito. A concatenação termina quando o caractere nulo da `cadeia2` é copiado. Devolve um apontador para a `cadeia1`;

**`char *strncat(char *cadeia1, const char *cadeia2, size_t n)`**

Concatena até `n` caracteres da cadeia de caracteres `cadeia2` no final da cadeia de caracteres `cadeia1`. O caractere nulo `'\0'` da `cadeia1` é sobrescrito. O caractere nulo é sempre adicionado na `cadeia1`. Devolve um apontador para a `cadeia1`;

**`int strcmp(const char *cadeia1, const char *cadeia2)`**

Compara as cadeias de caracteres `cadeia1` e `cadeia2`. Devolve zero se `cadeia1` e `cadeia2` são iguais. Devolve um valor menor que zero ou maior que zero se `cadeia1` é menor que ou maior que `cadeia2`, respectivamente;

**`int strncmp(const char *cadeia1, const char *cadeia2, unsigned int n)`**

Compara no máximo os primeiros `n` caracteres de `cadeia1` e `cadeia2`. Finaliza a comparação após encontrar um caractere nulo. Devolve zero se os `n` primeiros caracteres, ou os caracteres até alcançar o caractere nulo, de `cadeia1` e `cadeia2` são iguais. Devolve um valor menor que zero ou maior que zero se `cadeia1` é menor que ou maior que `cadeia2`, respectivamente;

**`char *strcpy(char *cadeia1, const char *cadeia2)`**

Copia a cadeia de caracteres `cadeia2` na cadeia de caracteres `cadeia1`. Copia até o caractere nulo de `cadeia2`, inclusive. Devolve um apontador para `cadeia1`;

**`char *strncpy(char *cadeia1, const char *cadeia2, unsigned int n)`**

Copia até `n` caracteres de `cadeia2` em `cadeia1`. A cópia termina quando `n` caracteres são copiados ou quando o caractere nulo em `cadeia2` é alcançado. Se o caractere nulo é alcançado, os caracteres nulos são continuamente copiados para `cadeia1` até que `n` caracteres tenham sido copiados;

**`size_t strlen(const char *cadeia)`**

Computa o comprimento da `cadeia`, não incluindo o caractere nulo. Devolve o número de caracteres da `cadeia`;

**`char *strpbrk(const char *cadeia1, const char *cadeia2)`**

Procura o primeiro caractere na `cadeia1` que corresponde a algum caractere especificado na `cadeia2`. Se encontrado, um apontador para a localização deste caractere é devolvido. Caso contrário, o apontador para nulo é devolvido;

**`char *strchr(const char *cadeia, int c)`**

Busca pela primeira ocorrência do caractere `c` na cadeia de caracteres `cadeia`. Devolve um apontador para a posição que contém o primeiro caractere na `cadeia` que corresponde a `c` ou nulo se `c` não ocorre em `cadeia`;

**`char *strrchr(const char *cadeia, int c)`**

Busca pela última ocorrência do caractere `c` na cadeia de caracteres `cadeia`. Devolve um apontador para a posição que contém o último caracter na `cadeia` que corresponde `c` ou nulo se `c` não ocorre em `cadeia`;

**`size_t strspn(const char *cadeia1, const char *cadeia2)`**

Encontra a primeira seqüência de caracteres na `cadeia1` que contém qualquer caracter da `cadeia2`. Devolve o comprimento desta primeira seqüência de caracteres que corresponde à `cadeia2`;

**`size_t strcspn(const char *cadeia1, const char *cadeia2)`**

Encontra a primeira seqüência de caracteres na `cadeia1` que não contém qualquer caracter da `cadeia2`. Devolve o comprimento desta primeira seqüência de caracteres que não corresponde à `cadeia2`;

**`char *strstr(const char *cadeia1, const char *cadeia2)`**

Encontra a primeira ocorrência da `cadeia2` na `cadeia1`. Devolve um apontador para a localização da primeira ocorrência da `cadeia2` na `cadeia1`. Se não há correspondência, o apontador nulo é devolvido. Se `cadeia2` contém uma cadeia de caracteres de comprimento zero, então `cadeia1` é devolvida.

### 45.3.14 Data e hora

A biblioteca que contém funções para determinar, manipular e formatar horários e datas pode ser acessada através do arquivo-cabeçalho **`time.h`**.

## Exercícios

45.1 (a) Escreva uma função com a seguinte interface

```
double cosseno(double x, double epsilon)
```

que receba um número real $x$ e um número positivo real $\varepsilon > 0$, e calcule uma aproximação para $\cos x$, onde $x$ é dado em radianos, através da seguinte série:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^k \frac{x^{2k}}{(2k)!} + \ldots$$

incluindo todos os termos $T_k = \frac{x^{2k}}{(2k)!}$, até que $T_k < \varepsilon$.

(b) Escreva uma função com a seguinte interface:

```
double seno(double x, double epsilon)
```

que receba um número real $x$ e um número real $\varepsilon > 0$, e calcule uma aproximação para $\operatorname{sen} x$, onde $x$ é dado em radianos, através da seguinte série:

$$\operatorname{sen} x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \ldots$$

incluindo todos os termos $T_k = \frac{x^{2k+1}}{(2k+1)!}$, até que $T_k < \varepsilon$.

(c) Escreva um programa que receba um número real $x$ representando um ângulo em radianos e um número real $\varepsilon > 0$ representando uma precisão, e calcule o valor de $\tan x$ de duas formas: usando as funções dos itens acima e usando a função `tan` da biblioteca `math`. Compare os resultados.

45.2 (a) Escreva uma função com a seguinte interface:

```
int ocorre(char palavra[], char frase[], int pos)
```

que receba duas cadeias de caracteres `palavra` e `frase` e um inteiro `pos` e verifique se a cadeia de caracteres `palavra` ocorre na posição `pos` da cadeia de caracteres `frase`. Em caso positivo, a função deve devolver 1. Em caso negativo, deve devolver zero.

(b) Escreva um programa que receba duas cadeias de caracteres `padrao` e `texto`, com $m$ e $n$ caracteres respectivamente e $m \leqslant n$, e imprima o número de vezes que `padrao` ocorre em `texto`.

Use todas as funções da biblioteca `string` que puder.

AULA 46

# PRÉ-PROCESSADOR

O pré-processador é um módulo da linguagem C que edita um programa antes de sua compilação. É uma ferramenta poderosa e que diferencia a linguagem C das demais linguagens de programação do alto nível. As diretivas `#include` e `#define` que usamos em aulas anteriores são manipuladas pelo pré-processador, assim como outras que veremos nesta aula. Apesar de poderoso, o mal uso do pré-processador pode produzir programas praticamente ininteligíveis e/ou com erros muito difíceis de encontrar.

## 46.1 Funcionamento

As **diretivas de pré-processamento** controlam o comportamento do pré-processador. Uma diretiva do pré-processador é um comando que inicia com o caractere `#`. Até o momento, vimos duas das diretivas do pré-processador da linguagem C: `#include` e `#define`. Revisaremos essas diretivas adiante.

A figura 46.1 ilustra o papel do pré-processador durante o processo de compilação.

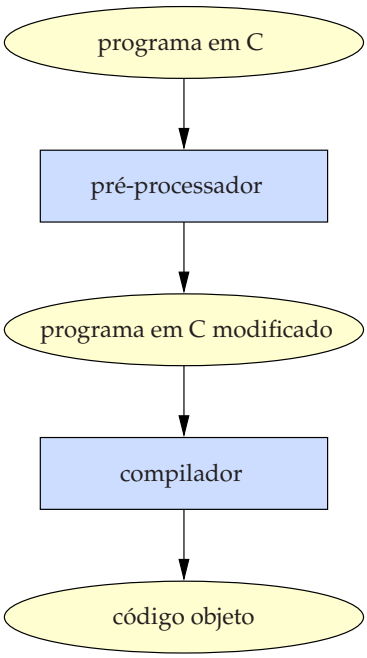



Figura 46.1: O papel do pré-processador durante a compilação.

A entrada para o pré-processador é um programa escrito na linguagem C, possivelmente contendo diretivas. O pré-processador executa então estas diretivas, eliminando-as durante este processo. A saída produzida pelo pré-processador é um outro programa na linguagem C, que representa uma versão editada do programa original, sem diretivas. A saída do pré-processador é a entrada para o compilador, que verifica erros no programa e realiza sua tradução para o código objeto, que contém apenas instruções diretas para a máquina.

## 46.2 Diretivas de pré-processamento

A maioria das diretivas de pré-processamento pertencem a uma das três categorias a seguir:

- **definição de macros:** a diretiva `#define` define uma macro, também chamada de constante simbólica; a diretiva `#undef` remove a definição de uma macro;
- **inclusão de arquivos:** a diretiva `#include` faz com que o conteúdo de um arquivo especificado seja incluído em um programa;
- **compilação condicional:** as diretivas `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` e `#endif` permitem que trechos de código seja incluídos ou excluídos do programa, dependendo das condições verificadas pelo pré-processador.

As outras diretivas `#error`, `#line` e `#pragma` são mais específicas e usadas com menos freqüência. Antes de descrevê-las com mais detalhes, listamos abaixo as regras que se aplicam a todas elas:

- **Diretivas sempre iniciam com o símbolo `#`.** Não é necessário que o símbolo `#` seja o primeiro caractere da linha. No entanto, apenas espaços podem precedê-lo. Após `#` o identificador de uma diretiva deve ser descrito, seguido por qualquer outra informação que a diretiva necessite.
- **Qualquer número de espaços e caracteres de tabulação horizontal podem separar os itens em uma diretiva.** Por exemplo, a diretiva a seguir está correta:

```
#    define      MAX         100
```

- **Diretivas sempre terminam no primeiro caractere de mudança de linha, a menos que a continuação seja explícita.** Para continuar uma diretiva em uma próxima linha, devemos finalizar a linha atual com o caractere `\`. Por exemplo, a diretiva abaixo define uma macro que representa a capacidade de um disco medida em bytes:

```
#define CAPACIDADE_DISCO (LADOS *               \
                          TRILHAS_POR_LADO *    \
                          SETORES_POR_TRILHA *  \
                          BYTES_POR_SETOR)
```

- **Diretivas podem aparecer em qualquer lugar em um programa.** Apesar de usualmente colocarmos as diretivas `#include` e `#define` no começo de um arquivo, outras diretivas são mais prováveis de ocorrer em outros pontos do programa.
- **Comentários podem ocorre na mesma linha de uma diretiva.** Em geral, colocamos um comentário no final da definição de uma macro para explicar seu significado. Por exemplo:

```
#define TAMANHO 100     /* Dimensão dos vetores */
```

## 46.3 Definições de macros

O uso mais freqüente da diretiva `#define` do pré-processador da linguagem C é a atribuição de nomes simbólicos para constantes. Macros como essas são conhecidas como macros simples. O pré-processador também suporta definição de macros parametrizadas. Nesta aula veremos apenas as macros simples.

A definição de uma **macro simples** tem a seguinte forma:

```
#define identificador lista-de-troca
```

onde `lista-de-troca` é uma seqüência qualquer de itens. Essa lista pode incluir identificadores, palavras reservadas, constantes numéricas, constantes de caracteres, literais, operadores e pontuação. Quando encontra uma definição de uma macro, o pré-processador toma nota que o `identificador` representa a `lista-de-troca`. Sempre que `identificador` ocorre posteriormente no arquivo, o pré-processador o substitui pela `lista-de-troca`.

Um erro freqüente na definição de macros é a inclusão de símbolos extras, que conseqüentemente farão parte da lista de troca. Abaixo são mostrados dois exemplos de erros como esse.

```
#define TAM = 100
#define N 10;
...
int u[TAM];
double v[N];
```

A tradução do trecho de código acima realizada pelo pré-processador gera o seguinte trecho de código modificado, correspondente às duas última linhas:

```
int u[= 100];
double v[10;];
```

É certo que o compilador acusará erros nessa duas linhas do programa.

Podemos usar macros para dar nomes a valores numéricos, caracteres e literais, como ilustrado abaixo:

```
#define COMPRIMENTO 80
#define VERDADEIRO  1
#define FALSO       0
#define PI          3.141592
#define ENTER       '\n'
#define ERRO        "Erro: memória insuficiente\n"
```

Usar a diretiva `#define` para criar nomes para constantes tem diversas vantagens, como tornar os programas mais fáceis de ler e de modificar, ajudar a evitar inconsistências e erros tipográficos, permitir pequenas mudanças na sintaxe da linguagem, renomear tipos e controlar a compilação condicional.

## 46.4 Inclusão de arquivos-cabeçalhos

Vamos recordar a aula 45. A diretiva `#include` ordena o pré-processador a abrir um arquivo especificado e a inserir seu conteúdo no arquivo atual. Assim, se queremos que vários arquivos de código fonte tenham acesso à mesma informação, devemos colocar essa informação em um arquivo e então usar a diretiva `#include` para trazer o conteúdo do arquivo em cada arquivo de código fonte. Arquivos que são incluídos dessa forma são chamados de **arquivos-cabeçalhos**, do inglês *header files* ou também *include files*. Por convenção, um arquivo como esse tem a extensão `.h`.

A diretiva `#include` pode ser usada de duas formas. A primeira forma é usada para arquivos-cabeçalhos que pertencem à biblioteca padrão da linguagem C:

```
#include <arquivo.h>
```

A segunda forma é usada para todos os outros arquivos-cabeçalhos, incluindo aqueles que são escritos por programadores(as):

```
#include "arquivo.h"
```

Programa 46.1: Um programa curioso.

```
#include <stdio.h>

#define PAR      0
#define IMPAR    1
#define Programa main
#define inicio   {
#define fim      }
#define escreva  printf
#define leia     scanf
#define se       if
#define senao    else
#define enquanto while
#define devolva  return

int verificaParidade(int numero)
inicio
  int paridade;
  se (numero % 2 == 0)
    paridade = PAR;
  senao
    paridade = IMPAR;
  devolva paridade;
fim

int Programa(void)
inicio
  int i, n, numero, somapar, somaimpar;
  escreva("Informe a quantidade de elementos: ");
  leia("%d", &n);
  somapar = 0;
  somaimpar = 0;
  i = 1;
  enquanto (i <= n)
  inicio
    escreva("Informe um número: ");
    leia("%d", &numero);
    se (verificaParidade(numero) == PAR)
      somapar = somapar + numero;
    senao
      somaimpar = somaimpar + numero;
    i++;
  fim
  escreva("\nSoma dos números pares: %d\n", somapar);
  escreva("Soma dos números ímpares: %d\n", somaimpar);
  devolva 0;
fim
```

A diferença entre as duas formas se dá pela maneira como o compilador busca o arquivo-cabeçalho. Na primeira, o compilador busca o arquivo-cabeçalho no(s) diretório(s) em que os arquivos-cabeçalhos do sistema se encontram. Nos sistemas baseados no UNIX, como o LINUX, os arquivos-cabeçalhos são mantidos usualmente no diretório `/usr/include`. Na segunda forma, a busca é realizada no diretório corrente e, em seguida, no(s) diretório(s) em que os arquivos-cabeçalhos do sistema se encontram.

Arquivos-cabeçalhos auxiliam no compartilhamento de definições de macros, de tipos e de protótipos de funções por dois ou mais arquivos de código fonte. Na aula 47, quando apresentamos formas de tratar programas muito grandes divididos em arquivos diferentes, mencionamos novamente a diretiva `#include`, destacando formas de seu uso no compartilhamento de definições de macros, de tipos e de protótipos de funções.

## 46.5 Compilação condicional

O pré-processador da linguagem C reconhece diretivas que permitem a inclusão ou exclusão de um trecho de programa dependendo do resultado da avaliação de um teste executado pelo pré-processador. Esse processo é chamado de **compilação condicional**. Em geral, as diretivas de compilação condicional são muito usadas para auxiliar na depuração dos programas. As diretivas `#if` e `#endif`, por exemplo, são usadas em conjunto para verificar se um trecho de código será ou não executado, dependendo da condição a ser avaliada. O formato geral da diretiva `#if` é

```
#if expressão-constante
```

e da diretiva `#endif` é simplesmente

```
#endif
```

Quando o pré-processador encontra a diretiva `#if`, ele avalia a `expressão-constante`. Se o resultado da avaliação é igual a zero, as linhas entre `#if` e `#endif` serão removidas do programa durante o pré-processamento. Caso contrário, as linhas entre `#if` e `#endif` serão mantidas e processadas pelo compilador. O seguinte trecho de código exemplifica o uso dessas diretivas e permite verificar seu uso no auxílio do processo de depuração.

```
#define DEBUG 1
...
#if DEBUG
printf("Valor de i = %d\n", i);
printf("Valor de j = %d\n", j);
#endif
```

Durante o pré-processamento, a diretiva `#if` verifica o valor de `DEBUG`. Como seu valor é diferente de zero, o pré-processador mantém as duas chamadas à função `printf` no programa, mas as linhas contendo as diretivas `#if` e `#endif` serão eliminadas após o pré-processamento. Se modificamos o valor de `DEBUG` para zero e recompilamos o programa, então o pré-processador remove todas as 4 linhas do programa. É importante notar que a diretiva `#if` trata identificadores não definidos como macros que têm o valor zero. Assim, se esquecemos de definir `DEBUG`, o teste

```
#if DEBUG
```

irá falhar, mas não produzirá uma mensagem de erro. O teste

```
#if !DEBUG
```

produzirá o resultado verdadeiro, nesse caso.

Nesse sentido, há duas outras diretivas para verificar se um identificador é definido ou não como uma macro naquele ponto do programa: `#ifdef` e `#ifndef`.

O formato geral dessas diretivas é

```
#ifdef identificador
Linhas a serem incluídas se o identificador está definido como uma macro
#endif
```

e

```
#ifndef identificador
Linhas a serem incluídas se o identificador não está definido como uma macro
#endif
```

As diretivas `#elif` e `#else` podem ser usadas em conjunto com as diretivas `#if`, `#ifdef` e `#ifndef` quando aninhamento de blocos são necessários. O formato geral dessas diretivas é apresentado abaixo:

```
#elif expressão-constante
```

e

```
#else
```

Por exemplo, podemos verificar uma série de condições como a seguir:

```
#if expressão1
Linhas a serem incluídas se expressão1 é diferente de zero
#elif expressão2
Linhas a serem incluídas se expressão2 é diferente de zero
#else
Linhas a serem incluídas em caso contrário
#endif
```

A compilação condicional, além de auxiliar na depuração de programas, também tem muitos outros usos como na construção de programas que podem ser transportados de uma máquina para outra ou de um sistema operacional para outro, na construção de programas que podem ser compilados por compiladores diferentes, no suporte à definições padronizadas de macros, entre outros.

## 46.6 Outras diretivas

As diretivas `#error`, `#line` e `#pragma` são mais específicas e por isso menos usadas.

A diretiva `#error` faz com que o pré-processador imprima uma mensagem de erro na saída padrão. A diretiva `#error` tem o seguinte formato geral:

```
#error mensagem
```

onde `mensagem` é uma seqüência qualquer de caracteres. Um exemplo de uso dessa diretiva é apresentado no trecho de código a seguir:

```
#ifdef LINUX
...
#else
#error Sistema operacional não especificado
#endif
```

Se o pré-processador encontra uma diretiva `#error`, isso é sinal de que uma falha grave ocorreu no programa e alguns compiladores terminam imediatamente a compilação.

A diretiva `#line` é usada para alterar a forma como as linhas do programa são numeradas. O formato geral dessa diretiva é apresentado abaixo:

```
#line n
```

e

```
#line n "arquivo"
```

No primeiro formato, as linhas do programa são numeradas a partir do número `n`. No segundo, as linhas do programa no `arquivo` são numeradas a partir do número `n`. Muitos compiladores usam essa informação quando precisam gerar mensagens de erro.

Por fim, a diretiva `#pragma` permite que uma diretiva seja criada pelo(a) programador(a). O formato geral dessa diretiva é dado abaixo:

```
#pragma diretiva
```

AULA 47

# PROGRAMAS EXTENSOS

Nesta aula aprenderemos como dividir e distribuir nossas funções em vários arquivos. Esta possibilidade é uma característica importante da linguagem C, permitindo que o(a) programador(a) possa ter sua própria biblioteca de funções e possa usá-la em conjunto com um ou mais programas.

Os programas que escrevemos até o momento são bem simples e, por isso mesmo pequenos, com poucas linhas de código. No entanto, programas pequenos são uma exceção. À medida que os problemas têm maior complexidade, os programas para solucioná-los têm, em geral, proporcionalmente mais linhas de código. Por exemplo, a versão 2.6.25 do núcleo do sistema operacional LINUX, de abril de 2008, tem mais de nove milhões de linhas de código na linguagem C e seria impraticável mantê-las todas no mesmo arquivo[1]. Nesta aula veremos que um programa na linguagem C consiste de vários arquivos-fontes e também de alguns arquivos-cabeçalhos. aprenderemos a dividir nossos programas em múltiplos arquivos.

## 47.1 Arquivos-fontes

Até a última aula, sempre consideramos que um programa na linguagem C consiste de um único arquivo. Na verdade, um programa pode ser dividido em qualquer número de **arquivos-fontes** que, por convenção, têm a extensão `.c`. Cada arquivo-fonte contém uma parte do programa, em geral definições de funções e variáveis. Um dos arquivos-fontes de um programa deve necessariamente conter uma função `main`, que é o ponto de início do programa. Quando dividimos um programa em arquivos, faz sentido colocar funções relacionadas e variáveis em um mesmo arquivo-fonte. A divisão de um programa em arquivos-fontes múltiplos tem vantagens significativas:

- agrupar funções relacionadas e variáveis em um único arquivo ajuda a deixar clara a estrutura do programa;
- cada arquivo-fonte pode ser compilado separadamente, com uma grande economia de tempo se o programa é grande é é modificado muitas vezes;
- funções são mais facilmente re-usadas em outros programas quando agrupadas em arquivos-fontes separados.

[1] O sistema operacional LINUX, por ser livre e de código aberto, permite que você possa consultar seu código fonte, além de modificá-lo a seu gosto.

## 47.2 Arquivos-cabeçalhos

Nas aulas 45 e 46 abordamos com algum detalhe os arquivos-cabeçalhos. Quando dividimos um programa em vários arquivos-fonte, como uma função definida em um arquivo pode chamar uma outra função definida em outro arquivo? Como dois arquivos podem compartilhar a definição de uma mesma macro ou a definição de um tipo? A diretiva `#include` nos ajuda a responder a essas perguntas, permitindo que essas informações possam ser compartilhadas entre arquivos-fontes.

Muitos programas grandes contêm definições de macros e definições de tipos que necessitam ser compartilhadas por vários arquivos-fontes. Essas definições devem ser mantidas em arquivos-cabeçalhos.

Por exemplo, suponha que estamos escrevendo um programa que usa macros com nomes `LOGIC`, `VERDADEIRO` e `FALSO`. Ao invés de repetir essas macros em cada arquivo-fonte do programa que necessita delas, faz mais sentido colocar as definições em um arquivo-cabeçalho com um nome como `logico.h` tendo as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
#define LOGIC      int
```

Qualquer arquivo-fonte que necessite dessas definições deve conter simplesmente a linha a seguir:

```
#include "logico.h"
```

Definições de tipos também são comuns em arquivos-cabeçalhos. Por exemplo, ao invés de definir a macro `LOGIC` acima, podemos usar `typedef` para criar um tipo `logic`. Assim, o arquivo `logico.h` terá as seguintes linhas:

```
#define VERDADEIRO 1
#define FALSO      0
typedef logic int;
```

A figura 47.1 mostra um exemplo de dois arquivos-fontes que incluem o arquivo-cabeçalho `logico.h`.

Colocar definições de macros e tipos em um arquivo-cabeçalho tem algumas vantagens. Primeiro, economizamos tempo por não ter de copiar as definições nos arquivos-fontes onde são necessárias. Segundo, o programa torna-se muito mais fácil de modificar, já que a modificação

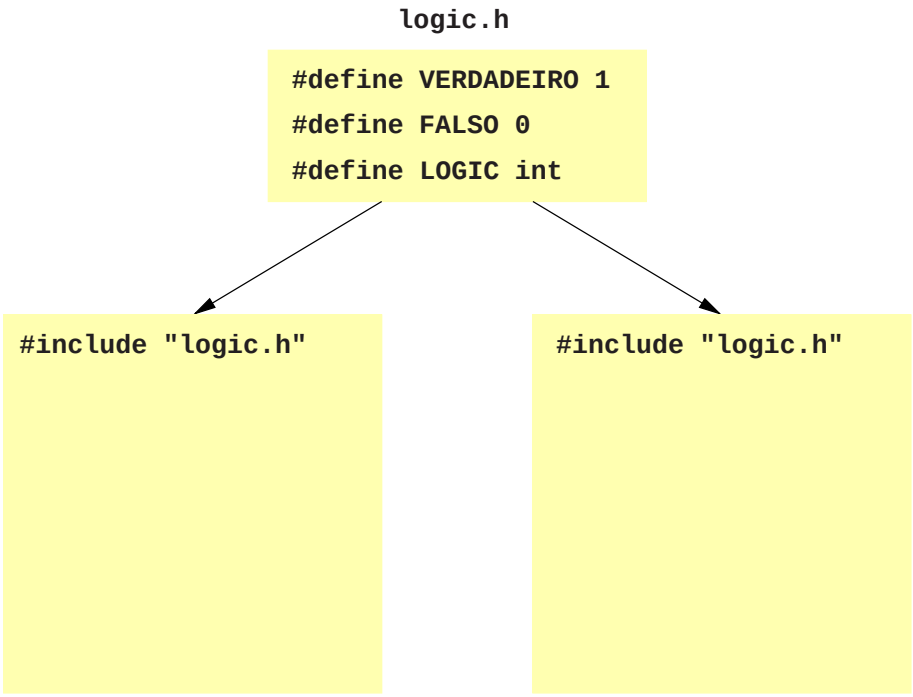



Figura 47.1: Inclusão de um arquivo-cabeçalho em dois arquivos-fontes.

da definição de uma macro ou de um tipo necessita ser feita em um único arquivo-cabeçalho. E terceiro, não temos de nos preocupar com inconsistências em conseqüência de arquivos-fontes contendo definições diferentes da mesma macro ou tipo.

Suponha agora que um arquivo-fonte contém uma chamada a uma função `soma` que está definida em um outro arquivo-fonte, com nome `calculos.c`. Chamar a função `soma` sem sua declaração pode ocasionar erros de execução. Quando chamamos uma função que está definida em outro arquivo, é sempre importante ter certeza que o compilador viu sua declaração, isto é, seu protótipo, antes dessa chamada.

Nosso primeiro impulso é declarar a função `soma` no arquivo-fonte onde ela foi chamada. Isso resolve o problema, mas pode criar imensos problemas de gerenciamento. Suponha que a função é chamada em cinqüenta arquivos-fontes diferentes. Como podemos assegurar que os protótipos das funções `soma` são idênticos em todos esses arquivos? Como podemos garantir que todos esses protótipos correspondem à definição de `soma` em `calculos.c`? Se `soma` deve ser modificada posteriormente, como podemos encontrar todos os arquivos-fontes onde ela é usada? A solução é evidente: coloque o protótipo da função `soma` em um arquivo-cabeçalho e inclua então esse arquivo-cabeçalho em todos os lugares onde `soma` é chamada. Como `soma` é definida em `calculos.c`, um nome natural para esse arquivo-cabeçalho é `calculos.h`. Além de incluir `calculos.h` nos arquivos-fontes onde `soma` é chamada, precisamos incluí-lo em `calculos.c` também, permitindo que o compilador verifique que o protótipo de `soma` em `calculos.h` corresponde à sua definição em `calculos.c`. É regra sempre incluir o arquivo-cabeçalho que declara uma função em um arquivo-fonte que contém a definição dessa função. Não fazê-lo pode ocasionar erros difíceis de encontrar.

Se `calculos.c` contém outras funções, muitas delas devem ser declaradas no mesmo arquivo-cabeçalho onde foi declarada a função `soma`. Mesmo porque, as outras funções em

`calculos.c` são de alguma forma relacionadas com `soma`. Ou seja, qualquer arquivo que contenha uma chamada à `soma` provavelmente necessita de alguma das outras funções em `calculos.c`. Funções cuja intenção seja usá-las apenas como suporte dentro de `calculos.c` não devem ser declaradas em um arquivo-cabeçalho.

Para ilustrar o uso de protótipos de funções em arquivos-cabeçalhos, vamos supor que queremos manter diversas definições de funções relacionadas a cálculos geométricos em um arquivo-fonte `geometricas.c`. As funções são as seguintes:

```
double perimetroQuadrado(double lado)
{
  return 4 * lado;
}
double perimetroTriangulo(double lado1, double lado2, double lado3)
{
  return lado1 + lado2 + lado3;
}
double perimetroCirculo(double raio)
{
  return 2 * PI * raio;
}
double areaQuadrado(double lado)
{
  return lado * lado;
}
double areaTriangulo(double base, double altura)
{
  return base * altura / 2;
}
double areaCirculo(double raio)
{
  return PI * raio * raio;
}
double volumeCubo(double lado)
{
  return lado * lado * lado;
}
double volumeTetraedro(double lado, double altura)
{
  return (double)1/3 * areaTriangulo(lado, lado * sqrt(altura) / 2) * altura;
}
double volumeEsfera(double raio)
{
  return (double)4/3 * PI * raio * raio;
}
```

Os protótipos dessas funções, além de uma definição de uma macro, serão mantidos em um arquivo-cabeçalho com nome `geometricas.h`:

```
double perimetroQuadrado(double lado);
double perimetroCirculo(double raio);
double perimetroTriangulo(double lado1, double lado2, double lado3);
double areaQuadrado(double lado);
double areaCirculo(double raio);
double areaTriangulo(double base, double altura);
double volumeCubo(double lado);
double volumeEsfera(double raio);
double volumeTetraedro(double lado, double altura);
```

Além desses protótipos, a macro `PI` também deve ser definida neste arquivo. Então, um arquivo-fonte `calc.c` que calcula medidas de figuras geométricas e que contém a função `main` pode ser construído. A figura 47.2 ilustra essa divisão.

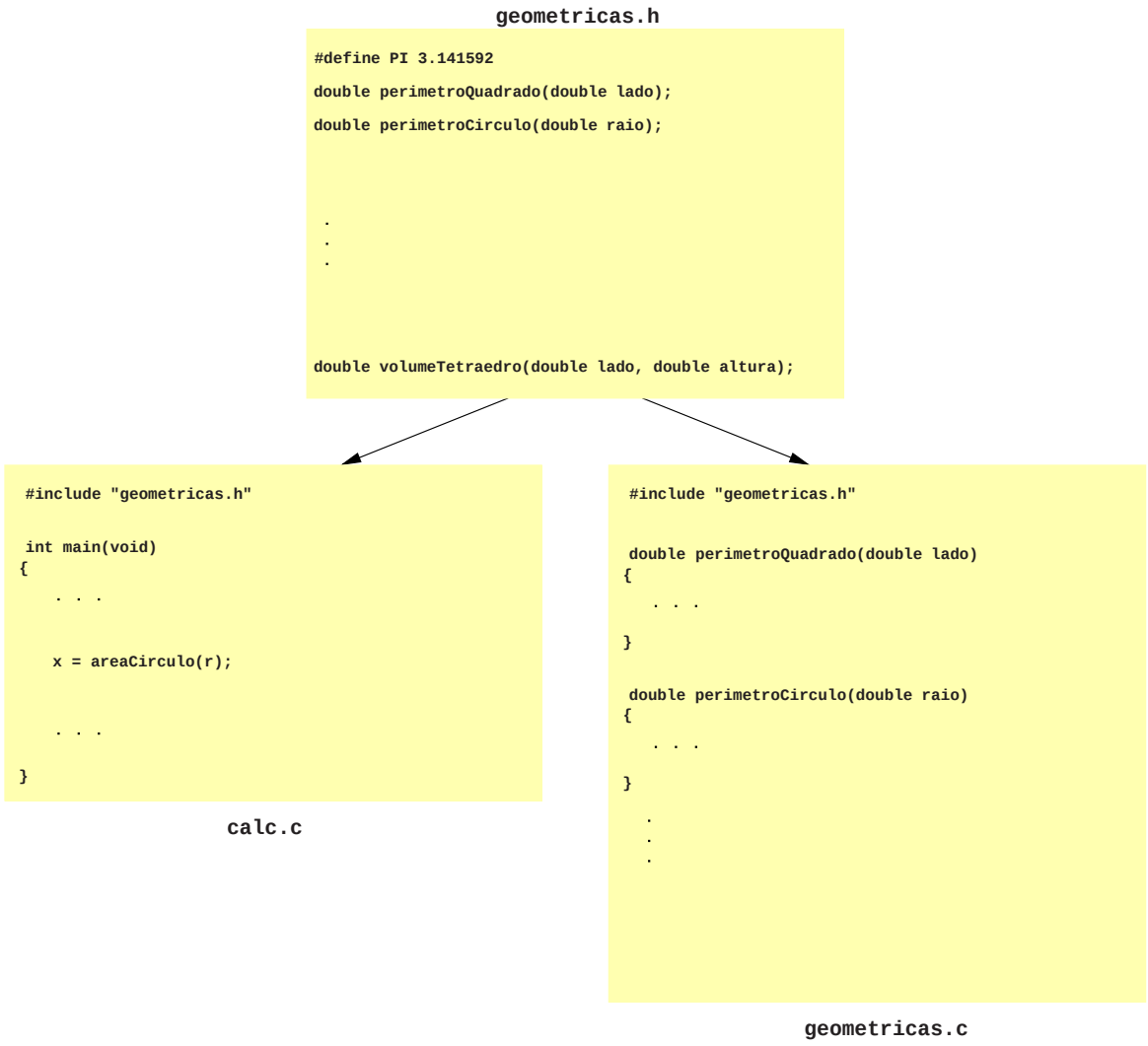



Figura 47.2: Relação entre protótipos, arquivo-fonte e arquivo-cabeçalho.

## 47.3 Divisão de programas em arquivos

Como vimos na seção anterior, podemos dividir nossos programas em diversos arquivos que possuem uma conexão lógica entre si. Usaremos agora o que conhecemos sobre arquivos-cabeçalhos e arquivos-fontes para descrever uma técnica simples de dividir um programa em arquivos. Consideramos aqui que o programa já foi projetado, isto é, já decidimos quais funções o programa necessita e como arranjá-las em grupos logicamente relacionados.

Cada conjunto de funções relacionadas será colocado em um arquivo-fonte em separado. `geometricas.c` da seção anterior é um exemplo de arquivo-fonte desse tipo. Além disso, criamos um arquivo-cabeçalho com o mesmo nome do arquivo-fonte, mas com extensão `.h`. O arquivo-cabeçalho `geometricas.h` da seção anterior é um exemplo. Nesse arquivo-cabeçalho, colocamos os protótipos das funções incluídas no arquivo-fonte, lembrando que as funções que são projetadas somente para dar suporte às funções do arquivo-fonte não devem ter seus protótipos descritos no arquivo-cabeçalho. Então, devemos incluir o arquivo-cabeçalho em cada arquivo-fonte que necessite chamar uma função definida no arquivo-fonte. Além disso, incluímos o arquivo-cabeçalho no próprio arquivo-fonte para que o compilador possa verificar que os protótipos das funções no arquivo-cabeçalho são consistentes com as definições do arquivo-fonte. Esse é o caso do exemplo da seção anterior, com arquivo-fonte `geometricas.c` e arquivo-cabeçalho `geometricas.h`. Veja novamente a figura 47.2.

A função principal `main` deve ser incluída em um arquivo-fonte que tem um nome representando o nome do programa. Por exemplo, se queremos que o programa seja conhecido como `calc` então a função `main` deve estar em um arquivo-fonte com nome `calc.c`. É possível que existam outras funções no mesmo arquivo-fonte onde `main` se encontra, funções essas que não são chamadas em outros arquivos-fontes do programa.

Para gerar um arquivo-executável de um programa dividido em múltiplos arquivos-fontes, os mesmos passos básicos que usamos para um programa em um único arquivo-fonte são necessários:

- **compilação:** cada arquivo-fonte do programa deve ser compilado separadamente; para cada arquivo-fonte, o compilador gera um arquivo contendo código objeto, que têm extensão `.o`;
- **ligação:** o ligador combina os arquivos-objetos criados na fase compilação, juntamente com código das funções da biblioteca, para produzir um arquivo-executável.

Muitos compiladores nos permitem construir um programa em um único passo. Com o compilador GCC, usamos o seguinte comando para construir o programa `calc` da seção anterior:

```
gcc -o calc calc.c geometricas.c
```

Os dois arquivos-fontes são primeiro compilados em arquivos-objetos. Esse arquivos-objetos são automaticamente passados para o ligador que os combina em um único arquivo. A opção `-o` especifica que queremos que nosso arquivo-executável tenha o nome `calc`.

Digitar os nomes de todos os arquivos-fontes na linha de comando de uma janela de um terminal logo torna-se uma tarefa tediosa. Além disso, podemos desperdiçar uma quantidade de tempo quando reconstruímos um programa se sempre recompilamos todos os arquivos-fontes, não apenas aqueles que são afetados pelas nossas modificações mais recentes.

Para facilitar a construção de grandes programas, o conceito de *makefiles* foi proposto nos primórdios da criação do sistema operacional UNIX. Um *makefile* é um arquivo que contém informação necessária para construir um programa. Um *makefile* não apenas lista os arquivos que fazem parte do programa, mas também descreve as dependências entre os arquivos. Por exemplo, da seção anterior, como `calc.c` inclui o arquivo `geometricas.h`, dizemos que `calc.c` 'depende' de `geometricas.h`, já que uma mudança em `geometricas.h` fará com que seja necessária a recompilação de `calc.c`.

Abaixo, listamos um *makefile* para o programa `calc`.

```
calc: calc.o geometricas.o
        gcc -o calc calc.o geometricas.o -lm
calc.o: calc.c geometricas.h
        gcc -c calc.c -lm
geometricas.o: geometricas.c geometricas.h
        gcc -c geometricas.c -lm
```

No arquivo acima, existem 3 grupos de linhas. Cada grupo é conhecido como um **regra**. A primeira linha em cada regra fornece um arquivo-**alvo**, seguido pelos arquivos dos quais ele depende. A segunda linha é um **comando** a ser executado se o alvo deve ser reconstruído devido a uma alteração em um de seus arquivos de dependência.

Na primeira regra, `calc` é o alvo:

```
calc: calc.o geometricas.o
        gcc -o calc calc.o geometricas.o -lm
```

A primeira linha dessa regra estabelece que `calc` depende dos arquivos `calc.o` e `geometricas.o`. Se qualquer um desses dois arquivos foi modificado desde da última construção do programa, então `calc` precisa ser reconstruído. O comando na próxima linha indica como a reconstrução deve ser feita, usando o GCC para ligar os dois arquivos-objetos.

Na segunda regra, `calc.o` é o alvo:

```
calc.o: calc.c geometricas.h
        gcc -c calc.c -lm
```

A primeira linha indica que `calc.o` necessita ser reconstruído se ocorrer uma alteração em `calc.c` ou `geometricas.h`. A próxima linha mostra como atualizar `calc.o` através da recompilação de `calc.c`. A opção `-c` informa o compilador para compilar `calc.c` em um arquivo-objeto, sem ligá-lo.

Tendo criado um *makefile* para um programa, podemos usar o utilitário `make` para construir ou reconstruir o programa. verificando a data e a hora associada com cada arquivo do programa, `make` determina quais arquivos estão desatualizados. Então, ele invoca os comandos necessários para reconstruir o programa.

Algumas dicas para criar *makefiles* seguem abaixo:

- cada comando em um *makefile* deve ser precedido por um caractere de tabulação horizontal `TAB`;
- um *makefile* é armazenado em um arquivo com nome `Makefile`; quando o utilitário `make` é usado, ele automaticamente verifica o conteúdo do diretório atual buscando por esse arquivo;
- use

  ```
  make alvo
  ```

  onde `alvo` é um dos alvos listados no *makefile*; se nenhum alvo é especificado, `make` construirá o alvo da primeira regra.

O utilitário `make` é complicado o suficiente para existirem dezenas de livros e manuais que nos ensinam a usá-lo. Com as informações desta aula, temos as informações básicas necessárias para usá-lo na construção de programas extensos divididos em diversos arquivos. Mais informações sobre o utilitário `make` devem ser buscadas no manual do GNU/Make.

## Exercícios

47.1 (a) Escreva uma função com a seguinte interface:

```
void preencheAleatorio(int v[], int n)
```

que receba um vetor $v$ de números inteiros e um número inteiro $n$ e gere $n$ números inteiros aleatórios armazenando-os em $v$. Use a função `rand` da biblioteca `stdlib`.

(b) Crie um arquivo-fonte com todas as funções de ordenação que vimos nas aulas 42, 43 e 44. Crie também um arquivo-cabeçalho correspondente.

(c) Escreva um programa que receba um inteiro $n$, com $1 \leqslant n \leqslant 10000$, gere um seqüência de $n$ números aleatórios e execute os métodos de ordenação que conhecemos sobre esse vetor, medindo seu tempo de execução. Use as funções `clock` e `difftime` da biblioteca `time`.

(d) Crie um *makefile* para compilar e ligar seu programa.

AULA 48

# OPERAÇÕES SOBRE BITS

A linguagem C possui muitas características de uma linguagem de programação de alto nível e isso significa que podemos escrever programas que são independentes das máquinas que irão executá-los. A maioria das aplicações possui naturalmente essa propriedade, isto é, podemos propor soluções computacionais – programas escritos na linguagem C – que não têm de se adaptar às máquinas em que serão executados. Por outro lado, alguns programas mais específicos, como compiladores, sistemas operacionais, cifradores, processadores gráficos e programas cujo tempo de execução e/ou uso do espaço de armazenamento são críticos, precisam executar operações no nível dos bits.

Outra característica importante da linguagem C, que a diferencia de outras linguagens de programação alto nível, é que ela permite o uso de operações sobre bits específicos e sobre trechos de bits, e ainda permite o uso de estruturas de armazenamento para auxiliar nas operações de baixo nível. Nesta aula, tratamos com algum detalhe de cada um desses aspectos da linguagem C.

## 48.1 Operadores bit a bit

A linguagem C possui seis **operadores bit a bit**, que operam sobre dados do tipo inteiro e do tipo caractere no nível de seus bits. A tabela abaixo mostra esses operadores e seus significados.

| Operador | Significado |
|---|---|
| **~** | complemento |
| **&** | E |
| **^** | OU exclusivo |
| **\|** | OU inclusivo |
| **<<** | deslocamento à esquerda |
| **>>** | deslocamento à direita |

O operador de complemento **~** é o único operador unário dessa tabela, os restantes todos são binários. Os operadores **~**, **&**, **^** e **|** executam operações booleanas sobre os bits de seus operandos.

O operador **~** gera o complemento de seu operando, isto é, troca os bits 0 por 1 e os bits 1 por 0. O operador **&** executa um E booleano bit a bit nos seus operandos. Os operadores **^** e **|** executam um OU booleano bit a bit nos seus operandos, sendo que o operador **^** produz 0 caso os dois bits correspondentes dos operandos sejam 1 e, nesse mesmo caso, o operador **|** produz

1. Os operadores de deslocamento **<<** e **>>** são operadores binários que deslocam os bits de seu operando à esquerda um número de vezes representado pelo operando à direita. Por exemplo, `i << j` produz como resultado o valor de `i` deslocado de `j` posições à esquerda. Para cada bit que é deslocado à esquerda "para fora" do número, um bit 0 é adicionado à direita desse número. O mesmo vale inversamente para o operador de deslocamento à direita **>>**.

O programa 48.1 mostra um exemplo simples do uso de todos os operadores sobre bits.

Programa 48.1: Uso de operadores sobre bits.

```
#include <stdio.h>
void imprime(unsigned short int n, char msg[])
{
  char binario[17] = {'0','0','0','0','0','0','0','0',
                      '0','0','0','0','0','0','0','0','\0'};
  unsigned short int m;
  int i;
  m = n;
  for (i = 15; n != 0; i--, n = n / 2)
    binario[i] = '0' + n % 2;
  printf("%9s %5d (%s)\n", msg, m, binario);
}
int main(void)
{
  unsigned short int i, j, k;
  i = 51;
  j = 15;
  imprime(i, "i =");
  imprime(j, "j =");
  k = ~i;
  imprime(k, "~i =");
  k = i & j;
  imprime(k, "i & j =");
  k = i ^ j;
  imprime(k, "i ^ j =");
  k = i | j;
  imprime(k, "i | j =");
  k = i << 4;
  imprime(k, "i << 4 =");
  k = j >> 2;
  imprime(k, "j >> 2 =");
  return 0;
}
```

A execução do programa 48.1 provoca a seguinte saída:

```
     i =    51 (0000000000110011)
     j =    15 (0000000000001111)
    ~i = 65484 (1111111111001100)
 i & j =     3 (0000000000000011)
 i ^ j =    60 (0000000000111100)
 i | j =    63 (0000000000111111)
i << 4 =   816 (0000001100110000)
j >> 2 =     3 (0000000000000011)
```

Os operadores sobre bits têm precedências distintas uns sobre os outros, como mostra a tabela abaixo:

| Operador | Precedência |
|---|---|
| ~ | 1 (maior) |
| & << >> | 2 |
| ^ | 3 |
| \| | 4 (menor) |

Observe ainda que a precedência dos operadores sobre bits é menor que precedência dos operadores relacionais. Isso significa que um sentença como a seguir

```
if (status & 0x4000 != 0)
```

tem o significado a seguir

```
if (status & (0x4000 != 0))
```

o que muito provavelmente não é a intenção do(a) programador(a) neste caso. Dessa forma, para que a intenção se concretize, devemos escrever:

```
if ((status & 0x4000) != 0)
```

Muitas vezes em programação de baixo nível, queremos acessar bits específicos de uma seqüência de bits. Por exemplo, quando trabalhamos com computação gráfica, queremos colocar dois ou mais pixeis em um único byte. Podemos extrair e/ou modificar dados que são armazenados em um número pequeno de bits com o uso dos operadores sobre bits.

Suponha que `i` é uma variável do tipo `unsigned short int`, ou seja, um compartimento de memória de 16 bits que armazena um número inteiro. As operações mais freqüentes sobre um único bit que podem ser realizadas sobre a variável `i` são as seguintes:

- **Ligar um bit:** suponha que queremos ligar o bit 4 de `i`, isto é o quinto bit menos significativo, o quinto da direita para a esquerda. A forma mais fácil de ligar o bit 4 é executar um OU inclusivo entre `i` e a constante `0x0010`:

```
i = i | 0x0010;
```

  Mais geralmente, se a posição do bit que queremos ligar está armazenada na variável `j`, podemos usar o operador de deslocamento à esquerda e executar a seguinte expressão, seguida da atribuição:

```
i = i | 1 << j;
```

- **Desligar um bit:** para desligar o bit 4 de `i`, devemos usar uma máscara com um bit 0 na posição 4 e o bit 1 em todas as outras posições:

```
i = 0x00ff;
i = i & ~0x0010;
```

  Usando a mesma idéia, podemos escrever uma sentença que desliga um bit cuja posição está armazenada em uma variável `j`:

```
i = i & ~(1 << j);
```

- **Testando um bit:** o trecho de código a seguir verifica se o bit 4 da variável `i` está ligado:

```
if (i & 0x0010)
```

  Para testar se o bit `j` está ligado, temos de usar a seguinte sentença:

```
if (i & 1 << j)
```

Para que o trabalho com bits torne-se um pouco mais fácil, freqüentemente damos nomes às posições dos bits de interesse. Por exemplo, se os bits das posições 1, 2 e 4 correspondem às cores azul, verde e vermelho, podemos definir macros que representam essas três posições dos bits:

```
#define AZUL     1
#define VERDE    2
#define VERMELHO 4
```

Ligar, desligar e testar o bit `AZUL` pode ser feito como abaixo:

```
i = i | AZUL;
i = i & ~AZUL;
if (i & AZUL) ...
```

Podemos ainda ligar, desligar e testar vários bits ao mesmo tempo:

```
i = i | AZUL | VERDE;
i = i & ~(AZUL | VERMELHO);
if (i & (VERDE | VERMELHO)) ...
```

Trabalhar com um grupo de vários bits consecutivos, chamados de **trecho de bits**[1] é um pouco mais complicado que trabalhar com um único bit. As duas operações mais comuns sobre trechos de bits são as seguintes:

- **Modificar um trecho de bits:** para alterar um trecho de bits é necessário inicialmente limpá-lo usando a operação E bit a bit e posteriormente armazenar os novos bits no trecho de bits com a operação de OU inclusivo. A operação a seguir mostra como podemos armazenar o valor binário 101 nos bits 4–6 da variável `i`:

  ```
  i = i & ~0x0070 | 0x0050;
  ```

  Suponha ainda que a variável `j` contém o valor a ser armazenado nos bits 4–6 de `i`. Então, podemos executar a seguinte sentença:

[1] Tradução livre de *bit-field*.

```
i = i & ~0x0070 | j << 4;
```

- **Recuperar um trecho de bits:** quando o trecho de bits atinge o bit 0 da variável, a recuperação de seu valor pode ser facilmente obtida da seguinte forma. Suponha que queremos recuperar os bits 0–2 da variável `i`. Então, podemos fazer:

```
j = i & 0x0007;
```

Se o trecho de bits não tem essa propriedade, então podemos deslocar o trecho de bits à direita para depois extraí-lo usando a operação E. Para extrair, por exemplo, os bits 4–6 de `i`, podemos usar a seguinte sentença:

```
j = (i >> 4) & 0x0007;
```

## 48.2 Trechos de bits em registros

Além das técnicas que vimos na seção anterior para trabalhar com trechos de bits, a linguagem C fornece como alternativa a declaração de registros cujos campos são trechos de bits. Essa característica pode ser bem útil especialmente em casos em que a complexidade de programação envolvendo trechos de bits gera arquivos-fontes complicados e confusos.

Suponha que queremos armazenar uma data na memória. Como os números envolvidos são relativamente pequenos, podemos pensar em armazená-la em 16 bits, com 5 bits para o dia, 4 bits para o mês e 7 bits para o ano, como mostra a figura 48.1.

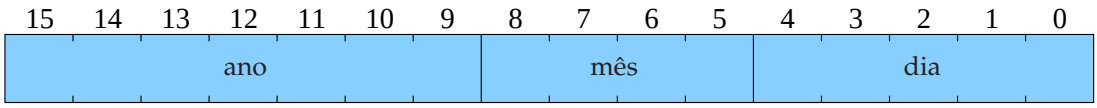



Figura 48.1: A forma de armazenamento de uma data.

Usando trechos de bits, podemos definir um registro na linguagem C com o mesmo formato:

```
struct data {
  unisgned int dia: 5;
  unisgned int mes: 4;
  unisgned int ano: 7;
};
```

O número após cada campo indica o comprimento em bits desse campo. O tipo de um trecho de bits pode ser um de dois possíveis: `int` (`signed int`) e `unsigned int`. O melhor é declarar todos os trechos de bits que são campos de um registro como `signed int` ou `unsigned int`.

Podemos usar os trechos de bits exatamente como usamos outros campos de um registro, como mostra o exemplo a seguir:

```
struct data data_arquivo;
data_arquivo.dia = 28;
data_arquivo.mes = 12;
data_arquivo.ano = 8;
```

Podemos considerar que um valor armazenado no campo `ano` seja relativo ao ano de 1980[2]. Ou seja a atribuição do valor `8` ao campo `ano`, como feita acima, tem significado 1988. Depois dessas atribuições, a variável `data_arquivo` tem a aparência mostrada na figura 48.2.

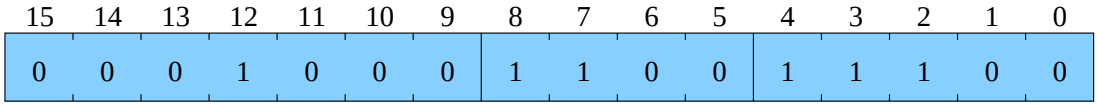



Figura 48.2: Variável `data_arquivo` com valores armazenados.

Poderíamos usar operadores sobre bits para obter o mesmo resultado, talvez até mais rapidamente. No entanto, escrever um programa legível é usualmente mais importante que ganhar alguns microssegundos.

Trechos de bits têm uma restrição que não se aplica a outros campos de um registro. Como os trechos de bits não têm um endereço no sentido usual, a linguagem C não nos permite aplicar o operador de endereço `&` para um trecho de bits. Por conta disso, funções como `scanf` não podem armazenar valores diretamente em um campo de um registro que é um trecho de bits. É claro que podemos usar `scanf` para ler uma entrada em uma variável e então atribui seu valor a um campo de um registro que é um trecho de bits.

## Exercícios

48.1 Uma das formas mais simples de criptografar dados é usar a operação de OU exclusivo sobre cada caractere com uma chave secreta. Suponha, por exemplo, que a chave secreta é o caractere **&**. Supondo que usamos a tabela ASCII, se fizermos um OU exclusivo do caractere **z** com a chave, obtemos o caractere \:

| | | |
|---|---|---|
| | 00100110 | código binário ASCII para **&** |
| XOR | 01111010 | código binário ASCII para **z** |
| | 01011100 | código binário ASCII para \ |

[2] 1980 é o ano em que o mundo começou, segundo a Microsoft. O registro `data` é a forma como o sistema operacional MS–DOS armazena a data que um arquivo foi criado ou modificado.

Para decifrar uma mensagem, aplicamos a mesma idéia. Cifrando uma mensagem previamente cifrada, obtemos a mensagem original. Se executamos um OU exclusivo entre o caractere **&** e o caractere \, por exemplo, obtemos o caractere original **z**:

| | | |
|---|---|---|
| | 00100110 | código binário ASCII para **&** |
| XOR | 01011100 | código binário ASCII para \ |
| | 01111010 | código binário ASCII para **z** |

Escreva um programa que receba uma cadeia de caracteres e cifre essa mensagem usando a técnica previamente descrita.

Programa 48.2: Solução do exercício 48.1.

```
#include <ctype.h>
#include <stdio.h>
#define CHAVE '&'
int main(void)
{
  int c_orig, c_novo;
  c_orig = getchar();
  while (c_orig != EOF) {
    c_novo = c_orig ^ CHAVE;
    if (isprint(c_orig) && isprint(c_novo))
      putchar(c_novo);
    else
      putchar(c_orig);
    c_orig = getchar();
  }
  return 0;
}
```

A mensagem original pode ser digitada pelo(a) usuário ou lida a partir de um arquivo com redirecionamento de entrada. A mensagem cifrada pode ser vista na saída padrão (monitor) ou pode ser armazenada em um arquivo usando redirecionamento de saída.

Suponha que o programa acima tenha sido salvo como `cript.c` e que um executável tenha sido gerado com nome `cript`. Então, para cifrar uma mensagem (texto) digitada em um arquivo `msg` e armazenar o resultado no arquivo `novo`, podemos fazer:

```
prompt$ ./cript < msg > novo
```

Para recuperar a mensagem original e visualizá-la no monitor, podemos fazer:

```
prompt$ ./cript < novo
```

AULA 49

# INTRODUÇÃO AOS APONTADORES

Apontadores ou ponteiros são certamente uma das características mais destacáveis da linguagem de programação C. Os apontadores agregam poder e flexibilidade à linguagem de maneira a diferenciá-la de outras linguagens de programação de alto nível, permitindo a representação de estruturas de dados complexas, modificação de valores passados como argumentos a funções, alocação dinâmica de espaços na memória, entre outros destaques. Nesta aula iniciaremos o contato com esses elementos.

## 49.1 Variáveis apontadoras

Para bem compreender os apontadores precisamos inicialmente compreender a idéia de indireção. Conceitualmente, um apontador permite acesso indireto a um valor armazenado em algum ponto da memória. Esse acesso é realizado justamente porque um **apontador** é uma variável que armazena um valor especial, que é um endereço de memória, e por isso nos permite acessar indiretamente o valor armazenado nesse endereço.

A memória de um computador pode ser vista como constituída de muitas posições (ou compartimentos ou células), dispostas continuamente, cada qual podendo armazenar um valor, na base binária. Ou seja, a memória nada mais é do que um grande vetor que pode armazenar valores na base binária e que, por sua vez, esse valores podem ser interpretados como valores de diversos tipos. Os índices desse vetor, numerados seqüencialmente a partir de 0 (zero), são chamados de **endereços de memória**. Quando escrevemos um programa em uma linguagem de programação de alto nível, o nome ou identificador de uma variável é associado a um índice desse vetor, isto é, a um endereço da memória. A tradução do nome da variável para um endereço de memória, e vice-versa, é feita de forma automática e transparente pelo compilador da linguagem de programação.

Em geral, a memória de um computador é dividida em bytes, com cada byte sendo capaz de armazenar 8 bits de informação. Cada byte tem um único endereço que o distingue de outros bytes da memória. Se existem $n$ bytes na memória, podemos pensar nos endereços como números de intervalo de 0 a $n - 1$, como mostra a figura 49.1.

Um programa executável é constituído por trechos de código e dados, ou seja, por instruções de máquina que correspondem às sentenças no programa original na linguagem C e de variáveis no programa original. Cada variável do programa ocupa um ou mais bytes na memória. O endereço do primeiro byte de uma variável é dito ser o endereço da variável. Na figura 49.2, a variável `i` ocupa os bytes dos endereços 2000 e 2001. Logo, o endereço da variável `i` é 2000.

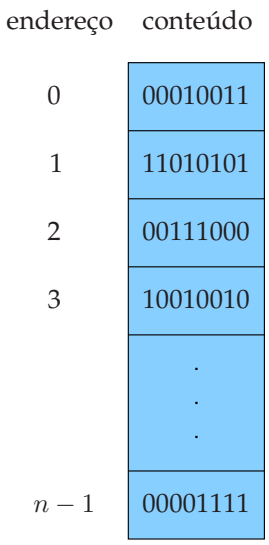



Figura 49.1: Uma ilustração da memória e de seus endereços.

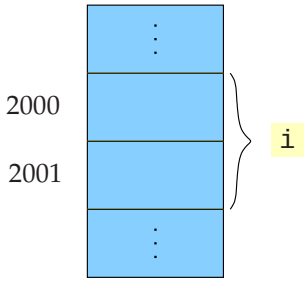



Figura 49.2: Endereço da variável `i`.

Os apontadores têm um papel importante em toda essa história. Apesar de os endereços serem representados por números, como ilustrado nas figuras 49.1 e 49.2, o intervalo de valores que esse objetos podem assumir é diferente do intervalo que os números inteiros, por exemplo, podem assumir. Isso significa, entre outras coisas, que não podemos armazenar endereços em variáveis do tipo inteiro. Endereços são armazenados em variáveis especiais, chamadas de variáveis apontadoras.

Quando armazenamos o endereço de uma variável `i` em uma variável apontadora `p`, dizemos que `p` **aponta para** `i`. Em outras palavras, um apontador nada mais é que um endereço e uma variável apontadora é uma variável que pode armazenar endereços.

Ao invés de mostrar endereços como números, usaremos uma notação simplificada de tal forma que, para indicar que uma variável apontadora `p` armazena o endereço de uma variável `i`, mostraremos o conteúdo de `p` – um endereço – como uma flecha orientada na direção de `i`, como mostra a figura 49.3.

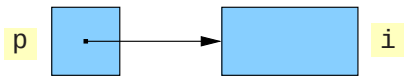



Figura 49.3: Representação de uma variável apontadora.

Uma variável apontadora pode ser declarada da mesma forma que uma variável qualquer, mas com um asterisco precedendo seu identificador. Por exemplo,

```
int *p;
```

Essa declaração indica que `p` é uma variável apontadora capaz de apontar para objetos do tipo `int`. A linguagem C obriga que toda variável apontadora aponte apenas para objetos de um tipo particular, chamado de **tipo referenciado**.

Diante do exposto até este ponto, daqui por diante não mais distinguiremos os termos apontador e variável apontadora, ficando então subentendido valor (conteúdo) e variável.

## 49.2 Operadores de endereçamento e de indireção

A linguagem C possui dois operadores para uso específico com apontadores. Para obter o endereço de uma variável, usamos o **operador de endereçamento (ou de endereço)** `&`. Se `v` é uma variável, então `&v` é seu endereço na memória. Para ter acesso ao objeto que um apontador aponta, temos de usar o **operador de indireção** `*`. Se `p` é um apontador, então `*p` representa o objeto para o qual `p` aponta no momento.

Declarar uma variável apontadora reserva um espaço na memória para um apontador mas não a faz apontar para um objeto. É crucial inicializar `p` antes de usá-lo. Uma forma de inicializar um apontador é atribuir-lhe o endereço de alguma variável usando o operador `&`:

```
int i, *p;
p = &i;
```

Atribuir o endereço da variável `i` para a variável `p` faz com que `p` aponte para `i`, como ilustra-a figura 49.4.

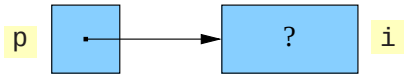



Figura 49.4: Variável apontadora `p` contendo o endereço da variável `i`.

É possível inicializar uma variável apontadora no momento de sua declaração, como abaixo:

```
int i, *p = &i;
```

Uma vez que uma variável apontadora aponta para um objeto, podemos usar o operador de indireção `*` para acessar o valor armazenado no objeto. Se `p` aponta para `i`, por exemplo, podemos imprimir o valor de `i` como segue:

```
printf("%d\n", *p);
```

Observe que a função `printf` mostrará o valor de `i` e não o seu endereço. Observe também que aplicar o operador `&` a uma variável produz um apontador para a variável e aplicar o operador `*` para um apontador retoma o valor original da variável:

```
j = *&i;
```

Na verdade, a atribuição acima é idêntica à seguinte:

```
j = i;
```

Enquanto dizemos que `p` aponta para `i`, dizemos também que `*p` é um apelido para `i`. Não apenas `*p` tem o mesmo valor que `i`, mas alterar o valor de `*p` altera o valor de `i`.

Uma observação importante que auxilia a escrever e ler programas com apontadores é sempre "traduzir" os operadores unários de endereço & e indireção * para *endereço da variável* e *conteúdo da variável apontada por*, respectivamente. Sempre que usamos um desses operadores no sentido de estabelecer indireção e apontar para valores, é importante traduzi-los desta forma.

Programa 49.1: Um exemplo do uso de apontadores.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    char c, *p;
5    p = &c;
6    c = 'a';
7    printf("c=%c, *p=%c\n", c, *p);
8    c = '/';
9    printf("c=%c, *p=%c\n", c, *p);
10   *p = 'Z';
11   printf("c=%c, *p=%c\n", c, *p);
12   return 0;
13 }
```

Observe que no programa 49.1, após a declaração das variáveis `c` e `p`, temos a inicialização do apontador `p`, que recebe o endereço da variável `c`, sendo essas duas variáveis do mesmo tipo `char`. É importante sempre destacar que o valor, ou conteúdo, de um apontador na linguagem C não tem significado até que contenha, ou aponte, para algum endereço válido.

A primeira chamada da função `printf` no programa 49.1 apenas mostra o conteúdo da variável `c`, que foi inicializada com o caractere `'a'`, e também o conteúdo da variável apontada por `p`. Como a variável `p` aponta para a variável `c`, o valor apresentado na saída é também aquele armazenado na variável `c`, isto é, o caractere `'a'`. A segunda chamada da função `printf` é precedida pela alteração do conteúdo da variável `c` e, como a variável `p` mantém-se apontando para a variável `c`, a chamada da função `printf` faz com que o caractere `'/'` seja apresentado na saída. É importante notar que, a menos que o conteúdo da variável `p` seja modificado, a expressão `*p` sempre acessa o conteúdo da variável `c`. Por fim, a última chamada à função `printf` é precedida de uma atribuição que modifica o conteúdo da variável apontada por `p`, isto é, a atribuição a seguir:

```
*p = 'Z';
```

faz também com que a variável `c` receba o caractere `'Z'`.

A linguagem C permite ainda que o operador de atribuição copie apontadores, supondo que possuam o mesmo tipo. Suponha que a seguinte declaração tenha sido feita:

```
int i, j, *p, *q;
```

A sentença

```
p = &i;
```

é um exemplo de atribuição de um apontador, onde o endereço de `i` é copiado em `p`. Um outro exemplo de atribuição de apontador é dado a seguir:

```
q = p;
```

Essa sentença copia o conteúdo de `p`, o endereço de `i`, para `q`, fazendo com que `q` aponte para o mesmo lugar que `p` aponta, como podemos visualizar na figura 49.5.

Ambos os apontadores `p` e `q` apontam para `i` e, assim, podemos modificar o conteúdo de `i` indiretamente através da atribuição de valores para `*p` e `*q`.

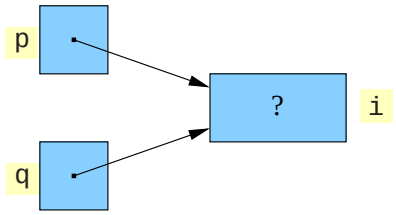



Figura 49.5: Dois apontadores para um mesmo endereço.

## 49.3 Apontadores em expressões

Apontadores podem ser usados em expressões aritméticas de mesmo tipo que seus tipos referenciados. Por exemplo, um apontador para uma variável do tipo inteiro pode ser usado em uma expressão aritmética envolvendo números do tipo inteiro, supondo o uso correto do operador de indireção `*`.

É importante observar também que os operadores `&` e `*`, por serem operadores unários, têm precedência sobre os operadores binários das expressões aritméticas em que se envolvem. Por exemplo, em uma expressão aritmética envolvendo números do tipo inteiro, os operadores binários `+`, `-`, `*` e `/` têm menor prioridade que o operador unário de indireção `*`.

Vejamos a seguir, como um exemplo simples, o programa 49.2 que usa um apontador como operando em uma expressão aritmética.

Programa 49.2: Outro exemplo do uso de apontadores.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int i, j, *apt1, *apt2;
5    apt1 = &i;
6    i = 5;
7    j = 2 * *apt1 + 3;
8    apt2 = apt1;
9    printf("i=%d, j=%d, *apt1=%d, *apt2=%d\n", i, j, *apt1, *apt2);
10   return 0;
11 }
```

## Exercícios

49.1 Se `i` é uma variável e `p` é uma variável apontadora que aponta para `i`, quais das seguintes expressões são apelidos para `i`?

(a) `*p`

(b) `&p`

(c) `*&p`
(d) `&*p`
(e) `*i`
(f) `&i`
(g) `*&i`
(h) `&*i`

49.2 Se `i` é uma variável do tipo `int` e `p` e `q` são apontadores para `int`, quais das seguintes atribuições são corretas?

(a) `p = i;`
(b) `*p = &i;`
(c) `&p = q;`
(d) `p = &q;`
(e) `p = *&q;`
(f) `p = q;`
(g) `p = *q;`
(h) `*p = q;`
(i) `*p = *q;`

49.3 Entenda o que o programa 49.3 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 49.3: Programa do exercício 49.3.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int a, b, *apt1, *apt2;
5    apt1 = &a;
6    apt2 = &b;
7    a = 1;
8    (*apt1)++;
9    b = a + *apt1;
10   *apt2 = *apt1 * *apt2;
11   printf("a=%d, b=%d, *apt1=%d, *apt2=%d\n", a, b, *apt1, *apt2);
12   return 0;
13 }
```

49.4 Entenda o que o programa 49.4 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

49.5 Entenda o que o programa 49.5 faz, simulando sua execução passo a passo. Depois disso, implemente-o.

Programa 49.4: Programa do exercício 49.4.

```
#include <stdio.h>
int main(void)
{
  int a, b, c, *apt;
  a = 3;
  b = 7;
  printf("a=%d, b=%d\n", a, b);
  apt = &a;
  c = *apt;
  apt = &b;
  a = *apt;
  apt = &c;
  b = *apt;
  printf("a=%d, b=%d\n", a, b);
  return 0;
}
```

Programa 49.5: Programa do exercício 49.5.

```
#include <stdio.h>
int main(void)
{
  int i, j, *p, *q;
  p = &i;
  q = p;
  *p = 1;
  printf("i=%d, *p=%d, *q=%d\n", i, *p, *q);
  q = &j;
  i = 6;
  *q = *p;
  printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
  return 0;
}
```

AULA 50

# APONTADORES E FUNÇÕES

Nesta aula revemos apontadores e funções na linguagem C. Até o momento, aprendemos algumas "regras" de como construir funções que têm parâmetros de entrada e saída ou argumentos passados por referência. Esses argumentos/parâmetros, como veremos daqui por diante, são na verdade apontadores. Um endereço de uma variável é passado como argumento para uma função. O parâmetro correspondente que recebe o endereço é então um apontador. Qualquer alteração realizada no conteúdo do parâmetro tem reflexos externos à função, no argumento correspondente.

## 50.1 Parâmetros de entrada e saída?

A abstração que fizemos antes para compreendermos o que são parâmetros de entrada e saída na linguagem C será revista agora e revelará algumas novidades. Vamos tomar um exemplo, que de alguma forma já vimos antes, no programa 50.1.

Programa 50.1: Exemplo de parâmetros de entrada e saída e apontadores.

```
1  #include <stdio.h>
2  void troca(int *a, int *b)
3  {
4    int aux;
5    aux = *a;
6    *a = *b;
7    *b = aux;
8  }
9  int main(void)
10 {
11   int x, y;
12   printf("Informe dois valores: ");
13   scanf("%d%d", &x, &y);
14   printf("Antes da troca : x = %d e y = %d\n", x, y);
15   troca(&x, &y);
16   printf("Depois da troca: x = %d e y = %d\n", x, y);
17   return 0;
18 }
```

Agora que entendemos os conceitos básicos que envolvem os apontadores, podemos olhar o programa 50.1 e compreender o que está acontecendo, especialmente no que se refere ao uso de apontadores como argumentos de funções. Suponha que alguém está executando esse programa. A execução inicia na linha 11 da função `main` e seu efeito é ilustrado na figura 50.1.


Figura 50.1: Execução da linha 11.

Em seguida, uma mensagem é emitida na saída após a execução da linha 12. Depois disso, na linha 13, suponha que o(a) usuário(a) do programa informe dois valores quaisquer do tipo inteiro como, por exemplo, 3 e 8. Isso se reflete na memória como na figura 50.2.


Figura 50.2: Execução da linha 13.

Na linha 14 da função `main`, a execução da função `printf` permite que o(a) usuário(a) verifique na saída os conteúdos das variáveis que acabou de informar. Na linha 15 do programa, a função `troca` é chamada com os endereços das variáveis `x` e `y` como argumentos. Isto é, esses endereços são copiados nos argumentos correspondentes que compõem a interface da função. O fluxo de execução do programa é então desviado para o trecho de código da função `troca`. Os parâmetros da função `troca` são dois apontadores para valores do tipo inteiro com identificadores `a` e `b`. Esses dois parâmetros recebem os endereços das variáveis `x` e `y` da função `main`, respectivamente, como se pode observar na figura 50.3.

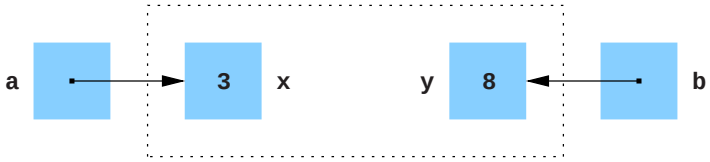



Figura 50.3: Execução da linha 15.

A seguir, na linha 4 do programa, a primeira linha do corpo da função `troca`, ocorre a declaração da variável `aux` e um espaço identificado por `aux` é reservado na memória, como vemos na figura 50.4.

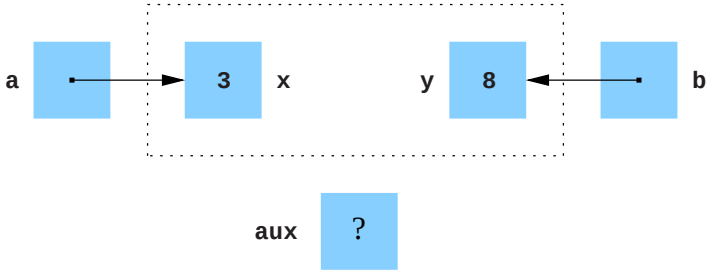



Figura 50.4: Execução da linha 4.

Depois da declaração da variável `aux`, a execução da linha 5 faz com que o *conteúdo da variável apontada por* `a` seja armazenado na variável `aux`. Veja a figura 50.5.

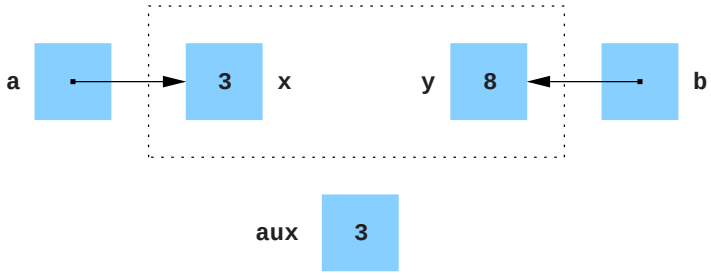



Figura 50.5: Execução da linha 5.

Na execução da linha 6, o *conteúdo da variável apontada por* `a` recebe o *conteúdo da variável apontada por* `b`, como mostra a figura 50.6.

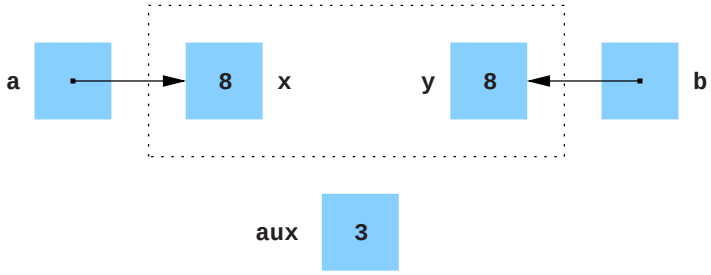



Figura 50.6: Execução da linha 6.

Por fim, na execução da linha 7, o *conteúdo da variável apontada por* `b` recebe o conteúdo da variável `aux`, conforme a figura 50.7.

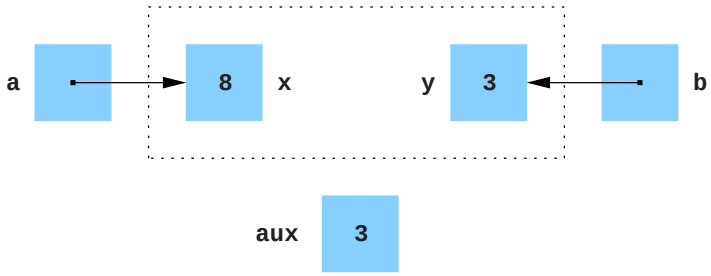



Figura 50.7: Execução da linha 7.

Após o término da função `troca`, seus parâmetros e variáveis locais são destruídos e o fluxo de execução volta para a função `main`, no ponto logo após onde foi feita a chamada da função `troca`, isto é, na linha 16. A memória neste momento encontra-se no estado ilustrado na figura 50.8.

Na linha 17 da função `main` é a chamada da função `printf`, que mostra os conteúdo das variáveis `x` e `y`, que foram trocados, conforme já constatado e ilustrado na figura 50.8. O programa então salta para a próxima linha e chega ao fim de sua execução.

Esse exemplo destaca que são realizadas cópias do valores dos argumentos – que nesse


Figura 50.8: Estado da memória após o término da função `troca`.

caso são endereços das variáveis da função `main` – para os parâmetros respectivos da função `troca`. No corpo dessa função, sempre que usamos o operador de indireção para acessar algum valor, estamos na verdade acessando o conteúdo da variável correspondente dentro da função `main`, que chamou a função `troca`. Isso ocorre com as variáveis `x` e `y` da função `main`, quando copiamos seus endereços nos parâmetros `a` e `b` da função `troca`.

Cópia? Como assim cópia? Nós aprendemos que parâmetros passados desta mesma forma são parâmetros de entrada e saída, ou seja, são parâmetros passados por referência e não por cópia. Esse exemplo mostra uma característica muito importante da linguagem C, que ficou dissimulada nas aulas anteriores: *só há passagem de argumentos por cópia na linguagem C*, ou ainda, *não há passagem de argumentos por referência na linguagem C*. O que fazemos de fato é simular a passagem de um argumento por referência usando apontadores. Assim, passando (por cópia) o endereço de uma variável como argumento para uma função, o parâmetro correspondente deve ser um apontador e, mais que isso, um apontador para a variável correspondente cujo endereço foi passado como argumento. Dessa forma, qualquer modificação indireta realizada no corpo dessa função usando esse apontador será realizada na verdade no conteúdo da variável apontada pelo parâmetro, que é simplesmente o conteúdo da variável passada como argumento na chamada da função.

Não há nada de errado com o que aprendemos nas aulas anteriores sobre argumentos de entrada e saída, isto é, passagem de argumentos por referência. No entanto, vale ressaltar que passagem de argumentos por referência é um tópico conceitual quando falamos da linguagem de programação C. O correto é repetir que *só há passagem de argumentos por cópia na linguagem C*.

## 50.2 Devolução de apontadores

Além de passar apontadores para funções também podemos fazê-las devolver apontadores. Funções como essas são comuns, por exemplo, quando tratamos de cadeias de caracteres.

A função abaixo recebe dois apontadores para inteiros e devolve um apontador para o maior dos números inteiros, isto é, devolve o endereço onde se encontra o maior dos números.

```
1  int *max(int *a, int *b)
2  {
3    if (*a > *b)
4      return a;
5    else
6      return b;
7  }
```

Quando chamamos a função `max`, passamos apontadores para duas variáveis do tipo `int` e armazenamos o resultado em uma variável apontadora:

```
int i, j, *p;
...
p = max(&i, &j);
```

Na execução da função `max`, temos que `*a` é um apelido para `i` e `*b` é um apelido para `j`. Se `i` tem valor maior que `j`, então `max` devolve o endereço de `i`. Caso contrário, `max` devolve o endereço de `j`. Depois da chamada, `p` aponta para `i` ou para `j`.

Não é possível que uma função devolva o endereço de uma variável local sua, já que ao final de sua execução, essa variável será destruída.

## Exercícios

50.1 (a) Escreva uma função com a seguinte interface:

```
void minmax(int v[], int n, int *max, int *min)
```

que receba um vetor $v$ com $n > 0$ números inteiros e devolva um maior e um menor dos elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um menor elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

50.2 (a) Escreva uma função com a seguinte interface:

```
void doisMaiores(int v[], int n, int *p_maior, int *s_maior)
```

que receba um vetor $v$ com $n > 0$ números inteiros e devolva um maior e um segundo maior elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um segundo maior elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

50.3 (a) Escreva uma função com a seguinte interface:

```
void somaprod(int a, int b, int *soma, int *prod)
```

que receba dois números inteiros $a$ e $b$ e devolva a soma e o produto destes dois números.

(b) Escreva um programa que receba $n$ números inteiros, com $n > 0$ par, calcule a soma e o produto deste conjunto usando a função do item (a) e determine quantos deles são maiores que esta soma e quantos são maiores que o produto. Observe que os números na entrada podem ser negativos.

Simule no papel a execução de seu programa antes de implementá-lo.

50.4 (a) Escreva uma função com a seguinte interface:

```
int *maximo(int v[], int n)
```

que receba um vetor $v$ de $n$ números inteiros e devolva o endereço do elemento de $v$ onde reside um maior elemento de $v$.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.

AULA 51

# APONTADORES E VETORES

Nas aulas 49 e 50 aprendemos o que são os apontadores e também como são usados como parâmetros de funções e devolvidos de funções. Nesta aula veremos outra aplicação para os apontadores. A linguagem C nos permite usar expressões aritméticas de adição e subtração com apontadores que apontam para elementos de vetores. Essa é uma forma alternativa de trabalhar com vetores e seus índices. Para nos tornarmos melhores programadores da linguagem C é necessário conhecer bem essa relação íntima entre apontadores e vetores. Além disso, o uso de apontadores para trabalhar com vetores é vantajoso em termos de eficiência do programa executável resultante.

## 51.1 Aritmética com apontadores

Nas aulas 49 e 50 vimos que apontadores podem apontar para elementos de um vetor. Suponha, por exemplo, que temos declaradas as seguintes variáveis:

```
int v[10], *p;
```

Podemos fazer o apontador `p` apontar para o elemento `v[0]` do vetor fazendo a seguinte atribuição, como mostra a figura 51.1:

```
p = &v[0];
```

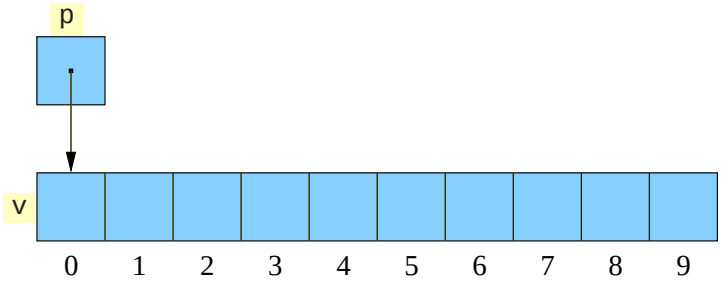



Figura 51.1: `p = &v[0];`

Podemos acessar `v[0]` através de `p`, como ilustrado na figura 51.2.

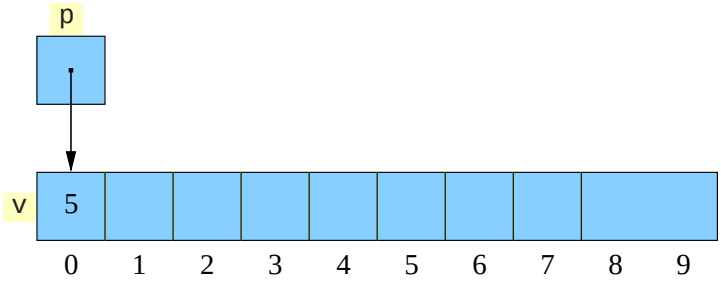



Figura 51.2: `*p = 5;`

Podemos ainda executar **aritmética com apontadores** ou **aritmética com endereços** sobre `p` e assim acessamos outros elementos do vetor `v`. A linguagem C possibilita três formas de aritmética com apontadores: (*i*) adicionar um número inteiro a um apontador; (*ii*) subtrair um número inteiro de um apontador; e (*iii*) subtrair um apontador de outro apontador.

Vamos olhar para cada uma dessas operações. Suponha que temos declaradas as seguintes variáveis:

```
int v[10], *p, *q, i;
```

Adicionar um inteiro `j` a um apontador `p` fornece um apontador para o elemento posicionado `j` posições após `p`. Mais precisamente, se `p` aponta para o elemento `v[i]`, então `p+j` aponta para `v[i+j]`. A figura 51.3 ilustra essa idéia.

Do mesmo modo, se `p` aponta para o elemento `v[i]`, então `p-j` aponta para `v[i-j]`, como ilustrado na figura 51.4.

Ainda, quando um apontador é subtraído de outro, o resultado é a distância, medida em elementos do vetor, entre os apontadores. Dessa forma, se `p` aponta para `v[i]` e `q` aponta para `v[j]`, então `p` − `q` é igual a `i` − `j`. A figura 51.5 ilustra essa situação.

Podemos comparar variáveis apontadoras entre si usando os operadores relacionais usuais (`<`, `<=`, `>`, `>=`, `==` e `!=`). Usar os operadores relacionais para comparar dois apontadores que apontam para um mesmo vetor é uma ótima idéia. O resultado da comparação depende das posições relativas dos dois elementos do vetor. Por exemplo, depois das atribuições dadas a seguir:

```
p = &v[5];
q = &v[1];
```

o resultado da comparação `p <= q` é falso e o resultado de `p >= q` é verdadeiro.

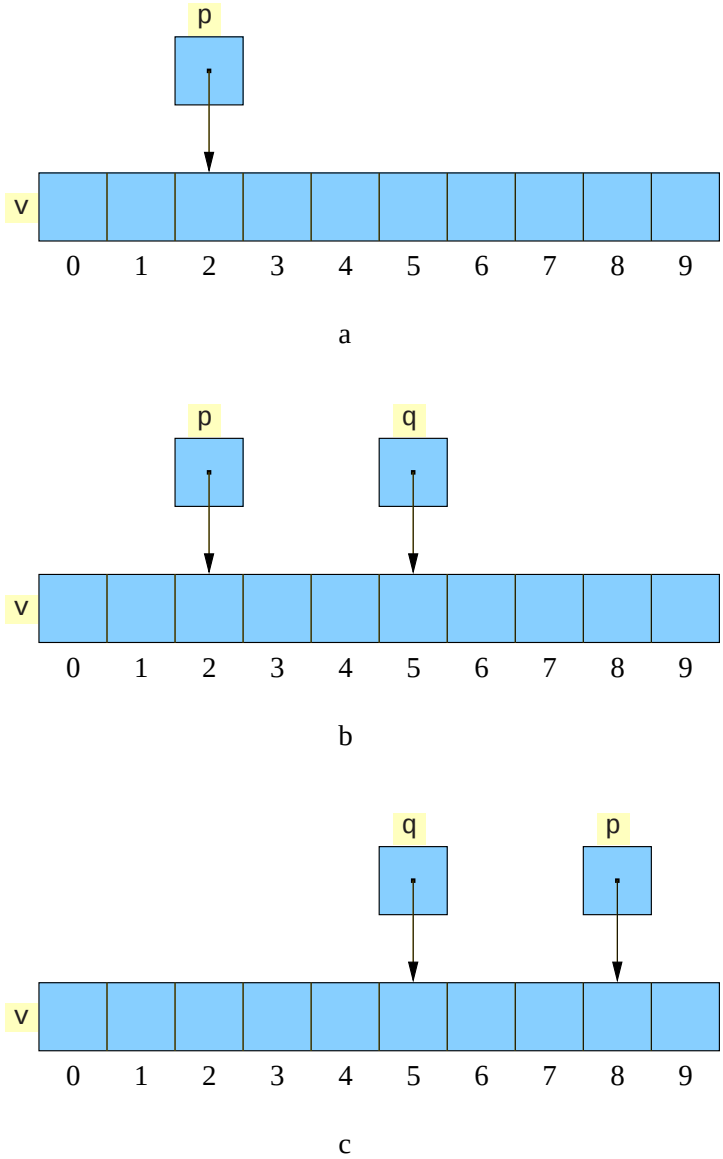



Figura 51.3: (a) `p = &v[2];` (b) `q = p + 3;` (c) `p = p + 6;`

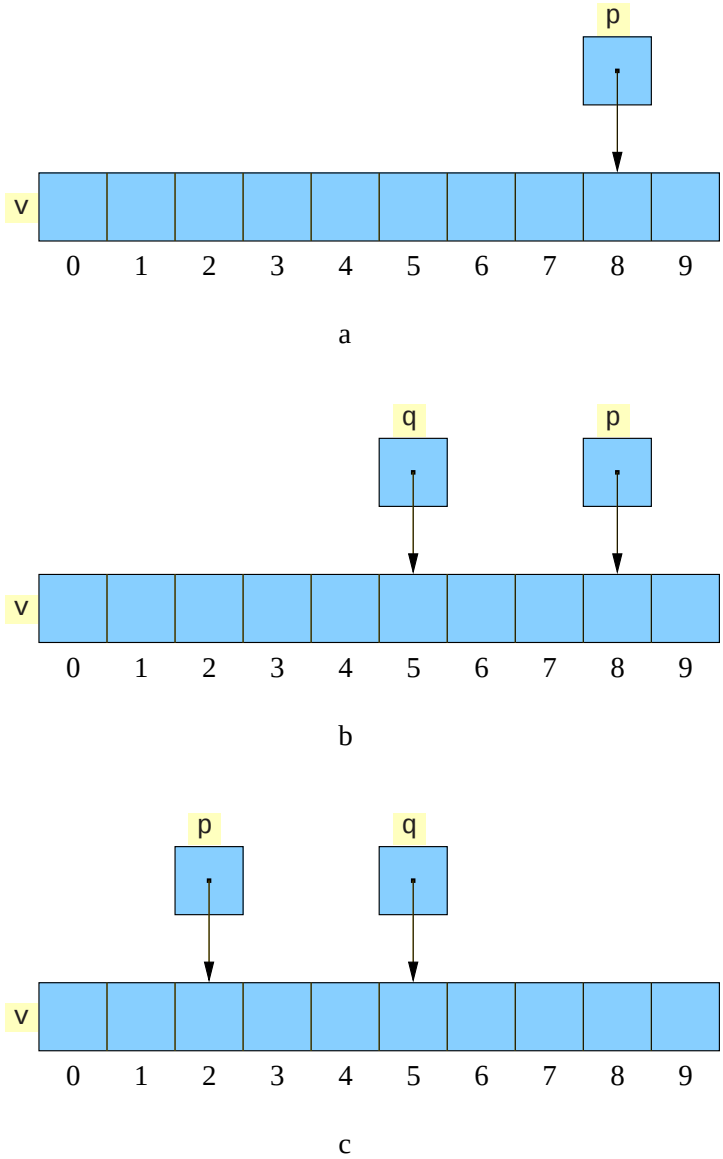



Figura 51.4: (a) `p = &v[8];` (b) `q = p - 3;` (c) `p = p - 6;`

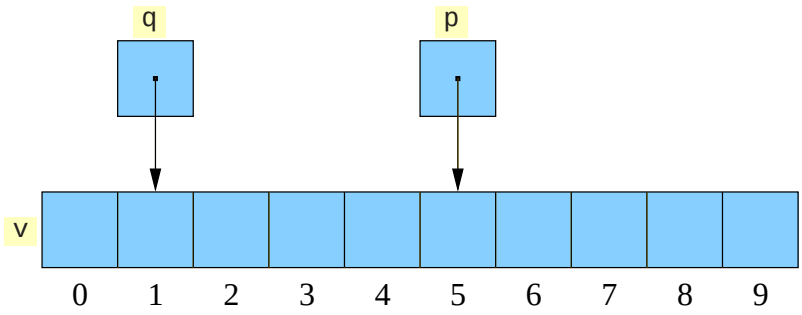



Figura 51.5: `p = &v[5];` e `q = &v[1];` A expressão `p - q` tem valor 4 e a expressão `q - p` tem valor $-4$.

## 51.2 Uso de apontadores para processamento de vetores

Usando aritmética de apontadores podemos visitar os elementos de um vetor através da atribuição de um apontador para seu início e do seu incremento em cada passo, como mostrado no trecho de código abaixo:

```
#define DIM 100
...
int v[DIM], soma, *p;
...
soma = 0;
for (p = &v[0]; p < &v[DIM]; p++)
  soma = soma + *p;
```

A condição `p <&v[DIM]` na estrutura de repetição `for` necessita de atenção especial. Apesar de estranho, é possível aplicar o operador de endereço para `v[DIM]`, mesmo sabendo que este elemento não existe no vetor `v`. Usar `v[DIM]` dessa maneira é perfeitamente seguro, já que a sentença `for` não tenta examinar o seu valor. O corpo da estrutura de repetição `for` será executado com `p` igual a `&v[0]`, `&v[1]`, ..., `&v[DIM-1]`, mas quando `p` é igual a `&v[DIM]` a estrutura de repetição termina.

Como já vimos, podemos também combinar o operador de indireção `*` com operadores de incremento `++` ou decremento `--` em sentenças que processam elementos de um vetor. Considere inicialmente o caso em que queremos armazenar um valor em um vetor e então avançar para o próximo elemento. Usando um índice, podemos fazer diretamente:

```
v[i++] = j;
```

Se `p` está apontando para um elemento de um vetor, a sentença correspondente usando esse apontador seria:

```
*p++ = j;
```

Devido à precedência do operador `++` sobre o operador `*`, o compilador enxerga essa sentença como

```
*(p++) = j;
```

O valor da expressão `*p++` é o valor de `*p`, antes do incremento. Depois que esse valor é devolvido, a sentença incrementa `p`.

A expressão `*p++` não é a única combinação possível dos operadores `*` e `++`. Podemos escrever `(*p)++` para incrementar o valor de `*p`. Nesse caso, o valor devolvido pela expressão é também `*p`, antes do incremento. Em seguida, a sentença incrementa `*p`. Ainda, podemos escrever `*++p` ou ainda `++*p`. No primeiro caso, incrementa `p` e o valor da expressão é `*p`, depois do incremento. No segundo, incrementa `*p` e o valor da expressão é `*p`, depois do incremento.

O trecho de código acima, que realiza a soma dos elementos do vetor `v` usando aritmética com apontadores, pode então ser reescrito como a seguir, usando uma combinação dos operadores `*` e `++`.

```
soma = 0;
p = &v[0];
while (p < &v[DIM])
  soma = soma + *p++;
```

## 51.3 Uso do identificador de um vetor como apontador

Apontadores e vetores estão intimamente relacionados. Como vimos nas seções anteriores, usamos aritmética de apontadores para trabalhar com vetores. Mas essa não é a única relação entre eles. Outra relação importante entre apontadores e vetores fornecida pela linguagem C é que o identificador de um vetor pode ser usado como um apontador para o primeiro elemento do vetor. Essa relação simplifica a aritmética com apontadores e estabelece ganho de versatilidade em ambos, apontadores e vetores.

Por exemplo, suponha que temos o vetor `v` declarado como abaixo:

```
int v[10];
```

Usando `v` como um apontador para o primeiro elemento do vetor, podemos modificar o conteúdo de `v[0]` da seguinte forma:

```
*v = 7;
```

Podemos também modificar o conteúdo de `v[1]` através do apontador `v+1`:

```
*(v+1) = 12;
```

Em geral, `v+i` é o mesmo que `&v[i]` e `*(v+i)` é equivalente a `v[i]`. Em outras palavras, índices de vetores podem ser vistos como uma forma de aritmética de apontadores.

O fato de que o identificador de um vetor pode servir como um apontador facilita nossa programação de estruturas de repetição que percorrem vetores. Considere a estrutura de repetição do exemplo dado na seção anterior:

```
soma = 0;
for (p = &v[0]; p < &v[DIM]; p++)
  soma = soma + *p;
```

Para simplificar essa estrutura de repetição, podemos substituir `&v[0]` por `v` e `&v[DIM]` por `v+DIM`, como mostra o trecho de código abaixo:

```
soma = 0;
for (p = v; p < v+DIM; p++)
  soma = soma + *p;
```

Apesar de podermos usar o identificador de um vetor como um apontador, não é possível atribuir-lhe um novo valor. A tentativa de fazê-lo apontar para qualquer outro lugar é um erro, como mostra o trecho de código abaixo:

```
while (*v != 0)
  v++;
```

O programa 51.1 mostra um exemplo do uso desses conceitos, realizando a impressão dos elementos de um vetor na ordem inversa da qual forma lidos.

Programa 51.1: Imprime os elementos na ordem inversa da de leitura.

```
1  #include <stdio.h>
2  #define N 10
3  int main(void)
4  {
5    int v[N], *p;
6    printf("Informe %d números: ", N);
7    for (p = v; p < v+N; p++)
8      scanf("%d", p);
9    printf("Em ordem reversa: ");
10   for (p = v+N-1; p >= v; p--)
11     printf(" %d", *p);
12   printf("\n");
13   return 0;
14 }
```

Outro uso do identificador de um vetor como um apontador é quando um vetor é um argumento em uma chamada de função. Nesse caso, o vetor é sempre tratado como um apontador. Considere a seguinte função que recebe um vetor de $n$ números inteiros e devolve um maior elemento nesse vetor.

```
int max(int v[], int n)
{
  int i, maior;
  maior = v[0];
  for (i = 1; i < n; i++)
    if (v[i] > maior)
      maior = v[i];
  return maior;
}
```

Suponha que chamamos a função `max` da seguinte forma:

```
M = max(U, N);
```

Essa chamada faz com que o endereço do primeiro compartimento do vetor `U` seja atribuído à `v`. O vetor `U` não é de fato copiado.

Para indicar que não queremos que um parâmetro que é um vetor não seja modificado, podemos incluir a palavra reservada `const` precedendo a sua declaração.

Quando uma variável simples é passada para uma função, isto é, quando é um argumento de uma função, seu valor é copiado no parâmetro correspondente. Então, qualquer alteração no parâmetro correspondente não afeta a variável. Em contraste, um vetor usado como um argumento não está protegido contra alterações, já que não ocorre uma cópia do vetor todo. Desse modo, o tempo necessário para passar um vetor a uma função independe de seu tamanho. Não há perda por passar vetores grandes, já que nenhuma cópia do vetor é realizada. Além disso, um *parâmetro* que é um vetor pode ser declarado como um apontador. Por exemplo, a função `max` descrita acima pode ser declarada como a seguir:

```
int max(int *v, int n)
{
  ...
}
```

Neste caso, declarar `v` como sendo um apontador é equivalente a declarar `v` como sendo um vetor. O compilador trata ambas as declarações como idênticas.

Apesar de a declaração de um *parâmetro* como um vetor ser equivalente à declaração do mesmo *parâmetro* como um apontador, o mesmo não vale para uma variável. A declaração a seguir:

```
int v[10];
```

faz com que o compilador reserve espaço para 10 números inteiros. Por outro lado, a declaração abaixo:

```
int *v;
```

faz o compilador reservar espaço para uma variável apontadora. Nesse último caso, `v` não é um vetor e tentar usá-lo como tal pode causar resultados desastrosos. Por exemplo, a atribuição:

```
*v = 7;
```

armazena o valor 7 onde `v` está apontando. Como não sabemos para onde `v` está apontando, o resultado da execução dessa linha de código é imprevisível.

Do mesmo modo, podemos usar uma variável apontadora, que aponta para uma posição de um vetor, como um vetor. O trecho de código a seguir ilustra essa afirmação.

```
#define DIM 100
...
int v[DIM], soma, *p;
...
soma = 0;
p = v;
for (i = 0; i < DIM; i++)
  soma = soma + p[i];
```

O compilador trata `p[i]` como `*(p+i)`, que é uma forma possível de usar aritmética com apontadores. Essa possibilidade de uso, que parece um tanto estranha à primeira vista, é muito útil em alocação dinâmica de memória, como veremos em breve.

## Exercícios

51.1 Suponha que as seguintes declarações foram realizadas:

```
int v[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &v[1], *q = &v[5];
```

(a) Qual o valor de `*(p + 3)`?

(b) Qual o valor de `*(q - 3)`?

(c) Qual o valor de `q - p)`?

(d) A expressão `p < q` tem valor verdadeiro ou falso?

(e) A expressão `*p < *q` tem valor verdadeiro ou falso?

51.2 Qual o conteúdo do vetor `v` após a execução do seguinte trecho de código?

```
#define N 10
int v[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &v[0], *q = &v[N-1], temp;
while (p < q) {
  temp = *p;
  *p++ = *q;
  *q-- = temp;
}
```

51.3 Suponha que `v` é um vetor e `p` é um apontador. Considere que a atribuição `p = v;` foi realizada previamente. Quais das expressões abaixo não são permitidas? Das restantes, quais têm valor verdadeiro?

(a) `p == v[0]`

(b) `p == &v[0]`

(c) `*p == v[0]`

(d) `p[0] == v[0]`

51.4 Escreva um programa que leia uma mensagem e a imprima em ordem reversa. Use a função `getchar` para ler caractere por caractere, armazenando-os em um vetor. Pare quando encontrar um caractere de mudança de linha `'\n'`. Faça o programa de forma a usar um apontador, ao invés de um índice como um número inteiro, para controlar a posição corrente no vetor.

AULA 52

# APONTADORES E MATRIZES

Na aula 51, trabalhamos com apontadores e vetores. Nesta aula veremos as relações entre apontadores e matrizes. Neste caso, é necessário estender o conceito de indireção (simples) para indireção dupla. Veremos também como estender esse conceito para indireção múltipla e suas relações com variáveis compostas homogêneas multi-dimensionais.

## 52.1 Apontadores para elementos de uma matriz

Sabemos há muito tempo que a linguagem C armazena matrizes, que são objetos representados de forma bidimensional, como uma seqüência contínua de compartimentos de memória, com demarcações de onde começa e onde termina cada uma de suas linhas. Ou seja, uma matriz é armazenada como um vetor na memória, com marcas especiais em determinados pontos regularmente espaçados: os elementos da linha 0 vêm primeiro, seguidos pelos elementos da linha 1, da linha 2, e assim por diante. Uma matriz com $n$ linhas tem a aparência ilustrada como na figura 24.2.

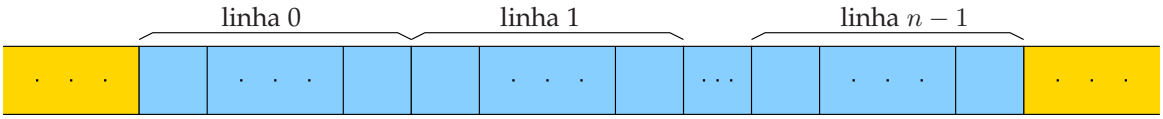



Figura 52.1: Representação da alocação de espaço na memória para uma matriz.

Essa representação pode nos ajudar a trabalhar com apontadores e matrizes. Se fazemos um apontador `p` apontar para a primeira célula de uma matriz, isto é, o elemento na posição (0,0), podemos então visitar todos os elementos da matriz incrementando o apontador `p` repetidamente.

Por exemplo, suponha que queremos inicializar todos os elementos de uma matriz de números inteiros com 0. Suponha que temos a declaração da seguinte matriz:

```
int A[LINHAS][COLUNAS];
```

A forma que aprendemos inicializar uma matriz pode ser aplicada à matriz `A` declarada acima e então o seguinte trecho de código realiza a inicialização desejada:

```
int i, j;
...
for (i = 0; i < LINHAS; i++)
  for (j = 0; j < COLUNAS; j++)
    A[i][j] = 0;
```

Mas se vemos a matriz `A` da forma como é armazenada na memória, isto é, como um vetor unidimensional, podemos trocar o par de estruturas de repetição por uma única estrutura de repetição:

```
int *p;
...
for (p = &A[0][0]; p <= &A[LINHAS-1][COLUNAS-1]; p++)
  *p = 0;
```

A estrutura de repetição acima inicia com `p` apontando para `A[0][0]`. Os incrementos sucessivos de `p` fazem-no apontar para `A[0][1]`, `A[0][2]`, e assim por diante. Quando `p` atinge `A[0][COLUNAS-1]`, o último elemento da linha 0, o próximo incremento faz com que `p` aponte para `A[1][0]`, o primeiro elemento da linha 1. O processo continua até que `p` ultrapasse o elemento `A[LINHAS-1][COLUNAS-1]`, o último elemento da matriz.

## 52.2 Processamento das linhas de uma matriz

Podemos processar uma linha de uma matriz – ou seja, percorrê-la, visitar os conteúdos dos compartimentos, usar seus valores, modificá-los, etc – também usando apontadores. Por exemplo, para visitar os elementos da linha `i` de uma matriz `A` podemos usar um apontador `p` apontar para o elemento da linha `i` e da coluna `0` da matriz `A`, como mostrado abaixo:

```
p = &A[i][0];
```

ou poderíamos fazer simplesmente

```
p = A[i];
```

já que, para qualquer matriz `A`, a expressão `A[i]` é um apontador para o primeiro elemento da linha `i`.

A justificativa para essa afirmação vem da aritmética com apontadores. Lembre-se que para um vetor `A`, a expressão `A[i]` é equivalente a `*(A + i)`. Assim, `&A[i][0]` é o mesmo que `&(*(A[i] + 0))`, que é equivalente a `&*A[i]` e que, por fim, é equivalente a `A[i]`. No trecho de código a seguir usamos essa simplificação para inicializar com zeros a linha `i` da matriz `A`:

```
int A[LINHAS][COLUNAS], *p, i;
...
for (p = A[i]; p < A[i] + COLUNAS; p++)
  *p = 0;
```

Como `A[i]` é um apontador para a linha `i` da matriz `A`, podemos passar `A[i]` para um função que espera receber um vetor como argumento. Em outras palavras, um função que foi projetada para trabalhar com um vetor também pode trabalhar com uma linha de uma matriz. Dessa forma, a função `max` da aula 51 pode ser chamada com a linha `i` da matriz `A` como argumento:

```
M = max(A[i], COLUNAS);
```

## 52.3 Processamento das colunas de uma matriz

O processamento dos elementos de uma coluna de uma matriz não é tão simples como o processamento dos elementos de uma linha, já que a matriz é armazenada linha por linha na memória. A seguir, mostramos uma estrutura de repetição que inicializa com zeros a coluna `j` da matriz `A`:

```
int A[LINHAS][COLUNAS], (*p)[COLUNAS], j;
...
for (p = &A[0]; p < A[LINHAS]; p++)
  (*p)[j] = 0;
```

Nesse exemplo, declaramos `p` como um apontador para um vetor de dimensão `COLUNAS`, cujos elementos são números inteiros. Os parênteses envolvendo `*p` são necessários, já que sem eles o compilador trataria `p` como um vetor de apontadores em vez de um apontador para um vetor. A expressão `p++` avança `p` para a próxima linha. Na expressão `(*p)[j]`, `*p` representa uma linha inteira de `A` e assim `(*p)[j]` seleciona o elemento na coluna `j` da linha. Os parênteses são essenciais na expressão `(*p)[i]`, já que, sem eles, o compilador interpretaria a expressão como `*(p[i])`.

## 52.4 Identificadores de matrizes como apontadores

Assim como o identificador de um vetor pode ser usado como um apontador, o identificador de uma matriz também pode e, na verdade, de qualquer o identificador de qualquer variável composta homogênea pode. No entanto, alguns cuidados são necessários quando ultrapassamos a barreira de duas ou mais dimensões.

Considere a declaração da matriz a seguir:

```
int A[LINHAS][COLUNAS];
```

Neste caso, `A` *não* é um apontador para `A[0][0]`. Ao contrário, é um apontador para `A[0]`. Isso faz mais sentido se olharmos sob o ponto de vista da linguagem C, que considera `A` não como uma matriz bidimensional, mas como um vetor. Quando usado como um apontador, `A` tem tipo `int (*)[COLUNAS]`, um apontador para um vetor de números inteiros de tamanho `COLUNAS`.

Saber que `A` aponta para `A[0]` é útil para simplificar estruturas de repetição que processam elementos de uma matriz. Por exemplo, ao invés de escrever:

```
for (p = &A[0]; p < A[LINHAS]; p++)
  (*p)[j] = 0;
```

para inicializar a coluna `j` da matriz `A`, podemos escrever

```
for (p = A; p < A + LINHAS; p++)
  (*p)[j] = 0;
```

Com essa idéia, podemos fazer o compilador acreditar que uma variável composta homogênea multi-dimensional é unidimensional, isto é, é um vetor. Por exemplo, podemos passar a matriz `A` como argumento para a função `max` da aula 51 da seguinte forma:

```
M = max(A[0], LINHAS*COLUNAS);
```

já que `A[0` aponta para o elemento na linha 0 e coluna 0 e tem tipo `int *` e assim essa chamada será executada corretamente.

## Exercícios

52.1 Escreva uma função que preencha uma matriz quadrada de dimensão $n$ com a matriz identidade $I_n$. Use um único apontador que percorra a matriz.

52.2 Reescreva a função abaixo usando aritmética de apontadores em vez de índices de matrizes. Em outras palavras, elimine as variáveis `i` e `j` e todos os `[]`. Use também uma única estrutura de repetição.

```
int somaMatriz(const int A[DIM][DIM], int n)
{
  int i, j, soma = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      soma = soma + A[i][j];
  return soma;
}
```

AULA 53

# APONTADORES E CADEIAS DE CARACTERES

Nas aulas 51 e 52 estudamos formas de trabalhar com variáveis compostas homogêneas e apontadores para seus elementos. No entanto, é importante ainda estudar a relação entre apontadores e as cadeias de caracteres que, como já vimos, são vetores especiais que contêm caracteres. Nesta aula aprenderemos algumas particularidades de apontadores para elementos de cadeias de caracteres na linguagem C, além de estudar a relação entre apontadores e constantes que são cadeias de caracteres.

## 53.1 Literais e apontadores

Devemos relembrar que uma **literal** é uma seqüência de caracteres envolvida por aspas duplas. Um exemplo de uma literal é apresentado a seguir[1]:

```
"O usuário médio de computador possui o cérebro de um macaco-aranha"
```

Nosso primeiro contato com literais foi ainda na aula 4. Literais ocorrem com freqüência na chamada das funções `printf` e `scanf`. Mas quando chamamos uma dessas funções e fornecemos uma literal como argumento, o que de fato estamos passando? Em essência, a linguagem C trata literais como cadeias de caracteres. Quando o compilador encontra uma literal de comprimento $n$ em um programa, ele reserva um espaço de $n + 1$ bytes na memória. Essa área de memória conterá os caracteres da literal mais o caracter nulo que indica o final da cadeia. O caracteres nulo é um byte cujos bits são todos zeros e é representado pela seqüência de caracteres `\0`. Por exemplo, a literal `"abc"` é armazenada como uma cadeia de quatro caracteres, como mostra a figura 53.1.

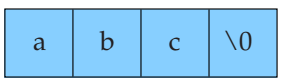



Figura 53.1: Representação de uma literal na memória.

Literais podem ser vazias. A literal `""` é armazenada como um único caractere nulo.

[1] Frase de Bill Gates, dono da Micro$oft, em uma entrevista para *Computer Magazine*.

Como uma literal é armazenada em um vetor, o compilador a enxerga como um apontador do tipo `char *`. As funções `printf` e `scanf`, por exemplo, esperam um valor do tipo `char *` como primeiro argumento. Se, por exemplo, fazemos a seguinte chamada:

```
printf("abc");
```

o endereço da literal `"abc"` é passado como argumento para a função `printf`, isto é, o endereço de onde se encontra o caractere `a` na memória.

Em geral, podemos usar uma literal sempre que a linguagem C permita o uso de um apontador do tipo `char *`. Por exemplo, uma literal pode ocorrer do lado direito de uma atribuição, como mostrado a seguir:

```
char *p;
p = "abc";
```

Essa atribuição não copia os caracteres de `"abc"`, mas faz o apontador `p` apontar para o primeiro caractere da literal, como mostra a figura 53.2.

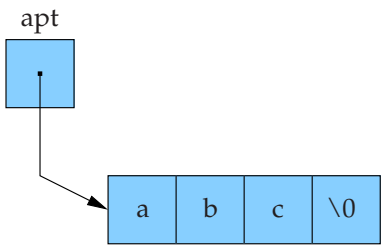



Figura 53.2: Representação da atribuição de uma literal a um apontador.

Observe ainda que não é permitido alterar uma literal durante a execução de um programa. Isso significa que a tentativa de modificar uma literal pode causar um comportamento indefinido do programa.

Relembrando, uma variável cadeia de caracteres é um vetor do tipo `char` que necessariamente deve reservar espaço para o caractere nulo. Quando declaramos um vetor de caracteres que será usado para armazenar cadeias de caracteres, devemos sempre declarar esse vetor com uma posição a mais que a mais longa das cadeias de caracteres possíveis, já que por convenção da linguagem C, toda cadeia de caracteres é finalizada com um caractere nulo. Veja, por exemplo, a declaração a seguir:

```
char cadeia[TAM+1];
```

onde `TAM` é uma macro definida com o tamanho da cadeia de caracteres mais longa que pode ser armazenada na variável `cadeia`.

Lembrando ainda, podemos inicializar um cadeia de caracteres no momento de sua declaração, como mostra o exemplo abaixo:

```
char data[13] = "9 de outubro";
```

O compilador então coloca os caracteres de `"9 de outubro"` no vetor `data` e adiciona o caractere nulo ao final para que `data` possa ser usada como uma cadeia de caracteres. A figura 53.3 ilustra essa situação.


Figura 53.3: Declaração e inicialização de uma cadeia de caracteres.

Apesar de `"9 de outubro"` se parecer com uma literal, a linguagem C de fato a vê como uma abreviação para um inicializador de um vetor. Ou seja, a declaração e inicialização acima é enxergada pelo compilador como abaixo:

```
char data[13] = {'9',' ','d','e',' ','o','u','t','u','b','r','o','\0'};
```

No caso em que o inicializador é menor que o tamanho definido para a cadeia de caracteres, o compilador preencherá as posições finais restantes da cadeia com o caractere nulo. Por outro lado, é sempre importante garantir que o inicializador tenha menor comprimento que o tamanho do vetor declarado. Também, podemos omitir o tamanho do vetor em uma declaração e inicialização simultâneas, caso em que o vetor terá o tamanho equivalente ao comprimento do inicializador mais uma unidade, que equivale ao caractere nulo.

Agora, vamos comparar a declaração abaixo:

```
char data[] = "9 de outubro";
```

que declara uma vetor `data`, que é uma cadeia de caracteres, com a declaração a seguir:

```
char *data = "9 de outubro";
```

que declara `data` como um apontador. Devido à relação estrita entre vetores e apontadores, podemos usar as duas versões da declaração de `data` como uma cadeia de caractere. Em particular, qualquer função que receba um vetor de caracteres ou um apontador para caracteres aceita qualquer uma das versões da declaração da variável `data` apresentada acima.

No entanto, devemos ter cuidado para não cometer o erro de acreditar que as duas versões da declaração de `data` são equivalentes e intercambiáveis. Existem diferenças significativas entre as duas, que destacamos abaixo:

- na versão em que a variável é declarada como um vetor, os caracteres armazenados em `data` podem ser modificados, como fazemos com elementos de qualquer vetor; na versão em que a variável é declarada como um apontador, `data` aponta para uma literal que, como já vimos, não pode ser modificada;
- na versão com vetor, `data` é um identificador de um vetor; na versão com apontador, `data` é uma variável que pode, inclusive, apontar para outras cadeias de caracteres durante a execução do programa.

Se precisamos que uma cadeia de caracteres seja modificada, é nossa responsabilidade declarar um vetor de caracteres no qual será armazenado essa cadeia. Declarar um apontador não é suficiente, neste caso. Por exemplo, a declaração abaixo:

```
char *p;
```

faz com que o compilador reserve espaço suficiente para uma variável apontadora. Infelizmente, o compilador não reserva espaço para uma cadeia de caracteres, mesmo porque, não há indicação alguma de um possível comprimento da cadeia de caracteres que queremos armazenar. Antes de usarmos `p` como uma cadeia de caracteres, temos de fazê-la apontar para um vetor de caracteres. Uma possibilidade é fazer `p` apontar para uma variável que é uma cadeia de caracteres, como mostramos a seguir:

```
char cadeia[TAM+1], *p;
p = cadeia;
```

Com essa atribuição, `p` aponta para o primeiro caractere de `cadeia` e assim podemos usar `p` como uma cadeia de caracteres. Outra possibilidade é fazer `p` apontar para uma cadeia de caracteres dinamicamente alocada, como veremos na aula 55.

Ainda poderíamos discorrer sobre processos para leitura e escrita de cadeias de caracteres, sobre acesso aos caracteres de uma cadeia de caracteres e também sobre o uso das funções da biblioteca da linguagem C que trabalham especificamente com cadeias de caracteres, o que já fizemos nas aulas 23 e 45. Ainda veremos a seguir dois tópicos importantes sobre cadeias de caracteres: vetores de cadeias de caracteres e argumentos de linha de comando.

## 53.2 Vetores de cadeias de caracteres

Uma forma de armazenar em memória um vetor de cadeias de caracteres é através da criação de uma matriz de caracteres e então armazenar as cadeias de caracteres uma a uma. Por exemplo, podemos fazer como a seguir:

```
char planetas[][9] = {"Mercurio", "Venus", "Terra",
                      "Marte", "Jupiter", "Saturno",
                      "Urano", "Netuno", "Plutao"};
```

Observe que estamos omitindo o número de linhas da matriz, que é fornecido pelo inicializador, mas a linguagem C exige que o número de colunas seja especificado, conforme fizemos na declaração.

A figura 53.4 ilustra a declaração e inicialização da variável `planetas`. Observe que todas as cadeias cabem nas colunas da matriz e, também, que há um tanto de compartimentos desperdiçados na matriz, preenchidos com o caractere `\0`, já que nem todas as cadeias são compostas por 8 caracteres.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | M | e | r | c | u | r | i | o | \0 |
| 1 | V | e | n | u | s | \0 | \0 | \0 | \0 |
| 2 | T | e | r | r | a | \0 | \0 | \0 | \0 |
| 3 | M | a | r | t | e | \0 | \0 | \0 | \0 |
| 4 | J | u | p | i | t | e | r | \0 | \0 |
| 5 | S | a | t | u | r | n | o | \0 | \0 |
| 6 | U | r | a | n | o | \0 | \0 | \0 | \0 |
| 7 | N | e | t | u | n | o | \0 | \0 | \0 |
| 8 | P | l | u | t | a | o | \0 | \0 | \0 |

Figura 53.4: Matriz de cadeias de caracteres `planetas`.

A ineficiência de armazenamento aparente nesse exemplo é comum quando trabalhamos com cadeias de caracteres, já que coleções de cadeias de caracteres serão, em geral, um misto entre curtas e longas cadeias. Uma possível forma de sanar esse problema é usar um vetor cujos elementos são apontadores para cadeias de caracteres, como podemos ver na declaração abaixo:

```
char *planetas[] = {"Mercurio", "Venus", "Terra",
                    "Marte", "Jupiter", "Saturno",
                    "Urano", "Netuno", "Plutao"};
```

Note que há poucas diferenças entre essa declaração e a declaração anterior da variável `planetas`: removemos um par de colchetes com um número no interior deles e colocamos um asterisco precedendo o identificador da variável. No entanto, o efeito dessa declaração na memória é muito diferente, como podemos ver na figura 53.5.

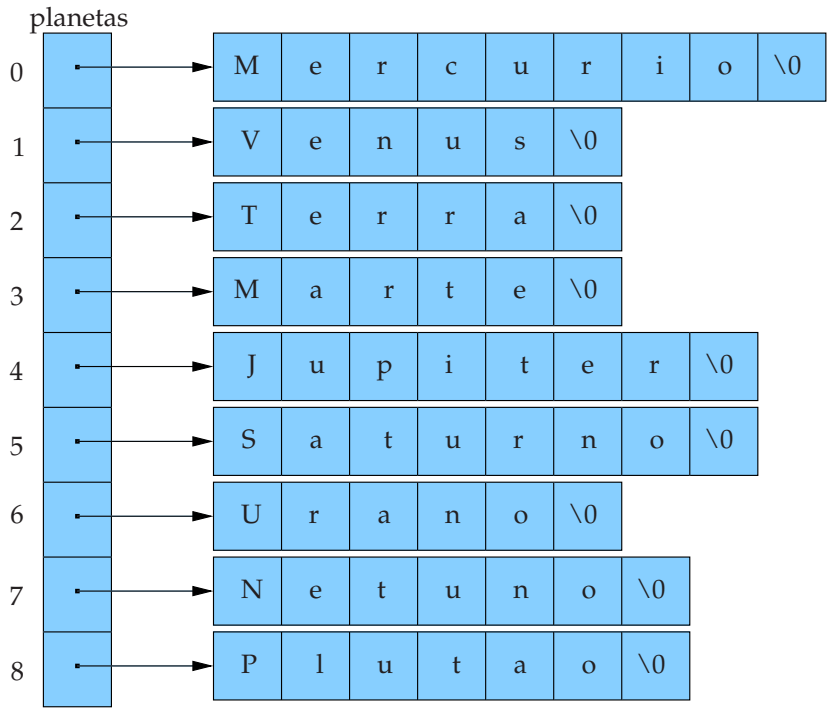



Figura 53.5: Vetor `planetas` de apontadores para cadeias de caracteres.

Cada elemento do vetor `planetas` é um apontador para uma cadeia de caracteres, terminada com um caractere nulo. Não há mais desperdício de compartimentos nas cadeias de caracteres, apesar de termos de alocar espaço para os apontadores no vetor `planetas`. Para acessar um dos nomes dos planetas necessitamos apenas do índice do vetor. Para acessar um caractere do nome de um planeta devemos fazer da mesma forma como acessamos um elemento em uma matriz. Por exemplo, para buscar cadeias de caracteres no vetor `planetas` que iniciam com a letra M, podemos usar o seguinte trecho de código:

```
for (i = 0; i < 9; i++)
  if (planetas[i][0] == 'M')
    printf("%s começa com M\n", planetas[i]);
```

## 53.3 Argumentos na linha de comandos

Quando executamos um programa, em geral, devemos fornecer a ele alguma informação como por exemplo um nome de um arquivo, uma opção que modifica seu comportamento, etc. Por exemplo, considere o comando UNIX `ls`. Se executamos esse comando como abaixo:

```
prompt$ ls
```

em uma linha de comando, o resultado será uma listagem de nomes dos arquivos no diretório atual. Se digitamos o comando seguido de uma opção, como abaixo:

```
prompt$ ls -l
```

então o resultado é uma listagem detalhada[2] que nos mostra o tamanho de cada arquivo, seu proprietário, a data e hora em que houve a última modificação no arquivo e assim por diante. Para modificar ainda mais o comportamento do comando `ls` podemos especificar que ele mostre detalhes de apenas um arquivo, como mostrado abaixo:

```
prompt$ ls -l exerc1.c
```

Nesse caso, o comando `ls` mostrará informações detalhadas sobre o arquivo `exerc1.c`.

Informações de comando de linha estão disponíveis para todos os programas, não apenas para comandos do sistema operacional. Para ter acesso aos **argumentos de linha de comando**, chamados de **parâmetros do programa** na linguagem C padrão, devemos definir a função `main` como uma função com dois parâmetros que costumeiramente têm identificadores `argc` e `argv`. Isto é, devemos fazer como abaixo:

```
int main(int argc, char *argv[])
{
  ...
}
```

O parâmetro `argc`, abreviação de "contador de argumentos", é o número de argumentos de linha de comando, incluindo também o nome do programa. O parâmetro `argv`, abreviação de "vetor de argumentos", é um vetor de apontadores para os argumentos da linha de

[2] `l` do inglês *long*.

comando, que são armazenados como cadeias de caracteres. Assim, `argv[0]` aponta para o nome do programa, enquanto que `argv[1]` até `argv[argc-1]` apontam para os argumentos da linha de comandos restantes. O vetor `argv` tem um elemento adicional `argv[argc]` que é sempre um apontador nulo, um apontador especial que aponta para nada, representado pela macro `NULL`.

Se um(a) usuário(a) digita a linha de comando abaixo:

```
prompt$ ls -l exerc1.c
```

então `argc` conterá o valor 3, `argv[0]` apontará para a cadeia de caracteres com o nome do programa, `argv[1]` apontará para a cadeia de caracteres `"-l"`, `argv[2]` apontará para a cadeia de caracteres `exerc1.c` e `argv[3]` apontará para nulo. A figura 53.6 ilustra essa situação. Observe que o nome do programa não foi listado porque pode incluir o nome do diretório ao qual o programa pertence ou ainda outras informações que dependem do sistema operacional.

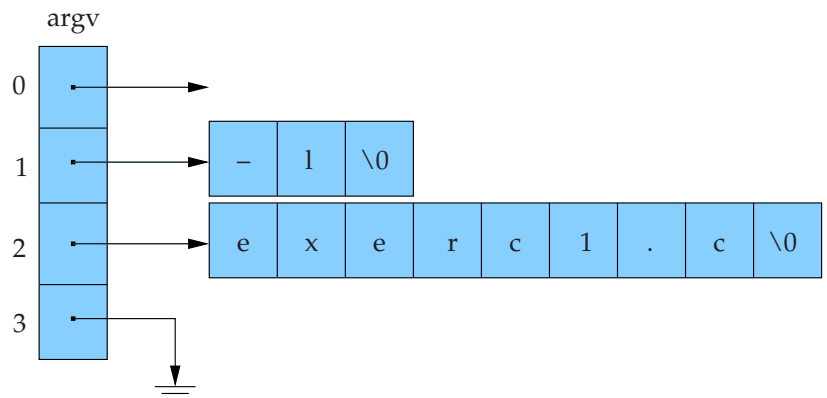



Figura 53.6: Representação de `argv`.

Como `argv` é um vetor de apontadores, o acesso aos argumentos da linha de comandos é realizado, em geral, como mostrado na estrutura de repetição a seguir:

```
int i;
for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

O programa 53.1 ilustra como acessar os argumentos de uma linha de comandos. O programa é projetado para verificar se as cadeias de caracteres fornecidas na linha de comandos são nomes de planetas do sistema solar.

Se o programa 53.1 tem o nome `planetas.c` e seu executável correspondente tem nome `planetas`, então podemos executar esse programa com uma seqüência de cadeias de caracteres, como mostramos no exemplo abaixo:

Programa 53.1: Verifica nomes de planetas.

```
1  #include <stdio.h>
2  #include <string.h>
3  #define NUM_PLANETAS 9
4  int main(int argc, char *argv[])
5  {
6    char *planetas[] = {"Mercurio", "Venus", "Terra",
7                        "Marte", "Jupiter", "Saturno",
8                        "Urano", "Netuno", "Plutao"};
9    int i, j, achou;
10   for (i = 1; i < argc; i++) {
11     achou = 0;
12     for (j = 0; j < NUM_PLANETAS && !achou; j++)
13       if (strcmp(argv[i], planetas[j]) == 0)
14         achou = 1;
15     if (achou)
16       printf("%s é o planeta %d\n", argv[i], j+1);
17     else
18       printf("%s não é um planeta\n", argv[i]);
19   }
20   return 0;
21 }
```

```
prompt$ ./planetas Jupiter venus Terra Joaquim
```

O resultado da execução dessa execução é dado a seguir:

```
Jupiter é o planeta 5
venus não é um planeta
Terra é o planeta 3
Joaquim não é um planeta
```

## Exercícios

53.1 As chamadas de funções abaixo supostamente escrevem um caractere de mudança de linha na saída, mas algumas delas estão erradas. Identifique quais chamadas não funcionam e explique o porquê.

(a) `printf("%c", '\n');`

(b) `printf("%c", "\n");`

(c) `printf("%s", '\n');`

(d) `printf("%s", "\n");`

(e) `printf('\n');`

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

(g) `putchar('\n');`

(h) `putchar("\n");`

53.2 Suponha que declaramos um apontador `p` como abaixo:

```
char *p = "abc";
```

Quais das chamadas abaixo estão corretas? Mostre a saída produzida por cada chamada correta e explique por que a(s) outra(s) não está(ão) correta(s).

(a) `putchar(p);`

(b) `putchar(*p);`

(c) `printf("%s", p);`

(d) `printf("%s", *p);`

53.3 Suponha que declaramos as seguintes variáveis:

```
char s[MAX+1];
int i, j;
```

Suponha também que a seguinte chamada foi executada:

```
scanf("%d%s%d", &i, s, &j);
```

Se o usuário digita a seguinte entrada:

```
12abc34 56def78
```

qual serão os valores de `i`, `j` e `s` depois dessa chamada?

53.4 A função abaixo supostamente cria uma cópia idêntica de uma cadeia de caracteres. O que há de errado com a função?

```
char *duplica(const char *p)
{
  char *q;
  strcpy(q, p);
  return q;
}
```

53.5 O que imprime na saída o programa abaixo?

```
#include <stdio.h>
int main(void)
{
  char s[] = "Dvmuvsb", *p;
  for (p = s; *p; p++)
    --*p;
  printf("%s\n", s);
  return 0;
}
```

53.6 (a) Escreva uma função com a seguinte interface:

```
void maiuscula(char cadeia[])
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use `cadeia` apenas como vetor, juntamente com os índices necessários.

(b) Escreva uma função com a seguinte interface:

```
void maiuscula(char *cadeia)
```

que receba uma cadeia de caracteres (terminada com um caractere nulo) contendo caracteres arbitrários e substitua os caracteres que são letras minúsculas nessa cadeia por letras maiúsculas. Use apenas apontadores e aritmética com apontadores.

53.7 (a) Escreva uma função que receba uma cadeia de caracteres e devolva o número total de caracteres que ela possui.

(b) Escreva uma função que receba uma cadeia de caracteres e devolva o número de vogais que ela possui.

(c) Escreva uma função que receba uma cadeia de caracteres e devolva o número de consoantes que ela possui.

(d) Escreva um programa que receba diversas cadeias de caracteres e faça a média do número de vogais, de consoantes e de símbolos de pontuação que elas possuem.

Use apenas apontadores nas funções em (a), (b) e (c).

53.8 Escreva um programa que encontra a maior e a menor palavra de uma seqüência de palavras informadas pelo(a) usuário(a). O programa deve terminar se uma palavra de quatro letras for fornecida na entrada. Considere que nenhuma palavra tem mais que 20 letras.

Um exemplo de entrada e saída do programa pode ser assim visualizado:

```
Informe uma palavra: laranja
Informe uma palavra: melao
Informe uma palavra: tomate
Informe uma palavra: cereja
Informe uma palavra: uva
Informe uma palavra: banana
Informe uma palavra: maca

Maior palavra: laranja
Menor Palavra: uva
```

53.9 Escreva um programa com nome `reverso.c` que mostra os argumentos da linha de comandos em ordem inversa. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./reverso garfo e faca
```

deve produzir a seguinte saída:

```
faca e garfo
```

53.10 Escreva um programa com nome `soma.c` que soma todos os argumentos informados na linha de comandos, considerando que todos eles são números inteiros. Por exemplo, executando o programa da seguinte forma:

```
prompt$ ./soma 81 25 2
```

deve produzir a seguinte saída:

```
108
```

AULA 54

# APONTADORES E REGISTROS

Nesta aula trabalharemos com apontadores e registros. Primeiro, veremos como declarar e usar apontadores para registros. Essas tarefas são equivalentes as que já fizemos quando usamos apontadores para números inteiros, por exemplo. Além disso, vamos adicionar também apontadores como campos de registros. Quando registros contêm apontadores podemos usá-los como estruturas de dados poderosas tais como listas encadeadas, árvores, etc, como veremos daqui por diante.

## 54.1 Apontadores para registros

Suponha que definimos uma etiqueta de registro `data` como a seguir:

```
struct data {
  int dia;
  int mes;
  int ano;
};
```

A partir dessa definição, podemos declarar variáveis do tipo `struct data`, como abaixo:

```
struct data hoje;
```

E então, assim como fizemos com apontadores para inteiros, caracteres e números de ponto flutuante, podemos declarar um apontador para o registro `data` da seguinte forma:

```
struct data *p;
```

Podemos, a partir dessa declaração, fazer uma atribuição à variável `p` como a seguir:

```
p = &hoje;
```

Além disso, podemos atribuir valores aos campos do registro de forma indireta, como fazemos abaixo:

```
(*p).dia = 11;
```

Essa atribuição tem o efeito de armazenar o número inteiro 11 no campo `dia` da variável `hoje`, indiretamente através do apontador `p` no entanto. Nessa atribuição, os parênteses envolvendo `*p` são necessários porque o operador `.`, de seleção de campo de um registro, tem maior prioridade que o operador `*` de indireção. É importante relembrar também que essa forma de acesso indireto aos campos de um registro pode ser substituída, e tem o mesmo efeito, pelo operador `->` como mostramos no exemplo abaixo:

```
p->dia = 11;
```

O programa 54.1 ilustra o uso de apontadores para registros.

Programa 54.1: Uso de um apontador para um registro.

```
1  #include <stdio.h>
2  struct data {
3    int dia;
4    int mes;
5    int ano;
6  };
7  int main(void)
8  {
9    struct data hoje, *p;
10   p = &hoje;
11   p->dia = 13;
12   p->mes = 9;
13   p->ano = 2007;
14   printf("A data de hoje é %d/%d/%d\n", hoje.dia, hoje.mes, hoje.ano);
15   return 0;
16 }
```

Na linha 9 há a declaração de duas variáveis: um registro com identificador `hoje` e um apontador para registros com identificador `p`. Na linha 10, `p` recebe o endereço da variável

`hoje`. Observe que a variável `hoje` é do tipo `struct data`, isto é, a variável `hoje` é do mesmo tipo da variável `p` e, portanto, essa atribuição é válida. Em seguida, valores do tipo inteiro são armazenados na variável `hoje`, mas de forma indireta, com uso do apontador `p`. Por fim, os valores atribuídos são impressos na saída. A figura 54.1 mostra as variáveis `hoje` e `p` depois das atribuições realizadas durante a execução do programa 54.1.

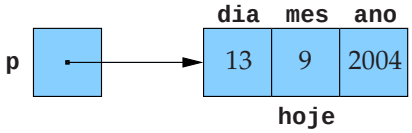



Figura 54.1: Representação do apontador `p` e do registro `hoje`.

## 54.2 Registros contendo apontadores

Podemos também usar apontadores como campos de registros. Por exemplo, podemos definir uma etiqueta de registro como abaixo:

```
struct reg_apts {
  int *apt1;
  int *apt2;
};
```

A partir dessa definição, podemos declarar variáveis (registros) do tipo `struct reg_apts` como a seguir:

```
struct reg_apts bloco;
```

Em seguida, a variável `bloco` pode ser usada como sempre fizemos. Note apenas que `bloco` não é um apontador, mas um registro que contém dois campos que são apontadores. Veja o programa 54.2, que mostra o uso dessa variável.

Observe atentamente a diferença entre `(*p).dia` e `*reg.apt1`. No primeiro caso, `p` é um apontador para um registro e o acesso indireto a um campo do registro, via esse apontador, tem de ser feito com a sintaxe `(*p).dia`, isto é, o conteúdo do endereço contido em `p` é um registro e, portanto, a seleção do campo é descrita fora dos parênteses. No segundo caso, `reg` é um registro – e não um apontador para um registro – e como contém campos que são apontadores, o acesso ao conteúdo dos campos é realizado através do operador de indireção `*`. Assim, `*reg.apt1` significa que queremos acessar o conteúdo do endereço apontado por `reg.apt1`. Como o operador de seleção de campo `.` de um registro tem prioridade pelo operador de indireção `*`, não há necessidade de parênteses, embora pudéssemos usá-los da forma `*(reg.apt1)`. A figura 54.2 ilustra essa situação.

Programa 54.2: Uso de um registro que contém campos que são apontadores.

```
#include <stdio.h>
struct apts_int {
  int *apt1;
  int *apt2;
};
int main(void)
{
  int i1, i2;
  struct apts_int reg;
  i2 = 100;
  reg.apt1 = &i1;
  reg.apt2 = &i2;
  *reg.apt1 = -2;
  printf("i1 = %d, *reg.apt1 = %d\n", i1, *reg.apt1);
  printf("i2 = %d, *reg.apt2 = %d\n", i2, *reg.apt2);
  return 0;
}
```

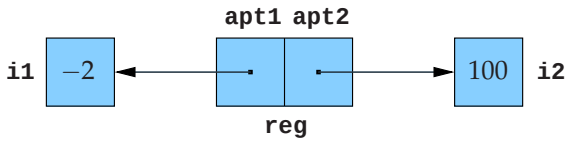



Figura 54.2: Representação do registro `reg` contendo dois campos apontadores.

## Exercícios

54.1 Qual a saída do programa descrito abaixo?

```
#include <stdio.h>
struct dois_valores {
  int vi;
  float vf;
};
int main(void)
{
  struct dois_valores reg1 = {53, 7.112}, reg2, *p = &reg1;
  reg2.vi = (*p).vf;
  reg2.vf = (*p).vi;
  printf("1: %d %f\n2: %d %f\n", reg1.vi, reg1.vf, reg2.vi, reg2.vf);
  return 0;
}
```

54.2 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>
struct apts {
  char *c;
  int *i;
  float *f;
};
int main(void)
{
  char caractere;
  int inteiro;
  float real;
  struct apts reg;
  reg.c = &caractere;
  reg.i = &inteiro;
  reg.f = &real;
  scanf("%c%d%f", reg.c, reg.i, reg.f);
  printf("%c\n%d\n%f\n", caractere, inteiro, real);
  return 0;
}
```

54.3 Simule a execução do programa descrito abaixo.

```
#include <stdio.h>
struct celula {
  int valor;
  struct celula *prox;
};
int main(void)
{
  struct celula reg1, reg2, *p;
  scanf("%d%d", &reg1.valor, &reg2.valor);
  reg1.prox = &reg2;
  reg2.prox = NULL;
  for (p = &reg1; p != NULL; p = p->prox)
    printf("%d ", p->valor);
  printf("\n");
  return 0;
}
```

AULA 55

# USO AVANÇADO DE APONTADORES

Nas aulas 49 a 54 vimos formas importantes de uso de apontadores: como parâmetros de funções simulando passagem por referência e como elementos da linguagem C que podem acessar indiretamente outros compartimentos de memória, seja uma variável, uma célula de um vetor ou de uma matriz ou ainda um campo de um registro, usando, inclusive uma aritmética específica para tanto.

Nesta aula veremos outros usos para apontadores: como auxiliares na alocação dinâmica de espaços de memória, como apontadores para funções e como apontadores para outros apontadores.

## 55.1 Alocação dinâmica de memória

As estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo. Por exemplo, uma vez que um programa foi compilado, a quantidade de elementos de um vetor ou de uma matriz é fixa. Isso significa que, para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente. Felizmente, a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa. Usando alocação dinâmica, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa. Apesar de disponível para qualquer tipo de dados, a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas.

Aprendemos até aqui a declarar uma variável composta homogênea especificando um identificador e sua(s) dimensão(ões). Por exemplo, veja as declarações a seguir:

```
int vet[100];
float mat[40][60];
```

Nesse caso, temos a declaração de um vetor com identificador `vet` e 100 posições de memória que podem armazenar números inteiros e uma matriz com identificador `mat` de 40 linhas e 60 colunas que são compartimentos de memória que podem armazenar números de ponto flutuante. Todos os compartimentos dessas variáveis compostas homogêneas ficam disponíveis para uso durante a execução do programa.

Em diversas aplicações, para que os dados de entrada sejam armazenados em variáveis compostas homogêneas com dimensão(ões) adequadas, é necessário saber antes essas dimensão(ões), o que é então solicitado a um(a) usuário(a) do programa logo de início. Por exemplo, o(a) usuário(a) da aplicação pode querer informar a quantidade de elementos que será armazenada no vetor `vet`, um valor que será mantido no programa para verificação do limite de armazenamento e que não deve ultrapassar o limite máximo de 100 elementos. Note que a previsão de limitante máximo deve sempre ser especificada também. Do mesmo modo, o(a) usuário(a) pode querer informar, antes de usar essa estrutura, quantas linhas e quantas colunas da matriz `mat` serão usadas, sem ultrapassar o limite máximo de 40 linhas e 60 colunas. Se o(a) usuário(a), por exemplo, usar apenas 10 compartimentos do vetor `vet` ou apenas $3 \times 3$ compartimentos da matriz `mat` durante a execução do programa, os compartimentos restantes não serão usados, embora tenham sido alocados na memória durante suas declarações, ocupando espaço desnecessário na memória do sistema computacional.

Essa alocação que acabamos de descrever e que conhecemos bem é chamada de **alocação estática de memória**, o que significa que, antes da execução, o compilador, quando encontra uma declaração como essa, reserva na memória um número fixo de compartimentos correspondentes à declaração. Esse espaço é fixo e não pode ser alterado durante a execução do programa.

Alocação estática e compartimentos não usados na memória podem não ter impacto significativo em programas pequenos, que fazem pouco uso da memória, como os nossos programas desenvolvidos até aqui. No entanto, devemos sempre ter em mente que a memória é um recurso limitado e que em programas maiores e que armazenam muitas informações em memória temos, de alguma forma, de usá-la de maneira eficiente, economizando compartimentos sempre que possível.

Nesse sentido, se pudéssemos declarar variáveis compostas homogêneas com o número exato de compartimentos que serão de fato usados durante a execução do programa, então não haveria esse desperdício mencionado. No exemplo da declaração de `vet` e `mat`, economizaríamos espaço significativo de memória, caso necessitássemos apenas de 10 compartimentos no vetor `vet` e 9 compartimentos na matriz `mat`. No entanto, em outra execução subseqüente do mesmo programa que declara essas variáveis, podem ser necessárias capacidades diferentes para ambas as variáveis. Dessa forma, fixar um valor torna-se inviável e o que melhor podemos fazer em alocação estática de memória e prever um limitante máximo para esssas quantidades, conforme vimos fazendo até aqui.

Felizmente para nós, programadores e programadoras da linguagem C, é possível alocar dinamicamente um ou mais blocos de memória na linguagem C. **Alocação dinâmica de memória** significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso. Caso haja espaço suficiente disponível na memória principal, a solicitação é atendida e o espaço solicitado fica reservado na memória para aquele uso específico. Caso contrário, isto é, se não há como atender a solicitação de espaço, uma mensagem de erro em tempo de execução é emitida e o(a) programador(a) tem de estar preparado para esse tipo de situação, incluindo testes de verificação de situações como essa no arquivo-fonte, informando ao(à) usuário(a) a impossibilidade de uso do espaço solicitado de memória, solicitando ainda alguma decisão do(a) usuário(a) e, provavelmente, encerrando a execução do programa.

Vejamos um exemplo no programa 55.1 que faz alocação dinâmica de um vetor.

Programa 55.1: Um exemplo de alocação dinâmica de memória.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5    int i, n;
6    int *vetor, *apt;
7    printf("Informe a dimensão do vetor: ");
8    scanf("%d", &n);
9    vetor = (int *) malloc(n * sizeof(int));
10   if (vetor != NULL) {
11     for (i = 0; i < n; i++) {
12       printf("Informe o elemento %d: ", i+1);
13       scanf("%d", (vetor+i));
14     }
15     printf("\nVetor          : ");
16     for (apt = vetor; apt < (vetor+n); apt++)
17       printf("%d ", *apt);
18     printf("\nVetor invertido: ");
19     for (i = n-1; i >= 0; i--)
20       printf("%d ", vetor[i]);
21     printf("\n");
22   }
23   else
24     printf("Impossível alocar o espaço requisitado\n");
25   return 0;
26 }
```

Na segunda linha do programa 55.1, incluímos o arquivo-cabeçalho `stdlib.h`, que contém a declaração da função `malloc`, usada na linha 9 desse programa. A função `malloc` tem sua interface apresentada a seguir:

```
void *malloc(size_t tamanho)
```

A função `malloc` reserva uma certa quantidade específica de memória e devolve um apontador do tipo `void`. No programa acima, reservamos $n$ compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`. Essa expressão, na verdade, reflete o número de bytes que serão alocados continuamente na memória, que depende do sistema computacional onde o programa é compilado. O operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão. Nesse caso, nas máquinas que usamos no laboratório, a expressão `sizeof(int)` devolve 4 bytes. Esse número é multiplicado por $n$, o número de compartimentos que desejamos para armazenar números inteiros. O endereço da primeira posição de

memória onde encontram-se esses compartimentos é devolvido pela função `malloc`. Essa função devolve um apontador do tipo `void`. Por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um apontador para um número inteiro. Por fim, esse endereço é armazenado em `vetor`, que foi declarado como um apontador para números inteiros. A partir daí, podemos usar cmdprgvet da forma como preferirmos, como um apontador ou como um vetor.

O programa 55.2 é um exemplo de alocação dinâmica de memória de uma matriz de números inteiros. Esse programa é um pouco mais complicado que o programa 55.1, devido ao uso distinto que faz da função `malloc`.

Programa 55.2: Um exemplo de alocação dinâmica de uma matriz.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5    int i, j, m, n, **matriz, **apt;
6    printf("Informe a dimensão da matriz: ");
7    scanf("%d %d", &m, &n);
8    matriz = (int **) malloc(m * sizeof(int *));
9    if (matriz == NULL) {
10     printf("Não há espaço suficiente na memória\n");
11     return 0;
12   }
13   for (apt = matriz, i = 0; i < m; i++, apt++) {
14     *apt = (int *) malloc(n * sizeof(int));
15     if (*apt == NULL) {
16       printf("Não há espaço suficiente na memória\n");
17       return 0;
18     }
19   }
20   for (i = 0; i < m; i++)
21     for (j = 0; j < n; j++) {
22       printf("Informe o elemento (%d,%d): ", i+1, j+1);
23       scanf("%d", &matriz[i][j]);
24     }
25   printf("\nMatriz:\n");
26   apt = matriz;
27   for (i = 0; i < m; i++) {
28     for (j = 0; j < n; j++)
29       printf("%d ", *(*(apt+i)+j));
30     printf("\n");
31   }
32   printf("\n");
33   return 0;
34 }
```

Observe por fim que as linhas em que ocorrem a alocação dinâmica de memória no programa 55.2 podem ser substituídas de forma equivalente pelas linhas a seguir:

```
matriz = (int **) malloc(m * sizeof(int *));
for (i = 0; i < m; i++)
  matriz[i] = (int *) malloc(n * sizeof(int));
```

Além da função `malloc` existem duas outras funções para alocação de memória na `stdlib.h`: `calloc` e `realloc`, mas a primeira é mais freqüentemente usada que essas outras duas. Essas funções solicitam blocos de memória de um espaço de armazenamento conhecido como ***heap***[1] ou ainda **lista de espaços disponíveis**. A chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um apontador nulo. Pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário. Considere o exemplo a seguir:

```
p = malloc(...);
q = malloc(...);
p = q;
```

Após as duas primeiras sentenças serem executadas, `p` aponta para um bloco de memória e `q` aponta para um outro. No entanto, após a atribuição de `q` para `p` na última sentença, as duas variáveis apontam para o mesmo bloco de memória, o segundo bloco, sem que nenhuma delas aponte para o primeiro. Além disso, não poderemos mais acessar o primeiro bloco, que ficará perdido na memória, ocupando espaço desnecessário. Esse bloco é chamado de **lixo**.

A função `free` é usada para ajudar os programadores da linguagem C a resolver o problema de geração de lixo na memória durante a execução de programas. Essa função tem a seguinte interface na `stdlib.h`:

```
void free(void *apt)
```

Para usar a função `free` corretamente, devemos lhe passar um apontador para um bloco de memória que não mais necessitamos, como fazemos abaixo:

```
p = malloc(...);
q = malloc(...);
free(p);
p = q;
```

[1] Uma tradução possível do inglês pode ser amontoado ou coleção de coisas dispostas juntas.

A chamada à função `free` devolve o bloco de memória apontado por `p` para o *heap*, que fica disponível para uso em chamadas subseqüentes das funções de alocação de memória.

O argumento da função `free` deve ser um apontador que foi previamente devolvido por uma função de alocação de memória.

## 55.2 Apontadores para apontadores

Como vimos até aqui, um apontador é uma variável cujo conteúdo é um endereço. Dessa forma, podemos acessar o conteúdo de uma posição de memória através de um apontador de forma indireta. Temos visto exemplos de apontadores e suas aplicações em algumas das aulas anteriores e também na seção anterior.

Por outro lado, imagine por um momento que uma variável de um tipo básico qualquer contém um endereço de uma posição de memória que, por sua vez, também contém um endereço de uma outra posição de memória. Então, essa variável é definida como um **apontador para um apontador**, isto é, um apontador para um compartimento de memória que contém um apontador. Apontadores para apontadores têm diversas aplicações na linguagem C, especialmente no uso de matrizes, como veremos adiante.

O conceito de indireção dupla é então introduzido neste caso: uma variável que contém um endereço de uma posição de memória, isto é, um apontador, que, por sua vez, contém um endereço para uma outra posição de memória, ou seja, um outro apontador. Certamente, podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla, etc. No entanto, a compreensão de um programa fica gradualmente mais difícil à medida que indireções múltiplas vão sendo utilizadas. Entender bem um programa com apontadores para apontadores, ou apontadores para apontadores para apontadores, e assim por diante, é bastante complicado e devemos usar esses recursos de forma criteriosa.

Considere agora o programa 55.3.

Inicialmente, o programa 55.3 faz a declaração de seis variáveis do tipo inteiro: `x` e `y`, que armazenam valores desse tipo; `apt1` e `apt2`, que são apontadores; e `aptapt1` e `aptapt2` que são apontadores para apontadores. O símbolo `**` antes do identificador das variáveis `aptapt1` e `aptapt2` significa que as variáveis são apontadores para apontadores, ou seja, podem armazenar um endereço onde se encontra um outro endereço onde, por sua vez, encontra-se um valor, um número inteiro nesse caso.

Após essas declarações, o programa segue com atribuições de valores às variáveis `x` e `y` e com atribuições dos endereços das variáveis `x` e `y` para os apontadores `apt1` e `apt2`, respectivamente, como já vimos em outros exemplos.

Note então que as duas atribuições abaixo:

```
aptapt1 = &apt1;
aptapt2 = &apt2;
```

Programa 55.3: Um exemplo de indireção dupla.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int x, y, *apt1, *apt2, **aptapt1, **aptapt2;
5    x = 1;
6    y = 4;
7    printf("x=%d  y=%d\n", x, y);
8    apt1 = &x;
9    apt2 = &y;
10   printf("*apt1=%d  *apt2=%d\n", *apt1, *apt2);
11   aptapt1 = &apt1;
12   aptapt2 = &apt2;
13   printf("**aptapt1=%d  **aptapt2=%d\n", **aptapt1, **aptapt2);
14   return 0;
15 }
```

fazem das variáveis `aptapt1` e `aptapt2` apontadores para apontadores. Ou seja, `aptapt1` contém o endereço da variável `apt1` e `aptapt2` contém o endereço da variável `apt2`. Por sua vez, a variável `apt1` contém o endereço da variável `x` e a variável `apt2` contém o endereço da variável `y`, o que caracteriza as variáveis `aptapt1` e `aptapt2` como apontadores para apontadores.

Observe finalmente que para acessar o conteúdo do endereço apontado pelo endereço apontado por `aptapt1` temos de usar o símbolo de indireção dupla `**`, como pode ser verificado na última chamada à função `printf` do programa.

Veja a figura 55.1 para um exemplo esquemático do que ocorre na memória quando apontadores para apontadores são usados em um programa.

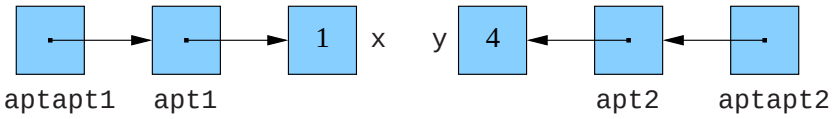



Figura 55.1: Exemplo esquemático de indireção dupla.

## 55.3 Apontadores para funções

Como vimos até aqui, apontadores podem conter endereços de variáveis de tipos básicos, de elementos de vetores e matrizes, de campos de registros ou de registros inteiros. Um apontador pode ainda ser usado para alocação dinâmica de memória. No entanto, a linguagem C não requer que apontadores contenham endereços apenas de dados. É possível ter em um programa apontadores para funções, já que as funções ocupam posições de memória e, por isso, possuem um endereço na memória, assim como todas as variáveis.

Podemos usar apontadores para funções assim como usamos apontadores para variáveis. Em particular, passar um apontador para uma função como um argumento de outra função é bastante comum na linguagem C. Suponha que estamos escrevendo a função `integral` que integra uma função matemática `f` entre os pontos `a` e `b`. Gostaríamos de fazer a função `integral` tão geral quanto possível, passando a função `f` como um argumento seu. Isso é possível na linguagem C pela definição de `f` como um apontador para uma função. Considerando que queremos integrar funções que têm um parâmetro do tipo `double` e que devolvem um valor do tipo `double`, uma possível interface da função `integral` é apresentada a seguir:

```
double integral(double (*f)(double), double a, double b)
```

Os parênteses em torno de `*f` indicam que `f` é um apontador para uma função, não uma função que devolve um apontador. Também é permitido definir `f` como se fosse uma função:

```
double integral(double f(double), double a, double b)
```

Do ponto de vista do compilador, as interfaces acima são idênticas.

Quando chamamos a função `integral` devemos fornecer um nome de uma função como primeiro argumento. Por exemplo, a chamada a seguir integra a função seno de $0$ a $\pi/2$:

```
result = integral(sin, 0.0, PI / 2);
```

O argumento `sin` e o nome da função seno incluída em `math.h`.

Observe que não há parênteses após `sin`. Quando o nome de uma função não é seguido por parênteses, o compilador produz um apontador para a função em vez de gerar código para uma chamada da função. No exemplo acima, não há uma chamada à função `sin`. Ao invés disso, estamos passando para a função `integral` um apontador para a função `sin`. Podemos pensar em apontadores para funções como pensamos com apontadores para vetores e matrizes. Relembrando, se, por exemplo, `v` é o identificador de um vetor, então `v[i]` representa um elemento do vetor enquanto que `v` representa um apontador para o vetor, ou melhor, para o primeiro elemento do vetor. Da forma similar, se `f` é o identificador de uma função, a linguagem C trata `f(x)` como uma chamada da função, mas trata `f` como um apontador para a função.

Dentro do corpo da função `integral` podemos chamar a função apontada por `f` da seguinte forma:

```
y = (*f)(x);
```

Nessa chamada, `*f` representa a função apontada por `f` e `x` é o argumento dessa chamada. Assim, durante a execução da chamada `integral(sin, 0.0, PI / 2)`, cada chamada de `*f` é, na verdade, uma chamada de `sin`.

Também podemos armazenar apontadores para funções em variáveis ou usá-los como elementos de um vetor, de uma matriz, de um campo de um registro. Podemos ainda escrever funções que devolvem apontadores para funções. Como um exemplo, declaramos abaixo uma variável que pode armazenar um apontador para uma função:

```
void (*aptf)(int);
```

O apontador `aptf` pode apontar para qualquer função que tenha um único parâmetro do tipo `int` e que devolva um valor do tipo `void`. Se `f` é uma função com essas características, podemos fazer `aptf` apontar para `f` da seguinte forma:

```
aptf = f;
```

Observe que não há operador de endereçamento antes de `f`. Uma vez que `pf` aponta para `f`, podemos chamar `f` indiretamente através de `aptf` como a seguir:

```
(*aptf)(i);
```

O programa 55.4 imprime uma tabela mostrando os valores das funções seno, cosseno e tangente no intervalo e incremento escolhidos pelo(a) usuário(a). O programa usa as funções `sin`, `cos` e `tan` de `math.h`.

Programa 55.4: Um exemplo de apontador para função.

```
#include <math.h>
#include <stdio.h>
void tabela(double (*f)(double), double a, double b, double incr)
{
  int i, num_intervalos;
  double x;
  num_intervalos = ceil((b - a) / incr);
  for (i = 0; i <= num_intervalos; i++) {
    x = a + i * incr;
    printf("%11.6f %11.6f\n", x, (*f)(x));
  }
}
int main(void)
{
  double inicio, fim, incremento;
  printf("Informe um intervalo [a, b]: ");
  scanf("%lf%lf", &inicio, &fim);
  printf("Informe o incremento: ");
  scanf("%lf", &incremento);
  printf("\n      x         cos(x)"
         "\n  ---------   ---------\n");
  tabela(cos, inicio, fim, incremento);
  printf("\n      x         sen(x)"
         "\n  ---------   ---------\n");
  tabela(sin, inicio, fim, incremento);
  printf("\n      x         tan(x)"
         "\n  ---------   ---------\n");
  tabela(tan, inicio, fim, incremento);
  return 0;
}
```

## Exercícios

55.1 Refaça o exercício 21.1 usando alocação dinâmica de memória.

55.2 Refaça o exercício 22.2 usando alocação dinâmica de memória.

55.3 Refaça o exercício 23.2 usando alocação dinâmica de memória.

55.4 Refaça o exercício 24.3 usando alocação dinâmica de memória.

55.5 Refaça o exercício 26.1 usando alocação dinâmica de memória.

55.6 Simule a execução do programa a seguir.

```
#include <stdio.h>
int f1(int (*f)(int))
{
  int n = 0;
  while ((*f)(n))
    n++;
  return n;
}
int f2(int i)
{
  return i * i + i - 12;
}
int main(void)
{
  printf("Resposta: %d\n", f1(f2));
  return 0;
}
```

55.7 Escreva uma função com a seguinte interface:

```
int soma(int (*f)(int), int inicio, int fim)
```

Uma chamada `soma(g, i, j)` deve devolver `g(i) + ... + g(j)`.

AULA 56

# ARQUIVOS

Nos programas que fizemos até aqui, a entrada e a saída sempre ocorreram em uma janela de terminal ou console do sistema operacional. Na linguagem C, não existem palavras-chaves definidas para tratamento de operações de entrada e saída. Essas tarefas são realizadas através de funções. Pouco usamos outras funções de entrada e saída da linguagem C além das funções `scanf` e `printf`, localizadas na biblioteca padrão de entrada e saída, com arquivo-cabeçalho `stdio.h`. Nesta aula, aprenderemos funções que realizam tarefas de entrada e saída em arquivos.

## 56.1 Seqüências de caracteres

Na linguagem C, o termo **seqüência de caracteres**, do inglês *stream*, significa qualquer fonte de entrada ou qualquer destinação para saída. Os programas que produzimos até aqui sempre obtiveram toda sua entrada a partir de uma seqüência de caracteres, em geral associada ao teclado, e escreveram sua saída em outra seqüência de caracteres, associada com o monitor.

Programas maiores podem necessitar de seqüências de caracteres adicionais, associadas a arquivos armazenados em uma variedade de meios físicos tais como discos e memórias ou ainda portas de rede e impressoras. As funções de entrada e saída mantidas em `stdio.h` trabalham do mesmo modo com todas as seqüências de caracteres, mesmo aquelas que não representam arquivos físicos.

O acesso a uma seqüência de caracteres na linguagem C se dá através de um **apontador de arquivo**, cujo tipo é `FILE *`, declarado em `stdio.h`. Algumas seqüências de caracteres são representadas por apontadores de arquivos com nomes padronizados. Podemos ainda declarar apontadores de arquivos conforme nossas necessidades, como fazemos abaixo:

```
FILE *apt1, *apt2;
```

Veremos mais sobre arquivos do sistema na seção 56.3.5.

A biblioteca representada pelo arquivo-cabeçalho `stdio.h` suporta dois tipos de arquivos: texto e binário. Os bytes em um **arquivo-texto** representam caracteres, fazendo que seja possível examinar e editar o seu conteúdo. O arquivo-fonte de um programa na linguagem C, por exemplo, é armazenado em um arquivo-texto. Por outro lado, os bytes em um **arquivo-binário** não representam necessariamente caracteres. Grupos de bytes podem representar outros tipos

de dados tais como inteiros e números com ponto flutuante. Um programa executável, por exemplo, é armazenado em um arquivo-binário.

Arquivos-texto possuem duas principais características que os diferem dos arquivos-binários: são divididos em linhas e podem conter um marcador especial de fim de arquivo. Cada linha do arquivo-texto normalmente termina com um ou dois caracteres especiais, dependendo do sistema operacional.

## 56.2 Redirecionamento de entrada e saída

Como já fizemos nos trabalhos da disciplina e em algumas aulas, a leitura e escrita em arquivos podem ser facilmente executadas nos sistemas operacionais em geral. Como percebemos, nenhum comando especial teve de ser adicionado aos nossos programas na linguagem C para que a leitura e a escrita fossem executadas de/para arquivos. O que fizemos até aqui foi redirecionar a entrada e/ou a saída de dados do programa. Como um exemplo simples, vejamos o programa 56.1, uma solução para o exercício 12.2, onde um número inteiro na base decimal é fornecido como entrada e na saída é apresentado o mesmo número na base binária.

Programa 56.1: Uma solução para o exercício 12.2.

```
1  #include <stdio.h>
2  int main(void)
3  {
4    int pot10, numdec, numbin;
5    scanf("%d", &numdec);
6    pot10 = 1;
7    numbin = 0;
8    while (numdec > 0) {
9      numbin = numbin + (numdec % 2) * pot10;
10     numdec = numdec / 2;
11     pot10 = pot10 * 10;
12   }
13   printf("%d\n", numbin);
14   return 0;
15 }
```

Supondo que o programa 56.1 tenha sido armazenado no arquivo-fonte `decbin.c` e o programa executável equivalente tenha sido criado após a compilação com o nome `decbin`, então, se queremos que a saída do programa executável `decbin` seja armazenada no arquivo `resultado`, então podemos digitar em uma linha de terminal o seguinte:

```
prompt$ ./decbin > resultado
```

Esse comando instrui o sistema operacional a executar o programa `decbin` redirecionando a saída, que normalmente seria apresentada no terminal, para um arquivo com nome `resultado`. Dessa forma, qualquer informação a ser apresentada por uma função de saída, como `printf`, não será mostrada no terminal, mas será escrita no arquivo `resultado`.

Por outro lado, podemos redirecionar a entrada de um programa executável, de tal forma que chamadas a funções que realizam entrada de dados, não mais solicitem essas informações ao usuário a partir do terminal, mas as obtenha a partir de um arquivo. Por exemplo, o programa 56.1 usa a função `scanf` para ler um número inteiro a partir de um terminal. Podemos redirecionar a entrada do programa `decbin` quando está sendo executado, fazendo que essa entrada seja realizada a partir de um arquivo. Por exemplo, se temos um arquivo com nome `numero` que contém um número inteiro, podemos digitar o seguinte em uma linha de terminal:

```
prompt$ ./decbin < numero
```

Com esse redirecionamento, o programa 56.1, que solicita um número a ser informado pelo usuário, não espera até que um número seja digitado. Ao contrário, pelo redirecionamento, a entrada do programa é tomada do arquivo `numero`. Ou seja, a chamada à função `scanf` tem o efeito de ler um valor do arquivo `numero` e não do terminal, embora a função `scanf` não "saiba" disso.

Podemos redirecionar a entrada e a saída de um programa simultaneamente da seguinte maneira:

```
prompt$ ./decbin < numero > resultado
```

Esse comando faz com que o programa `decbin` seja executado tomando a entrada de dados a partir do arquivo `resultado` e escrevendo a saída no arquivo `resultado`.

O redirecionamento de entrada e saída é uma ferramenta útil, já que, podemos manter um arquivo de entradas e realizar diversos testes sobre um programa executável a partir desse arquivo de entradas. Além disso, se temos, por exemplo, um arquivo alvo que contém soluções correspondentes às entradas, podemos comparar esse arquivo de soluções com as saídas de nosso programa, que também podem ser armazenadas em um arquivo.

## 56.3 Funções de entrada e saída da linguagem C

Até esta aula, excluindo o redirecionamento de entrada e saída, tínhamos sempre armazenado quaisquer informações na memória principal do nosso sistema computacional. Como já mencionado, uma grande quantidade de problemas pode ser resolvida com as operações de entrada e saída que conhecemos e com o redirecionamento de entrada e saída que acabamos de aprender. Entretanto, existem problemas onde há necessidade de obter, ou armazenar, dados

de/para dois ou mais arquivos. Os arquivos são mantidos em um dispositivo do sistema computacional conhecido como memória secundária, que pode ser implementado como um disco rígido, um disquete, um disco compacto (CD), um disco versátil digital (DVD), um cartão de memória, um disco amovível (*USB flash memory*), entre outros. A linguagem C tem um conjunto de funções específicas para tratamento de arquivos que se localiza na biblioteca padrão de entrada e saída, cujo arquivo-cabeçalho é `stdio.h`.

### 56.3.1 Funções de abertura e fechamento

Para que se possa realizar qualquer operação sobre um arquivo é necessário, antes de tudo, de **abrir** esse arquivo. Como um programa pode ter de trabalhar com diversos arquivos, então todos eles deverão estar abertos durante a execução desse programa. Dessa forma, há necessidade de identificação de cada arquivo, o que é implementado na linguagem C com o uso de um **apontador para arquivo**.

A função `fopen` da biblioteca padrão da linguagem C é uma função que realiza a abertura de um arquivo no sistema. A função recebe como parâmetros duas cadeias de caracteres: a primeira é o nome do arquivo a ser aberto e a segunda determina o modo no qual o arquivo será aberto. O nome do arquivo deve constar no sistema de arquivos do sistema operacional. A função `fopen` devolve um apontador único para o arquivo que pode ser usado para identificar esse arquivo a partir desse ponto do programa. Esse aponador é posicionado no início ou no final do arquivo, dependendo do modo como o arquivo foi aberto. Se o arquivo não puder ser aberto por algum motivo, a função devolve o valor `NULL`. Um apontador para um arquivo deve ser declarado com um tipo pré-definido `FILE`, também incluso no arquivo-cabeçalho `stdio.h`. A interface da função `fopen` é então descrita da seguinte forma:

```
FILE *fopen(char *nome, char *modo)
```

As opções para a cadeia de caracteres `modo` são descritas na tabela a seguir:

| modo | Descrição |
|---|---|
| **r** | modo de leitura de texto |
| **w** | modo de escrita de texto† |
| **a** | modo de adicionar texto‡ |
| **r+** | modo de leitura e escrita de texto |
| **w+** | modo de leitura e escrita de texto† |
| **a+** | modo de leitura e escrita de texto‡ |
| **rb** | modo de leitura em binário |
| **wb** | modo de escrita em binário† |
| **ab** | modo de adicionar em binário‡ |
| **r+b** ou **rb+** | modo de leitura e escrita em binário |
| **w+b** ou **wb+** | modo de leitura e escrita em binário† |
| **a+b** ou **ab+** | modo de leitura e escrita em binário‡ |

onde:

† trunca o arquivo existente com tamanho 0 ou cria novo arquivo;

‡ abre ou cria o arquivo e posiciona o apontador no final do arquivo.

Algumas observações sobre os modos de abertura de arquivos se fazem necessárias. Primeiro, observe que se o arquivo não existe e é aberto com o modo de leitura ( `r` ) então a abertura falha. Também, se o arquivo é aberto com o modo de adicionar ( `a` ), então todas as operações de escrita ocorrem o final do arquivo, desconsiderando a posição atual do apontador do arquivo. Por fim, se o arquivo é aberto no modo de atualização ( `+` ) então a operação de escrita não pode ser imediatamente seguida pela operação de leitura, e vice-versa, a menos que uma operação de reposicionamento do apontador do arquivo seja executada, tal como uma chamada a qualquer uma das funções `fseek` , `fsetpos` , `rewind` ou `fflush` .

Por exemplo, o comando de atribuição a seguir

```
aptarq = fopen("entrada", "r");
```

tem o efeito de abrir um arquivo com nome `entrada` no modo de leitura. A chamada à função `fopen` devolve um identificador para o arquivo aberto que é atribuído ao apontador `aptarq` do tipo `FILE` . Então, esse apontador é posicionado no primeiro caracter do arquivo. A declaração prévia do apontador `aptarq` deve ser feita da seguinte forma:

```
FILE *aptarq;
```

A função `fclose` faz o oposto que a função `fopen` faz, ou seja, informa o sistema que o programador não necessita mais usar o arquivo. Quando um arquivo é fechado, o sistema realiza algumas tarefas importantes, especialmente a escrita de quaisquer dados que o sistema possa ter mantido na memória principal para o arquivo na memória secundária, e então dissocia o identificador do arquivo. Depois de fechado, não podemos realizar tarefas de leitura ou escrita no arquivo, a menos que seja reaberto. A função `fclose` tem a seguinte interface:

```
int fclose(FILE *aptarq)
```

Se a operação de fechamento do arquivo apontado por `aptarq` obtém sucesso, a função `fclose` devolve o valor 0 (zero). Caso contrário, o valor `EOF` é devolvido.

### 56.3.2 Funções de entrada e saída

A função `fgetc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja lido de um arquivo. A interface dessa função é apresentada a seguir:

```
int fgetc(FILE *aptarq)
```

A função `fgetc` lê o próximo caracter do arquivo apontado por `aptarq`, avançando esse apontador em uma posição. Se a leitura é realizada com sucesso, o caracter lido é devolvido pela função. Note, no entanto, que a função, ao invés de especificar o valor de devolução como sendo do tipo `unsigned char`, especifica-o como sendo do tipo `int`. Isso se deve ao fato de que a leitura pode falhar e, nesse caso, o valor devolvido é o valor armazenado na constante simbólica `EOF`, definida no arquivo-cabeçalho `stdio.h`. O valor correspondente à constante simbólica `EOF` é obviamente um valor diferente do valor de qualquer caracter e, portanto, um valor negativo. Do mesmo modo, se o fim do arquivo é encontrado, a função `fgetc` também devolve `EOF`.

A função `fputc` da biblioteca padrão de entrada e saída da linguagem C permite que um único caracter seja escrito em um arquivo. A interface dessa função é apresentada a seguir:

```
int fputc(int caracter, FILE *aptarq)
```

Se a função `fputc` tem sucesso, o apontador `aptarq` é incrementado e o caracter escrito é devolvido. Caso contrário, isto é, se ocorre um erro, o valor `EOF` é devolvido.

Existe outro par de funções de leitura e escrita em arquivos com identificadores `fscanf` e `fprintf`. As interfaces dessas funções são apresentadas a seguir:

```
int fscanf(FILE *aptarq, char *formato, ...)
```

e

```
int fprintf(FILE *aptarq, char *formato, ...)
```

Essas duas funções são semelhantes às respectivas funções `scanf` e `printf` que conhecemos bem, a menos de um parâmetro a mais que é informado, justamente o primeiro, que é o apontador para o arquivo que ser quer realizar as operações de entrada e saída formatadas. Dessa forma, o exemplo de chamada a seguir

```
fprintf(aptaqr, "O número %d é primo.\n", numero);
```

realiza a escrita no arquivo apontado por `aptarq` da mensagem entre aspas duplas, substituindo o valor numérico correspondente armazenado na variável `numero`. O número de caracteres escritos no arquivo é devolvido pela função `fprintf`. Se um erro ocorrer, então o valor `-1` é devolvido.

Do mesmo modo,

```
fscanf(aptaqr, "%d", &numero);
```

realiza a leitura de um valor que será armazenado na variável `numero` a partir de um arquivo identificado pelo apontador `aptarq`. Se a leitura for realizada com sucesso, o número de valores lidos pela função `fscanf` é devolvido. Caso contrário, isto é, se houver falha na leitura, o valor `EOF` é devolvido.

Há outras funções para entrada e saída de dados a partir de arquivos, como as funções `fread` e `fwrite`, que não serão cobertas nesta aula. O leitor interessado deve procurar as referências bibliográficas do curso.

### 56.3.3 Funções de controle

Existem diversas funções de controle que dão suporte a operações sobre os arquivos. Dentre as mais usadas, listamos uma função que descarrega o espaço de armazenamento temporário da memória principal para a memória secundária e as funções que tratam do posicionamento do apontador do arquivo.

A função `fflush` faz a descarga de qualquer informação associada ao arquivo que esteja armazenada na memória principal para o dispositivo de memória secundária associado. A função `fflush` tem a seguinte interface:

```
int fflush(FILE *aptarq)
```

onde `aptarq` é o apontador para um arquivo. Se o apontador `aptarq` contém um valor nulo, então todos espaços de armazenamento temporários na memória principal de todos os arquivos abertos são descarregados nos dispositivos de memória secundária. Se a descarga é realizada com sucesso, a função devolve o valor 0 (zero). Caso contrário, a função devolve o valor `EOF`.

Sobre as funções que tratam do posicionamento do apontador de um arquivo, existe uma função específica da biblioteca padrão da linguagem C que realiza um teste de final de arquivo. Essa função tem identificador `feof` e a seguinte interface:

```
int feof(FILE *aptarq)
```

O argumento da função `feof` é um apontador para um arquivo do tipo `FILE`. A função devolve um valor inteiro diferente de 0 (zero) se o apontador `aptarq` está posicionado no final do arquivo. Caso contrário, a função devolve o valor 0 (zero).

A função `fgetpos` determina a posição atual do apontador do arquivo e tem a seguinte interface:

```
int fgetpos(FILE *aptarq, fpos_t *pos)
```

onde `aptarq` é o apontador associado a um arquivo e `pos` é uma variável que, após a execução dessa função, conterá o valor da posição atual do apontador do arquivo. Observe que `fpos_t` é um novo tipo de dado, definido no arquivo-cabeçalho `stdio.h`, adequado para armazenamento de uma posição qualquer de um arquivo. Se a obtenção dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `fsetpos` posiciona o apontador de um arquivo em alguma posição escolhida e tem a seguinte interface:

```
int fsetpos(FILE *aptarq, fpos_t *pos)
```

onde `aptarq` é o apontador para um arquivo e `pos` é uma variável que contém a posição para onde o apontador do arquivo será deslocada. Se a determinação dessa posição for realizada com sucesso, a função devolve 0 (zero). Caso contrário, a função devolve um valor diferente de 0 (zero).

A função `ftell` determina a posição atual de um apontador em um dado arquivo. Sua interface é apresentada a seguir:

```
long int ftell(FILE *aptarq)
```

A função `ftell` devolve a posição atual no arquivo apontado por `aptarq`. Se o arquivo é binário, então o valor é o número de *bytes* a partir do início do arquivo. Se o arquivo é de texto, então esse valor pode ser usado pela função `fseek`, como veremos a seguir. Se há sucesso na sua execução, a função devolve a posição atual no arquivo. Caso contrário, a função devolve o valor `-1L`.

A função `fseek` posiciona o apontador de um arquivo para uma posição determinada por um deslocamento. A interface da função é dada a seguir:

```
int fseek(FILE *aptarq, long int desloca, int apartir)
```

O argumento `aptarq` é o apontador para um arquivo. O argumento `desloca` é o número de *bytes* a serem saltados a partir do conteúdo do argumento `apartir`. Esse conteúdo pode ser um dos seguintes valores pré-definidos no arquivo-cabeçalho `stdio.h`:

| | |
|---|---|
| **SEEK_SET** | A partir do início do arquivo |
| **SEEK_CUR** | A partir da posição atual |
| **SEEK_END** | A partir do fim do arquivo |

Em um arquivo de texto, o contéudo de `apartir` dever ser `SEEK_SET` e o conteúdo de `desloca` deve ser 0 (zero) ou um valor devolvido pela função `ftell`. Se a função é executada com sucesso, o valor 0 (zero) é devolvido. Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rewind` faz com que o apontador de um arquivo seja posicionado para o início desse arquivo. A interface dessa função é a seguinte:

```
void rewind(FILE *aptarq)
```

### 56.3.4 Funções sobre arquivos

Há funções na linguagem C que permitem que um programador remova um arquivo do disco ou troque o nome de um arquivo. A função `remove` tem a seguinte interface:

```
int remove(char *nome)
```

A função `remove` elimina um arquivo, com nome armazenado na cadeia de caracteres `nome`, do sistema de arquivos do sistema computacional. O arquivo não deve estar aberto no programa. Se a remoção é realizada, a função devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido.

A função `rename` tem a seguinte interface:

```
int rename(char *antigo, char *novo)
```

A função `rename` faz com que o arquivo com nome armazenado na cadeia de caracteres `antigo` tenha seu nome trocado pelo nome armazenado na cadeia de caracteres `novo`. Se a função realiza a tarefa de troca de nome, então `rename` devolve o valor 0 (zero). Caso contrário, um valor diferente de 0 (zero) é devolvido e o arquivo ainda pode ser identificado por seu nome antigo.

### 56.3.5 Arquivos do sistema

Sempre que um programa na linguagem C é executado, três arquivos ou seqüências de caracteres são automaticamente abertos pelo sistema, identificados pelos apontadores `stdin`, `stdout` e `stderr`, definidos no arquivo-cabeçalho `stdio.h` da biblioteca padrão de entrada e saída. Em geral, `stdin` está associado ao teclado e `stdout` e `stderr` estão associados ao monitor. Esses apontadores são todos do tipo `FILE`.

O apontador `stdin` identifica a entrada padrão do programa e é normalmente associado ao teclado. Todas as funções de entrada definidas na linguagem C que executam entrada de dados e não têm um apontador do tipo `FILE` como um argumento tomam a entrada a partir do arquivo apontado por `stdin`. Assim, ambas as chamadas a seguir:

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

e

```
fscanf(stdin, "%d", &numero);
```

são equivalentes e lêem um número do tipo inteiro da entrada padrão, que é normalmente o terminal.

Do mesmo modo, o apontador `stdout` se refere à saída padrão, que também é associada ao terminal. Assim, as chamadas a seguir:

```
printf("Programar é bacana!\n");
```

e

```
fprintf(stdout, "Programa é bacana!\n");
```

são equivalentes e imprimem a mensagem acima entre as aspas duplas na saída padrão, que é normalmente o terminal.

O apontador `stderr` se refere ao arquivo padrão de erro, onde muitas das mensagens de erro produzidas pelo sistema são armazenadas e também é normalmente associado ao terminal. Uma justificativa para existência de tal arquivo é, por exemplo, quando as saídas todas do programa são direcionadas para um arquivo. Assim, as saídas do programa são escritas em um arquivo e as mensagens de erro são escritas na saída padrão, isto é, no terminal. Ainda há a possibilidade de escrever nossas próprias mensagens de erro no arquivo apontado por `stderr`.

## 56.4 Exemplos

Nessa seção apresentaremos dois exemplos que usam algumas das funções de entrada e saída em arquivos que aprendemos nesta aula.

O primeiro exemplo, apresentando no programa 56.2, é bem simples e realiza a cópia do conteúdo de um arquivo em outro arquivo.

O segundo exemplo, apresentado no programa 56.3, mescla o conteúdo de dois arquivos texto em um arquivo resultante. Neste exemplo, cada par de palavras do arquivo resultante é composto por uma palavra de um arquivo e uma palavra do outro arquivo. Se um dos arquivos contiver menos palavras que o outro arquivo, esse outro arquivo é descarregado no arquivo resultante.

Programa 56.2: Um exemplo de cópia de um arquivo.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  char nomebase[MAX+1], nomecopia[MAX+1];
  int c;
  FILE *aptbase, *aptcopia;
  printf("Informe o nome do arquivo a ser copiado: ");
  scanf("%s", nomebase);
  printf("Informe o nome do arquivo resultante: ");
  scanf("%s", nomecopia);
  aptbase = fopen(nomebase, "r");
  if (aptbase != NULL) {
    aptcopia = fopen(nomecopia, "w");
    if (aptcopia != NULL) {
      c = fgetc(aptbase);
      while (c != EOF) {
        fputc(c, aptcopia);
        c = fgetc(aptbase);
      }
      fclose(aptbase);
      fclose(aptcopia);
      printf("Arquivo copiado.\n");
    }
    else
      printf("Não é possível abrir o arquivo %s para escrita.\n", nomecopia);
  }
  else
    printf("Não é possível abrir o arquivo %s para leitura.\n", nomebase);
  return 0;
}
```

Programa 56.3: Um segundo exemplo de uso de arquivos.

```
#include <stdio.h>
#define MAX 100
int main(void)
{
  char nomeum[MAX+1], nomedois[MAX+1], palavra1[MAX+1], palavra2[MAX+1];
  int i1, i2;
  FILE *aptum, *aptdois, *aptresult;
  printf("Informe o nome do primeiro arquivo: ");
  scanf("%s", nomeum);
  printf("Informe o nome do segundo arquivo: ");
  scanf("%s", nomedois);
  aptum = fopen(nomeum, "r");
  aptdois = fopen(nomedois, "r");
  if (aptum != NULL && aptdois != NULL) {
    aptresult = fopen("mesclado", "w");
    if (aptresult != NULL) {
      i1 = fscanf(aptum, "%s", palavra1);
      i2 = fscanf(aptdois, "%s", palavra2);
      while (i1 != EOF && i2 != EOF) {
        fprintf(aptresult, "%s %s ", palavra1, palavra2);
        i1 = fscanf(aptum, "%s", palavra1);
        i2 = fscanf(aptdois, "%s", palavra2);
      }
      while (i1 != EOF) {
        fprintf(aptresult, "%s ", palavra1);
        i1 = fscanf(aptum, "%s", palavra1);
      }
      while (i2 != EOF) {
        fprintf(aptresult, "%s ", palavra2);
        i2 = fscanf(aptdois, "%s", palavra2);
      }
      fprintf(aptresult, "\n");
      fclose(aptum);
      fclose(aptdois);
      fclose(aptresult);
      printf("Arquivo mesclado.\n");
    }
    else
      printf("Não é possível abrir um arquivo para escrita.\n");
  }
  else
    printf("Não é possível abrir os arquivos para leitura.\n");
  return 0;
}
```

## Exercícios

56.1 Escreva um programa que leia o conteúdo de um arquivo cujo nome é fornecido pelo usuário e copie seu conteúdo em um outro arquivo, trocando todas as letras minúsculas por letras maiúsculas.

56.2 Suponha que temos dois arquivos cujas linhas são ordenadas lexicograficamente. Por exemplo, esses arquivos podem conter nomes de pessoas, linha a linha, em ordem alfabética. Escreva um programa que leia o conteúdo desses dois arquivos, cujos nomes são fornecidos pelo usuário, e crie um novo arquivo resultante contendo todas as linhas dos dois arquivos ordenadas lexicograficamente. Por exemplo, se os arquivos contêm as linhas

| **Arquivo 1** | **Arquivo 2** |
|---|---|
| Antônio | Carlos |
| Berenice | Célia |
| Diana | Fábio |
| Solange | Henrique |
| Sônia | |
| Zuleica | |

o arquivo resultante deve ser

Antônio
Berenice
Carlos
Célia
Diana
Fábio
Henrique
Solange
Sônia
Zuleica

AULA 57

# LISTAS LINEARES

Listas lineares são as primeiras estruturas de dados que aprendemos. Essas estruturas são muito usadas em diversas aplicações importantes para organização de informações na memória tais como representações alternativas para expressões aritméticas, compartilhamento de espaço de memória, entre outras. Nesta aula, aprenderemos a definição dessa estrutura, sua implementação em alocação estática e suas operações básicas.

## 57.1 Definição

Informalmente, uma lista linear é uma estrutura de dados que armazena um conjunto de informações que são relacionadas entre si. Essa relação se expressa apenas pela ordem relativa entre os elementos. Por exemplo, nomes e telefones de uma agenda telefônica, as informações bancárias dos funcionários de uma empresa, etc, são informações que podem ser armazenadas em uma lista linear. Cada informação contida na lista é na verdade um registro contendo os dados relacionados. Em geral, usamos um desses dados como uma chave para realizar diversas operações sobre essa lista, tais como busca de um elemento, inserção, remoção, etc. Já que os dados acoplados à chave, também chamados de dados satélites, são irrelevantes e participam apenas das movimentações dos registros, podemos imaginar que uma lista linear é composta apenas pelas chaves desses registros e que essas chaves são representadas por números inteiros.

Agora formalmente, uma **lista linear** $L$ é um conjunto de $n \geqslant 0$ registros $l_1, l_2, \ldots, l_n$ determinada pela ordem relativa desses elementos:

(*i*) se $n > 0$ então $l_1$ é o primeiro registro;

(*ii*) o registro $l_i$ é precedido pelo registro $l_{i-1}$ para todo $i$, $1 < i \leqslant n$.

As operações básicas sobre uma lista linear são as seguintes:

- busca;
- inclusão; e
- remoção.

De acordo com a política de armazenamento das informações nos registros da lista, temos listas lineares especiais tais como pilhas e filas, como veremos adiante.

Dependendo da aplicação, muitas outras operações também podem ser realizadas sobre essa estrutura como, por exemplo, alteração de um elemento, combinação de duas listas, ordenação da lista de acordo com as chaves, determinação do elemento de menor ou maior chave, determinação do tamanho ou número de elementos da lista, etc.

As listas lineares podem ser armazenadas na memória de duas maneiras distintas:

**Alocação estática ou seqüencial:** os elementos são armazenados em posições consecutivas de memória, com uso da vetores;

**Alocação dinâmica ou encadeada:** os elementos podem ser armazenados em posições não consecutivas de memória, com uso de apontadores.

A aplicação, ou problema que queremos resolver, é que define o tipo de armazenamento a ser usado, dependendo das operações sobre a lista, do número de listas envolvidas e das características particulares das listas. Nas aulas 41 e 32 vimos especialmente a operação de busca em uma lista linear em alocação seqüencial.

Os registros de uma lista linear em alocação encadeada, também chamados de células, encontram-se dispostos em posições aleatórias da memória e são ligados por apontadores que indicam a posição do próximo elemento da lista. Assim, um campo é acrescentado a cada registro da lista indicando o endereço do próximo elemento da lista. Veja a figura 57.1 para um exemplo de um registro de uma lista.

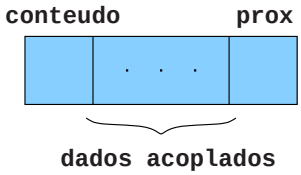



Figura 57.1: Representação de um registro de uma lista linear em alocação encadeada.

Os dados acoplados, ou dados satélites, são irrelevantes nas nossas aplicações e por isso não são considerados. A definição de um registro de nossa lista linear é então descrita como a seguir:

```
struct cel {
  int conteudo;
  struct cel *prox;
};
```

É conveniente tratar as células de uma lista linear em alocação encadeada como um tipo novo de dados, que chamaremos de `celula`:

```
typedef struct cel celula;
```

Uma célula `c` e um apontador `p` para uma célula podem ser declarados da seguinte forma:

```
celula c;
celula *p;
```

Se `c` é uma célula então `c.conteudo` é o conteúdo da célula e `c.prox` é o endereço da célula seguinte. Se `p` é o endereço de uma célula então `p->conteudo` é o conteúdo da célula apontada por `p` e `p->prox` é o endereço da célula seguinte. Se `p` é o endereço da última célula da lista então `p->prox` vale `NULL`.

Uma ilustração de uma lista linear em alocação encadeada é mostrada na figura 57.2.

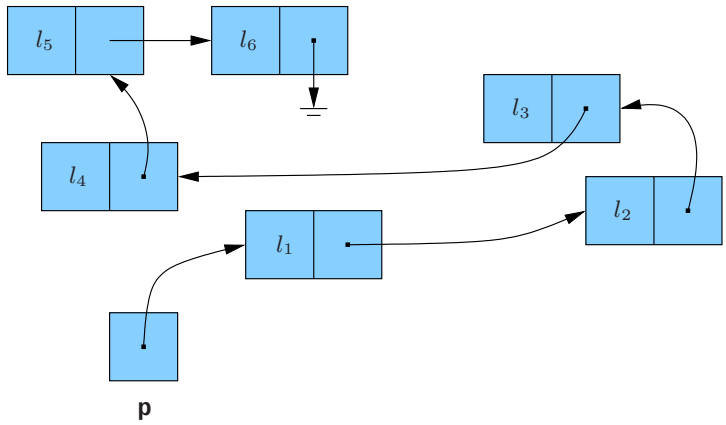



Figura 57.2: Representação de uma lista linear em alocação encadeada na memória.

Dizemos que o **endereço** de uma lista encadeada é o endereço de sua primeira célula. Se `p` é o endereço de uma lista, podemos dizer simplesmente "`p` é uma lista" e "considere a lista `p`". Reciprocamente, a expressão "`p` é uma lista" deve ser interpretada como "`p` é o endereço da primeira célula de uma lista".

Uma lista linear pode ser vista de duas maneiras diferentes, dependendo do papel que sua primeira célula representa. Em uma lista linear **com cabeça**, a primeira célula serve apenas para marcar o início da lista e portanto o seu conteúdo é irrelevante. A primeira célula é a **cabeça** da lista. Em uma lista linear **sem cabeça** o conteúdo da primeira célula é tão relevante quanto o das demais. Nas nossas aulas, trataremos das listas lineares sem cabeça, embora as listas lineares com cabeça sejam ligeiramente mais simples de manipular. Assim, sempre que nos referirmos a uma lista linear, na verdade estaremos nos referindo implicitamente a uma lista linear sem cabeça. Deixaremos as listas lineares com cabeças para os exercícios.

Uma lista linear está vazia se não tem célula alguma. Para criar uma lista vazia `lista` basta escrever as seguintes sentenças:

```
celula *lista;
lista = NULL;
```

Para imprimir o conteúdo de todas as células de uma lista linear `lista` podemos usar a seguinte função:

```
void imprime_lista(celula *lista)
{
  celula *p;
  for (p = lista; p != NULL; p = p->prox)
    printf("%d\n", p->conteudo);
}
```

A seguir vamos discutir e implementar as operações básicas de busca, inserção e remoção sobre listas lineares em alocação encadeada.

## 57.2 Busca

O processo de busca de um objeto $x$ em uma lista linear em alocação encadeada, isto é, de verificar se $x$ é igual ao conteúdo de alguma célula da lista, pode ser descrito como apresentamos abaixo. A função `buscaEnc` recebe uma lista linear apontada por `lista` e um número inteiro `x` e devolve o endereço de um registro contendo `x`, caso `x` ocorra em `lista`. Caso contrário, a função devolve `NULL`.

```
1  celula *buscaEnc(celula *lista, int x)
2  {
3    celula *p;
4    p = lista;
5    while (p != NULL && p->conteudo != x)
6      p = p->prox;
7    return p;
8  }
```

É interessante comparar a eficiência dessa função com a da função `busca` da aula 32, que realiza processamento equivalente, porém em uma lista linear em alocação seqüencial, isto é, em um vetor.

Uma chamada à função `buscaEnc` pode ser realizada como mostramos a seguir, supondo apontadores `p` e `lista` do tipo `celula` e `x` um número inteiro:

```
p = buscaEnc(lista, x);
```

## 57.3 Inserção

Observe que na alocação encadeada, os nós de uma lista linear se encontram dispostos em posições de memória não necessariamente contínuas. Com a realização de inserções e remoções, há necessidade de encontrar novas posições de memória para armazenamento de informações e liberar outras posições que possam ser reutilizadas posteriormente. Podemos gerenciar a memória na linguagem C usando funções que gerenciam seu espaço disponível. As funções `malloc` e `free`, da biblioteca `stdlib`, executam a tarefa de ocupação e liberação, respectivamente, de compartimentos da lista de espaços disponíveis da memória, como vimos na aula 55.

Uma função que insere um elemento `x` na lista linear `L` é apresentada a seguir. Se `x` já consta da lista, então nada fazemos. Caso contrário, inserimos `x` no início da lista.

```
1  void insereEnc(celula **L, int x)
2  {
3    celula *novo;
4    if (buscaEnc(*L, x) == NULL) {
5      novo = (celula *) malloc(sizeof(celula));
6      if (novo != NULL) {
7        novo->conteudo = x;
8        novo->prox = *L;
9        *L = novo;
10     }
11     else {
12       printf("Não há memória disponível!\n");
13       exit(EXIT_FAILURE);
14     }
15   }
16   else
17     printf("Valor já se encontra na lista!\n");
18 }
```

Uma chamada à função `insereEnc` pode ser feita como abaixo, para uma `lista` e um valor `x`:

```
insereEnc(&lista, x);
```

Observe ainda que as inserções são sempre realizadas no início da lista linear. Podemos modificar essa comportamento, por exemplo, mantendo um outro apontador para o final da lista linear e fazendo as inserções neste outro extremo. Deixamos essa idéia para ser implementada em um exercício.

## 57.4 Remoção

Para realizar a remoção de um registro da lista é necessário buscá-lo. A função `buscaEnc` descrita nesta aula devolve um apontador para o registro cujo conteúdo procurado se encontra, ou `NULL` caso esse valor não ocorra na lista. Um exemplo desta situação é apresentado na figura 57.3.

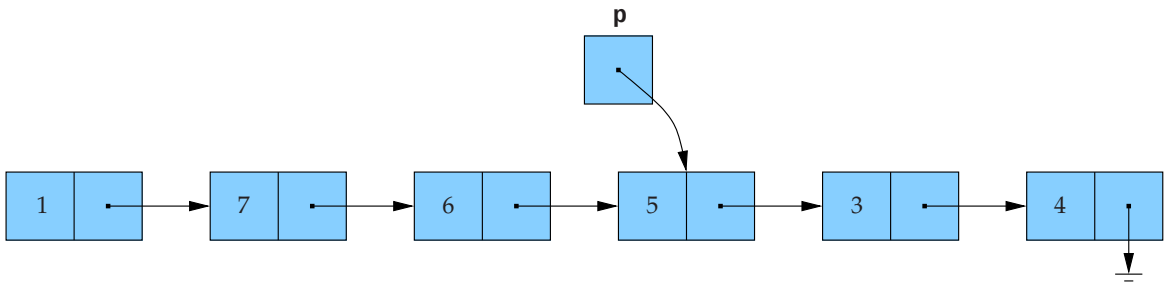



Figura 57.3: Busca em uma lista linear encadeada. O apontador é devolvido como resultado.

Depois de ter encontrado o registro que contém o elemento que queremos remover, o que devemos fazer? Infelizmente, esta função `buscaEnc` não devolve uma informação fundamental para que a remoção possa ser realizada imediatamente com sucesso: um apontador para o registro anterior ao registro que contém o valor que queremos remover. No exemplo da figura 57.3, precisamos de um apontador para o registro que contém o valor 6. Como só temos um apontador para o registro seguinte, não conseguimos realizar a remoção, a menos que façamos algum truque. Mesmo assim, ainda é possível realizar a remoção, mas vamos deixar idéias mais complicadas para um exercício. A idéia a ser apresentada aqui é modificar um pouco a função `buscaEnc` de modo que possa devolver dois apontadores: um para o registro que contém o valor buscado e outro para o registro imediatamente anterior a esse. A função `buscaEnc2` a seguir é resultado dessa idéia.

```
1  void buscaEnc2(celula *L, celula **apt, celula **ant, int x)
2  {
3    *ant = NULL;
4    *apt = L;
5    while (*apt != NULL && (*apt)->conteudo != x) {
6      *ant = *apt;
7      *apt = (*apt)->prox;
8    }
9  }
```

Observe que a função `buscaEnc2` devolve um apontador para o elemento procurado em `*apt` e um apontador para o elemento anterior ao procurado em `*ant`, se o elemento de fato se encontra na lista. Caso contrário, `*apt` conterá um apontador `NULL`, como antes.

Agora, com os apontadores `*apt` e `*ant`, podemos descrever a função `removeEnc`.

```
1  void removeEnc(celula **L, int x)
2  {
3    celula *apt, *ant;
4    buscaEnc2(*L, &apt, &ant, x);
5    if (apt != NULL) {
6      if (ant != NULL)
7        ant->prox = apt->prox;
8      else
9        *L = apt->prox;
10     free(p);
11   }
12   else
13     printf("Valor não encontrado\n");
14 }
```

Veja a figura 57.4 para um exemplo de remoção.

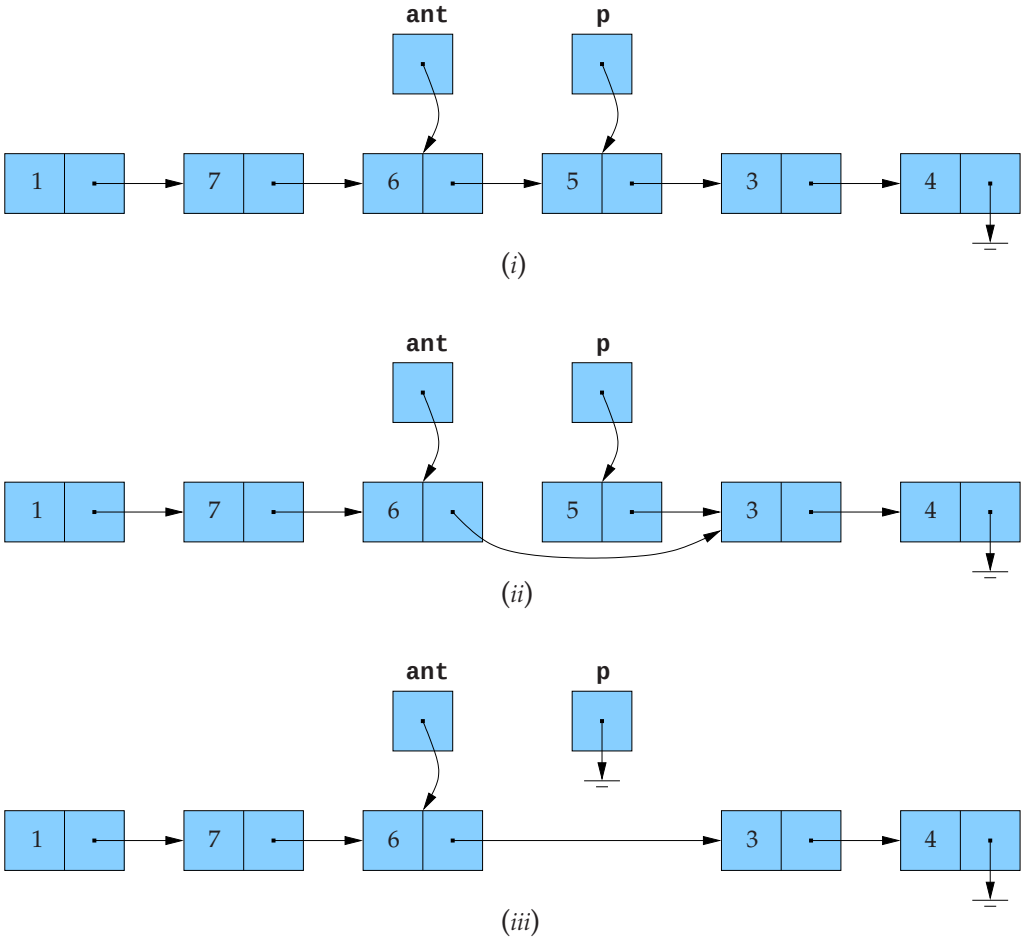



Figura 57.4: Remoção do valor 5 em uma lista linear encadeada. (*i*) Após a busca do valor. (*ii*) Modificação do apontador `ant`. (*iii*) Liberação da memória.

Uma chamada à função `removeEnc` é ilustrada abaixo, para uma `lista` e um valor `x`:

```
removeEnc(&lista, x);
```

Observe que os apontadores `apt` e `ant` são variáveis locais à função `removeEnc`, que auxiliam tanto na busca como na remoção propriamente. Além disso, se `ant == NULL` então o registro que contém `x`, que queremos remover, encontra-se na primeira posição da lista. Dessa forma, há necessidade de alterar o conteúdo do apontador `*L` para o registro seguinte desta lista.

## Exercícios

57.1 Escreva uma função `insereFim` que receba uma lista linear `*L`, um apontador `f` para o fim da lista e um valor `x` e realize a inserção desse valor no final da lista.

57.2 Se conhecemos apenas o apontador `apt` para um nó de uma lista linear em alocação encadeada, como na figura 57.5, e nada mais é conhecido, como podemos modificar a lista linear de modo que passe a conter apenas os valores $20, 4, 19, 47$, isto é, sem o conteúdo do nó apontado por `apt`?

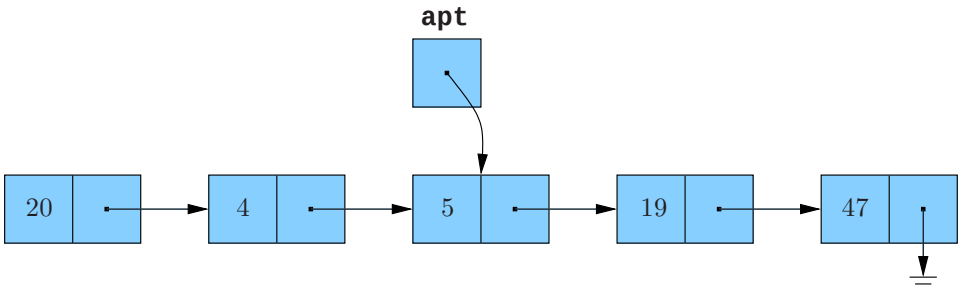



Figura 57.5: Uma lista linear encadeada com um apontador `apt`.

57.3 O esquema apresentado na figura 57.6 permite percorrer uma lista linear encadeada nos dois sentidos, usando apenas o campo `prox` que contém o endereço do próximo elemento da lista. Usamos dois apontadores `esq` e `dir`, que apontam para dois elementos vizinhos da lista. A idéia desse esquema é que à medida que os apontadores `esq` e `dir` caminham na lista, os campos `prox` são invertidos de maneira a permitir o tráfego nos dois sentidos.

Escreva funções para:

(a) mover `esq` e `dir` para a direita de uma posição.

(b) mover `esq` e `dir` para a esquerda de uma posição.

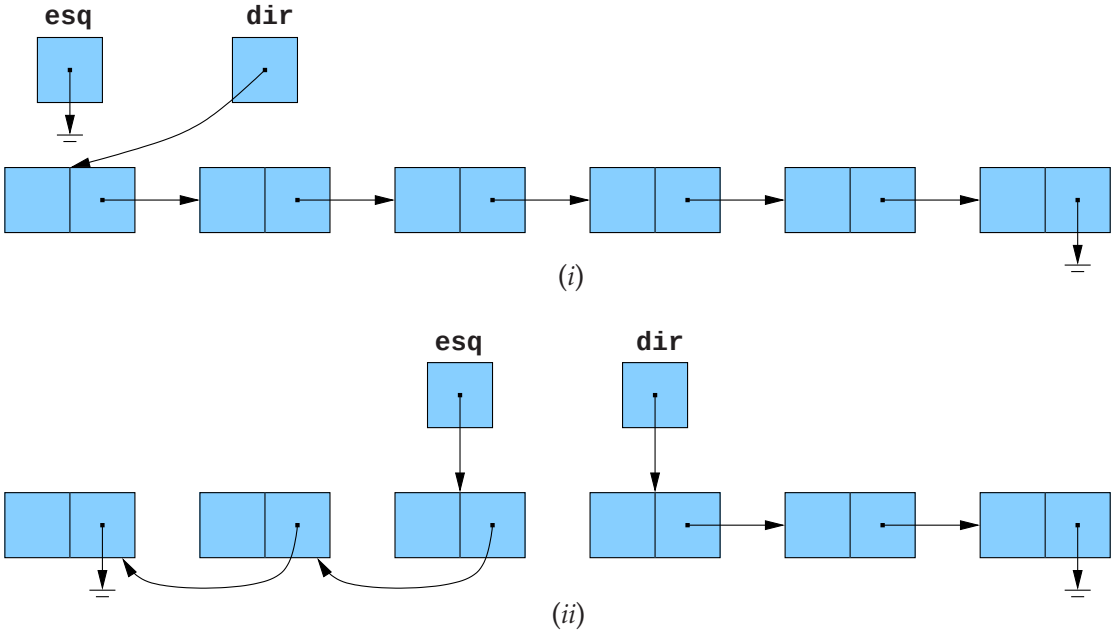



Figura 57.6: A figura (*ii*) foi obtida de (*i*) através da execução da função dada no exercício 57.4(a) por três vezes.

57.4 Uma lista linear encadeada com cabeça é tal que a primeira célula serve apenas para marcar o início da lista e portanto o seu conteúdo é irrelevante. A primeira célula é a cabeça da lista. Se `L` é o endereço da cabeça de lista então `L->prox` vale `NULL` se e somente se a lista está vazia. Para criar uma lista vazia deste tipo, devemos fazer o seguinte:

```
celula *L;
L = malloc(sizeof(celula));
L->prox = NULL;
```

Escreva funções de busca, inserção e remoção em uma lista linear com cabeça.

57.5 (a) Escreva uma função que copie um vetor para uma lista linear encadeada.

(b) Escreva uma função que copie uma lista linear encadeada em um vetor.

57.6 Escreva uma função que decida se duas listas dadas têm o mesmo conteúdo.

57.7 Escreva uma função que conte o número de células de uma lista linear encadeada.

57.8 Seja `lista` uma lista linear com seus conteúdos dispostos em ordem crescente. Escreva as funções `buscaEncOrd`, `insereEncOrd` e `removeEncOrd` para realização da busca, inserção e remoção, respectivamente, em uma lista linear com essas característica. As operações de inserção e remoção devem manter a lista em ordem crescente.

57.9 Sejam duas listas lineares `L1` e `L2`, com seus conteúdos dispostos em ordem crescente. Escreva uma função `concatena` que receba `L1` e `L2` e construa uma lista `R` resul-

tante da intercalação dessas duas listas, de tal forma que a lista construída também esteja ordenada. A função `concatena` deve destruir as listas `L1` e `L2` e deve devolver `R`.

57.10 Seja `L` uma lista linear composta por registros contendo os valores $l_1, l_2, \ldots l_n$, nessa ordem.

(a) Escreva uma função `roda1` que receba `*L` e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $l_2, l_3, \ldots, l_n, l_1$, nessa ordem.

(b) Escreva uma função `inverte` que receba `*L` e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $l_n, l_{n-1}, \ldots, l_2, l_1$, nessa ordem.

(c) Escreva uma função `soma` que receba `*L` e modifique e devolva essa lista de tal forma que a lista resultante contenha as chaves $l_1 + l_n, l_2 + l_{n-1}, \ldots, l_{n/2} + l_{n/2+1}$, nessa ordem. Considere $n$ par.

57.11 Sejam $S_1$ e $S_2$ dois conjuntos disjuntos de números inteiros. Suponha que $S_1$ e $S_2$ estão implementados em duas listas lineares em alocação encadeada `L1` e `L2`, respectivamente. Escreva uma função `uniao` que receba as listas `L1` e `L2` representando os conjuntos $S_1$ e $S_2$ e devolva uma lista resultante `R` que representa a união dos conjuntos, isto é, uma lista linear encadeada que representa o conjunto $S = S_1 \cup S_2$.

57.12 Seja um polinômio $p(x) = a_0x^n + a_1x^{n-1} + \ldots + a_n$, com coeficientes de ponto flutuante. Represente $p(x)$ adequadamente por uma lista linear encadeada e escreva as seguintes funções.

(a) Escreva uma função `pponto` que receba uma lista `p` e um número de ponto flutuante `x0` e calcule e devolva $p(x_0)$.

(b) Escreva uma função `psoma` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) + q(x)$.

(c) Escreva uma função `pprod` que receba as listas lineares `p` e `q`, que representam dois polinômios, e calcule e devolva o polinômio resultante da operação $p(x) \cdot q(x)$.

AULA 58

# PILHAS

O armazenamento seqüencial de uma lista é usado, em geral, quando essas estruturas sofrem poucas modificações ao longo do tempo, isto é, poucas inserções e remoções são realizadas durante sua existência. Isso implica em poucas movimentações de registros. Por outro lado, a alocação encadeada é mais usada na situação inversa, ou seja, quando modificações na lista linear são mais freqüentes.

Caso os elementos a serem inseridos ou removidos de uma lista linear se localizem em posições especiais nessas estruturas, como por exemplo a primeira ou a última posição, então temos uma situação favorável para o uso de listas lineares em alocação seqüencial. Este é o caso da estrutura de dados denominada pilha. O que torna essa lista linear especial é a adoção de uma política bem definida de inserções e remoções, sempre em um dos extremos da lista. A implementação de uma pilha em alocação encadeada tem muitas aplicações importantes e também será discutida nesta aula.

## 58.1 Definição

Uma **pilha** é uma lista linear tal que as operações de inserção e remoção são realizadas em um único extremo dessa estrutura de dados.

O funcionamento dessa lista linear pode ser comparado a qualquer pilha de objetos que usamos com freqüência como, por exemplo, uma pilha de pratos de um restaurante. Em geral, os clientes do restaurante retiram pratos do topo da pilha e os funcionários colocam pratos limpos também no topo.

Observe que uma pilha tem sempre um indicador do extremo em que uma inserção ou remoção deve ser realizada. O indicador desse extremo, nesta estrutura, é chamado de **topo** da pilha.

Duas operações básicas são realizadas sobre uma pilha: inserção e remoção. Essas duas operações são também chamadas de empilhamento e desempilhamento de elementos. Observe que a operação de busca não foi mencionada e não faz parte do conjunto de operações básicas de uma pilha.

## 58.2 Operações básicas em alocação seqüencial

Suponha que uma pilha esteja armazenada em um vetor `P[0..MAX-1]`. A parte do vetor efetivamente ocupada pela pilha é `P[0..t]` onde `t` é o índice que define o topo da pilha.

Uma ilustração de uma pilha em alocação seqüencial é dada na figura 58.1.

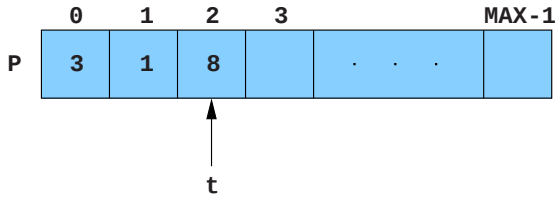



Figura 58.1: Representação de uma pilha `P` em alocação seqüencial.

Convencionamos que uma pilha está **vazia** se seu topo `t` vale $-1$ e **cheia** se seu topo `t` vale `MAX-1`.

Suponha que queremos inserir um elemento de valor 7 na pilha `P` da figura 58.1. Como resultado desta operação, esperamos que a pilha tenha a configuração mostrada na figura 58.2 após sua execução.

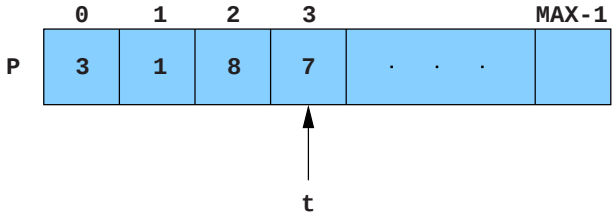



Figura 58.2: Inserção da chave 7 na pilha `P` da figura 58.1.

A operação de inserir um objeto em uma pilha, ou empilhar, é descrita a seguir.

```
1  void empilhaSeq(int P[MAX], int *t, int x)
2  {
3    if (*t != MAX-1) {
4      (*t)++;
5      P[*t] = x;
6    }
7    else
8      printf("Impossível inserir um novo elemento na pilha\n");
9  }
```

Suponha agora que queremos remover um elemento de uma pilha. Observe que, assim como na inserção, a remoção também deve ser feita em um dos extremos da pilha, isto é, no topo da pilha. Assim, a remoção de um elemento da pilha `P` que tem sua configuração ilustrada na figura 58.1 tem como resultado a pilha com a configuração mostrada na figura 58.3 após a sua execução.

A operação de remover, ou desempilhar, um objeto de uma pilha é descrita na função `desempilhaSeq` a seguir.

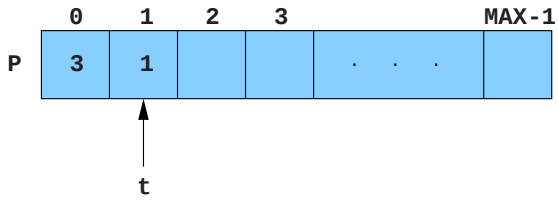



Figura 58.3: Remoção de um elemento da pilha `P` da figura 58.1.

```
1   int desempilhaSeq(int P[MAX], int *t)
2   {
3     int removido;
4     if (*t != -1) {
5       removido = P[*t];
6       (*t)--;
7       return removido;
8     }
9     else {
10      printf("Impossível remover um elemento da pilha\n");
11      return INT_MIN;
12    }
13  }
```

Agora que temos implementadas as operações básicas de inserção e remoção, ou de empilhar e desempilhar, em uma pilha, podemos usar essas funções em aplicações que necessitam dessa estrutura de dados. Diversas são as aplicações que necessitam de uma ou mais pilhas, dentre as quais podemos citar a conversão de notação de expressões aritméticas (notação infixa para notação posfixa), a pilha de execução de um programa no sistema operacional, a análise léxica de um programa realizada pelo compilador, etc.

Uma aplicação simples e interessante é apresentada a seguir. Considere, por exemplo, o problema de decidir se uma dada seqüência de parênteses e chaves é bem-formada. Por exemplo, a seqüência abaixo:

```
(  (  )  {  (  )  }  )
```

é bem-formada, enquanto que a a seqüência

```
(  {  )  }
```

é malformada.

Suponha que a seqüência de parênteses e chaves está armazenada em uma cadeia de caracteres `s`. A função `bemformada` a seguir recebe a cadeia de caracteres `s` e devolve `1` se `s` contém uma seqüência bem-formada de parênteses e chaves e devolve `0` se a seqüência está malformada.

```
int bemformada(char s[])
{
  char *p;
  int t;
  int n, i;
  n = strlen(s);
  p = (char *) malloc(n * sizeof(char));
  t = -1;
  for (i = 0; s[i] != '\0'; i++) {
    switch (s[i]) {
      case ')':
        if (t != -1 && p[t] == '(')
          t--;
        else
          return 0;
        break;
      case '}':
        if (t != -1 && p[t] == '{')
          t--;
        else
          return 0;
        break;
      default:
        t++;
        p[t] = s[i];
    }
  }
  free(p);
  return t == -1;
}
```

## 58.3 Operações básicas em alocação encadeada

As pilhas também podem ser implementadas em alocação encadeada. Existem muitas aplicações práticas onde as pilhas implementadas em alocação encadeada são importantes e bastante usadas.

Consideramos que as células da pilha são do tipo declarado abaixo:

```
typedef struct cel {
  int conteudo;
  struct cel *prox;
} celula;
```

Uma pilha vazia pode ser criada da seguinte forma:

```
celula *topo;
topo = NULL;
```

A operação de empilhar uma nova chave `x` em uma pilha `t` é descrita na função a seguir.

```
1  void empilhaEnc(celula **t, int x)
2  {
3    celula *nova;
4    nova = (celula *) malloc(sizeof(celula));
5    if (nova != NULL) {
6      nova->conteudo = x;
7      nova->prox = *t;
8      *t = nova;
9    }
10   else {
11     printf("Impossível inserir um novo elemento na pilha\n");
12     exit(EXIT_FAILURE);
13   }
14 }
```

Suponha agora que queremos remover um elemento do topo de uma pilha `t`. A operação de desempilhar um registro é descrita na função a seguir.

```
1  int desempilhaEnc(celula **t)
2  {
3    int x;
4    celula *aux;
5    if (*t != NULL) {
6      aux = *t;
7      x = (*t)->conteudo;
8      *t = (*t)->prox;
9      free(aux);
10     return x;
11   }
12   else {
13     printf("Impossível remover um elemento da pilha\n");
14     return INT_MIN;
15   }
16 }
```

## Exercícios

58.1 Reescreva a função `bemformada` armazenando a pilha em uma lista linear em alocação encadeada.

58.2 Uma palavra é um **palíndromo** se a seqüência de caracteres que a constitui é a mesma quer seja lida da esquerda para a direita ou da direita para a esquerda. Por exemplo, as palavras RADAR e MIRIM são palíndromos. Escreva um programa eficiente para reconhecer se uma dada palavra é palíndromo.

58.3 Suponha que exista um único vetor $M$ de registros de um tipo pilha pré-definido, com um total de **`MAX`** posições. Este vetor fará o papel da memória do computador. Este vetor $M$ será compartilhado por duas pilhas em alocação seqüencial. Implemente eficientemente as operações de empilhamento e desempilhamento para as duas pilhas de modo que nenhuma das pilhas estoure sua capacidade de armazenamento, a menos que o total de elementos em ambas as pilhas seja **`MAX`**.

58.4 Um estacionamento possui um único corredor que permite dispor 10 carros. Existe somente uma única entrada/saída do estacionamento em um dos extremos do corredor. Se um cliente quer retirar um carro que não está próximo à saída, todos os carros impedindo sua passagem são retirados, o cliente retira seu carro e os outros carros são recolocados na mesma ordem que estavam originalmente. Escreva um algoritmo que processa o fluxo de chegada/saída deste estacionamento. Cada entrada para o algoritmo contém uma letra `E` para entrada ou `S` para saída, e o número da placa do carro. Considere que os carros chegam e saem pela ordem especificada na entrada. O algoritmo deve imprimir uma mensagem sempre que um carro chega ou sai. Quando um carro chega, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não existe vaga, o carro não entra no estacionamento e vai embora. Quando um carro sai do estacionamento, a mensagem deve incluir o número de vezes que o carro foi movimentado para fora da garagem, para permitir que outros carros pudessem sair.

AULA 59

# FILAS

Uma fila é, assim como uma pilha, uma lista linear especial, onde há uma política de inserções e remoções bem definida. Em uma fila, uma inserção é realizada em um dos extremos da fila e a remoção no outro extremo. Em uma pilha, como vimos na aula passada, a inserção e a remoção são realizadas em um mesmo extremo. Nesta aula veremos como implementar as operações básicas sobre uma fila, considerando sua implementação em alocação seqüencial e encadeada.

## 59.1 Definição

Uma **fila** é uma lista linear tal que as operações de inserção são realizadas em um dos extremos da lista e a remoção é realizada no outro extremo. Uma ilustração de uma fila é mostrada na figura 59.1. O funcionamento dessa estrutura pode ser comparado a qualquer fila de objetos que usamos com freqüência como, por exemplo, uma fila de um banco. Em geral, as pessoas entram, ou são inseridas, no final da fila e as pessoas saem, ou são removidas, do início da fila.

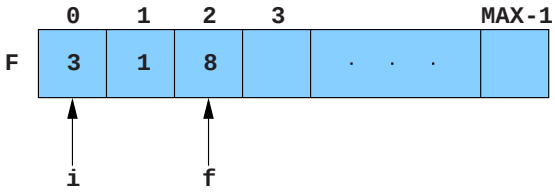



Figura 59.1: Representação de uma fila `F` em alocação seqüencial.

Note que uma fila tem sempre um indicador de onde começa e de onde termina. Quando queremos inserir um elemento na fila, esta operação é realizada no final da estrutura. Quando queremos remover um elemento da fila, essa operação é realizada no início da estrutura. Esses extremos são referenciados como `inicio` e `fim` de uma fila `F`. Se a fila `F` é implementada em alocação seqüencial, isto é, sobre um vetor, então esses extremos são índices desse vetor. Caso contrário, se for implementada em alocação encadeada, então esses extremos são apontadores para o primeiro e o último registros da fila.

Assim como nas pilhas, duas operações básicas são realizadas sobre uma fila: inserção e remoção. Essas duas operações são também chamadas de enfileiramento e desenfileiramento de elementos. Observe que a operação de busca não foi mencionada e não faz parte do conjunto de operações básicas de uma fila.

## 59.2 Operações básicas em alocação seqüencial

Uma fila está armazenada em um segmento `F[i..f]` de um vetor `F[0..MAX-1]`. Assim, devemos ter $0 \leqslant$ `i` $\leqslant$ `f` $\leqslant$ `MAX` $- 1$. O primeiro elemento da fila está na posição `i` e o último na posição `f`. A fila está **vazia** se `i` $=$ `f` $= -1$ e **cheia** se `f` $+ 1$ mod `MAX` $=$ `i`.

Suponha que queremos inserir um elemento de chave `7` na fila `F` da figura 59.1. Como resultado desta operação, esperamos que a fila tenha a configuração mostrada na figura 59.2 após sua execução.

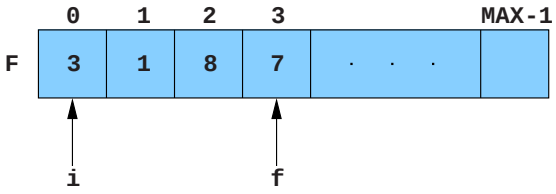



Figura 59.2: Inserção da chave 7 na fila `F` da figura 59.1.

Observe que o índice `i` move-se somente quando da remoção de um registro da fila e o índice `f` move-se somente quando da inserção de um registro na fila. Esses incrementos fazem com que a fila "movimente-se" da esquerda para direita, o que pode ocasionar a falsa impressão de transbordamento de memória. Consideramos então os registros alocados seqüencialmente como se estivessem em um círculo, onde o compartimento `F[MAX-1]` é seguido pelo compartimento `F[0]`.

A operação de enfileirar um novo número inteiro `x` em uma fila `F` é descrita a seguir.

```
1   void enfileiraSeq(int F[], int *i, int *f, int x)
2   {
3     int aux;
4     aux = (*f + 1) % MAX;
5     if (aux != *i) {
6       *f = aux;
7       F[*f] = x;
8       if (*i == -1)
9         *i = 0;
10    }
11    else
12      printf("Impossível inserir um novo elemento na fila\n");
13  }
```

A variável `aux` é um índice temporário para o final da fila. Se o índice armazenado em `aux` é igual ao índice `*i`, então não há mais espaço disponível na fila. Caso contrário, o índice `*f` é atualizado com esse valor e o novo elemento é colocado no final da fila.

Suponha agora que queremos remover um elemento da fila `F` que aparece na figura 59.1. Como resultado desta operação, esperamos que a fila tenha a configuração mostrada na figura 59.3 após sua execução.

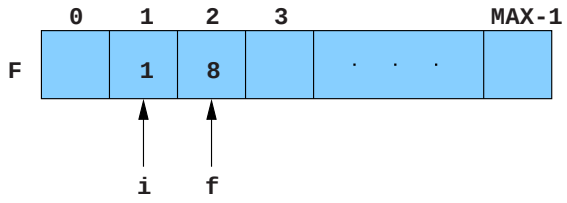



Figura 59.3: Remoção de um elemento da fila F da figura 59.1.

A operação de desenfileirar um número inteiro em uma fila `F` é descrita a seguir.

```
1  int desenfileiraSeq(int F[], int *i, int *f)
2  {
3    int removido;
4    removido = INT_MIN;
5    if (*i != -1) {
6      removido = F[*i];
7      if (*i != *f)
8        *i = (*i + 1) % MAX;
9      else {
10       *i = -1;
11       *f = -1;
12     }
13   }
14   else
15     printf("Impossível remover um elemento da fila\n");
16   return removido;
17 }
```

Como um exemplo de aplicação, suponha que temos $n$ cidades numeradas de $0$ a $n-1$ e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz `A` definida da seguinte forma: `A[x][y]` vale `1` se existe estrada da cidade `x` para a cidade `y` e vale `0` em caso contrário. A figura 59.4 ilustra um exemplo. A **distância** de uma cidade `o` a uma cidade `x` é o menor número de estradas que é preciso percorrer para ir de `o` a `x`. Estabelecemos então o seguinte problema: determinar a distância de uma dada cidade `o` a cada uma das outras cidades da rede. As distâncias são armazenadas em um vetor `d` de tal modo que `d[x]` seja a distância de `o` a `x`. Se for impossível chegar de `o` a `x`, podemos dizer que `d[x]` vale $\infty$. Usaremos ainda $-1$ para representar $\infty$ uma vez que nenhuma distância "real" pode ter valor $-1$.

O programa 59.1 usa uma fila para solucionar o problema das distâncias em uma rede. O programa classifica as cidades durante sua execução, da seguinte forma: uma cidade é considerada **ativa** se já foi visitada mas as estradas que nela começam ainda não forma exploradas. As cidades ativas são mantidas numa fila pelo programa. Em cada iteração, o programa remove da fila uma cidade `x` e insere na fila todas as cidades vizinhas a `x` que ainda não foram visitadas.

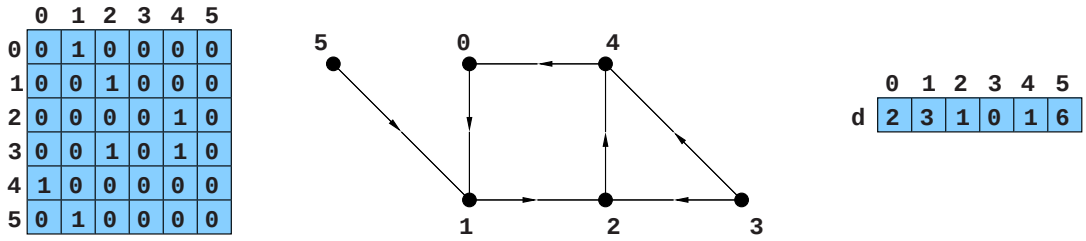



Figura 59.4: A matriz representa cidades $0, \ldots, 5$ interligadas por estradas de mão única. O vetor `d` dá as distâncias da cidade 3 a cada uma das demais.

Programa 59.1: Exemplo de uma função que usa uma fila.

```
1  int *distancias(int A[MAX][MAX], int n, int o)
2  {
3    int *d, x, y;
4    int *F, i, f;
5    d = (int *) malloc(n * sizeof(int));
6    for (x = 0; x < n; x++)
7      d[x] = -1;
8    d[o] = 0;
9    F = (int *) malloc(n * sizeof(int));
10   i = 0;
11   f = 0;
12   F[i] = o;
13   while (i <= f) {
14     x = F[i];
15     i++;
16     for (y = 0; y < n; y++)
17       if (A[x][y] == 1 && d[y] == -1) {
18         d[y] = d[x] + 1;
19         f++;
20         F[f] = y;
21       }
22   }
23   free(F);
24   return d;
25 }
```

## 59.3 Operações básicas em alocação encadeada

A título de recordação, lembramos que uma fila é uma lista linear tal que suas operações de inserção são realizadas em um dos extremos da lista e as operações de remoção são realizadas no extremo oposto. Observe então que uma fila tem sempre dois indicadores/marcadores da posição onde começa e da posição onde termina. A política de armazenamento de elementos

nas células de uma fila estabelece que quando queremos inserir um elemento em uma fila, essa operação é sempre realizada no extremo identificado como o final dessa estrutura. Por outro lado, quando queremos remover um elemento da fila, essa operação é sempre realizada no extremo identificado como início da estrutura. Em alocação encadeada, esses extremos são referenciados através de apontadores para células da fila, que contêm o endereço das suas primeira e última células.

Assim como nas pilhas, apenas duas operações básicas são realizadas sobre as filas: inserção e remoção. Essas duas operações são também chamadas de enfileiramento e desefileiramento de elementos. Observe que a operação básica de busca, que especificamos nas listas lineares, conforme a aula 57, não foi mencionada e também não faz parte do conjunto de operações básicas de uma fila.

Uma fila em alocação encadeada é dita **vazia** se e somente se ambos seus apontadores de início e fim contêm o valor nulo ( `NULL` ). Dessa forma, podemos criar facilmente uma fila vazia como especificado abaixo:

```
celula *ini, *fim;
ini = NULL;
fim = NULL;
```

A operação de enfileirar um novo elemento `x` em uma fila é dada através da função descrita abaixo.

```
1  void enfileiraEnc(celula **comeco, celula **final, int x)
2  {
3    celula *nova;
4    nova = (celula *) malloc(sizeof(celula));
5    if (nova != NULL) {
6      nova->conteudo = x;
7      nova->prox = NULL;
8      if (*final != NULL)
9        (*final)->prox = nova;
10     else
11       *comeco = nova;
12     *final = nova;
13   }
14   else
15     printf("Impossível inserir um novo elemento na fila\n");
16 }
```

Suponha agora que queremos desenfileirar um elemento de uma fila identificada pelos apontadores `*comeco` e `*final`. A operação de desenfileirar uma célula dessa fila é descrita a seguir.

```
1  int desenfileiraEnc(celula **comeco, celula **final)
2  {
3    int x;
4    celula *p;
5    if (*comeco != NULL) {
6      p = *comeco;
7      x = p->conteudo;
8      *comeco = p->prox;
9      free(p);
10     if (*comeco == NULL)
11       *final = NULL;
12     return x;
13   }
14   else {
15     printf("Impossível remover um elemento da fila\n");
16     return INT_MIN;
17   }
18 }
```

## Exercícios

59.1 Solucione o problema das distências em uma rede usando uma fila em alocação encadeada.

59.2 Implemente uma fila usando duas pilhas.

59.3 Implemente uma pilha usando duas filas.

59.4 Um estacionamento possui um único corredor que permite dispor 10 carros. Os carros chegam pelo sul do estacionamento e saem pelo norte. Se um cliente quer retirar um carro que não está próximo do extremo norte, todos os carros impedindo sua passagem são retirados, o cliente retira seu carro e os outros carros são recolocados na mesma ordem que estavam originalmente. Sempre que um carro sai, todos os carros do sul são movidos para frente, de modo que as vagas fiquem disponíveis sempre no extremo sul do estacionamento. Escreva um algoritmo que processa o fluxo de chegada/saída deste estacionamento. Cada entrada para o algoritmo contém uma letra 'E' para entrada ou 'S' para saída, e o número da placa do carro. Considere que os carros chegam e saem pela ordem especificada na entrada. O algoritmo deve imprimir uma mensagem sempre que um carro chega ou sai. Quando um carro chega, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não existe vaga, o carro deve esperar até que exista uma vaga, ou até que uma instrução fornecida pelo usuário indique que o carro deve partir sem que entre no estacionamento. Quando uma vaga torna-se disponível, outra mensagem deve ser impressa. Quando um carro sai do estacionamento, a mensagem deve incluir o número de vezes que o carro foi movimentado dentro da garagem, incluindo a saída mas não a chegada. Este número é 0 se o carro partiu da linha de espera, isto é, se o carro esperava uma vaga, mas partiu sem entrar no estacionamento.

59.5 Imagine um tabuleiro quadrado 10–por–10. As casas "livres" são marcadas com $0$ e as casas "bloqueadas" são marcadas com $-1$. As casas $(1,1)$ e $(10,10)$ estão livres. Ajude uma formiga que está na casa $(1,1)$ a chegar à casa $(10,10)$. Em cada passo, a formiga só pode se deslocar para uma casa livre que esteja à direita, à esquerda, acima ou abaixo da casa em que está.

59.6 Um **deque** é uma lista linear que permite a inserção e a remoção de elementos em ambos os seus extremos. Escreva quatro funções para manipular um deque: uma que realiza a inserção de um novo elemento no início do deque, uma que realiza a inserção de um novo elemento no fim do deque, uma que realiza a remoção de um elemento no início do deque e uma que realiza a remoção de um elemento no fim do deque.

AULA 60

# LISTAS LINEARES CIRCULARES

Algumas operações básicas em listas lineares, em especial aquelas implementadas em alocação encadeada, não são muito eficientes. Um exemplo emblemático desse tipo de operação é a busca, como pudemos perceber em aulas anteriores. As listas lineares circulares realizam algumas operações mais eficientemente, já que possuem uma informação a mais que as listas lineares ordinárias.

Uma lista linear circular é, em geral, implementada em alocação encadeada e difere de uma lista linear por não conter um apontador para nulo em qualquer de seus registros, o que não permite que se identifique um fim, ou mesmo um começo, da lista. Caso um registro especial, chamado cabeça da lista, faça parte da sua estrutura, então é possível identificar facilmente um início e um fim da lista linear. Nesta aula veremos a implementação das operações básicas de busca, inserção e remoção em listas lineares circulares em alocação encadeada.

## 60.1 Alocação encadeada

A adição de uma informação na estrutura de uma lista linear em alocação encadeada permite que operações sejam realizadas de forma mais eficiente. Quando o último registro da lista linear encadeada aponta para o primeiro registro, temos uma **lista linear circular**. Dessa forma, não podemos identificar um fim da lista linear. No entanto, o que aparenta uma dificuldade é, na verdade, uma virtude desse tipo de estrutura, como veremos nas funções que implementam suas operações básicas de busca, inserção e remoção. Veja a figura 60.1 para um exemplo de uma lista linear circular em alocação encadeada.

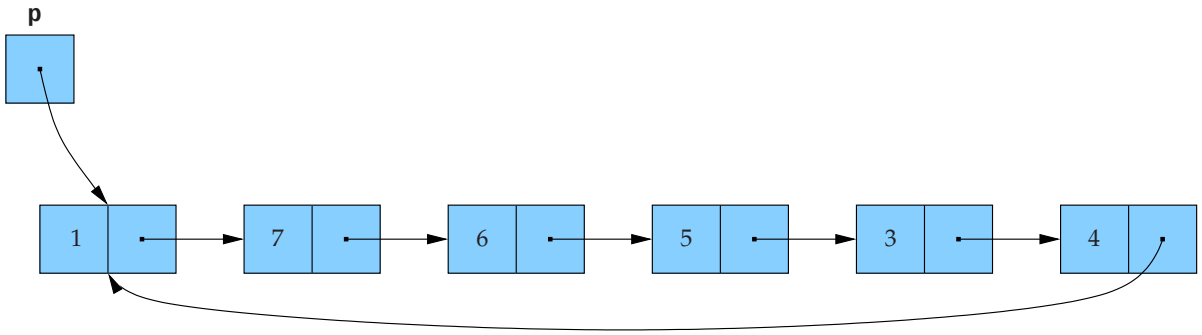



Figura 60.1: Representação de uma lista linear circular em alocação encadeada.

A definição de um tipo célula da lista linear circular permanece a mesma, como mostramos abaixo:

```
typedef struct cel {
  int conteudo;
  struct cel *prox;
} celula;
```

A seguir vamos discutir e implementar as operações básicas de busca, inserção e remoção sobre listas lineares circulares em alocação encadeada. É importante observar que a implementação dessas listas tornam-se mais fáceis se usamos o recurso de inserir uma célula especial no início da lista, chamada de cabeça.

## 60.2 Busca

No processo de busca, dada uma lista linear circular e um valor, queremos saber se o valor está presente na lista. A função `buscaCirc` recebe um apontador `circ` para a lista linear circular e um valor `x` e devolve uma célula contendo o valor procurado `x`, caso `x` ocorra em na lista. Caso contrário, a função devolve `NULL`.

```
1  celula *buscaCirc(celula *circ, int x)
2  {
3    celula *p;
4    if (circ != NULL) {
5      p = circ->prox;
6      while (p != circ) {
7        if (p->conteudo == x)
8          return p;
9        p = p->prox;
10     }
11     if (p->conteudo == x)
12       return p;
13   }
14   return NULL;
15 }
```

## 60.3 Inserção

A inserção em uma lista linear circular encadeada é simples e semelhante à inserção em uma lista linear encadeada. Temos apenas de prestar atenção para que a inserção sempre mantenha a propriedade da lista linear ser circular, o que se dá de modo natural se inserimos o novo elemento logo em seguida à célula referenciada pelo apontador da lista.

```
1  void insereCirc(celula **circ, int x)
2  {
3    celula *nova;
4    if (buscaCirc(*circ, x) == NULL) {
5      nova = (celula *) malloc(sizeof(celula));
6      nova->conteudo = x;
7      if (*circ != NULL) {
8        nova->prox = (*circ)->prox;
9        (*circ)->prox = nova;
10     }
11     else {
12       nova->prox = nova;
13       *circ = nova;
14     }
15   }
16   else
17     printf("Valor já se encontra na lista!\n");
18 }
```

## 60.4 Remoção

A função de remoção de um elemento em uma lista linear circular necessita de uma modificação na função de busca, de modo a devolver implicitamente dois apontadores, um para a célula contendo o elemento procurado e outro para a célula imediatamente anterior. A função `buscaCirc2` é uma modificação das funções `buscaCirc` e `buscaSeq2` e é deixada como exercício.

```
1  void removeCirc(celula **circ, int x)
2  {
3    celula *p, *ant;
4    buscaCirc2(*circ, &p, &ant, x);
5    if (p != NULL) {
6      if (p != ant) {
7        ant->prox = p->prox;
8        if (p == *circ)
9          *circ = p->prox;
10     }
11     else
12       *circ = NULL;
13     free(p);
14   }
15   else
16     printf("Elemento não encontrado!\n");
17 }
```

## Exercícios

60.1 Escreva a função `buscaCirc2`, que receba uma lista linear circular e um valor `x` e devolva dois apontadores `ant` e `p` que apontam para a célula anterior a célula que contém `x`, respectivamente.

60.2 Suponha que queremos implementar uma lista linear circular em alocação encadeada de tal forma que os conteúdos sempre permaneçam em ordem crescente. Escreva as funções para as operações básicas de busca, inserção e remoção para listas lineares circulares em alocação encadeada com os conteúdos ordenados.

60.3 Também para listas lineares circulares vale que as funções que implementam suas operações básicas são mais simples e eficientes se mantemos uma célula que indica seu início, chamada de cabeça. Veja a figura 60.2.

Escreva as funções para as operações básicas de busca, inserção e remoção para listas lineares circulares em alocação encadeada com cabeça.

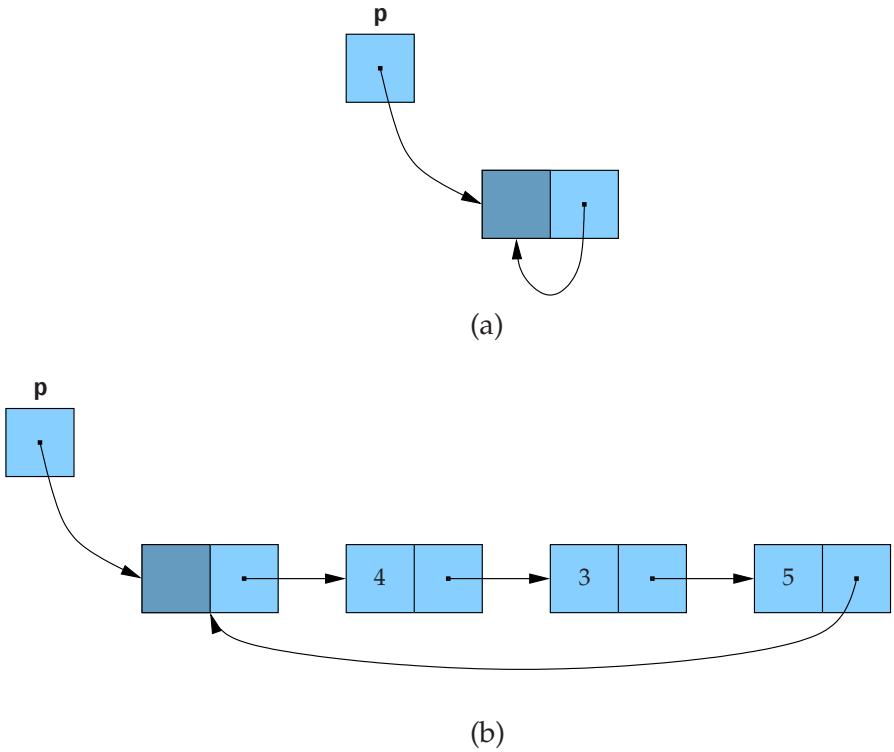



Figura 60.2: Lista linear circular em alocação encadeada com cabeça. (a) Lista vazia. (b) Lista com três células.

60.4 O **problema de Josephus** foi assim descrito através do relato de Flavius Josephus, um historiador judeu que viveu no primeiro século, sobre o cerco de Yodfat. Ele e mais 40 soldados aliados estavam encurralados em uma caverna rodeada por soldados romanos e, não havendo saída, optaram então pelo suicídio antes da captura. Decidiram que formariam um círculo e, a cada contagem de 3, um soldado seria morto. Josephus foi o último a se salvar.

Podemos agora, descrever um problema mais geral como segue. Imagine $n$ pessoas dispostas em círculo. Suponha que as pessoas são numeradas de $1$ a $n$ no sentido horário. Começando com a pessoa de número 1, percorra o círculo no sentido horário e elimine cada $m$-ésima pessoa enquanto o círculo tiver duas ou mais pessoas. Escreva e teste uma função que resolva o problema, imprimindo na saída o número do sobrevivente.

AULA 61

# LISTAS LINEARES DUPLAMENTE ENCADEADAS

Quando trabalhamos com listas lineares (simplesmente) encadeadas, muitas vezes precisamos de manter um apontador para um registro e também para o registro anterior para poder realizar algumas operações evitando percursos adicionais desnecessários. No entanto, algumas vezes isso não é suficiente, já que podemos precisar percorrer a lista linear nos dois sentidos. Neste caso, além do campo apontador para o próximo registro no registro base da lista, adicionamos um novo campo apontador para o registro anterior. O gasto de memória imposto pela adição deste novo campo se justifica pela economia em não ter de reprocessar a lista linear inteira.

## 61.1 Definição

Os registros de uma lista linear duplamente encadeada são ligados por apontadores que indicam a posição do registro anterior e do próximo registro da lista. Assim, um outro campo é acrescentado a cada registro da lista indicando, além do endereço do próximo registro, o endereço do registro anterior da lista. Veja a figura 61.1.

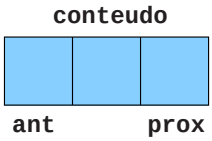



Figura 61.1: Representação de uma célula de uma lista linear duplamente encadeada.

A definição de uma célula de uma lista linear duplamente encadeada é então descrita como um tipo, como mostramos a seguir:

```
typedef struct cel {
  int conteudo;
  struct cel *ant;
  struct cel *prox;
} celula;
```

Uma representação gráfica de uma lista linear em alocação encadeada é mostrada na figura 61.2.

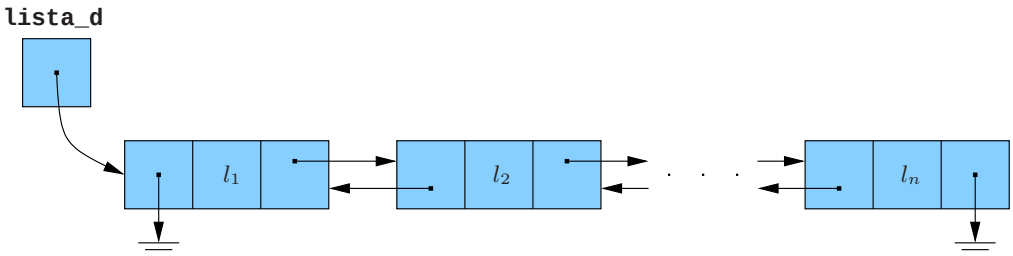



Figura 61.2: Representação de uma lista linear duplamente encadeada apontada por **`lista_dup`**. A primeira célula tem seu campo **`ant`** apontando para `NULL` e a última tem seu campo **`prox`** também apontando para **`NULL`**.

Uma lista linear duplamente encadeada pode ser criada e inicializada de forma idêntica a de uma lista linear, como mostramos a seguir:

```
celula *lista_d;
lista_d = NULL;
```

A seguir vamos discutir e implementar as operações básicas sobre listas lineares duplamente encadeadas.

## 61.2 Busca

No processo de busca de um valor em uma lista linear duplamente encadeada, é dada uma lista linear duplamente encadeada `dup` e um valor `x` e queremos saber se `x` está presente em `dup`. Uma função que realiza esse processo pode ser descrita como a seguir.

A função `buscaDup` recebe um apontador `dup` para uma lista duplamente encadeada e um valor `x` e devolve um registro contendo o valor procurado `x`, caso `x` ocorra em `dup`. Caso contrário, a função devolve `NULL`.

```
1  celula *buscaDup(celula *dup, int x)
2  {
3    celula *p;
4    p = dup;
5    while (p != NULL && p->conteudo != x)
6      p = p->prox;
7    return p;
8  }
```

Observe que a função `buscaDup` é praticamente uma cópia da função `buscaEnc`.

## 61.3 Inserção

A função que insere um elemento `x` em uma lista linear duplamente encadeada `*dup` é apresentada a seguir. A inserção continua sendo feita no início da lista.

```
1   void insereDup(celula **dup, int x)
2   {
3     celula *nova;
4     if (buscaDup(*dup, x) == NULL) {
5       nova = (celula *) malloc(sizeof(celula));
6       if (nova != NULL) {
7         nova->conteudo = x;
8         nova->ant = NULL;
9         nova->prox = *dup;
10        if (*dup != NULL)
11          (*dup)->ant = nova;
12        *dup = nova;
13      }
14      else {
15        printf("Não há memória disponível!\n");
16        exit(EXIT_FAILURE);
17      }
18    }
19    else
20      printf("Elemento já se encontra na lista!\n");
21  }
```

Observe novamente que as inserções são sempre realizadas no início da lista linear duplamente encadeada. Podemos modificar essa comportamento, por exemplo, mantendo um outro apontador para o final da lista e fazendo as inserções neste outro extremo. Deixamos essa idéia para ser implementada em um exercício.

## 61.4 Remoção

Como sabemos, para realizar a remoção de um elemento da lista é necessário buscá-lo. A função `buscaDup` descrita anteriormente devolve um apontador para o elemento onde o valor procurado se encontra ou `NULL` caso o valor não ocorra na lista. Se a busca tem sucesso, então o apontador devolvido aponta para a célula que se quer remover. Além disso, essa célula tem acesso à célula anterior e posterior da lista duplamente encadeada e isso é tudo o que precisamos saber para realizar a remoção satisfatoriamente. A função `removeDup` recebe como parâmetros um apontador `*dup` para o início da lista duplamente encadeada e um valor `x`, busca uma célula na lista que contém esse valor e, caso esse valor ocorra na lista, realiza sua remoção.

```
1  void removeDup(celula **dup, int x)
2  {
3    celula *anterior, *posterior, *removido;
4    removido = buscaDup(*dup, x);
5    if (removido != NULL) {
6      anterior = removido->ant;
7      posterior = removido->prox;
8      if (anterior != NULL)
9        anterior->prox = posterior;
10     else
11       *dup = posterior;
12     if (posterior != NULL)
13       posterior->ant = anterior;
14     free(removido);
15   }
16   else
17     printf("Valor não se encontra na lista!\n");
18 }
```

Observe que os apontadores `anterior`, `posterior` e `removido` são variáveis locais à função `removeDup` que auxiliam tanto na busca como na remoção propriamente. Além disso, se a célula a ser removida é a primeira da lista, há necessidade de alterar o conteúdo do apontador para o início da lista `dup` para a célula seguinte desta lista.

Observe ainda que o procedimento `removeDup` pode ser usado para remoção de uma célula em uma lista linear encadeada ordenada ou não ordenada.

## Exercícios

61.1 Considere uma lista linear duplamente encadeada contendo os elementos $l_1, l_2, \ldots, l_n$, como na figura 61.3.

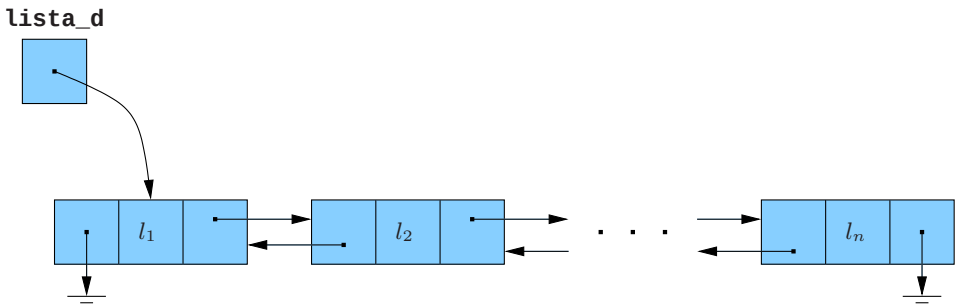



Figura 61.3: Uma lista linear duplamente encadeada.

Escreva uma função que altere os apontadores `ant` e `prox` da lista, *sem* mover suas informações, tal que a lista fique invertida como na figura 61.4.

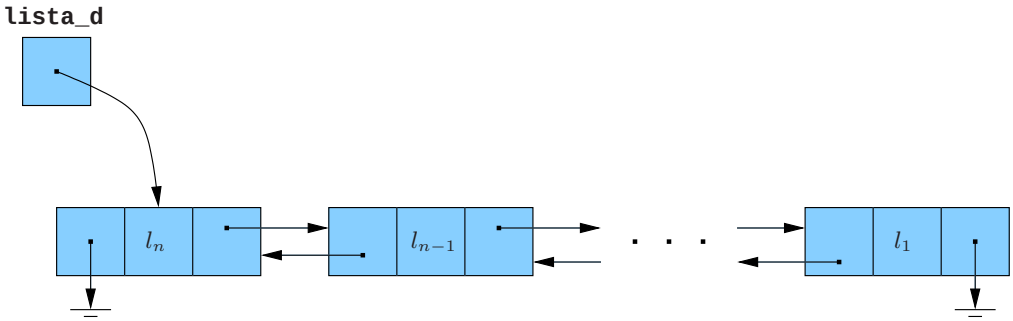



Figura 61.4: Inversão da lista linear duplamente encadeada da figura 61.3.

61.2 Escreva uma função que receba um apontador para o início de uma lista linear em alocação duplamente encadeada, um apontador para o fim dessa lista e um valor, e realize a inserção desse valor no final da lista.

61.3 Suponha que você queira manter uma lista linear duplamente encadeada com os conteúdos em ordem crescente. Escreva as funções de busca, inserção e remoção para essa lista.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] P. Breton. *História da Informática*. Editora da UNESP, 1991. Tradução de Elcio Fernandes do original *Histoire de L'informatique*, 1987, Editions La Decouvert. 1

[2] Computer history – from B.C. to today.
`http://www.computerhope.com/history/`.
último acesso em 28 de janeiro de 2009. 1

[3] R. X. Cringely. A history of the computer.
`http://www.pbs.org/nerds/timeline/index.html`.
último acesso em 28 de janeiro de 2009. 1

[4] P. Feofiloff. *Algoritmos em linguagem C*. Editora Campus/Elsevier, 2009. 4.5, 4.5, 7.3, 17, 18, 20.2

[5] B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison-Wesley Professional, 1999.

[6] B. W. Kernighan and D. M. Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1988.

[7] K. N. King. *C Programming – A Modern Approach*. W. W. Norton & Company, Inc., 2nd edition, 2008. 17, 18

[8] S. G. Kochan. *Unix Shell Programming*. Sams Publishing, 3rd edition, 2003.

[9] J. Kopplin. An illustrated history of computers.
`http://www.computersciencelab.com/ComputerHistory/History.htm`.
último acesso em 28 de janeiro de 2009. 1

[10] T. Kowaltowski. John von Neumann: suas contribuições à computação. *Revista de Estudos Avançados*, 10(26), Jan/Abr 1996. Instituto de Estudos Avançados da Universidade de São Paulo. 2

[11] H. Lukoff. *From Dits to Bits... a Personal History of the Electronic Computer*. Hawley Books, 1979. 2.1

[12] Departamento de Ciência da Computação – IME/USP, listas de exercícios – Introdução à Computação. `http://www.ime.usp.br/~macmulti/`.
último acesso em 28 de janeiro de 2009. 2

[13] A. Sapounov, E. Rosen, and J. Shaw. Computer history museum.
`http://www.computerhistory.org/`.
último acesso em 28 de janeiro de 2009. 1

[14] R. L. Shackelford. *Introduction to Computing and Algorithms*. Addison Wesley, 1997. 1

[15] S. S. Skiena and M. Revilla. *Programming Challenges*. Springer, 2003.

[16] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936. 2.1, 2.4

[17] J. von Neumann. First draft of a report on the EDVAC. Technical report, Moore School of Electrical Engineering – University of Pennsylvania, 1945. 2.1, 2.3

[18] Wikipedia – the free encyclopedia.
`http://en.wikipedia.org/wiki/Main_Page`.
último acesso em 28 de janeiro de 2009. 1

[19] K. Zuse. Verfahren zur selbsttätigen durchführung von rechnungen mit hilfe von rechenmaschinen. Patentanmeldung Z 23 139 – GMD Nr. 005/021, 1936. 2.1