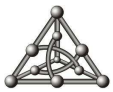
Programação de Computadores I

Fábio Henrique Viduani Martinez

Faculdade de Computação – UFMS



2011

SUMÁRIO

AULA 1

BREVE HISTÓRIA DA COMPUTAÇÃO

Nesta aula abordaremos uma pequena história da computação. 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 que, por descuido ou desconhecimento, infelizmente ainda não foram consultadas.

Este texto é baseado principalmente nas páginas da Rede Mundial de Computadores [1, 2, 3, 4, 5] e nos livros de Philippe Breton [6] e Russel Shackelford [7].

1.1 Pré-história

Desde muito tempo, é possível notar que o ser humano vem realizando cálculos complexos para solucionar seus mais diversos problemas. Um dos primeiros e mais notáveis exemplos que temos registro é a descoberta do número real que representa a relação entre o comprimento da circunferência de um círculo e o comprimento de seu raio. A mais antiga evidência do uso consciente de uma aproximação precisa para esse número, com valor $3 + \frac{1}{7}$, foi descoberta nos projetos das pirâmides da época do Reino Antigo no Egito, datando assim do ano de 2630 a.C. a 2468 a.C. As mais antigas evidências textuais do cálculo de aproximações desse número datam de 1900 a.C., realizadas pelos geômetras egípcios, babilônios, hindus e gregos. A letra grega π é até hoje usada como a representação desse número.

Além disso, a história do pensamento humano é marcada, desde o princípio, pelos registros de tentativas de sistematização e estruturação do mesmo. A lógica[1] é uma ciência filosófico-matemática descrita geralmente por um conjunto de regras que procura sistematizar o pensamento. A lógica enxerga o pensamento como a manifestação do conhecimento, que por sua vez é visto como a busca da verdade, e, para que essa meta seja atingida, estabelece regras a serem seguidas, chamadas de regras do pensar corretamente. Apesar de muitos povos antigos terem usado sofisticados sistemas de raciocínio, somente na China, Índia e Grécia os métodos de raciocínio tiveram um desenvolvimento sustentável e é possível que a lógica tenha emergido nesses três países por volta do século IV a.C. A lógica moderna descende da lógica clássica, também conhecida como lógica aristotélica, desenvolvida por Aristóteles[2]. A lógica moderna é baseada na lógica clássica e é usada extensivamente em diversas áreas do conhecimento, em especial na Ciência da Computação. Até os dias de hoje, programas e computadores são projetados com base nessa estruturação do pensamento chamada lógica.

[1] Do grego clássico *λόγος*, ou *logos*, que significa palavra, pensamento, idéia, argumento, relato, razão lógica ou princípio lógico.

[2] Aristotélēs (384 a.C. − 322 a.C.), nascido na Grécia, filósofo, aluno de Platão, mestre de Alexandre, o Grande.

Do mesmo modo, 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[3] 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 baseado especialmente no fato de auxiliar a memória humana durante as operações aritméticas mais básicas. Alguém bem treinado no uso de um ábaco pode executar adições e subtrações com a mesma velocidade de quem usa uma calculadora eletrônica. No entanto, as duas outras operações básicas são realizadas de forma bem mais lenta. Esse instrumento ainda é consideravelmente utilizado em certos países do Oriente.

No século IX, as contribuições de Muḥammad ibn Mūsā al-Khwārizmī[4] para a matemática, geografia, astronomia e cartografia estabelecem a base para inovações na álgebra e na trigonometria. A palavra "álgebra" deriva de seu livro do ano de 830 chamado "O Livro Compêndio sobre Cálculo por Completamento e Equalização" (*al-Kitab al-mukhtasar fi hisab* ***al-jabr*** *wa'l-muqabala*), onde apresenta um método sistemático para solução de equações lineares e quadráticas. Por esse trabalho é conhecido como o fundador da Álgebra, título que divide com Diophantus. Seu outro livro "Sobre o Cálculo com Números Hindus" (*Ketab fi Isti'mal al-'Adad al-Hindi*), escrito no ano de 825 e traduzido em seguida para o latim, foi responsável pelo espalhamento do sistema de numeração hindu pelo Oriente Médio e Europa. Esses e outros de seus livros foram responsáveis por enormes avanços na área da Matemática no mundo ocidental naquela época. Um outro livro seu chamado "A Imagem da Terra" (*Kitab surat al-ard*), traduzido como "Geografia", apresenta uma revisão do trabalho de Ptolomeu com coordenadas mais precisas especialmente do mar Mediterrâneo, Ásia e África. Seu nome al-Khwārizmī foi traduzido para o latim como *Algoritmi*, o que levou o surgimento da palavra algoritmo (*algorithm* em inglês) e algarismo em português. al-Khwārizmī também projetou dispositivos mecânicos como o astrolábio e o relógio solar.

Figura 1.2: Página de *Álgebra*.

[3] *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).

[4] Abū Abdallā Muḥammad ibn Mūsā al-Khwārizmī (780 – 850), nascido na Pérsia, matemático, astrônomo e geógrafo.

Ainda sobre os primeiros dispositivos de auxílio nas atividades humanas, o primeiro relógio de bolso foi criado por Peter Henlein[5] por volta de 1504. No ano de 1600, William Gilbert[6] ficou conhecido por suas investigações em magnetismo e eletricidade, tendo cunhado o termo "eletricidade" a partir da palavra grega *elecktra*. John Napier[7], 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[8] e utilizado nos projetos Mercury, Gemini e Apollo da NASA, que levaram o homem à lua.

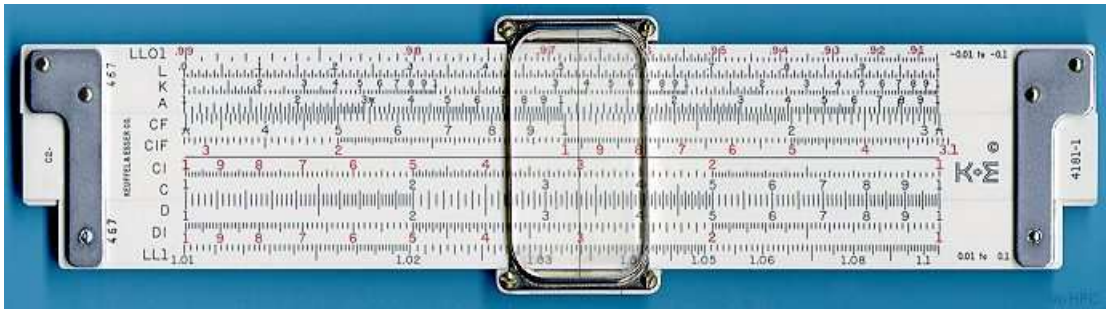


Figura 1.3: Régua de cálculo.

Leonardo da Vinci[9] e Wilhelm Schickard[10], 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[11] 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[12], co-inventor do *Cálculo* juntamente com Isaac Newton[13], planejou construir uma calculadora[14] 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[15], o telégrafo e a eletricidade, também investigada por Benjamin Franklin[16].

[5] Peter Henlein (1479 – 1542), nascido na Alemanha, serralheiro, artesão e relojoeiro.

[6] William Gilbert (1544 – 1603), nascido na Inglaterra, físico e filósofo. É lhe atribuído o título de "Pai da Engenharia Elétrica" ou "Pai da Eletricidade e Magnetismo".

[7] John Napier (1550 – 1617), nascido na Escócia, matemático, físico, astrônomo e 8º Lorde de Merchistoun.

[8] William Oughtred (1575 – 1660), nascido na Inglaterra, matemático.

[9] Leonardo di ser Piero da Vinci (1452 – 1519), nascido na Itália, erudito, arquiteto, anatomista, escultor, engenheiro, inventor, matemático, músico, cientista e pintor.

[10] Wilhelm Schickard (1592 – 1635), nascido na Alemanha, erudito.

[11] Blaise Pascal (1623 – 1662), nascido na França, matemático, físico e filósofo.

[12] Gottfried Wilhelm Leibniz (1646 – 1716), nascido na Alemanha, erudito.

[13] Sir Isaac Newton (1643 – 1727), nascido na Inglaterra, físico, matemático, astrônomo e filósofo.

[14] Ele chamou sua máquina de *stepped reckoner*.

[15] Daniel Gabriel Fahrenheit (1686 – 1736), nascido na Alemanha, físico e engenheiro.

[16] 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.

Joseph Jacquard[17] 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 desemprego de operários da indústria têxtil.

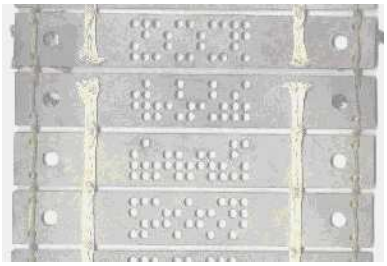


Figura 1.4: Cartões perfurados da máquina de Jacquard.

A *Máquina de Diferenças*[18], projetada por Charles Babbage[19], 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*[20] 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[21] 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.

O livro "Uma Investigação das Leis do Pensamento sobre as quais são fundadas as Teorias Matemáticas da Lógica e das Probabilidades" (*An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*), publicado em 1854 por George Boole[22], tornou-se muito influente cerca de 70 anos após sua publicação, quando Claude Shannon[23] e Victor Shestakov[24] independentemente descobriram uma interpretação para a sua álgebra em circuitos de reles eletromecânicos, demonstrando que a aplicação prática

[17] Joseph Marie Jacquard (1752 – 1834), nascido na França, alfaiate e inventor.
[18] Do inglês *Difference Engine*.
[19] Charles Babbage (1791 – 1871), nascido na Inglaterra, matemático, filósofo, engenheiro mecânico e precursor da Ciência da Computação.
[20] Do inglês *Analytic Engine*.
[21] 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.
[22] George Boole (1815 – 1864), nascido na Irlanda, matemático e filósofo.
[23] Claude Elwood Shannon (1916 – 2001), nascido nos Estados Unidos, engenheiro eletrônico e matemático, conhecido como "o pai da Teoria da Informação". Sua dissertação de mestrado já foi considerada como a mais importante dissertação de mestrado de todos os tempos.
[24] Victor Ivanovich Shestakov (1907 – 1987), nascido nos Rússia, lógico e engenheiro elétrico.

em eletricidade da álgebra de Boole poderia construir e resolver qualquer relação numérica e lógica. A álgebra de Boole, devido as suas aplicações, é ainda hoje ensinada nos cursos introdutórios das áreas exatas e tecnológicas, especialmente na Engenharia Elétrica, Engenharia Eletrônica e na Ciência da Computação.

Figura 1.5: Mesa de Hollerith.

Em 1890 o governo norte-americano tinha grande urgência de realizar o censo da população dos Estados Unidos e a previsão era de que, da maneira 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[25], que propôs a Mesa de Hollerith[26], uma máquina que consistia de um leitor de cartões perfurados que era sensível a pequenos buracos realizados nesses 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[27], a empresa foi renomeada INTERNATIONAL BUSINESS MACHINES – IBM em 1924.

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

[25] Herman Hollerith (1860 – 1929), nascido nos Estados Unidos, estatístico e empresário.
[26] Do inglês *Hollerith desk*.
[27] Thomas John Watson (1874 – 1956), nascido nos Estados Unidos, empresário.

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.

Interessante observar que em 1921 o dramaturgo Karel Čapek[28] introduziu e popularizou a palavra *robô* em sua peça *RUR – Rossumovi Univerzální Roboti*.

Konrad Zuse[29] 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 reles. 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, reles, 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 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[30]. 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.

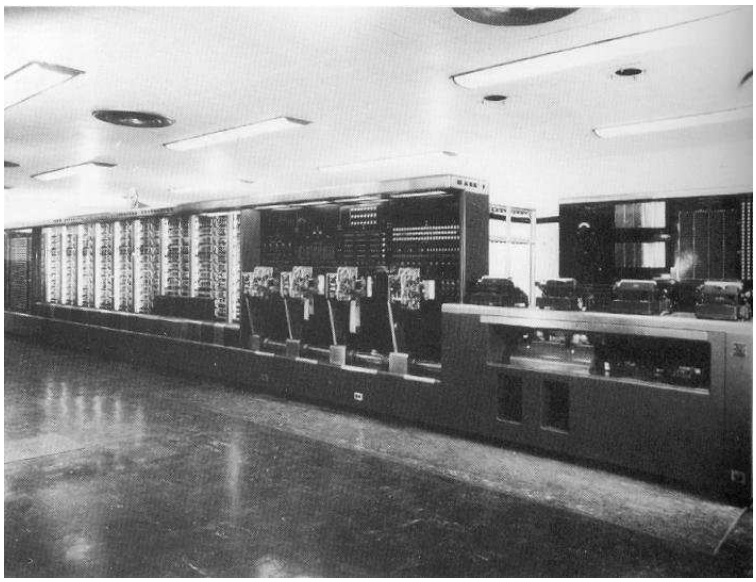


Figura 1.6: O computador eletromecânico Harvard Mark I.

[28] Karel Čapek ($1890 - 1938$), nascido no Império Austro-Húngaro, hoje República Tcheca, escritor.
[29] Konrad Zuse ($1910 - 1995$), nascido na Alemanha, engenheiro e pioneiro da computação.
[30] 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.

Uma das primeiras pessoas a programar o Mark I foi uma mulher, Grace Hopper[31], 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.

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[32] 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[33]. Seu trabalho, que estabelece as bases teóricas da Ciência da Computação, tem influências dos trabalhos de David Hilbert[34] e Kurt Gödel[35]. 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 processado criminalmente por assumir sua homossexualidade, tendo então aceitado um tratamento com hormônios femininos e castração química, como uma pena alternativa à prisão. Turing morreu em 1952 antes de completar 41 anos por um aparente auto-administrado envenenamento por cianeto. Em 10 de setembro de 2009, o primeiro ministro britânico Gordon Brown fez um pedido formal público de desculpas pelo comportamento do governo britânico nessa questão (*Treatment of Alan Turing was "appalling"*). Diversos eventos celebrando o aniversário de 100 anos de seu nascimento estão programados pelo mundo em 2012 (*The Alan Turing Year*).

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

[32] Alan Mathison Turing (1912 – 1954), nascido na Inglaterra, matemático, lógico e criptoanalista.

[33] Esta afirmação é chamada Tese de Church-Turing. Uma linguagem de programação ou um computador abstrato é *Turing-completo* se satisfaz essa tese. Alonzo Church, um influente matemático e lógico norte-americano, foi seu orientador de doutorado.

[34] David Hilbert (1862 – 1943), nascido na Alemanha, reconhecido como um dos mais influentes matemáticos dos séculos XIX e XX.

[35] Kurt Gödel (1906 – 1978), nascido no Império Austro-Húngaro, lógico, matemático e filósofo.

Figura 1.7: Válvula.

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 reles em seu funcionamento e, portanto, não era eletrônico. Entre 1937 e 1942 foi concebido o Computador Atanasoff-Berry (ABC), projetado por John Atanasoff[36] e Clifford Berry[37], 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[38] 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. Após a guerra, Winston Churchill, então primeiro-ministro britânico, ordenou a destruição desses computadores. Em 1993 deu-se início no Museu Nacional da Computação em Bletchley Park, Manchester, Inglaterra, um trabalho de reconstrução de um dos 10 computadores Colossus usados pelos Aliados na Segunda Guerra Mundial. Esse trabalho ainda não foi concluído devido às enormes dificuldades encontradas pela falta do projeto detalhado da máquina e pela forma em que se deu tal destruição[39]. 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.

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[40] e John Eckert[41] 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.

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[42]. Financiado pelo exército dos Estados Unidos, o ENIAC tinha como principal tarefa

[36] John Vincent Atanasoff (1903 – 1995), nascido nos Estados Unidos, físico.
[37] Clifford Edward Berry (1918 – 1963), nascido nos Estados Unidos, engenheiro elétrico.
[38] Thomas Harold Flowers (1905 – 1998), nascido na Inglaterra, engenheiro.
[39] Veja mais detalhes em The past is the future at Bletchley Park.
[40] John William Mauchly (1907 – 1980), nascido nos Estados Unidos, físico e pioneiro da computação.
[41] John Adam Presper Eckert Jr. (1919 – 1995), nascido nos Estados Unidos, físico e pioneiro da computação.
[42] Diríamos que o ENIAC era um computador com velocidade de 100 KHz.

Figura 1.8: O primeiro computador eletrônico ENIAC.

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.

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[43] 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[44] 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.

[43] Margittai Neumann János Lajos (1903 – 1957), nascido na Áustria-Hungria, matemático e erudito. John von Neumann tem muitas contribuições em diversas áreas. Como matemático, foi assistente de David Hilbert. 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.

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

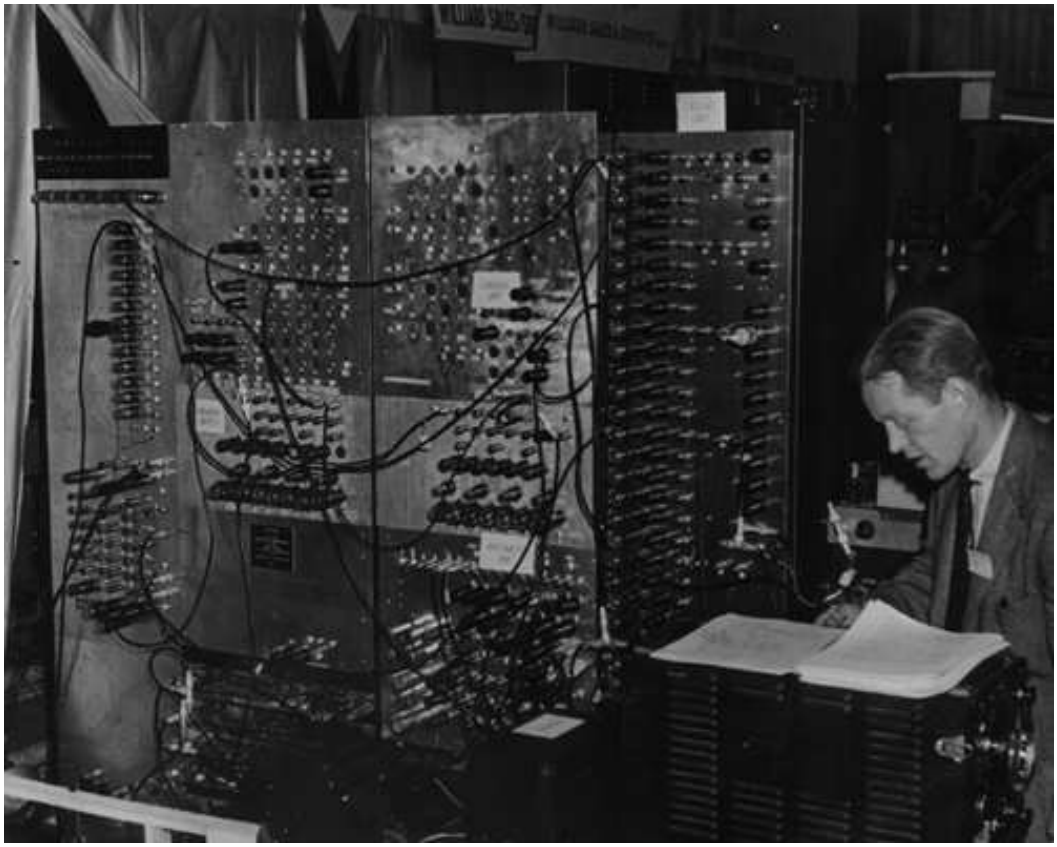


Figura 1.9: O EDVAC.

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 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

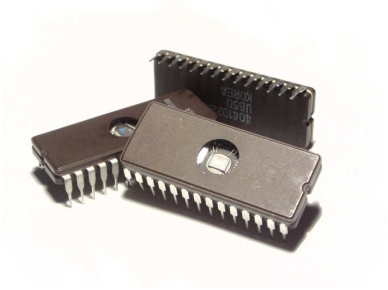


Figura 1.10: Circuito integrado.

Kilby[45] da TEXAS INSTRUMENTS e por Robert Noyce[46] da FAIRCHILD SEMICONDUCTOR. Um circuito integrado é um circuito eletrônico miniaturizado, consistindo de dispositivos semicondutores, produzido na superfície de um fino substrato de material semicondutor. É importante observar que a partir dos anos 50 a IBM praticamente engoliu as empresas concorrentes e tornou-se a maior empresa de produção de computadores da época. Devido a esse domínio, o governo norte-americano aplicou procedimentos antitruste contra a IBM de 1969 a 1982.

De 1960 a 1980

Figura 1.11: 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[47] inventou e patenteou o primeiro *mouse* de computador. No mesmo ano é desenvolvido o código padrão norte-americano para troca de informações[48] para padronizar a troca de dados entre computadores. Em 1967, a IBM cria o primeiro disco flexível[49] para armazenamento de dados. Além disso, as linguagens de programação BASIC, FORTRAN e COBOL foram propostas nessa década. Com o desenvolvimento da indústria de computadores fervilhando, Gordon Moore[50], 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"[51]. 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.

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 ma-

[45] 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.

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

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

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

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

[50] Gordon Earle Moore (1929–), nascido nos Estados Unidos, químico e físico.

[51] A afirmação original de Gordon Moore pode ser encontrada no artigo intitulado "*Cramming more components onto integrated circuits*" da *Electronics Magazine*, número 19, de abril de 1965.

neira é conhecida como compartilhamento de tempo[52] 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[53] 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.

Figura 1.12: Máquinas perfuradoras de cartões IBM 26.

A década de 70 também foi muito prolífica no desenvolvimento da computação. Linguagens de programação usadas até hoje foram projetadas nessa época como Pascal, concebida em 1971 por Niklaus Wirth[54], e C, proposta em 1972 por Dennis Ritchie[55]. Avanços no projeto de dispositivos também foram obtidos. Datam dessa época o CD[56], 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[57] e Paul Allen[58]. No mesmo ano, Steve Wozniak[59] e Steve Jobs[60] fundam a APPLE COMPUTERS, também uma das maiores empresas fabricantes de computadores do mundo.

[52] Do inglês *time sharing processing*.
[53] Do inglês *batch processing*.
[54] Niklaus E. Wirth (1934−), nascido na Suíça, engenheiro eletrônico e cientista da computação.
[55] Dennis MacAlistair Ritchie (1941−), nascido nos Estados Unidos, físico e matemático.
[56] Do inglês *compact disk*.
[57] William Henry Gates III (1955−), nascido nos Estados Unidos, empresário.
[58] Paul Gardner Allen (1953−), nascido nos Estados Unidos, empresário.
[59] Stephan Gary Wozniak (1950−), nascido nos Estados Unidos, engenheiro da computação e filantropo.
[60] Steven Paul Jobs (1955−), nascido nos Estados Unidos, empresário.

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[61], 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 1.13: 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 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.

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, 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.

[61] Do inglês *personal computer* – PC.

Figura 1.14: O computador pessoal Apple I.

Ao mesmo tempo que a indústria dos computadores avançava, progressos notáveis também foram obtidos nos fundamentos da Computação, em grande parte devido a Donald Knuth[62], a Stephen Cook[63] e a Leonid Levin[64]. Knuth é autor do livro "A Arte da Programação de Computadores" (*The Art of Computer Programming*), uma série em vários volumes, que contribuiu para o desenvolvimento e sistematização de técnicas matemáticas formais para a análise rigorosa da complexidade computacional dos algoritmos e a popularização da notação assintótica. É conhecido, a partir desses trabalhos, como "o Pai da Análise de Algoritmos". Além das contribuições fundamentais em Teoria da Computação, Knuth é o criador do sistema processador de textos TEX. Cook e Levin são conhecidos por terem demonstrado formalmente a existência do primeiro problema NP-completo: o problema da satisfabilidade de fórmulas booleanas. Isso significa que qualquer problema na classe NP pode ser transformado de maneira eficiente, isto é, pode ser reduzido em tempo polinomial por uma máquina de Turing determinística, para o problema de determinar se uma fórmula booleana é satisfatível. Uma conseqüência importante do Teorema de Cook-Levin é que se existe um algoritmo determinístico eficiente, ou de tempo polinomial, que soluciona o problema da satisfabilidade de fórmulas booleanas, então existe um algoritmo determinístico de tempo polinomial que soluciona *todos* os problemas da classe NP. A contribuição desses dois pesquisadores tem um significado fundamental para toda a área da Computação e pode ser traduzida informal e simplificadamente como a necessidade que temos de saber se um problema é ou não computável de forma eficiente em uma máquina geral e abstrata. Esse problema é conhecido como o "Problema P versus NP", um dos sete *Millenium Problems*[65] do *Clay Mathematics Institute of Cambridge*.

[62] Donald Ervin Knuth (1938−), nascido nos Estados Unidos, cientista da computação, Professor Emérito de Arte de Programação de Computadores da Universidade de Stanford.

[63] Stephen Arthur Cook (1939−), nascido nos Estados Unidos, matemático e cientista da computação, *University Professor* da Universidade de Toronto.

[64] Leonid Anatolievich Levin (1948−), nascido na Ucrânia, cientista da computação.

[65] Este problema tem um prêmio estabelecido de 1 milhão de dólares para o(s) pesquisador(es) que o soluciona-

Figura 1.15: O computador pessoal IBM PC.

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.

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[66]. 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

rem. Veja mais informações em P *vs.* NP *problem*.

[66] Também conhecido como *notebook*.

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[67] propõe um sistema de hipertexto que é o primeiro impulso da Rede Mundial de Computadores[68]. 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.

Richard Stallman[69], 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[70] 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.

Figura 1.16: O símbolo do Linux.

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 os utilizavam. 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[71] e David Filo[72] fundam a Yahoo!, uma empresa de serviços de Internet que engloba um portal de Internet, uma ferra-

[67] Sir Timothy John Berners-Lee (1955−), nascido na Inglaterra, físico.
[68] Do inglês *World Wide Web* – WWW.
[69] Richard Matthew Stallman (1953−), nascido nos Estados Unidos, físico, ativista político e ativista de software.
[70] Linus Benedict Torvalds (1969−), nascido na Finlândia, cientista da computação.
[71] Jerry Chih-Yuan Yang (1968−), nascido em Taiwan, engenheiro elétrico e empresário.
[72] David Filo (?−), nascido nos Estados Unidos, engenheiro da computação e empresário.

menta 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 da ferramenta de busca Google são desenvolvidos em 1996 como um projeto de pesquisa dos alunos de doutorado Larry Page[73] e Sergey Brin[74] 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[75] 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[76], 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. A década de 90 também é marcada pela união, compra e venda de diversas empresas de computadores. A indústria de jogos para computadores pessoais também teve seu crescimento acentuado nessa época.

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

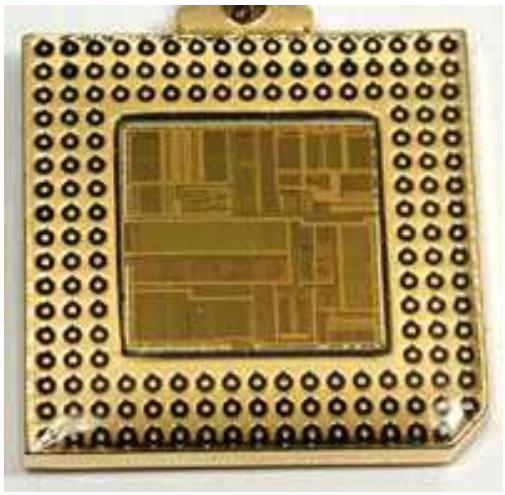


Figura 1.17: Processador Pentium.

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

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.

De 2000 até hoje

O ano de 2000 inicia receoso para governos e 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 com Bill Gates deixando o cargo de Executivo-Chefe.

A MICROSOFT lança o Windows XP em 2001, o Windows Vista, em janeiro de 2007 e recentemente o Windows 7 em 2009. A APPLE disponibiliza o MacOS X 10.0 em 2001 e seguidas novas atualizações até 2009, com o MAC OS X 10.6. 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, o INTEL Core 2 Duo e o INTEL Core 2 Quad de 2008. 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. Em seguida, lança a série K10 chamada Phenom, com 2, 3 e 4 núcleos. 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

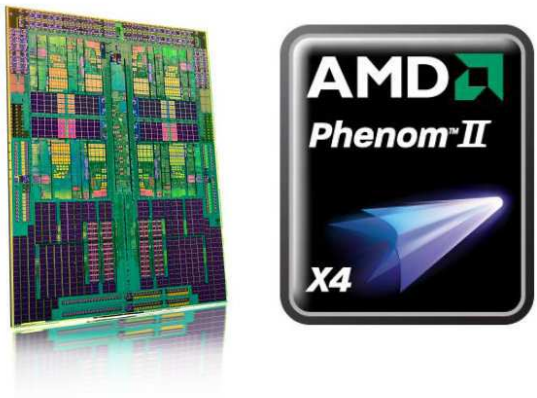


Figura 1.18: Phenom X4 da AMD.

500 supercomputadores mais rápidos do mundo, a TOP500, de novembro de 2009, há um amplo domínio de mais de 80% de uso de processadores com núcleos múltiplos da INTEL nesses computadores. Também interessante notar que esses supercomputadores usam alguma distribuição do sistema operacional Linux em mais de 82% deles, sendo que 99% usam um sistema operacional baseado no Unix, como o Linux, e apenas 1% usam um sistema operacional baseado no Windows.

Figura 1.19: Cray XT5-HE Opteron Six Core 2.6 GHz, o computador mais rápido do mundo na lista TOP500 de 2009, instalado no *Oak Ridge National Laboratory*. Usa o processador AMD x86_64 Opteron Six Core 2600 MHz de 10,4 GFlops e o sistema operacional Linux.

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

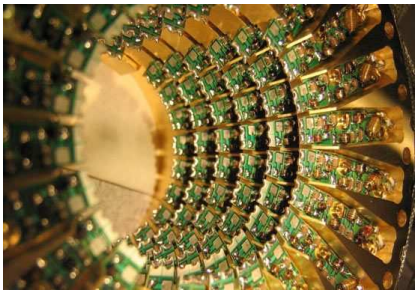


Figura 1.20: Computador quântico.

Exercícios

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

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 que têm papel importante na História da Computação: Augusta Ada Byron e Grace Murray Hopper.

1.4 Leia o verbete sobre Alan Mathison Turing, "o Pai da Ciência da Computação", na Wikipedia. Aproveite para ler o pedido formal de desculpas feito em 2009 pelo Primeiro-ministro da Inglaterra, Gordon Brown, e para vasculhar a página com as festividades do centenário de seu nascimento em 2012.

1.5 Leia uma apresentação simples e motivadora do problema P versus NP na página do *Clay Mathematics Institute*.

1.6 Leia o verbete *Cook-Levin Theorem* na Wikipedia e tente entender informalmente o que ele realmente quer dizer.

1.7 Visite a página *GNU Operating System* e leia o verbete GNU na Wikipedia e veja como este sistema vem se desenvolvendo ao longo do tempo.

1.8 Visite as páginas *Free Software Foundation*, Fundação Software Livre América Latina e o portal de software livre (*Portal: Free Software*) da Wikipedia. Procure saber quais projetos mantidos por esta fundação mais lhe interessou.

1.9 Leia o verbete Linux na Wikipedia, o sistema operacional que usamos na universidade.

1.10 Visite a página TOP500 para obter informações dos computadores mais rápidos hoje existentes.

AULA 2

INTRODUÇÃO À COMPUTAÇÃO

Na aula 1 vimos um pequeno histórico sobre a Computação. Nesta aula, vamos mencionar brevemente como essas máquinas são organizadas internamente além de invertigarmos o conceito de algoritmo e programa. 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 [8, 9, 10].

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 [11][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 [12][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 [13], 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 [14][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.

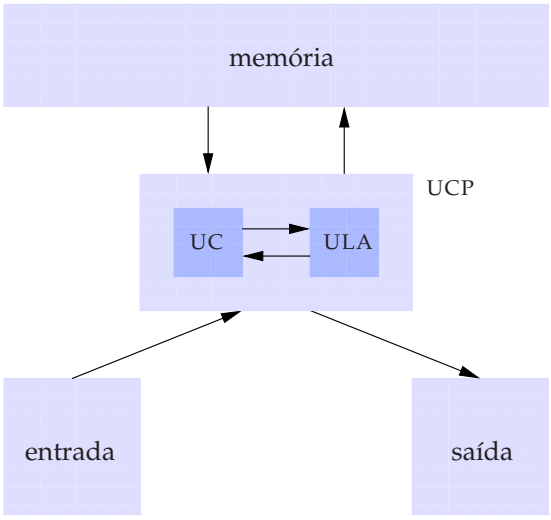


Figura 2.1: Arquitetura de von Neumann.

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 (ULA) e pela unidade de controle (UC). 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 UC coordena as tarefas da UCP.

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 rede 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.

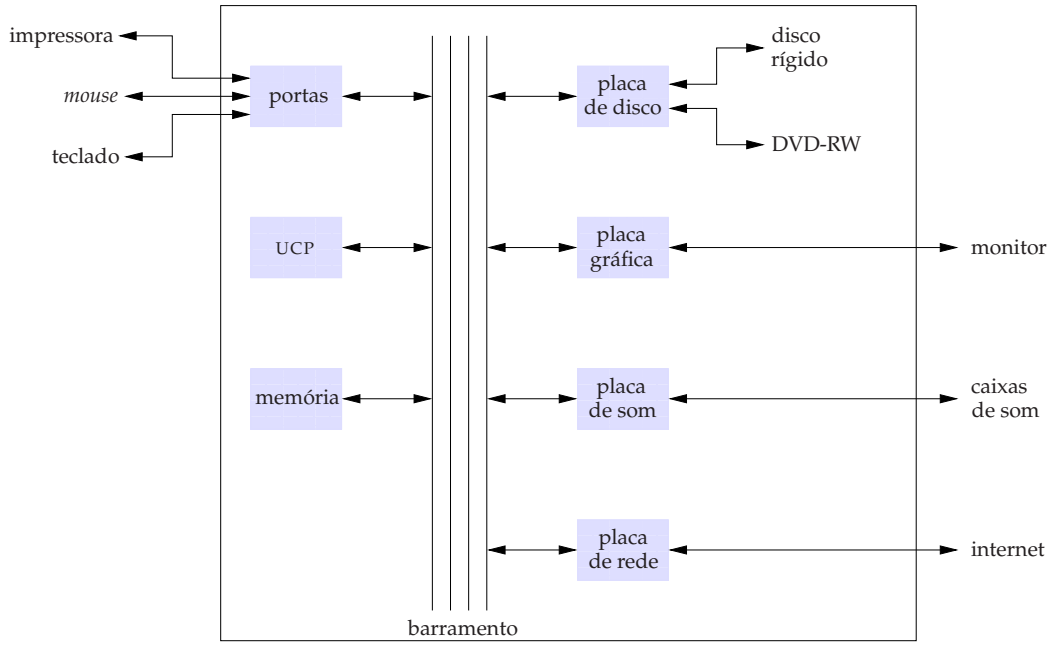


Figura 2.2: Uma outra ilustração da arquitetura de von Neumann.

É importante salientar que toda transmissão de dados, manipulação, armazenagem e recuperação é realizada de fato por um computador através de pulsos elétricos e magnéticos representando seqüências de dígitos binários, chamadas de bits, isto é, seqüências de 0's e 1's. Cada seqüência, por sua vez, é organizada em um ou mais bytes, que são grupos de oito bits. Neste contexto, as instruções e os dados manipulados pelo computador nada mais são do que seqüências de bytes que possuem significado para o computador.

2.3 Algoritmos e programas

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[9] em 1928. O termo "algoritmo", como mencionamos na aula 1, tem sua origem no cientista persa Muḥammad ibn Mūsā al-Khwārizmī[10]. A latinização de seu nome deu origem ao termo algoritmo e algarismo na língua portuguesa. 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-arábico de numeração no Oriente Médio e na Europa.

[9] David Hilbert (1862 – 1943), nascido na Alemanha, reconhecido como um dos mais influentes matemáticos dos séculos XIX e XX.

[10] Abū Abdallā Muḥammad ibn Mūsā al-Khwārizmī (780 – 850), nascido na Pérsia, matemático, astrônomo e geógrafo.

Um algoritmo é então um conjunto de idéias abstratas para solução de um problema. Mais formalmente, podemos estabelecer que um algoritmo é uma seqüência finita de instruções para resolver um problema, a qual possui as seguintes propriedades:

- Garantia de término: o problema a ser resolvido possui condições específicas que, quando satisfeitas, a execução do algoritmo é encerrada e o problema é então tido como "resolvido". Além disso, estas condições devem ser satisfeitas após uma quantidade finita de tempo, a ser contado a partir do início da execução do algoritmo;
- Exatidão: a intenção de cada instrução no algoritmo deve ser suficientemente clara, de forma que não haja ambigüidade na interpretação da intenção.
- Efetividade: cada instrução deve ser básica o suficiente para ser executada, pelo menos em princípio, por qualquer agente usando apenas lápis e papel.

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. Inicialmente, tomamos dois números inteiros a e b. 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). Esquematicamente, podemos escrever a seguinte seqüência de instruções para resolver o problema:

PASSO 1 Chame o maior número de a e o menor de b

PASSO 2 Divida a por b e chame o resto de r

PASSO 3 Se r é igual a zero então o máximo divisor comum é igual a b e a execução das instruções encerra aqui. Caso contrário, siga para a próxima instrução.

PASSO 4 Atribua o valor de b a a e o valor de r a b

PASSO 5 Volte para o passo 2

É importante destacar que essa seqüência de instruções sempre obtém um valor $r = 0$ em algum dos passos e, portanto, sempre termina. Além disso, a seqüência de instruções está correta, já que sempre produz uma resposta correta para um par de números inteiros fornecidos como entrada. Dessa forma, acabamos de descrever um algoritmo para solução do problema do máximo divisor comum, o algoritmo de Euclides.

Como era de se esperar, nem toda seqüência de instruções para resolver um determinado problema pode ser considerada um algoritmo. Por exemplo, se a instrução "Divida x por y se todo número inteiro par maior que 2 é a soma de dois números primos" estiver presente na seqüência, ela só poderá ser executada se soubermos se a proposição "todo número inteiro par maior que 2 é a soma de dois números primos" é verdadeira ou falsa. Entretanto, esta proposição, conhecida como conjectura de Goldbach[11], foi proposta em 1742 e continua sem solução até hoje. Logo, nossa instrução não pode ser executada por qualquer agente hoje em dia e, portanto, não pode fazer parte de um algoritmo.

[11] Veja mais em *Goldbach's conjecture*.

Um outro exemplo de instrução que não pode fazer parte de um algoritmo é "Escreva todos os números ímpares". Neste caso, temos uma instrução que não pode ser executada porque a execução nunca terminará, apesar de sabermos exatamente como determinar os números ímpares. Observe que se modificássemos a instrução para "Escreva todos os números ímpares menores do que 100", ela poderia, perfeitamente, fazer parte de um algoritmo.

Um problema para o qual existe uma solução na forma de algoritmo é dito um problema algorítmico. O problema de encontrar o máximo divisor comum de dois números naturais quaisquer é, portanto, um problema algorítmico. Problemas algorítmicos, em geral, possuem muitas ocorrências. Por exemplo, para o problema de encontrar o máximo divisor comum de dois números naturais, cada ocorrência é uma dupla distinta de números naturais cujo máximo divisor comum queremos encontrar. Um algoritmo é dito correto quando ele sempre termina e produz a resposta correta para todas as ocorrências de um dado problema.

Algoritmo, então, pode ser imaginado como a especificação de um processo "mecânico" que, quando executado, leva-nos à solução de algum problema. Embora o termo algoritmo esteja relacionado intimamente com Computação, algoritmos têm sido parte de nossas vidas desde a primeira vez que uma pessoa explicou para outra como fazer alguma coisa. As pessoas utilizam algoritmos quando seguem receitas culinárias ou instruções para programar um tocador de músicas. Entretanto, nem todo algoritmo pode ser executado por um computador. Um computador pode executar apenas aqueles algoritmos cujas instruções envolvam tarefas que ele possa entender e executar. Este não é o caso, por exemplo, de instruções como "corte o filé em cubos" e "ligue o tocador de música".

Computadores executam algoritmos que manipulam apenas dados e não coisas físicas, tais como filé e tocador de música. A execução de um algoritmo por um computador é denominada processamento de dados e consiste de três partes: uma entrada, um processo ou processamento e uma saída. A entrada é um conjunto de informações que é requisitada para que as instruções do algoritmo possam ser executadas. O processamento é a seqüência de instruções que compõe o algoritmo. E a saída é o resultado obtido com a execução do processo para a entrada fornecida. Por exemplo, a entrada e a saída para uma computação do algoritmo para o problema de encontrar o máximo divisor comum de dois números naturais são, respectivamente, dois números naturais e o máximo divisor comum entre eles.

Quando escrevemos algoritmos para serem executados por computador, temos de fazer algumas suposições sobre o modelo de computação *entrada–processamento–saída*. A primeira delas é que, a fim de realizar qualquer computação, o algoritmo deve possuir um meio de obter os dados da entrada. Esta tarefa é conhecida como leitura da entrada. A segunda, é que o algoritmo deve possuir um meio de revelar o resultado da computação. Isto é conhecido como escrita dos dados da saída. Todo e qualquer computador possui dispositivos através dos quais a leitura e a escrita de dados são realizadas.

2.3.1 Algoritmos e resolução de problemas

Todo algoritmo está relacionado com a solução de um determinado problema. Portanto, construir um algoritmo para um dado problema significa, antes de mais nada, encontrar uma solução para o problema e descrevê-la como uma sequência finita de ações.

A tarefa de encontrar a solução de um problema qualquer é, em geral, realizada de forma empírica e um tanto quanto desorganizada; ocorrem vários procedimentos mentais, dos quais

raramente tomamos conhecimento. A organização do procedimento de resolução de problemas é extremamente desejável, pois somente assim podemos verificar onde o procedimento não está eficiente. Identificadas as deficiências, procuramos formas de corrigi-las e, consequentemente, aumentamos a nossa capacidade de resolver problemas.

A capacidade para resolver problemas pode ser vista como uma habilidade a ser adquirida. Esta habilidade, como qualquer outra, pode ser obtida pela combinação de duas partes:

- Conhecimento: adquirido pelo estudo. Em termos de resolução de problemas, está relacionado a que táticas, estratégias e planos usar e quando usar;
- Destreza: adquirida pela prática. A experiência no uso do conhecimento nos dá mais agilidade na resolução de problemas.

Independente do problema a ser resolvido, ao desenvolvermos um algoritmo devemos seguir os seguintes passos:

- Análise preliminar: entender o problema com a maior precisão possível, identificando os dados e os resultados desejados;
- Solução: desenvolver um algoritmo para o problema;
- Teste de qualidade: executar o algoritmo desenvolvido com uma entrada para a qual o resultado seja conhecido;
- Alteração: se o resultado do teste de qualidade não for satisfatório, altere o algoritmo e submeta-o a um novo teste de qualidade;
- Produto final: algoritmo concluído e testado, pronto para ser aplicado.

2.3.2 Resolução de problemas e abstração

Talvez, o fator mais determinante para o sucesso em resolver um problema seja a abstração. Abstração[12] é alguma coisa independente de qualquer ocorrência particular ou o processo de identificar certas propriedades ou características de uma entidade material e usá-las para especificar uma nova entidade que representa uma simplificação da entidade da qual ela foi derivada. Esta "nova entidade" é o que chamamos de abstração.

Para entender o papel da abstração na resolução de problemas, considere a seguinte ocorrência de um problema que lembra nossos tempos de criança:

> Maria tinha cinco maçãs e João tinha três. Quantas maçãs eles tinham juntos?

Provavelmente, um adulto resolveria este problema fazendo uma abstração das maçãs como se elas fossem os números 5 e 3 e faria a soma de tais números. Uma criança poderia imaginar as cinco maçãs de Maria como cinco palitinhos e as três de João como três palitinhos. Daí, faria uma contagem dos palitinhos para chegar à solução. Em ambos os casos, os elementos do problema foram substituídos por outros (números e palitinhos) e a solução foi encontrada através da manipulação dos novos elementos.

[12] Segundo o *Webster's New Dictionary of American Language*.

O processo de abstração pode ser visto como constando de níveis. Isto diz respeito ao grau de simplificação de uma abstração. Por exemplo, se a planta de uma casa é entregue a um experiente mestre-de-obras sem menção de, por exemplo, como fazer uma mistura ideal de cimento para fundação, provavelmente, isso não será um problema para ele. Entretanto, se a mesma planta da casa for entregue a um advogado, ele certamente não saberá construir a casa. Isso quer dizer que a planta da casa dada ao advogado deveria conter maiores detalhes do processo de construção da casa do que aquela dada para o mestre-de-obras. Aqui, a planta da casa é uma abstração e o nível de detalhe da planta é proporcional ao nível de simplificação da abstração.

Algoritmos bem projetados são organizados em níveis de abstração, pois um mesmo algoritmo deve ser entendido por pessoas com diferentes graus de conhecimento. Quando um algoritmo está assim projetado, as instruções estão organizadas de tal forma que podemos entender o algoritmo sem, contudo, ter de entender os detalhes de todas as instruções de uma só vez. Para tal, o processo de construção de algoritmos conta com ferramentas, tais como módulos, que agrupam instruções que realizam uma determinada tarefa no algoritmo, independente das demais, tal como fazer a leitura da entrada, dispensando-nos de entender o detalhe de cada instrução separadamente, mas sim fornecendo-nos uma visão da funcionalidade do grupo de instruções.

2.3.3 Algoritmos e computadores

Após o desenvolvimento de um algoritmo para solução de algum problema, o próximo passo é informá-lo a um computador. No entanto, como mencionamos, os computadores não compreendem, e portanto são incapazes de obedecerem e executarem, instruções em uma linguagem natural como a língua portuguesa. 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. Um programa em linguagem de máquina poderia se parecer com a seguinte seqüência de bytes:

01000011 00111010 00111011 01000001 00101001 01000100

Como a maioria dos problemas resolvidos por computadores não envolve o conhecimento dos dispositivos internos do computador, a programação em linguagem de máquina é, na maioria das vezes, inadequada, pois o desenvolvedor perde mais tempo com os detalhes da máquina do que com o próprio problema. Entretanto, para programas onde o controle de tais dispositivos é essencial, o uso de linguagem de máquina é mais apropriado ou, às vezes, indispensável.

O próximo passo na evolução das linguagens de programação foi a criação da linguagem montadora ou *assembly*. Nesta linguagem, as instruções da linguagem de máquina recebem nomes compostos por letras, denominados mnemônicos, que são mais significativos para nós humanos. Por exemplo, a instrução na linguagem montadora do processador INTEL 8088 que

soma o valor no registrador CL com o valor no registrador BH e armazena o resultado em CL é dada por:

ADD CL,BH

Esta instrução equivale a seguinte sequência de dois bytes na linguagem de máquina do processador INTEL 8088:

00000010 11001111

Para que o computador pudesse executar um programa escrito em linguagem montadora foi desenvolvido um compilador denominado montador ou *assembler*, o qual realiza a tradução automática de um código escrito em linguagem montadora para o seu correspondente em linguagem de máquina.

O sucesso da linguagem montadora animou os pesquisadores a criarem linguagens em que a programação fosse realizada através de instruções na língua inglesa, deixando para o próprio computador a tarefa de traduzir o código escrito em tais linguagens para sua linguagem de máquina. Isto foi possível devido à criação de compiladores mais complexos do que os montadores.

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, Lisp, Python, entre outras, são linguagens de programação de alto nível. Há ainda uma classificação das linguagens de programação de alto nível pela forma que diferem nos conceitos e abstrações usados para representar os elementos de um programa, tais como objetos, funções, variáveis, restrições, etc, e os passos que compõem uma computação, como por exemplo a atribuição, a avaliação, o fluxo de informações, etc. A essa classificação se dá o nome de pardigma de programação. Dentre os existentes e mais comuns, podemos citar o paradigma estruturado, orientado a objetos e funcional.

Infelizmente, programas escritos em uma linguagem programação de alto nível 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 2.3.

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

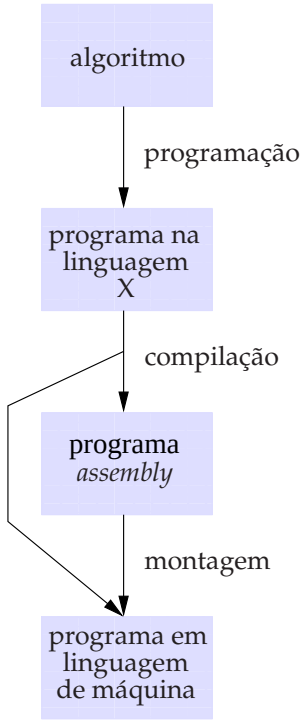


Figura 2.3: Passos desde a concepção de um algoritmo até o programa executável por um computador.

Exercícios

2.1 Leia o verbete *Computer* na Wikipedia, que traz muitas informações sobre essas máquinas.

2.2 Leia o verbete *von Neumann architecture* na Wikipedia, sobre o modelo de organização interna de computadores mais conhecido.

2.3 Dê uma olhada no trabalho de John von Neumann na referência [14][13], sobre a "arquitetura de von Neumann".

2.4 Dê uma olhada no trabalho de Alan Turing na referência [11][14] com a descrição da máquina de Turing.

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

2.6 Sobre a linguagem de programação que adotaremos neste curso, visite as páginas *C History* e *C Programming Language* e leia o verbete *C (programming language)* na Wikipedia.

[13] Artigo disponível na seção "Bibliografia complementar" na página da disciplina no Moodle.
[14] 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 parte prática 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.

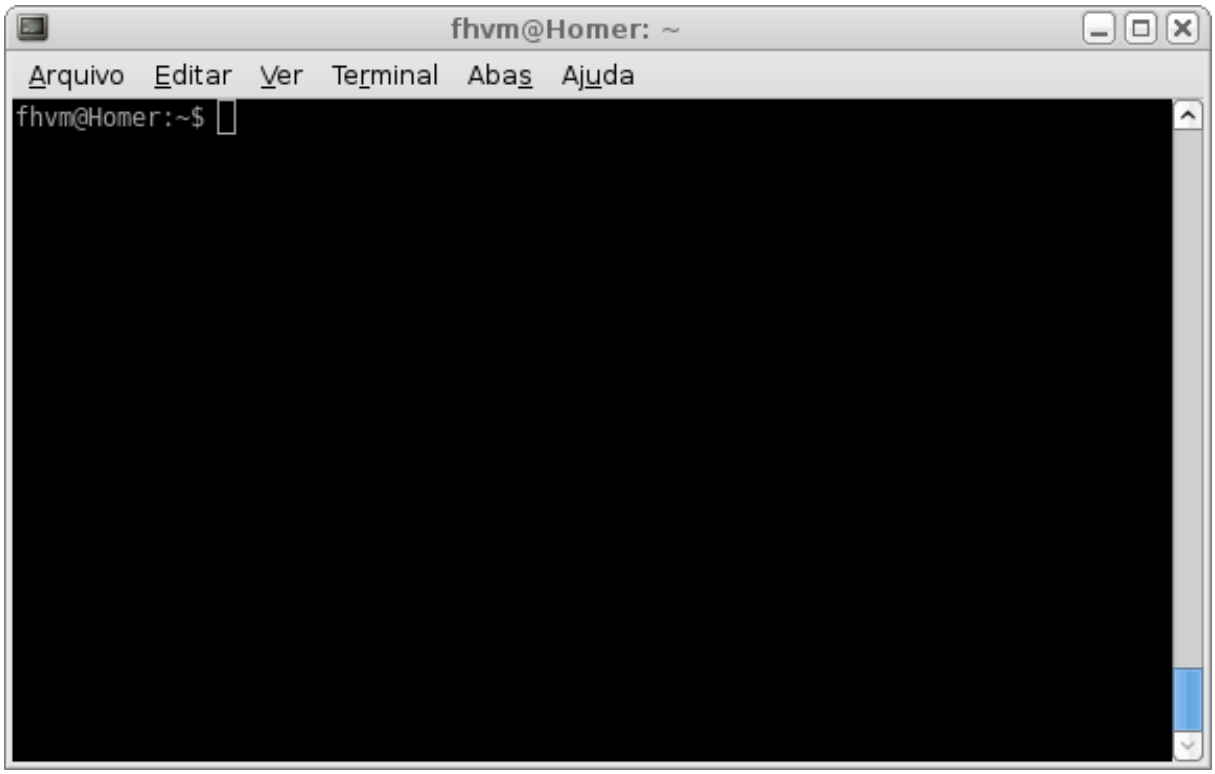


Figura 3.1: Uma interface, ou janela, de linhas de comandos do sistema de janelas GNOME, chamada no computador `Homer` pelo usuário `fhvm`. Esses nomes compõem o sinal de pronto do sistema.

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 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 Algoritmos e Programação I (e II). 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
```

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

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

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.

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 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.

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: **`algprogi`**, **`ftc`** e **`temp`**;
- entre no diretório **`algprogi`**;
- crie quatro subdiretórios dentro do diretório **`algprogi`** com os seguintes nomes: **`teoricas`**, **`praticas`**, **`trabalhos`** e **`provas`**;
- entre no diretório **`praticas`**;
- 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 **`algprogi`** e **`temp`**;
- verifique o conteúdo dos diretórios **`algprogi/praticas`**, **`algprogi`** 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 **`algprogi/praticas`**;

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

- 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 ~/algprogi** 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 Algoritmos e Programação I até aqui;
- salve o arquivo no diretório **algprogi/teoricas**;
- 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, tomando contato com a estrutura seqüencial de programação e com funções de entrada e saída.

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.

```
#include <stdio.h>

int main(void)
{
   printf("Programar é bacana!\n");
   return 0;
}
```

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. Assim, é só digitar **a.out** (ou **./a.out**) 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 do tipo inteiro (**int**). Esses conceitos de função, parâmetros 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 corpo do programa. Em seguida, a próxima linha contém uma chamada à função **printf**:

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

Passamos um único **argumento** à função `printf`, a seqüência de símbolos ou de caracteres `"Programar é bacana!\n"`. Note que todos os símbolos dessa seqüência são mostrados na saída, a menos de `"` e `\n`. O caractere `\n` 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` 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 corpo do programa.

4.4 Próximo programa

Vamos digitar um próximo programa.

Programa 4.2: Segundo programa.

```
#include <stdio.h>

int main(void)
{
   int num1, num2, soma;

   num1 = 25;
   num2 = 30;
   soma = num1 + num2;
   printf("A soma de %d e %d é %d\n", num1, num2, soma);

   return 0;
}
```

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 **operador 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 con-

sulta 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 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 cadeia de caracteres de formatação e, a cada um deles está associado uma das variáveis `num1`, `num2` e `soma`, na ordem em que aparecem. Essas variáveis são os argumentos restantes da função `printf`.

4.5 Documentação

Uma boa documentação de um programa, conforme [15], 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 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 resultante.

Vejamos o programa 4.3, que nada mais é que a codificação do programa 4.2 com uma documentação do programa principal.

Programa 4.3: Segundo programa.

```
#include <stdio.h>

/* Este programa faz a adição de dois números inteiros
   fixos e mostra o resultado da operação na saída. */
int main(void)
{
   int num1, num2, soma;

   num1 = 25;
   num2 = 30;
   soma = num1 + num2;
   printf("A soma de %d e %d é %d\n", num1, num2, soma);

   return 0;
}
```

Ainda conforme [15], 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.

4.6 Entrada e saída

Veremos agora um exemplo de programa na linguagem C que usa pela primeira vez a função de leitura de dados `scanf`. Então, abra seu Emacs, pressione a seqüência de teclas `C-x C-f`, dê um nome para seu programa, como por exemplo `soma.c`, e digite o código apresentado no programa 4.4 a seguir.

Programa 4.4: Entrada e saída de dados.

```
#include <stdio.h>

/* Recebe dois números inteiros e imprime sua soma */
int main(void)
{
   int num1, num2, soma;

   printf("Informe um número: ");
   scanf("%d", &num1);
   printf("Informe outro número: ");
   scanf("%d", &num2);

   soma = num1 + num2;

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

   return 0;
}
```

Vamos olhar mais de perto este programa. Observe que, pela primeira vez, usamos a função `scanf` que, como mencionamos acima, é uma função da linguagem C responsável pela leitura de dados.

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 são `num1`, `num2` e `soma`. Os conteúdos desses compartimentos de memória podem *variar* durante a execução do programa. De fato, no momento da declaração, os compartimentos associados aos nomes `num1`, `num2` e `soma` já contêm, cada um, algum valor que é desconhecido e que referenciamos como **lixo**.

Em seguida, há duas chamadas de funções para escrita e leitura:

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

A função para escrita de dados `printf` já é bem conhecida, vista nesta e nas aulas anteriores. A função de leitura `scanf` tem dois argumentos: uma cadeia de caracteres de formatação `"%d"` e uma variável `num1` correspondente a este formato. Diferentemente da função `printf`, uma variável do tipo inteiro que é um argumento da função `scanf` deve *sempre* vir precedida com o símbolo `&`. Na instrução 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 uma instrução diferente:

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

Aqui temos uma **instrução/comando de atribuição** da linguagem C, definida pelo símbolo/operador `=`. Uma instrução 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. Esta instrução 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. O lado direito de uma instrução de atribuição pode conter expressões aritméticas tão complicadas quanto quiseremos. O lado esquerdo, ao contrário, é sempre descrito por uma única variável.

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`.

Veremos agora um segundo exemplo usando as funções de entrada e saída que acabamos de aprender. Este segundo exemplo é muito semelhante ao exemplo anterior. A diferença está apenas no formato da expressão aritmética: no primeiro exemplo, uma adição é realizada; neste segundo exemplo, uma multiplicação é realizada. O operador aritmético de multiplicação é o símbolo `*`. Então, informe, compile e execute o seguinte programa 4.5.

Programa 4.5: 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

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 4.6?

Programa 4.6: Programa do exercício 4.2.

```
#include <stdio.h>
int main(void)
{
   printf("Alô! ");
   printf("Alô! ");
   printf("Tem alguém aí?");
   printf("\n");
   return 0;
}
```

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 4.7 está correto. Em caso negativo, liste os erros de digitação que você encontrou.

Programa 4.7: Programa do exercício 4.4.

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

4.5 Escreva 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.

4.6 Escreva 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.

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

4.8 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 5

ESTRUTURAS CONDICIONAIS

Até aqui, implementamos nossos primeiros programas e aprendemos a trabalhar com as funções de entrada e saída da linguagem C. Dessa forma, até esta aula, fizemos uso constante da estrutura seqüencial de programação. Nessa estrutura, um programa executa uma instrução após a outra, em geral linha após linha, sem que ocorra desvio algum nesse fluxo de execução.

Nesta aula, veremos estruturas condicionais na linguagem C, tendo oportunidade de implementá-las e de visualizar o funcionamento de programas que as contêm. Do mesmo modo, ressaltamos que uma estrutura condicional é uma ferramenta importante que auxilia o desenvolvedor/programador a decidir que trechos de um programa devem ser executados ou não, dependendo da avaliação de uma condição expressa.

Esta aula é baseada nas referências [15, 16].

5.1 Estrutura condicional simples

Como vimos na aula 4, nossa linguagem de programação é usada para executar uma seqüência de comandos ou instruções tais como uma leitura de dados, uma impressão de dados e atribuição de valores a variáveis. Além de ter esse poder, os programas também podem ser usados para tomar decisões. Na linguagem C, uma decisão 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) {
    instrução1;
    instrução2;
    ⋮
    instruçãon;
}
```

para $n \geqslant 1$. Na linguagem C, no caso em que $n = 1$, isto é, se o bloco de instruções da estrutura condicional contém uma única instrução, então os símbolos delimitadores de bloco `{` 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*, a instrução, ou o bloco de instruções, será executada(o); caso contrário, será ignorada(o).

Vejamos um pequeno exemplo no programa 5.1.

Programa 5.1: Estrutura condicional **`if`**.

```
/* Recebe um número inteiro positivo que representa uma idade
   e emite uma mensagem na saída se a idade é inferior a 30 */
#include <stdio.h>

int main(void)
{
   int idade;

   printf("Quantos anos você tem?");
   scanf("%d", &idade);

   if (idade < 30)
      printf("Puxa! Você é bem jovem!\n");

   printf("Até breve!\n");

   return 0;
}
```

5.2 Estrutura condicional composta

Uma estrutura condicional composta na linguagem C tem o seguinte formato geral:

```
if (condição) {
   instrução1;
   instrução2;
   ⋮
   instruçãom;
}
else {
   instrução1;
   instrução2;
   ⋮
   instruçãon;
}
```

para $m, n \geqslant 1$. Como nas estruturas condicionais simples, se $m = 1$ ou $n = 1$, os símbolos delimitadores de bloco `{` e `}` são opcionais. A linguagem C tem como regra sempre envolver um bloco de instruções com os delimitadores `{` e `}`. Exceção se faz quando o bloco de instruções contém uma única instrução: neste caso, a inclusão dos delimitadores é opcional.

Uma decisão é tomada de acordo com a avaliação da *condição*, que é uma expressão lógica: caso o resultado dessa avaliação seja *verdadeiro*, o primeiro bloco de instruções será executado e, ao término desse bloco, a instrução da próxima linha após a estrutura condicional composta será executada. Caso contrário, isto é, se o resultado da avaliação da expressão lógica é *falso*,

o segundo bloco de instruções, logo após a palavra-chave **`else`**, será executado. Ao final da execução das instruções desse bloco, a instrução da próxima linha após a estrutura condicional composta será executada. Veja um exemplo no programa 5.2.

Programa 5.2: Estrutura condicional **`if-else`**.

```
#include <stdio.h>

/* Recebe um número inteiro e emite uma mensagem de acordo com esse número */
int main(void)
{
   int idade;

   printf("Quantos anos você tem? ");
   scanf("%d", &idade);

   if (idade < 30)
      printf("Puxa! Você é bem jovem!\n");
   else
      printf("Puxa! Você já é velhinho!\n");
   printf("Até breve!\n");

   return 0;
}
```

5.2.1 Aplicação: troca de conteúdos

O programa 5.3 recebe dois números inteiros quaisquer e os imprime em ordem crescente.

Programa 5.3: Troca de conteúdos de variáveis.

```
#include <stdio.h>

/* Recebe dois números inteiros x e y e coloca o menor desses
   valores em x e o maior em y, mostrando o resultado na saída */
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;
}
```

5.3 Revisão de números inteiros

Como já mencionamos, 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, em algoritmos e em 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.

Durante a execução de algoritmos e programas podemos usar compartimentos de memória para armazenamento de informações. Como valores podem ser atribuídos e sobrescritos nesses compartimentos e, portanto, podem variar durante a execução do algoritmo ou programa, esses compartimentos de memória são chamados de **variáveis**. Além disso, podemos atribuir 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(a) desenvolvedor(a)/programador(a) de modo a refletir de alguma forma o seu conteúdo. Nas aulas anteriores, temos usado diversas variáveis para armazenar valores que são números inteiros, como `num1`, `soma` e `produto`, por exemplo. Nos nossos programas, podemos usar outros tipos de dados além de inteiros, como já mencionamos e veremos mais adiante.

Há uma regra fundamental sobre variáveis em programas: *qualquer variável usada em um programa deve ser previamente declarada*. Há também regras para um desenvolvedor determinar os identificadores das variáveis. Essas regras são as mesmas vistas na aula teórica e são apresentadas a seguir:

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

Dessa forma, os seguintes identificadores são nomes válidos de variáveis:

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

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

Por outro lado, os identificadores listados a seguir não são nomes válidos de variáveis em programas:

```
preço$
soma total
4quant
int
```

No primeiro caso, o identificador `preço$` contém um símbolo não válido: $. O segundo identificador também tem o mesmo problema, já que um identificador não pode conter um caractere espaço. O terceiro exemplo tem um identificador que inicia com um caracteres não válido, um dígito numérico. O último exemplo é um identificador não válido já que `int` é uma palavra-chave 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 todas diferentes.

A declaração de variáveis do tipo inteiro pode ser realizada com uma única instrução `int` e mais uma lista de identificadores de variáveis, separados por um espaço e uma vírgula, finalizada por ponto e vírgula; ou ainda, pode ser realizada uma a uma, com uma instrução `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. No entanto, um programa escrito com identificadores de variáveis muito longos pode ser difícil de ser escrito e compreendido. Por exemplo,

```
TotalDeDinheiroQueTenho = TotalDoDinheiroQueGuardeiAnoPassado +
      TotalDeDinheiroQueGuardeiEsteAno - TotalDeImpostosEmReais;
```

é bem menos significativo que

```
TotalGeral = TotalAno + TotalAnterior - Impostos;
```

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

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

é uma expressão aritmética constante.

Além disso, podemos definir uma constante numérica, conhecida na linguagem C como uma **macro**, usando a diretiva do pré-processador `#define`. A definição de uma constante ou macro na linguagem C 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 inicia com um caractere especial `#` e não é finalizada com o caractere `;`. 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 NUMERADOR 4
#define MIN -10000
#define MAX 100
```

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 e então o programa executável é gerado.

A linguagem C, assim como a linguagem algorítmica, possui cinco operadores aritméticos binários que podemos usar com números inteiros: `+` para adição, `-` para subtração, `*` para multiplicação, `/` para quociente da 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 5.4 a seguir é um exemplo de programa que usa todos esses operadores em expressões aritméticas com números inteiros.

Programa 5.4: Operações aritméticas sobre números inteiros.

```
/* Realiza operações aritméticas com números inteiros */
#include <stdio.h>

#define NUM 25

int main(void)
{
   int x, y, z, r;

   y = 10;
   r = NUM + y;
   printf("A soma de %d e %d é %d\n", NUM, y, r);

   x = 38;
   r = x - NUM;
   printf("A subtração de %d e %d é %d\n", x, NUM, r);

   x = 51;
   y = 17;
   r = x * y;
   printf("A multiplicação de %d por %d é %d\n", x, y, r);

   x = 100;
   y = NUM;
   r = x / y;
   printf("O quociente da divisão de %d por %d é %d\n", x, y, r);

   x = 17;
   y = 3;
   r = x % y;
   printf("O resto da divisão de %d por %d é %d\n", x, y, r);

   x = -7;
   r = -x;
   printf("%d com sinal trocado é %d\n", x, r);

   x = 10;
   y = 4;
   z = 15;
   r = x + y * z;
   printf("A expressão %d+%d*%d é %d\n", x, y, z, r);

   return 0;
}
```

Há ainda um operador **unário** `-` na linguagem C, que age sobre um único operando 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, as instruções de atribuição `x = -7;` e `r = -x;` fazem com que o valor armazenado na variável `r` 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, quocientes de divisões e restos de divisões têm precedência sobre a adição e subtração e, por isso, o resultado da expressão:

```
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,

```
(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, quociente da divisão e resto da divisão	1 (máxima)
`+ -`	Adição e subtração	2 (mínima)

5.3.1 Representação de números inteiros

Esta seção é completamente inspirada no apêndice C do livro [15].

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

5.1 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 5.5: Solução do exercício 5.1.

```
#include <stdio.h>

/* Recebe um número inteiro que representa uma
   temperatura e imprime uma mensagem de aviso */
int main(void)
{
   int temperatura;

   printf("Informe uma temperatura (em graus Celsius):"));
   scanf("%d", &temperatura);

   if (temperatura > 30)
      printf("Hoje é um dia bem quente!\n");
   else
      printf("Hoje não é um dia tão quente.\n");

   return 0;
}
```

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

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

5.3 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 nosso operador `%`.

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.

5.4 Escreva um programa que receba um número inteiro a e verifique se a é par ou ímpar.

5.5 Escreva um programa que receba um número inteiro a e verifique se a é positivo, negativo ou igual a 0.

5.6 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 6

ESTRUTURA DE REPETIÇÃO `while`

As estruturas de repetição são, juntamente com a estrutura seqüencial e as estruturas condicionais que vimos nas aulas passadas, elementos fundamentais em algoritmos e programação. São essas três estruturas básicas que permitem que um(a) desenvolvedor(a)/programador(a) consiga resolver centenas e centenas de problemas de maneira efetiva. De posse apenas das estruturas seqüencial e condicional, um(a) programador(a) fica muito restrito à solução de uma pequena parcela de problemas. Assim, nesta aula motivaremos o estudo das estruturas de repetição e apresentaremos a estrutura de repetição `while` da linguagem C.

Esta aula é baseada em [16, 9].

6.1 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.

```
#include <stdio.h>

/* Escreve os 10 primeiros números inteiros positivos na saída */
int main(void)
{

   printf("1\n");
   printf("2\n");
   printf("3\n");
   printf("4\n");
   printf("5\n");
   printf("6\n");
   printf("7\n");
   printf("8\n");
   printf("9\n");
   printf("10\n");

   return 0;
}
```

Além de bem simples, o programa acima está correto, isto é, realiza corretamente o propósito de imprimir os dez primeiros números inteiros positivos na saída. Mas e se quiséssemos, por exemplo, imprimir os 1000 primeiros números inteiros positivos? Um programa semelhante ao que acabamos de apresentar ficaria um pouco mais complicado e monótono de escrever, já que teria muitas e muitas linhas de código.

Uma das propriedades fundamentais dos computadores, talvez a que mais nos auxilia, é que eles possuem a capacidade de executar repetitivamente um conjunto de instruções. Essa capacidade de repetição permite que um(a) desenvolvedor(a)/programador(a) escreva algoritmos e programas mais concisos, contendo processos repetitivos que poderiam necessitar de centenas ou milhares de linhas se não fossem assim descritos. Na linguagem C podemos ter três estruturas de repetição diferentes. A seguir veremos a estrutura de repetição **`while`**, mais simples, mais didática, mais ilustrativa e provavelmente a mais usada estrutura de repetição da linguagem C.

6.2 Estrutura de repetição `while`

O formato geral da estrutura de repetição **`while`** é apresentado a seguir:

```
while (condição) {
   instrução1;
   instrução2;
   ⋮
   instruçãon;
}
```

onde **`while`** é uma palavra-chave da linguagem C, ***`condição`*** é uma expressão lógica envolvida por parênteses que é avaliada e possui o valor verdadeiro ou falso, $\textbf{\textit{instrução}}_1$ a $\textbf{\textit{instrução}}_n$ fazem parte do bloco de instruções da estrutura de repetição **`while`** com as chaves **`{`** e **`}`** delimitando esse bloco de instruções. Do mesmo modo como vimos nas estruturas condicionais, se o bloco de instruções associado à estrutura de repetição **`while`** tem uma única instrução, esses delimitadores são opcionais.

A estrutura de repetição **`while`** funciona da seguinte forma. Quando um programa encontra a palavra-chave **`while`**, a expressão descrita na ***`condição`*** é avaliada. Se o resultado da avaliação dessa expressão é verdadeiro, o programa desvia seu fluxo de execução para o bloco de instruções interno à estrutura de repetição. Então, as instruções são executadas uma a uma até a última delas. Quando o programa atinge o final do bloco de instruções, o fluxo de execução é desviado para a linha que contém a palavra-chave **`while`** e a expressão é novamente avaliada. Se o resultado é verdadeiro, as instruções são todas executadas novamente e o processo se repete. Caso contrário, isto é, se o resultado da avaliação da expressão é falso, o fluxo de execução do programa é desviado para a primeira instrução após o bloco de instruções envolvido por chaves.

Veja o programa 6.1 para um exemplo de uso da estrutura de repetição **`while`**.

Programa 6.1: Primeiro exemplo usando a estrutura de repetição **`while`**.

```
#include <stdio.h>

/* Mostra os 100 primeiros números inteiros positivos */
int main(void)
{
   int numero;

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

   return 0;
}
```

Exercícios

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

Programa 6.2: Programa para o exercício 6.1.

```
#include <stdio.h>

/* Recebe um número inteiro n > 0 e mostra a so-
   ma dos n primeiros números inteiros positivos */
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("Soma dos %d primeiros inteiros é %d\n", n, soma);

   return 0;
}
```

6.2 Dado n, imprimir os n primeiros naturais ímpares.

Exemplo:

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

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

6.4 Dado n, imprimir as n primeiras potências de 2.

Exemplo:

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

6.5 Dados x inteiro e n um natural, calcular x^n.

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

6.7 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.

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

6.9 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.

6.10 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).

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

6.12 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.

6.13 Dados o número n de estudantes de uma turma de Algoritmos e Programação 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 7

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. Importante destacar que a linguagem C não tem um tipo lógico primitivo, que é então simulado através do tipo inteiro, como veremos.

Esta aula é baseada nas referências [15, 16].

7.1 Expressões aritméticas

Uma **expressão aritmética** na linguagem C é qualquer seqüência de símbolos formada exclusivamente por constantes numéricas, variáveis numéricas, operadores aritméticos e parênteses.

Como já vimos, uma **constante numérica do tipo inteiro** é qualquer número inteiro descrito em nosso programa, como por exemplo 34 ou -7.

Uma **variável numérica do tipo inteiro** é aquela que foi declarada com uma instrução `int` no início do programa.

Os **operadores aritméticos** são divididos em duas classes: operadores aritméticos unários e operadores aritméticos binários. Um **operador aritmético unário** é um operador que age sobre um único número inteiro e devolve um resultado. Há dois operadores aritméticos unários na linguagem C: o operador `-`, que troca o sinal da expressão aritmética que o sucede, e o operador `+`, 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 números inteiros: `+` para adição, `-` para subtração, `*` para multiplicação, `/` para quociente da 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, quociente da 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, como abaixo:

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

Suponha ainda que as seguintes atribuições de valores constantes tenham sido executadas 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 5.

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

7.2 Expressões relacionais

Uma **expressão relacional**, ou simplesmente uma **relação**, é uma comparação entre dois valores do mesmo tipo primitivo de dados. Esses valores são representados na relação através de constantes, variáveis ou expressões. Essa definição é bem abrangente e acomoda outros tipos de dados além do tipo `int`, o único tipo primitivo de dados que conhecemos até o momento. Com o tipo `int` a definição de relação pode ser escrita como uma comparação entre dois valores, representados por constantes numéricas, variáveis numéricas ou expressões aritméticas.

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 um tipo lógico ou booleano não existe na linguagem C. Isso significa que literalmente não existem constantes lógicas e variáveis lógicas na linguagem. 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 o resultado da expressão é interpretado como *verdadeiro*. Caso contrário, isto é, se este valor é igual a zero, então o resultado da expressão é interpretado como *falso*. Para facilitar, muitas vezes representaremos 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 relacional.

7.3 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** é formada por uma ou mais proposições. Neste 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
                !x
      a == 2 && a < b + c
   b + c >= 5 - a || b != 3
a + b > c || 2 * x == b && 4 < x
```

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

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

7.1 Dado p inteiro, verificar se p é primo.

Programa 7.1: Solução para o exercício 7.1.

```
#include <stdio.h>

/* Recebe um inteiro positivo p e verifica se p é primo */
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 7.2: Solução para o exercício 7.1 usando uma variável indicadora de passagem.

```
#include <stdio.h>

/* Recebe um inteiro positivo p e verifica se p é primo */
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 7.3: Terceira solução para o exercício 7.1.

```
#include <stdio.h>

/* Recebe um inteiro positivo p e verifica se p é primo */
int main(void)
{
   int p, divisor, primo;

   printf("Informe um número: ");
   scanf("%d", &p);
   divisor = 2;
   primo = 1;
   while (divisor <= p/2 && primo) {
      if (! (p % divisor))
         primo = 0;
      else
         divisor = divisor + 1;
   }
   if (primo)
      printf("%d é primo\n", p);
   else
      printf("%d não é primo", p);

   return 0;
}
```

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

7.3 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.

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

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

7.6 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.

7.7 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.

7.8 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)

7.9 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 $3/4$.

7.10 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		

7.11 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.

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

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

7.14 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.

7.15 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.

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

Exemplo:

$$
\begin{aligned}
1^3 &= 1 \\
2^3 &= 3 + 5 \\
3^3 &= 7 + 9 + 11 \\
4^3 &= 13 + 15 + 17 + 19 \\
&\vdots
\end{aligned}
$$

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

7.17 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

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

AULA 8

OUTRAS ESTRUTURAS DE REPETIÇÃO

Duas outras estruturas de repetição estão disponibilizadas na linguagem C. Uma delas é a estrutura de repetição `for`, que pode ser vista como uma outra forma de apresentação da estrutura de repetição `while`. Essas estruturas possuem exepressões lógicas iniciais que controlam o fluxo de execução de seus respectivos blocos de instruções. Uma outra estrutura apresentada aqui é a estrutura repetição `do-while`, cuja a expressão lógica de controle é posicionada no final de seu bloco de instruções.

Esta aula é baseada nas referências [16, 15].

8.1 Estrutura de repetição `for`

Retomando o programa 6.1, isto é, o primeiro exemplo com uma estrutura de repetição na linguagem C que vimos até agora, vamos refazê-lo, desta vez usando a estrutura de repetição `for`. Vejamos então o programa 8.1 a seguir.

Programa 8.1: Primeiro exemplo.

```
#include <stdio.h>

/* Mostra os 100 primeiros números inteiros positivos */
int main(void)
{
   int numero;

   for (numero = 1; numero <= 100; numero = numero + 1)
      printf("%d\n", numero);

   printf("\n");

   return 0;
}
```

A novidade neste programa é a estrutura de repetição `for`. O formato geral da estrutura de repetição `for` é dado a seguir:

```
for (inicialização; condição; passo) {
   instrução_1;
   ⋮
   instrução_n;
}
```

A estrutura de repetição `for` dispõe a inicialização, a condição e o passo todos na mesma linha de instrução. O funcionamento desta estrutura é dado da seguinte forma. Sempre que a palavra-chave `for` é encontrada, a *inicialização* é executada. Em seguida, uma *condição/expressão* é avaliada: se o resultado da avaliação é verdadeiro, então o bloco de instruções delimitado pelas chaves é executado. Ao final da execução do bloco de instruções, o *passo* é executado e o processo todo se repete. Se, em algum momento, o resultado da avaliação for falso, o fluxo de execução do programa é desviado para a primeira instrução após o bloco de instruções da estrutura de repetição.

No ponto onde nos encontramos 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 do valor armazenado em seus operandos, respectivamente. De fato, nada mais é necessário compreender sobre esses operadores simples. No entanto, infelizmente, a compreensão das formas de uso desses operadores pode ser facilmente confundida. 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` em duas linhas consecutivas 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` em duas linhas consecutivas na saída.

Para ajudar em nossa compreensão, podemos pensar que a expressão `++cont` significa "incremente `cont` imediatamente" enquanto que a expressão `cont++` significa "use agora o valor de `cont` e depois o incremente". O "depois" nesse caso depende de algumas questões técnicas da linguagem C que ainda não podemos entender, mas seguramente a variável `cont` será incrementada antes da próxima instrução/sentença ser executada.

O operador de decremento `--` tem as mesmas propriedades do operador de incremento `++`, que acabamos de descrever.

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

O programa 8.2 é uma outra versão do programa 8.1, que soluciona o problema da soma dos 100 primeiros números inteiros positivos. Desta vez, o programa usa o operador de incremento na estrutura de repetição `for`.

Programa 8.2: Soma os 100 primeiros números inteiros positivos.

```
#include <stdio.h>

/* Mostra os 100 primeiros números inteiros positivos */
int main(void)
{
   int numero, soma;

   soma = 0;
   for (numero = 1; numero <= 100; ++numero)
      soma = soma + numero;

   printf("A soma dos 100 primeiros inteiros é %d\n", soma);

   return 0;
}
```

8.2 Estrutura de repetição `do-while`

Suponha agora 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. Observe, antes de tudo, que este problema pode ser facilmente resolvido usando a estrutura de repetição `while`. E dadas as suas semelhanças, um programa que soluciona esse problema usando a estrutura de repetição `for` também pode ser facilmente desenvolvido.

Vejamos agora então uma solução um pouco diferente, dada no programa 8.3. Esta solução contém uma terceira estrutura de repetição da linguagem C, a estrutura de repetição `do-while`, que ainda não conhecíamos.

Programa 8.3: Exemplo usando a estrutura de repetição **do-while**.

```
#include <stdio.h>

/* Recebe uma seqüência de números inteiros finaliza-
   da por 0 (zero) e mostra a soma desses números */
int main(void)
{
   int numero, soma;

   soma = 0;
   do {
      printf("Informe um número: ");
      scanf("%d", numero);
      soma = soma + numero;
   } while (numero != 0);

   printf("A soma dos números informados é %d\n", soma);

   return 0;
}
```

O programa 8.3 é muito semelhante aos demais programas que vimos construindo até o momento. A principal e mais significativa diferença é o uso da estrutura de repetição `do-while`. Quando o programa encontra essa estrutura pela primeira vez, com uma linha contendo a palavra-chave `do`, o fluxo de execução segue para a primeira instrução do bloco de instruções dessa estrutura, envolvido por chaves. As instruções desse bloco são então executados uma a uma, até o fim do bloco. Logo em seguida, após o fim do bloco com o caractere `}`, encontramos a palavra-chave `while` e, depois, entre parênteses, uma expressão lógica. Se o resultado da avaliação dessa expressão lógica for verdadeiro, então o fluxo de execução do programa é desviado para a primeira instrução do bloco de instruções desta estrutura de repetição e todas as instruções são executadas novamente. Caso contrário, se o resultado da avaliação da expressão lógica é falso, o fluxo de execução é desviado para a próxima instrução após a linha contendo o final do bloco de execução desta estrutura de repetição.

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

Exercícios

8.1 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.

Programa 8.4: Solução do exercício 8.1.

```
#include <stdio.h>

/* Imprime os números inteiros positivos de 4 dígitos cuja raiz quadra-
   da é igual à soma dos seus dois primeiros e dois últimos dígitos */
int main(void)
{
   int numero, DD, dd;

   for (numero = 1000; numero <= 9999; numero++) {
      DD = numero / 100;
      dd = numero % 100;
      if ( (DD + dd) * (DD + dd) == numero )
         printf("%d\n", numero);
   }

   return 0;
}
```

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

Programa 8.5: Solução do exercício 8.8.

```
/* Recebe um inteiro e imprime a quantidade de dígitos que possui */

#include <stdio.h>

int main(void)
{
   int n, digitos;

   printf("Informe n: ");
   scanf("%d", n);
   do {
      n = n / 10;
      digitos++;
   } while (n > 0);
   printf("O número tem %d dígitos\n", digitos);

   return 0;
}
```

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

Exemplo:

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

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

Exemplo:

Dado 18 a saída deverá ser 10010.

8.5 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.

8.6 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:

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

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

8.7 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.

8.8 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 9

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.

Esta aula é baseada nos livros [15, 16].

9.1 Constantes e variáveis do tipo ponto flutuante

Números inteiros não são suficientes ou adequados para solucionar todos os problemas algorítmicos. 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`	1.17549×10^{-38}	3.40282×10^{38}	6 dígitos
`double`	2.22507×10^{-308}	1.79769×10^{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 9.1. 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 9.1: Calcula a média de 100 números do tipo ponto flutuante.

```
#include <stdio.h>

/* Recebe uma seqüência de 100 números reais e mostra a média desses números */
int main(void)
{
   int i;
   float numero, soma, media;

   soma = 0.0;

   for (i = 1; i <= 100; i++) {
      printf("Informe um número: ");
      scanf("%f", &numero);
      soma = soma + numero;
   }
   media = soma / 100;

   printf("A média dos 100 números é %f\n", media);

   return 0;
}
```

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).

9.2 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 9.2.

Programa 9.2: Relação entre valores do tipo inteiro e do tipo ponto flutuante.

```
#include <stdio.h>

int main(void)
{
   int i1, i2;
   float f1, f2;

   i1 = 190;
   f1 = 100.5;
   i2 = i1 / 100;
   printf("i2 = %d\n", i2);
   f2 = i1 / 100;
   printf("f2 = %f\n", f2);
   f2 = i1 / 100.0;
   printf("f2 = %f\n", f2);
   f2 = f1 / 100;
   printf("f2 = %f\n", f2);
   return 0;
}
```

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 9.3, tenha sido desenvolvido, compilado e executado.

Programa 9.3: Operações e tipos inteiro e de ponto flutuante.

```
#include <stdio.h>

int main(void)
{
   int i1, i2;
   float f1;

   i1 = 3;
   i2 = 2;
   f1 = i1 / i2;

   printf("f1 = %f\n", f1);

   return 0;
}
```

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 seu operando, no caso uma expressão aritmética, para um valor do tipo inteiro.

Vejamos agora o programa 9.4.

Programa 9.4: Uso de operadores conversores de tipo.

```
#include <stdio.h>

int main(void)
{
   int i1, i2, i3, i4;
   float f1, f2;

   f1 = 10.8;
   f2 = 1.5;
   i1 = f1 / f2;
   printf("i1 = %d\n", i1);

   i2 = (int) f1 / f2;
   printf("i2 = %d\n", i2);

   i3 = f1 / (int) f2;
   printf("i3 = %d\n", i3);

   i4 = (int) f1 / (int) f2;
   printf("i4 = %d\n", i4);

   return 0;
}
```

A saída do programa 9.4 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 da instrução 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 `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 da instrução de atribuição temos uma variável do tipo inteiro `i2`, o valor `6` é armazenado nesta variável e mostrado na saída através do `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 da instrução de atribuição temos uma variável do tipo inteiro `i3`, o valor armazenado nesta variável é `10`, que também é mostrado na saída através de `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 com `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 9.5, que computa a área de um círculo cujo valor do raio é informado pelo usuário em radianos.

Programa 9.5: Cálculo da área do círculo.

```
#include <stdio.h>

/* Recebe o raio de um círculo e mostra sua área */
int main(void)
{
   float pi;
   double raio, area;

   pi = 3.141592f;

   printf("Digite o valor do raio: ");
   scanf("%lf", &raio);

   area = (double) pi * raio * raio;

   printf("A área do círculo é %f\n", area);

   return 0;
}
```

Exercícios

9.1 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.

9.2 Os pontos (x, y) que pertencem à figura H (veja a figura 9.1) 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.

Programa 9.6: Uma solução para o exercício 9.1.

```
#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;
}
```

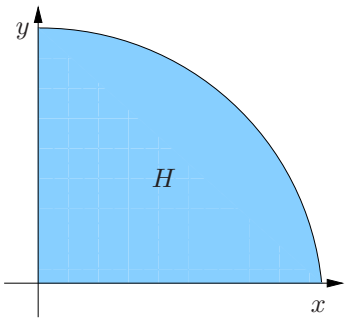


Figura 9.1: Área H de um quarto de um círculo.

9.3 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.

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

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

9.5 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 $\geqslant 5.0$.

9.6 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

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

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

9.8 Escreva um algoritmo que calcule 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 esquerda para a direita;

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

(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.

9.9 Uma maneira de calcular o valor do número π é utilizar a seguinte série:

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

Escreva um algoritmo que calcule e imprima o valor de π 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.

9.10 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 $|\frac{x^k}{k}| < 0.0001$, com k ímpar.

9.11 Uma maneira de calcular o valor de 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 ε, calcular o valor de 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.

9.12 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.

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

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

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 10

CARACTERES

Nesta aula vamos estudar um novo tipo primitivo de dados, o tipo caractere. O tipo caractere é um tipo primitivo que podemos trabalhar com um conjunto pequeno e previsível de valores. Na linguagem C existem dois tipos de caracteres: caractere com sinal e caractere 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. O estudo dos caracteres será estendido, mais útil e melhor compreendido na aula 13, quando veremos as cadeias de caracteres.

Esta aula é baseada especialmente nas referências [15, 16].

10.1 Representação gráfica

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

A impressão de um caractere na saída padrão é a sua representação como um símbolo gráfico. Por exemplo, o símbolo gráfico do caractere `97` é `a`. Alguns caracteres têm representações gráficas especiais, como o caractere `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 proposta da tabela ASCII, e também da tabela EBCDIC, logo mostraram-se aparentes e outros métodos foram desenvolvidos para estendê-las. 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 100 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.

10.2 Constantes e variáveis

Uma variável do tipo caractere com sinal pode ser declarada na linguagem C com a palavra reservada `char`. Uma variável do tipo caractere 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 em uma próxima aula. Dessa forma,

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

são declarações válidas de variáveis do tipo caractere.

Uma constante do tipo caractere 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 caractere através da representação gráfica de um caractere, envolvendo-o por aspas simples. Por exemplo, `'z'`, `';'` e `'3'` são exemplos de constantes do tipo caractere. 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 caractere 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 caractere.

caractere	constante	símbolo gráfico
0	`'\0'`	caractere 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 caractere 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 caractere na saída padrão com a função `printf` devemos usar o especificador de conversão de tipo caractere `%c` na seqüência de caracteres de formatação. Para ler um caractere a partir da entrada padrão com a função `scanf` também devemos usar o mesmo especificador de conversão de tipo caractere `%c` na seqüência de caracteres de formatação. Um exemplo de uso do tipo básico caractere é mostrado no programa 10.1.

Programa 10.1: Um programa usando o tipo `char`.

```
#include <stdio.h>

int main(void)
{
   char c;

   c = 'a';
   printf("%c\n", c);

   c = 97;
   printf("%c\n", c);

   printf("Informe um caractere: ");
   scanf("%c", &c);
   printf("%c = %d\n", c, c);

   return 0;
}
```

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

10.3 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 caractere 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);
⋮
```

Tente verificar a saída gerada pelas duas chamadas da função `printf` no trecho de código acima.

Exercícios

10.1 Verifique se o programa 10.2 está correto.

Programa 10.2: 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;
}
```

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

10.3 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.

10.4 *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`.

10.5 Escreva um programa que receba dois números inteiros a e b e um caractere 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 11

TIPOS PRIMITIVOS DE DADOS

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 caractere também é suportado pela linguagem e um valor do tipo caractere também é um número inteiro em um intervalo finito mais restrito que o dos tipos inteiros.

Nesta aula faremos uma revisão dos tipos primitivos de dados 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. Esta aula é uma revisão das aulas sobre o tipo inteiro (aula 7), o tipo ponto flutuante (aula 9) e o tipo carctere, que vimos em conjunto com nas aulas teóricas da disciplina. É também uma extensão desses conceitos, podendo ser usada como uma referência para tipos primitivos de dados.

As referências usadas nesta aula são [15, 16].

11.1 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 mais atuais são de 64 bits e a tabela abaixo mostra os intervalos de cada um dos possíveis tipos inteiros. A principal diferença entre máquinas de 32 e de 64 bits está no tipo `long int`. Em máquinas de 32 bits, o tipo `long int` e o tipo `int` são equivalentes. Em máquinas de 64 bits, há diferenciação entre esses dois tipos, como apresentado abaixo.

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`	−9.223.372.036.854.775.808	9.223.372.036.854.775.807
`unsigned long int`	0	18.446.744.073.709.551.615

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 constantes 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 do tipo inteiro não tem sinal, devemos adicionar a letra `U` ou `u` ao final da constante. Por exemplo, as constantes apresentadas 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 caractere `%` e representando um valor a ser preenchido durante a impressão. A informação que sucede o caractere `%` '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 11.1 usa os tipos primitivos 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 11.1: Exemplo de tipos e especificadores de tipos inteiros e também de formatação de impressão com especificadores de conversão.

```
#include <stdio.h>

int main(void)
{
   short int i1;
   unsigned short int i2;
   int i3;
   unsigned int i4;
   long int i5;
   unsigned long int i6;

   i1 = 159;
   i2 = 630u;
   i3 = -991023;
   i4 = 98979U;
   i5 = 8393202L;
   i6 = 34268298UL;

   printf("%hd %d %ho\n", i1, i1, i1);
   printf("%hu %i %hx\n", i2, i2, i2);
   printf("%+d % d %09d\n", i3, i3, i3);
   printf("%X %d %u\n", i4, i4, i4);
   printf("%+ld %ld %10ld\n", i5, i5, i5);
   printf("%lu %10.8lu %-lu\n", i6, i6, i6);

   return 0;
}
```

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, juntamente com a chamada da 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 uma leitura sempre que o usuário informa um **branco**, que é o caractere 32 (espaço ou `' '`), o caractere 9 (tabulação horizontal ou `'\t'`), o caractere 10 (tabulação vertical ou `'\v'`), o caractere 11 (retorno de carro/cursor ou `'\r'`), o caractere 12 (mudança de linha ou `'\n'`) ou o caractere 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.

11.2 Números com ponto flutuante

É fácil notar que apenas os números inteiros não são capazes de nos auxiliar na solução dos diversos problemas, mesmo quando encontramos formas de estender seu intervalo de valores, como no caso da linguagem C. Dessa forma, valores reais se fazem muito 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 9, mostra as características dos tipos numéricos com ponto flutuante quando implementados sob esse padrão.

Tipo	Menor valor (positivo)	Maior valor	Precisão
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 dígitos
`double`	2.22507×10^{-308}	1.79769×10^{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 valor é multiplicado. Se um expoente é dado, deve vir precedido da letra `e` ou `E`. Opcionalmente, um sinal `+` ou `-` pode ocorrer logo após a letra.

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`.

11.3 Caracteres

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

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

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

Uma constante do tipo caractere 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 caractere através da representação gráfica de um caractere, envolvendo-o por aspas simples. Por exemplo, `'z'`, `';'` e `'3'` são exemplos de constantes do tipo caractere. 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 caractere 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 caractere.

caractere	constante	símbolo gráfico
0	`'\0'`	caractere 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 caractere 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` possam ler ou escrever um único caractere. É importante reiterar que a função `scanf` não salta caracteres brancos antes de ler um caractere. Se o próximo caractere 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 caractere, 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 11.2 mostra um exemplo de uso da função `scanf` e de uma cadeia de caracteres de formatação contendo um especificador de tipo `%c`.

Programa 11.2: Conta o número de vogais minúsculas na frase digitada pelo usuário.

```
#include <stdio.h>

/* Recebe um frase e conta o número de vogais minúsculas que ela possui */
int main(void)
{
   char c;
   int conta;

   printf("Digite uma frase: ");
   conta = 0;
   do {
      scanf("%c", &c);
      if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
         conta++;
   } while (c != '\n');
   printf("A frase tem %d vogais minúsculas\n", conta);

   return 0;
}
```

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 caractere 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 caractere 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 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 caractere espaço e `•` representa o caractere 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 caractere usual na cadeia de caracteres de formatação de uma função `scanf` depende se o caractere 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 caractere 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 caractere da entrada; se os dois caracteres são idênticos, a função `scanf` descarta o caractere 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 caractere `/` 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 caractere da entrada e devolve esse caractere. A função `putchar` toma um caractere 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 caractere, 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 caractere que, após avaliada, seu valor será exibido na saída. A função `putchar` não devolve qualquer valor.

11.4 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 primitivos 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 9 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.

Veremos agora os dois primeiros casos, sendo que os dois últimos serão vistos adiante. 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 fiquem mais seguramente acomodados. Logo, o menor número de bits que mais seguramente acomode 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. Assim, 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 11.3 para um exemplo bastante ilustrativo.

Programa 11.3: Dificuldade na conversão de valores em expressões.

```
#include <stdio.h>

int main(void)
{
   int i;
   unsigned int j;

   i = -1;
   j = 1;
   printf("i = %+d i = %u\n", i, (unsigned int) i);
   printf("j = %+d j = %u\n", (int) j, j);
   if (i < j)
      printf("i < j\n");
   else
      printf("j < i\n");
   return 0;
}
```

A saída do programa 11.3 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 9. 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 primitivo de dados que será usado para converter o valor da `expressão`, após sua avaliação.

Por exemplo, supondo que `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 apenas o valor da parte fracionária da variável `f`.

11.5 Tipos de dados definidos pelo programador

A linguagem C, como vimos até aqui, possui poucos tipos primitivos de dados. No entanto, ela permite que um(a) programador(a) possa definir seus próprios tipos de dados, de acordo com sua conveniência. Para tanto, devemos usar a palavra reservada `typedef` no seguinte formato geral:

```
typedef tipo-primitivo tipo-novo;
```

onde `tipo-primitivo` é o nome de um dos tipos primitivos de dados da linguagem C e `tipo-novo` é o nome do novo tipo de dados definido pelo(a) programador(a). Um exemplo de definição e uso de um tipo é apresentado no trecho de código a seguir:

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

Na primeira linha, `typedef` é uma palavra reservada da linguagem C, `int` é o tipo primitivo de dados para números inteiros com sinal e `Logic` é o nome do novo tipo de dados criado pelo(a) programador(a). 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, entre outras coisas. Observe ainda que o compilador trata `Logic` como um sinônimo para `int`. Dessa forma, as variáveis `primo` e `crescente` são, na verdade, variáveis do tipo `int`.

Como principais vantagens podemos dizer que definições de tipos em programas permitem fazer com que um código torne-se mais compreensível, mais fácil de modificar e mais fácil de transportar de uma arquitetura de computadores para outra.

11.6 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 11.4.

Programa 11.4: Exemplo do uso do operador unário `sizeof`.

```
#include <stdio.h>

int main(void)
{
   printf("Caracteres: %lu\n", sizeof(char));
   printf("Inteiros:\n");
   printf(" short: %lu\n", sizeof(short int));
   printf(" int: %lu\n", sizeof(int));
   printf(" long int: %lu\n", sizeof(long int));
   printf("Números de ponto flutuante:\n");
   printf(" float: %lu\n", sizeof(float));
   printf(" double: %lu\n", sizeof(double));
   printf(" long double: %lu\n", sizeof(long double));
   return 0;
}
```

A saída do programa 11.4, executado em um computador com processador de 64 bits, é apresentada abaixo:

```
Caracteres:    1
Inteiros:
  short:       2
  int:         4
  long int:    8
Números de ponto flutuante:
  float:       4
  double:      8
  long double: 16
```

11.7 Exercícios

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

11.1 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 lógica) {
   case constante:
      ⋮
      instruções
      ⋮
      break;
   case constante:
      ⋮
      instruções
      ⋮
      break;
   ⋮
   default:
      ⋮
      instruções
      ⋮
      break;
}
```

Veja o programa 11.5. Observe especialmente a leitura dos dados.

Programa 11.5: Solução do exercício 11.1.

```
#include <stdio.h>

/* Recebe um operador aritmético e dois operandos inteiros e
   devolve o resultado da operação sobre os dois operandos */
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;
}
```

11.2 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.

11.3 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

Escreva um programa que receba um número real positivo x e um número real ε 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 ε. Mostre também na saída a quantidade de passos realizados para obtenção da raiz de x.

AULA 12

VETORES

Os tipos primitivos de dados na linguagem C se caracterizam pelo fato que seus valores não podem ser decompostos. Isso significa que um valor armazenado em uma variável de um tipo primitivo é único e não faz parte de uma composição de valores organizada de alguma maneira. Tipos primitivos de dados também são de básicos ou elementares. 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**. Nesta aula iniciaremos o estudo sobre variáveis compostas, partindo de uma variável conhecida como **variável composta homogênea unidimensional** ou simplesmente **vetor**.

Esta aula é inspirada nas referências [16, 15].

12.1 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 12.1.

Programa 12.1: Média de 5 notas.

```
#include <stdio.h>

/* Recebe 5 notas (números reais), calcula a média desses valores e mostra
   essa média e quantos dos valores são superiores e inferiores à média */
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", nota1, nota2, nota3, nota4, nota5);
   media = (nota1 + nota2 + nota3 + nota4 + nota5) / 5;
   printf("Media das provas: %f\n", media);

   menor = 0;
   if (nota1 < media)
      menor = menor + 1;
   if (nota2 < media)
      menor = menor + 1;
   if (nota3 < media)
      menor = menor + 1;
   if (nota4 < media)
      menor = menor + 1;
   if (nota5 < media)
      menor = menor + 1;

   maior = 0;
   if (nota1 > media)
      maior = maior + 1;
   if (nota2 > media)
      maior = maior + 1;
   if (nota3 > media)
      maior = maior + 1;
   if (nota4 > media)
      maior = maior + 1;
   if (nota5 > media)
      maior = maior + 1;

   printf("Quantidade com nota inferior à média: %d\n", menor);
   printf("Quantidade com nota superior à média: %d\n", maior);

   return 0;
}
```

A solução que apresentamos no programa 12.1 é uma solução correta para o problema. Isto é, dadas as cinco notas dos(as) estudantes em uma prova, o programa computa 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 1000 estudantes? Certamente, as estruturas seqüencial e condicional não seriam apropriadas para resolver esse problema, já que o(a) programador(a) 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.

12.2 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 primitivo. 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 primitivo; 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 na linguagem C é dada a seguir:

```
tipo identificador[dimensão];
```

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
- `dimensão` 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 12.1 mostramos o efeito da declaração do vetor `nota` na memória de um computador.

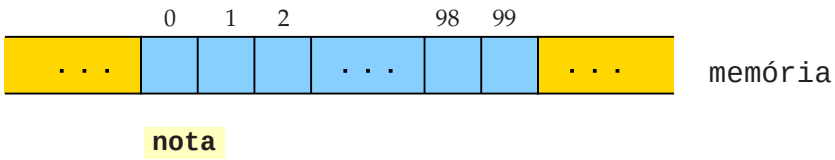


Figura 12.1: Vetor com 100 posições.

Observe 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`. Ou ainda, podemos fazer `nota[2*i+j]` para acessar a posição de índice $2i + j + 1$ do 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.

12.3 Declaração com inicialização

Podemos atribuir valores iniciais a quaisquer variáveis de qualquer tipo primitivo 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 primitivos:

```
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 primitivos, 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.

12.4 Exemplo com vetores

Vamos agora resolver o problema da seção 12.1 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 o programa 12.2.

Observe que o programa 12.2 é mais conciso e de mais fácil compreensão que o programa 12.1. 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.

Programa 12.2: Solução do problema proposto na seção 12.1 usando um vetor.

```
#include <stdio.h>

/* Recebe 5 notas (números reais), calcula a média desses valores e mostra
   essa média e quantos dos valores são superiores e inferiores à média */
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;
}
```

12.5 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 que calcula a área do círculo visto em uma aula prática pode ser reescrito com o uso de uma macro, como podemos ver no programa 12.3, onde uma macro com identificador `PI` é definida e usada no código.

Programa 12.3: Cálculo da área do círculo.

```
#include <stdio.h>

#define PI 3.141592f

/* Recebe um raio de um círculo (real) e mostra a área desse círculo */
int main(void)
{
   double raio, area;

   printf("Digite o valor do raio: ");
   scanf("%lf", &raio);

   area = PI * raio * raio;
   printf("A área do círculo de raio %f é %f\n", raio, area);

   return 0;
}
```

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 12.2 pode ser ainda modificado como no programa 12.4 com a inclusão de uma macro que indica a quantidade de notas a serem processadas.

A macro `MAX` é usada quatro vezes no programa 12.4: 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.

Programa 12.4: Solução do problema proposto na seção 12.1 usando uma macro e um vetor.

```
#include <stdio.h>

#define MAX 5

/* Recebe 5 notas (números reais), calcula a média desses valores e mostra
   essa média e quantos dos valores são superiores e inferiores à média */
int main(void)
{
   int i, menor, maior;
   float nota[MAX], soma, media;

   for (i = 0; i < MAX; i++) {
      printf("Informe a nota do(a) estudante %d: ", i+1);
      scanf("%f", &nota[i]);
   }

   soma = 0.0;
   for (i = 0; i < MAX; i++)
      soma = soma + nota[i];
   media = soma / MAX;

   menor = 0;
   maior = 0;
   for (i = 0; i < MAX; i++) {
      if (nota[i] < media)
         menor++;
      if (nota[i] > media)
         maior++;
   }

   printf("\nMedia das provas: %2.2f\n", media);
   printf("Quantidade com nota inferior à média: %d\n", menor);
   printf("Quantidade com nota superior à média: %d\n", maior);

   return 0;
}
```

Exercícios

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

12.2 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.

12.3 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.

12.4 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

Programa 12.5: Solução do exercício 12.1.

```
#include <stdio.h>

#define MAX 100

/* Recebe um número inteiro n e mais n números inteiros
   e escreve esses números na ordem inversa da de leitura */
int main(void)
{
   int i, n, A[MAX];

   printf("Informe n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      printf("Informe o número %d: ", i+1);
      scanf("%d", &A[i]);
   }
   printf("Números na ordem inversa da leitura:\n");
   for (i = n-1; i >= 0; i--)
      printf("%d ", A[i]);
   printf("\n");

   return 0;
}
```

37 números (de 0 a 36), calcular a freqüência de cada número desta roleta nos n lançamentos realizados.

12.5 Dados dois vetores x e y, ambos com n elementos, $1 \leqslant n \leqslant 100$, determinar o produto escalar desses vetores.

12.6 Calcule o valor do polinômio $p(x) = a_0 + a_1x + \ldots + a_nx^n$ 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 $x_1, x_2, \ldots, x_k$.

12.7 Dado o polinômio $p(x) = a_0 + a_1x + \ldots a_nx^n$, 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 $p(x)$.

12.8 Dados dois polinômios reais

$$p(x) = a_0 + a_1x + \ldots + a_nx^n \quad \text{e} \quad q(x) = b_0 + b_1x + \ldots + b_mx^m$$

determinar o produto desses dois polinômios. Suponha que $1 \leqslant m, n \leqslant 100$.

12.9 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

12.10 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.

12.11 Dada uma seqüência de n números inteiros, com $1 \leqslant n \leqslant 100$, imprimi-la em ordem crescente de seus valores.

12.12 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.

12.13 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 $m = 2$.

12.14 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.

12.15 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.

AULA 13

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.

Diversas funções que manipulam cadeias de caracteres estão disponíveis na biblioteca `string`.

Esta aula é inspirada nas referências [15, 16].

13.1 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 13.1.

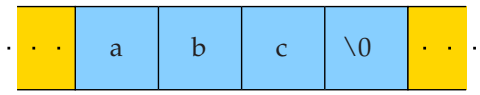


Figura 13.1: Armazenamento de uma literal na memória.

Por outro lado, uma literal também pode ser vazia. A literal `""` é uma literal vazia, representada na memória como na figura 13.2.

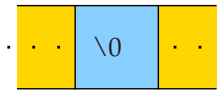


Figura 13.2: Literal vazia na memória.

13.2 Vetores de caracteres

Se queremos trabalhar com variáveis que comportam mais que um único caracter, temos de trabalhar com vetores de caracteres. No exercício 13.6, devemos definir 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.

13.3 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 13.1 que determina o comprimento de uma variável que é uma cadeia de caracteres. Qual será a saída deste programa?

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);
```

Programa 13.1: Determina o comprimento de uma cadeia de caracteres.

```
#include <stdio.h>

int main(void)
{
   char palavra[10] = "Ola!";
   int n;

   n = 0;
   while (palavra[n] != '\0')
      n++;
   printf("O comprimento da palavra é %d\n", n);

   return 0;
}
```

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 quando estudarmos 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 13.2.

Programa 13.2: Entrada e saída de uma cadeia de caracteres.

```
#include <stdio.h>

#define MAX 20

/* Recebe uma palavra com até MAX caracteres e imprime seu comprimento */
int main(void)
{
   char palavra[MAX];
   int n;

   printf("Informe uma palavra (com até %d caracteres): ", MAX);
   scanf("%s", palavra);
   n = 0;
   while (palavra[n] != '\0')
      n++;

   printf("A palavra [%s] tem %d caracteres\n", palavra, n);

   return 0;
}
```

Exercícios

13.1 Dada uma frase com no máximo 100 caracteres, determinar quantos caracteres espaço a frase contém.

13.2 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.

13.3 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.

Programa 13.3: Solução do exercício 13.1.

```
#include <stdio.h>

#define MAX 100

/* Dada uma frase, determina quantos espaços ela possui */
int main(void)
{
   char frase[MAX+1];
   int esp, i;

   printf("Informe uma frase: ");
   scanf("%[^\n]", frase);

   esp = 0;
   for (i = 0; frase[i] != '\0'; i++)
      if (frase[i] == ' ')
         esp++;

   printf("Frase tem %d espaços\n", esp);

   return 0;
}
```

13.4 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`.

13.5 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`.

13.6 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.

AULA 14

MATRIZES

Na aula 12, 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 como vimos com os vetores, as matrizes são *variáveis* porque podem ter os valores de suas células alterados durante a execução do algoritmo/programa, *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. A extensão de variáveis compostas homogêneas para três ou mais dimensões é simples e imediata, como veremos posteriormente. Nesta aula, que tem como referências os livros [15, 16], aprenderemos a declarar matrizes, a declarar e inicializar simultaneamente as matrizes e também a usar matrizes para solucionar problemas.

14.1 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 linha i e na coluna j é chamado de **elemento** i, j ou **(**i, j**)-ésimo** elemento de A. Este elemento é denotado por $A_{i,j}$ ou $A(i, j)$. 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. Ou seja, isso significa que 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.

Na linguagem C, 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. Veja a figura 14.1.

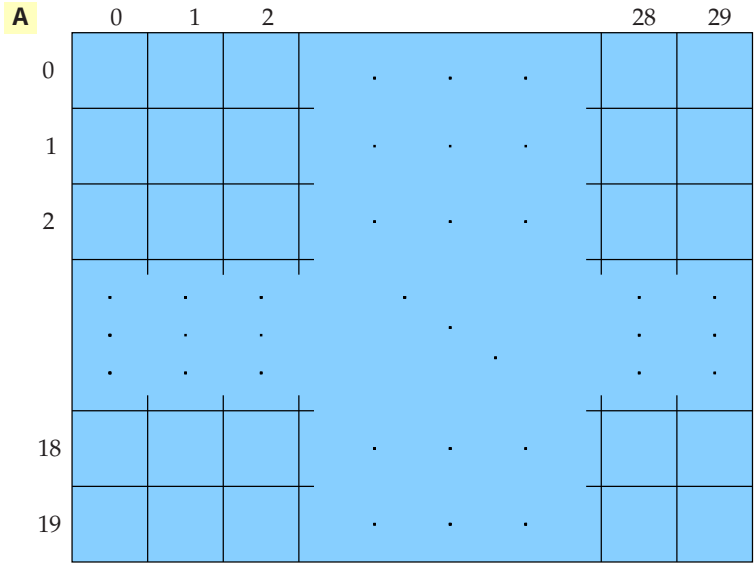


Figura 14.1: Matriz **A** com 20 linhas e 30 colunas.

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 14.2 para a matriz **A**.

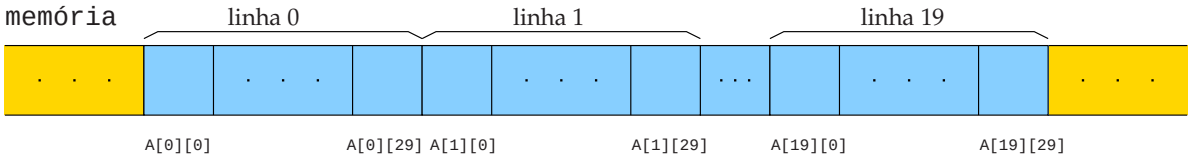


Figura 14.2: Matriz **A** com 20 linhas e 30 colunas disposta na memória.

14.2 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};
  ```

14.3 Exemplo

Apresentamos a seguir o programa 14.1, que resolve o seguinte problema:

> Dada uma matriz real B, de 5 linhas e 10 colunas, calcule o somatório dos elementos da oitava coluna e que calcule o somatório da terceira linha.

Programa 14.1: Exemplo de um programa que usa uma matriz.

```
#include <stdio.h>

/* Dada uma matriz de números reais de dimensão 5 x 10, mostra
   a soma da sua oitava coluna e da sua terceira linha */
int main(void)
{
   int i, j;
   float soma_c8, soma_l3, B[5][10];

   for (i = 0; i < 5; i++)
      for (j = 0; j < 10; j++) {
         printf("Informe B[%d][%d]: ", i, j);
         scanf("%f", B[i][j]);
      }
   soma_c8 = 0.0;
   for (i = 0; i < 4; i++)
      soma_c8 = soma_c8 + B[i][7];
   printf("Valor da soma da oitava coluna é %f\n", soma_c8);
   soma_l3 = 0.0;
   for (j = 0; j < 10; j++)
      soma_l3 = soma_l3 + B[2][j];
   printf("Valor da soma da terceira linha é %f\n", soma_l3);

   return 0;
}
```

Exercícios

14.1 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 $C_{m \times n} = A + B$.

14.2 Faça um programa que, dada uma matriz de números reais $A_{m \times n}$, determine A^t. Suponha que $1 \leqslant m, n \leqslant 100$.

14.3 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.

14.4 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 $1 \leqslant m, n \leqslant 100$.

Programa 14.2: Solução do exercício 14.1.

```
#include <stdio.h>

#define MAX 100

/* Dadas duas matrizes de números inteiros de dimensões m × n,
   calcula a matriz soma dessas matrizes, mostrando-a na saída */
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;
}
```

14.5 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.

14.6 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.

14.7 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.

14.8 Dada uma matriz de números reais $A_{m\times n}$, com $1 \leqslant m, n \leqslant 100$, verificar se existem elementos repetidos em A.

14.9 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.

14.10 (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.

[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.

14.11 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:

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

$$\begin{pmatrix} 1 & -1 & 2 & -1 & -1 & 3 & -1 & 4 \\ 5 & 6 & 0 & 0 & -1 & 7 & 0 & 0 \\ 8 & 0 & -1 & -1 & 9 & 0 & -1 & 0 \\ -1 & 10 & 0 & 11 & 0 & -1 & 12 & 0 \\ 13 & 0 & -1 & 14 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

14.12 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

$$a_{03} + a_{31} + a_{13} + a_{33} + a_{32} + a_{21} + a_{10} = 3 + 1 + 400 + 5 + 2 + 1 + 5 = 417\,.$$

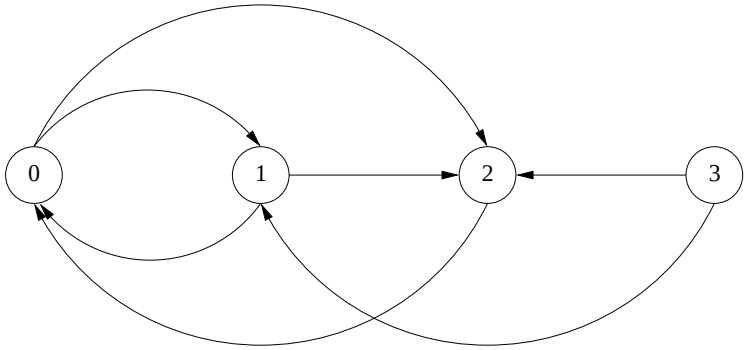
14.13 Dados um caça-palavra, representado por uma matriz A de letras de dimensão $m \times n$, com $1 \leqslant m, n \leqslant 50$, e uma lista de $k > 0$ palavras, encontrar a localização (linha e coluna) no caça-palavras em que cada uma das palavras pode ser encontrada.

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.

```
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
```

14.14 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 15

REGISTROS

Nas aulas 12 e 14 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, que é inspirada nas referências [15, 16].

15.1 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 15.1 mostra a disposição do registro `produto` na memória do computador.

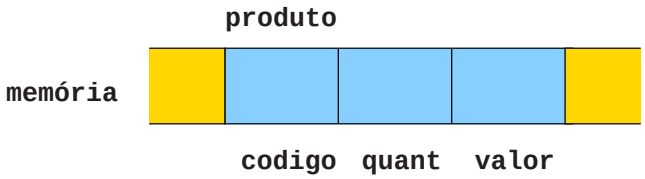


Figura 15.1: Representação do registro `produto` na memória.

Na linguagem C, a declaração de um registro sempre inicia com a palavra-chave `struct`. Logo em seguida, o delimitador de bloco `{` deve ocorrer. Depois disso, um bloco de declarações é iniciado. Nesse bloco, podemos declarar os campos do registro, linha após linha. Cada linha deve conter o tipo, de um dos tipos primitivos da linguagem C ou de um tipo criado pelo(a) programador(a), e mais o identificador do campo do registro. Finalizamos então as declarações dos campos do registro o delimitador de bloco `}`. Depois disso, realizamos de fato a declaração do registro, digitando o seu identificador. A variável do tipo registro com identificador `produto` declarada acima contém três campos, dois do tipo inteiro com identificadores `codigo` e `quant`, e outro do tipo ponto flutuante `valor`.

O formato geral de declaração de um registro é apresentado abaixo:

```
struct {
   ⋮
   bloco de declarações
   ⋮
} identificador;
```

Podemos declarar outras variáveis do tipo registro com os mesmos campos da variável `produto`, declarada acima, da seguinte forma:

```
struct {
   int codigo;
   int quant;
   float valor;
} produto, estoque, baixa;
```

Uma maneira alternativa de declaração dos campos de um registro é declarar campos de um mesmo tipo em uma única linha de código, seperando-os com vírgulas. Por exemplo, na linguagem C, a instrução a seguir:

```
struct {
   char sala, turma;
   int horas_inicio, minutos_inicio, horas_fim, minutos_fim;
   float largura, comprimento;
} aula;
```

declara um registro com identificador `aula` com seis campos, dois campos do tipo `char`, quatro outros campos do tipo `int` e dois campos 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 largura;
   float comprimento;
} 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 operador '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("Comprar produto\n");
else
   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;
```

15.2 Declaração e inicialização simultâneas

Declaração e inicialização simultâneas também são válidas com registros. As regras são idênticas às dos vetores. Por exemplo, o registro `produto` que declaramos acima pode ter sua inicialização realizada no momento de sua declaração como segue:

```
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`.

15.3 Operações sobre registros

Variáveis compostas heterogêneas possuem uma característica importante, que não é observada em variáveis compostas homogêneas.

Lembre-se que quando queremos copiar o conteúdo completo de todos os compartimentos de uma variável composta homogênea para os compartimentos respectivos de 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 um mesmo tipo de dados e mesma dimensão, é errado tentar fazer uma atribuição como abaixo:

```
A = B;
```

para copiar os valores do vetor `B` no vetor `A`. O compilador da linguagem C deve acusar um erro como esse. O correto então, neste caso, é fazer a cópia elemento a elemento de uma variável para outra. Supondo que n é a dimensão desses vetores, uma forma correta de realizar essa cópia é mostrada a seguir:

```
for (i = 0; i < n; i++)
   A[i] = B[i];
```

Por outro lado, quando tratamos de registros, 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;
mercadoria1.valor = 10.32 * TAXA + 0.53;
⋮
mercadoria2 = mercadoria1;
⋮
```

15.4 Exemplo

O programa 15.1 a seguir 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 15.1: Atualiza o horário em 1 segundo

```
#include <stdio.h>

/* Recebe um horário no formato hh:mm:ss e o atualiza
   em 1 segundo, mostrando o novo horário na saída */
int main(void)
{
   struct {
      int hh;
      int mm;
      int ss;
   } agora, prox;

   printf("Informe o horário atual (hh:mm:ss): ");
   scanf("%d:%d:%d", agora.hh, agora.mm, agora.ss);

   prox = agora;

   prox.ss = prox.ss + 1;
   if (prox.ss == 60) {
      prox.ss = 0;
      prox.mm = prox.mm + 1;
      if (prox.mm == 60) {
         prox.mm = 0;
         prox.hh = prox.hh + 1;
         if (prox.hh == 24)
            prox.hh = 0;
      }
   }

   printf("Próximo horário é %d:%d:%d\n", prox.hh, prox.mm, prox.ss);

   return 0;
}
```

Exercícios

15.1 Dada uma data no formato **`dd/mm/aaaa`**, escreva um programa que mostre a próxima data, isto é, a data que representa o dia seguinte à data fornecida.

Observação: 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.

15.2 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.

Programa 15.2: Solução do exercício 15.1.

```
#include <stdio.h>

/* Recebe uma data no formato dd:mm:aaaa e a atua-
   liza em 1 dia, mostrando a nova data na saída */
int main(void)
{
   struct {
      int dia;
      int mes;
      int ano;
   } data, prox;

   printf("Informe uma data (dd/mm/aa): ");
   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 % 100 == 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;
}
```

15.3 Dadas duas datas, calcule o número de dias decorridos entre estas duas datas.

Uma maneira provavelmente mais simples de computar essa diferença é usar a fórmula 15.1 para calcular um número de dias N baseado em uma data:

$$N = \left\lfloor \frac{1461 \times f(\text{ano}, \text{mês})}{4} \right\rfloor + \left\lfloor \frac{153 \times g(\text{mês})}{5} \right\rfloor + \text{dia} \tag{15.1}$$

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.

15.4 Seja N computado como na equação 15.1. 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 16

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 na memória para solução de problemas. Além disso, veremos ainda registros cujos campos podem ser variáveis não somente de tipos primitivos, de tipos definidos pelo usuário, ou ainda variáveis compostas homogêneas, mas também de variáveis compostas heterogêneas ou registros. O conjunto dessas combinações de variáveis que acabamos de mencionar 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.

Esta aula é baseada nas referências [16, 15].

16.1 Variáveis compostas homogêneas de registros

Como vimos nas aulas 12 e 14, 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, por exemplo, 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, numa mesma sentença de declaração, 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 16.1.

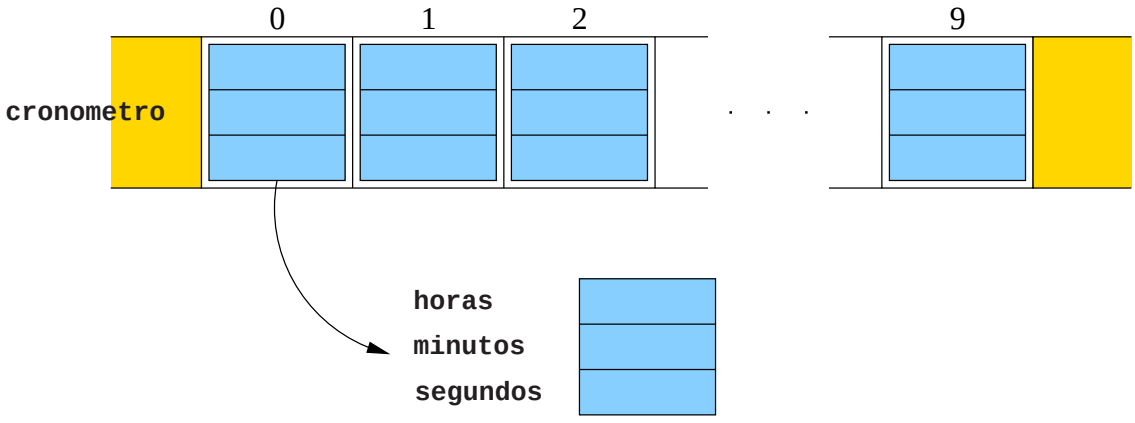


Figura 16.1: Efeito da declaração do vetor `cronometro` na memória.

Atribuições de valores do tipo inteiro aos campos do registro no primeiro compartimento deste vetor `cronometro` podem ser realizadas usando o identificador do vetor, o índice 0 (zero) envolvido por colchetes, o símbolo seletor de um campo `.` e o identificador do campo, como ilustrado nas atribuições abaixo:

```
cronometro[0].horas = 20;
cronometro[0].minutos = 39;
cronometro[0].segundos = 18;
```

Além disso, como fizemos na aula 15, podemos fazer a atribuição direta de registros para registros declarados da mesma maneira. 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 de atribuição acima estão corretas, apesar da primeira forma ser muito 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 16.1 soluciona o problema acima usando o método de ordenação das trocas sucessivas ou método da bolha.

Programa 16.1: Um programa usando um vetor de registros.

```
#include <stdio.h>

/* Recebe um inteiro n, 1 ⩽ n ⩽ 100, e n medidas de tempo
   hh:mm:ss, e mostra esses tempos em ordem crescente */
int main(void)
{
   int i, j, n;
   struct {
      int hh;
      int mm;
      int ss;
   } 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].hh, &cron[i].mm, &cron[i].ss);
   }

   for (i = n-1; i > 0; i--) {
      for (j = 0; j < i; j++) {
         if (cron[j].hh > cron[j+1].hh) {
            aux = cron[j];
            cron[j] = cron[j+1];
            cron[j+1] = aux;
         }
         else {
            if (cron[j].hh == cron[j+1].hh) {
               if (cron[j].mm > cron[j+1].mm) {
                  aux = cron[j];
                  cron[j] = cron[j+1];
                  cron[j+1] = aux;
               }
            }
            else {
               if (cron[j].mm == cron[j+1].mm) {
                  if (cron[j].ss > cron[j+1].ss) {
                     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].hh, cron[i].mm, cron[i].ss);

   return 0;
}
```

16.2 Registros contendo variáveis compostas homogêneas

Na aula 15 definimos registros que continham campos de tipos básicos de dados ou, no máximo, de tipos definidos pelo(a) programador(a). Na seção 16.1, estudamos vetores não mais de tipos básicos de dados, mas de registros, isto é, vetores contendo registros. Essa idéia pode ser estendida para matrizes ou ainda para variáveis compostas homogêneas de qualquer dimensão. Por outro lado, podemos também declarar registros que contêm variáveis compostas homogêneas como campos. Um exemplo bastante comum é a declaração de um vetor de caracteres, ou uma cadeia de caracteres, dentro de um registro, isto é, como um campo deste 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 16.2.

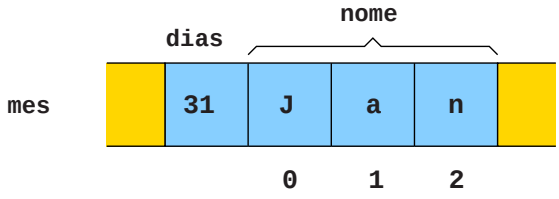


Figura 16.2: Variável `mes` na memória.

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;
```

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 16.2.

Programa 16.2: Exemplo do uso de um vetor como campo de um registro.

```
#include <stdio.h>

/* Recebe a descrição e o horário de início de duas atividades, no
   formato hh:mm:ss, e verifica qual delas será realizada primeiro */
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;
}
```

16.3 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 exemplo, podemos declarar uma variável do tipo registro com nome ou identificador `estudante` contendo um campo do tipo inteiro `rga`, um campo do tipo vetor de caracteres `nome` e um campo do tipo registro `nascimento`, contendo por sua vez três campos do tipo inteiro com identificadores `dia`, `mes` e `ano`:

```
struct {
   int rga;
   char nome[51];
   struct {
      int dia;
      int mes;
      int ano;
   } nascimento;
} estudante;
```

A figura 16.3 mostra o efeito da declaração da variável `estudante` na memória.

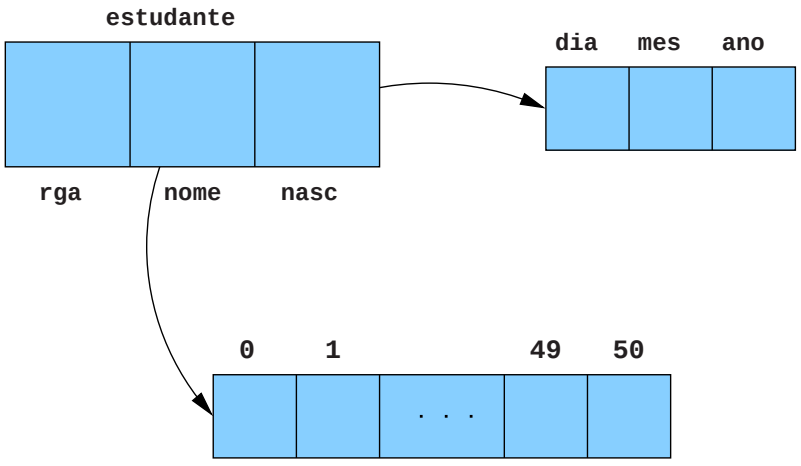


Figura 16.3: Efeitos da declaração do registro `estudante` na memória.

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;
```

Suponha finalmente que temos o seguinte problema.

> Dados um inteiro positivo n, uma seqüência de n nomes, telefones e datas de aniversário, e uma data no formato **`dd/mm`**, imprima os nomes e telefones das pessoas que aniversariam nesta data.

Uma solução para esse problema é apresentada no programa 16.3.

Podemos destacar novamente, assim como fizemos na aula 15, que registros podem ser atribuídos automaticamente para registros, não havendo necessidade de fazê-los campo a campo. Por exemplo, se temos declarados os registros:

```
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;
```

Programa 16.3: Um exemplo de uso de registros contendo registros.

```
#include <stdio.h>

#define DIM 100
#define MAX 50

/* Recebe um inteiro positivo n e n nomes, telefones e datas de
   aniversário, recebe uma data de consulta e mostra os nomes
   e telefones das pessoas que aniversariam nesta data */
int main(void)
{
   int i, n;
   struct {
      char nome[MAX+1];
      int telefone;
      struct {
         int dia;
         int mes;
         int ano;
      } aniver;
   } agenda[DIM];
   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);

   for (i = 0; i < n; i++)
      if (agenda[i].aniver.dia == data.dia && agenda[i].aniver.mes == data.mes)
          printf("%-50s %8d\n", agenda[i].nome, agenda[i].telefone);

   return 0;
}
```

Exercícios

16.1 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.

Programa 16.4: Solução do exercício 16.1.

```
#include <stdio.h>

#define MAX 100

/* Recebe um inteiro n, 1 <= n <= 100, n datas e uma data de referência,
   e verifica qual das n datas é mais próxima da data de referência */
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);
   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;
}
```

16.2 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.

16.3 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.

16.4 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.

16.5 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.

AULA 17

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.

Esta aula é baseada nas referências [16, 15].

17.1 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:

```
/* Recebe dois números reais e devolve a média aritmética deles */
double media(double a, double b)
{
   return (a + b) / 2;
}
```

A palavra-chave `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`. Observe que 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 resultante da avaliação da expressão aritmética `(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 no programa 17.1. Entre outras coisas, este programa mostra que uma função pode ser chamada tantas vezes quanto necessário.

Programa 17.1: Primeiro exemplo do uso de função.

```
#include <stdio.h>

/* Recebe dois números reais e devolve a média aritmética deles */
double media(double a, double b)
{
   return (a + b) / 2;
}

/* Recebe três números reais e calcula
   a média aritmética para cada par */
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;
}
```

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 uma mensagem uma certa quantidade de vezes:

```
/* Recebe um número inteiro e o imprime na saída com uma mensagem */
void imprimeContador(int n)
{
   printf("%d e contando...\n", 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:

```
/* Recebe um número inteiro e o imprime na saída com uma mensagem */
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 17.2 chama a função `imprimeContador` 10 vezes em uma estrutura de repetição `for`.

Programa 17.2: Segundo exemplo do uso de função.

```
#include <stdio.h>

/* Recebe um número inteiro e o imprime na saída com uma mensagem */
void imprimeContador(int n)
{
   printf("%d e contando...\n", n);

   return;
}


/* Imprime um contador inteiro descrescente de 10 a 1 */
int main(void)
{
   int i;

   for (i = 10; i > 0; i--)
      imprimeContador(i);

   return 0;
}
```

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 para o ponto em que foi chamada, no corpo da estrutura de repetição `for`, `imprimeContador` é chamada novamente e imprime a mensagem:

```
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:

```
/* Imprime uma mensagem na saída */
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 17.3 mostra o uso da função `imprimeMsg`.

Programa 17.3: Terceiro exemplo do uso de função.

```
#include <stdio.h>

/* Imprime uma mensagem na saída */
void imprimeMsg(void)
{
   printf("Programar é bacana!\n");
}

/* Imprime 10 vezes uma mensagem na saída */
int main(void)
{
   int i;

   for (i = 1; i <= 10; i++)
      imprimeMsg();

   return 0;
}
```

A execução do programa 17.3 começa com a primeira sentença na função `main`, que é a declaração da variável `i` do tipo `int`. Ems eguida, uma estrutura de repetição faz com que a função `imprimeMsg` seja executada 10 vezes. 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`.

17.2 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

   sentença_1;
   ⋮
   sentença_n;
}
```

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:

```
/* Recebe dois números reais e devolve a média aritmética deles */
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:

```
/* Recebe dois números reais e devolve a média aritmética deles */
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.

17.3 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 o valor 0 (zero) se o programa termina normalmente ou um valor diferente de 0 (zero) para indicar um término anormal. Por exemplo, quando uma seqüência de programas que se comunicam entre si é executada no sistema operacional LINUX, esse padrão de valores é adotado e verificado nesse processo. No entanto, não há uma regra rígida 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. É o que vimos sempre fazendo na última sentença de nossas funções `main`, quando escrevemos `return 0;`.

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. Depuração de programas é um tópico importante para uma linguagem de programação e teremos uma breve introdução ao assunto na aula 23.

17.4 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 entre dois números inteiros positivos antes, usando o algoritmo de Euclides, mas com o uso de uma única função, a função ou programa principal. Agora, faremos uma função que recebe esses dois números e devolve o máximo divisor comum. Uma possível função que soluciona o problema é mostrada a seguir.

```
/* Recebe dois números inteiros e devolve
   o máximo divisor comum entre eles */
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 17.4.

Programa 17.4: Exemplo de um programa com uma função `mdc`.

```
#include <stdio.h>

/* Recebe dois números inteiros e devolve
   o máximo divisor comum entre eles */
int mdc(int a, int b)
{
   int aux;

   while (b != 0) {
      aux = a % b;
      a = b;
      b = aux;
   }

   return a;
}

/* Recebe dois números inteiros e mostra
   o máximo divisor comum entre eles */
int main(void)
{
   int x, y;

   printf("Informe dois valores: ");
   scanf("%d%d", &x, &y);

   printf("O mdc entre %d e %d é %d\n", x, y, mdc(x, y));

   return 0;
}
```

17.5 Declaração de funções

Nos programas que vimos nas seções 17.1 e 17.4 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 17.1 colocando a definição da função `media` *depois* da definição da função `main`, como abaixo:

```
#include <stdio.h>

/* Recebe três valores reais e mostra a média aritméti-
   ca entre cada um dos 3 pares de números fornecidos */
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;
}

/* Recebe dois números reais e devol-
   ve a média aritmética entre eles */
double media(double a, double b)
{
   return (a + b) / 2;
}
```

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`:

```
#include <stdio.h>

double media(double a, double b);      /* declaração */

/* Recebe três valores reais e mostra a média aritméti-
   ca entre cada um dos 3 pares de números fornecidos */
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;
}

/* Recebe dois números reais e devol-
   ve a média aritmética entre eles */
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

17.1 (a) Escreva uma função com a seguinte interface:

```
double area_triangulo(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 17.5: Solução do exercício 17.1.

```
#include <stdio.h>

/* Recebe a base e a altura, dois números reais,
   e devolve a área do triângulo correspondente */
double area_triangulo(double base, double altura)
{
   return (base * altura) / 2;
}

/* Recebe um número inteiro n > 0 e uma seqüência
   de n pares de números reais, base e altura de
   um triângulo, e mostra a área de cada triângulo */
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", area_triangulo(b, a));
   }

   return 0;
}
```

17.2 (a) Escreva uma função com a seguinte interface:

```
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 9×6 é

$$\begin{array}{cccc} 9 & 6 & \rightarrow & 6 \\ 4 & 12 & & \\ 2 & 24 & & \\ 1 & 48 & \rightarrow & 48 \\ \hline & & & 54 \end{array}$$

(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).

17.3 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 num_lampadas(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 `num_lampadas`, 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`.

17.4 (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.

17.5 (a) Escreva uma função com interface

```
int verifica_primo(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.

17.6 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 conta_digitos(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.

17.7 (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

17.8 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 18

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, baseada nas referências [16, 15], focaremos nos argumentos e parâmetros das funções e no escopo de seus dados.

18.1 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 18.1.

Programa 18.1: Exemplo de passagem de parâmetros por referência.

```
#include <stdio.h>

/* Recebe um número de ponto flutuante x e devol-
   ve sua parte inteira e sua parte fracionária */
void decompoe(float x, int *parte_int, float *parte_frac)
{

   *parte_int = (int) x;
   *parte_frac = x - *parte_int;
   return;
}

/* Recebe um número de ponto flutuante e mostra na saída
   o número, sua parte inteira e sua parte fracionária */
int main(void)
{
   float num, b;
   int a;

   printf("Informe um número de ponto flutuante: ");
   scanf("%f", &num);
   decompoe(num, &a, &b);
   printf("Número: %f\n", num);
   printf("Parte inteira: %d\n", a);
   printf("Parte fracionária: %f\n", b);

   return 0;
}
```

Uma primeira observação importante é sobre a interface da função `decompoe`. 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. No corpo da função `decompoe`, observe que 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. 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.

18.2 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 18.1.

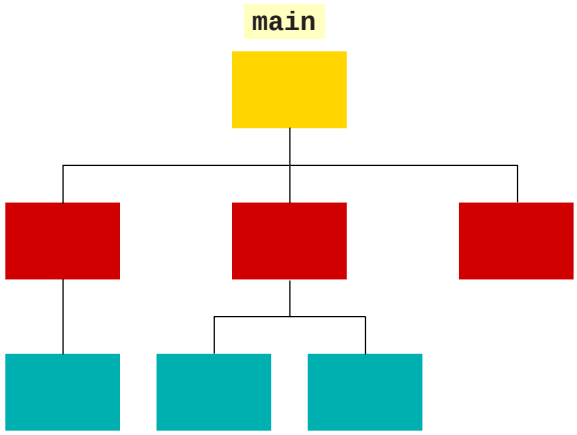


Figura 18.1: Um esquema de um programa contendo funções distribuídas em três níveis.

Exercícios

18.1 (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 número inteiro n e um vetor contendo n números inteiros, com $1 \leqslant n \leqslant 100$, ordene seus elementos em ordem crescente 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 18.2: Solução do exercício 18.1(b).

```
#include <stdio.h>

#define MAX 100

/* Recebe dois números inteiros a e b e devolve seus valores trocados */
void troca(int *a, int *b)
{
   int aux;

   aux = *a;
   *a = *b;
   *b = aux;

   return;
}

/* Recebe um um número inteiro n, com 1 <= n <= 100, e uma seqüência
   de n números inteiros, e mostra essa seqüência em ordem crescente */
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;
}
```

18.2 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 val, int *um, int *cin, int *dez, int *vin)
```

que receba um número não-negativo de ponto flutuante que representa 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 número não-negativo de ponto flutuante que representa um valor na moeda corrente determine a menor quantidade de cédulas para pagar esse valor. Use a função do item (a).

18.3 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 inteiro $n > 0$ e devolva três números inteiros: 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 número inteiro $n > 0$ e verifique se n é palíndromo. Suponha que n não contém o dígito 0.

18.4 (a) Escreva uma função com a seguinte interface:

```
int divisao(int *m, int *n, int d)
```

que receba três números inteiros positivos 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 números inteiros positivos m e n e calcule, usando a função do item (a), o mínimo múltiplo comum entre m e n.

18.5 (a) Escreva uma função com a seguinte interface:

```
int somabit(int b1, int b2, int *vaium)
```

que receba três números inteiros e devolva um número 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 receba dois números na base binária e, usando a função do item (a), calcule um número na base binária que é a soma dos dois números dados.

AULA 19

FUNÇÕES E VETORES

Nas aulas práticas e teóricas 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. Esta aula é baseada nas referências [16, 15].

19.1 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 19.1.

Programa 19.1: Um programa com uma função que tem um vetor como parâmetro.

```
#include <stdio.h>

/* Recebe um vetor com 10 números inteiros e devolve o menor deles */
int minimo(int A[10])
{
   int i, min;

   min = A[0];
   for (i = 1; i < 10; i++)
      if (A[i] < min)
         min = A[i];
   return min;
}

/* Recebe 10 números inteiros e mostra o menor deles na saída */
int main(void)
{
   int i, vet[10];

   for (i = 0; i < 10; i++)
      scanf("%d", &vet[i]);
   printf("O menor valor do conjunto é %d\n", minimo(vet));
   return 0;
}
```

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.

19.2 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. Vejamos o programa 19.2.

Programa 19.2: Um vetor é sempre um parâmetro passado por referência para uma função.

```
#include <stdio.h>

/* Recebe um vetor com n números inteiros e
   devolve o vetor com seus valores dobrados */
void dobro(int vetor[100], int n)
{
   int i;
   for (i = 0; i < n; i++)
      vetor[i] = vetor[i] * 2;
}

/* Recebe n números inteiros, com 1 <= n <= 100, e mos-
   tra o dobro de cada um desses valores informados */
int main(void)
{
   int i, n, valor[100];

   printf("Informe a quantidade de elementos: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      printf("Informe o %d-ésimo valor: ", i+1);
      scanf("%d", &valor[i]);
   }
   dobro(valor, n);
   printf("Resultado: ");
   for (i = 0; i < n; i++)
      printf("%d ", valor[i]);
   printf("\n");

   return 0;
}
```

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.

19.3 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:

```
/* Recebe um vetor com n números inteiros e
   devolve o vetor com seus valores dobrados */
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

19.1 (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 19.3: Solução do exercício 19.1.

```
#include <stdio.h>

#define MAX        100
#define VERDADEIRO 1
#define FALSO      0

/* Recebe um vetor A com m números inteiros e um vetor B com
   n números inteiros e verifica se A está contido em B */
int subconjunto(int A[MAX], int m, int B[MAX], int n)
{
   int i, j, contido;
   contido = VERDADEIRO;
   for (i = 0; i < m && contido; i++) {
      for (j = 0; j < n && B[j] != A[i]; j++)
         ;
      if (j == n)
         contido = FALSO;
   }
   return contido;
}

/* Recebe dois conjuntos de números inteiros e verifica se são iguais */
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;
}
```

19.2 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.

19.3 (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, 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 em ordem crescente de dimensão m e um vetor B de números inteiros 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 de números inteiros em ordem crescente.

AULA 20

FUNÇÕES E MATRIZES

Na aula 19 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. Esta aula é baseada nas referências [16, 15].

20.1 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

```
multiplica_escalar(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 20.1 recebe uma matriz $A_{4\times4}$ de números inteiros e um escalar e realiza o produto da matriz por esse escalar.

Programa 20.1: Uma função com um parâmetro que é uma matriz.

```
#include <stdio.h>

/* Recebe uma matriz de dimensão 4x4 de números inteiros e um escalar e
   devolve uma matriz resultante do produto entre a matriz e o escalar */
void multiplica_escalar(int A[4][4], int escalar)
{
   int i, j;

   for (i = 0; i < 4; i++)
      for (j = 0; j < 4; j++)
         A[i][j] = A[i][j] * escalar;
}

/* Recebe uma traiz de dimensão 4x4 de números
   inteiros e mostra seu conteúdo na saída */
void imprime_matriz(int A[4][4])
{
   int i, j;

   for (i = 0; i < 4; i++) {
      for (j = 0; j < 4; j++)
         printf("%3d ", A[i][j]);
      printf("\n");
   }
}

/* Recebe um número inteiro e uma matriz de dimensão 4x4
   de números inteiros e mostra a matriz de entrada e a
   matriz resultante do produto desse número pela matriz */
int main(void)
{
   int i, j, esc, matriz[4][4];

   scanf("%d", &esc);
   for(i = 0; i < 4; i++)
      for(j = 0; j < 4; j++)
         scanf("%d", &matriz[i][j]);

   imprime_matriz(matriz);
   multiplica_escalar(matriz, esc);
   imprime_matriz(matriz);

   return 0;
}
```

20.2 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 `multiplica_escalar` 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 multiplica_escalar(int A[][4], int escalar)
```

Exercícios

20.1 (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 20.2: Solução do exercício 20.1.

```
#include <stdio.h>

#define MAX 100

/* Recebe dois números inteiros a e b e devolve esses valores trocados */
void troca(int *a, int *b)
{
   int aux;

   aux = *a;
   *a = *b;
   *b = aux;
}

/* Recebe uma matriz de dimensão mxn de números inteiros
   e os índices i e j de duas linhas da matriz, troca o
   conteúdo dessas linhas e mostra a matriz resultante */
int main(void)
{
   int m, n, i, j, k, A[MAX][MAX];

   scanf("%d%d", &m, &n);
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         scanf("%d", &A[i][j]);
   scanf("%d%d", &i, &j);
   for (k = 0; k < n; k++)
      troca(&A[i][k], &A[j][k]);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%d ", A[i][j]);
      printf("\n");
   }

   return 0;
}
```

20.2 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.

20.3 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 21

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, como fazemos com variáveis de tipos primitivos.

Esta aula é baseada nas referências [16, 15].

21.1 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:

```
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, como 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.

21.2 Registros e passagem por cópia

O valor de um campo de um registro pode ser passado como argumento na chamada de uma função assim como fizemos com um valor associado a uma variável de um tipo básico ou a uma célula de um vetor ou matriz. Ou seja, a sentença

```
x = raiz_quadrada(poligono.diagonal);
```

chama a função `raiz_quadrada` com argumento `poligono.diagonal`, isto é, com o valor armazenado no campo `diagonal` do registro `poligono`.

Um registro todo pode ser argumento na chamada 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 a seguir:

```
s = total_segundos(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 ou contém um registro, é de fato um parâmetro de entrada da função, passado por cópia. Dessa forma, voltamos a repetir que somente as variáveis compostas homogêneas têm um comportamento deferenciado na linguagem C, no que ser refere a argumentos e parâmetros de funções neste caso específico.

Um exemplo com uma chamada de uma função contendo um registro como argumento é mostrado no programa 21.1.

Programa 21.1: Um programa com uma função com parâmetro do tipo registro.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss, converte essa medida em segundos e devolve esse valor */
int converte_segundos(struct marca tempo)
{
   return tempo.hh * 3600 + tempo.mm * 60 + tempo.ss;
}

/* Recebe uma medida de tempo no formato hh:mm:ss e mostra a quantida-
   de de segundos relativos à esse tempo com referência ao mesmo dia */
int main(void)
{
   int segundos;
   struct marca horario;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);

   segundos = converte_segundos(horario);

   printf("Passaram-se %d segundo(s) neste dia\n", segundos);

   return 0;
}
```

Devemos destacar pontos importantes neste programa. O primeiro, e talvez o mais importante, é a definição da etiqueta de registro `marca`. Em nenhuma outra oportunidade anterior havíamos desenvolvido um programa que continha uma declaração de uma etiqueta de registro. Observe ainda que a declaração da etiqueta `marca` do programa 21.1 foi feita *fora* dos corpos das funções `converte_segundos` 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 `marca` realizada dessa maneira se deve ao fato que a mesma é usada nas funções `converte_segundos` e `main`, obrigando que essa declaração seja realizada *globalmente*, ou seja, fora do corpo de qualquer função. Todas as outras declarações que fizemos são declarações *locais*, isto é, estão localizadas no interior de alguma função.

Os outros pontos de destaque do programa 21.1 são todos relativos ao argumento da chamada da função `converte_segundos` e do parâmetro da mesma, descrito em sua interface. Note que essa função tem como parâmetro o registro `tempo` do tipo `struct marca`. O argumento da função, na chamada dentro da função `main` é o registro `horario` de mesmo tipo `struct marca`.

Finalmente, é importante repetir 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 `converte_segundos`, mas, caso houvesse, isso não afetaria qualquer um dos campos do registro `horario` da função `main`.

21.3 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. Veja o programa 21.2.

Programa 21.2: Função com um registro como parâmetro de entrada e saída.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve o tempo neste registro atualizado em 1 segundo */
void adiciona_segundo(struct marca *tempo)
{
   (*tempo).ss++;
   if ((*tempo).ss == 60) {
      (*tempo).ss = 0;
      (*tempo).mm++;
      if ((*tempo).mm == 60) {
         (*tempo).mm = 0;
         (*tempo).hh++;
         if ((*tempo).hh == 24)
            (*tempo).hh = 0;
      }
   }
}

/* Recebe uma medida de tempo (hh:mm:ss) e mostra o tempo atualizado em 1s */
int main(void)
{
   struct marca horario;
   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
   adiciona_segundo(&horario);
   printf("Novo: %02d:%02d:%02d\n", horario.hh, horario.mm, horario.ss);
   return 0;
}
```

Uma observação importante sobre o programa 21.2 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 21.2 pode ser reescrito como o programa 21.3. Observe que esses dois programas têm a mesma finalidade, isto é, dada uma entrada, realizam o mesmo processamento e devolvem as mesmas saídas.

Programa 21.3: Uso do símbolo `->` para registros passados por referência.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve o tempo neste registro atualizado em 1 segundo */
void adiciona_segundo(struct marca *tempo)
{
   tempo->ss++;
   if (tempo->ss == 60) {
      tempo->ss = 0;
      tempo->mm++;
      if (tempo->mm == 60) {
         tempo->mm = 0;
         tempo->hh++;
         if (tempo->hh == 24)
            tempo->hh = 0;
      }
   }
}

/* Recebe uma medida de tempo no formato hh:mm:ss
   e mostra esse tempo atualizado em 1 segundo */
int main(void)
{
   struct marca horario;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &horario.hh, &horario.mm, &horario.ss);
   adiciona_segundo(&horario);
   printf("Novo horário %02d:%02d:%02d\n", horario.hh, horario.mm, horario.ss);
   return 0;
}
```

Observe que o argumento passado por referência para a função `adiciona_segundo` é idêntico nos programas 21.2 e 21.3. Mais que isso, a interface da função `adiciona_segundo` é idêntica nos dois casos e o parâmetro é declarado como `struct marca *tempo`. No corpo desta função do programa 21.3 é que usamos o símbolo `->` como uma abreviação, ou uma simplificação, da notação usada no corpo da função `adiciona_segundo` do programa 21.2. Assim, por exemplo, a expressão `(*tempo).ss` no programa 21.2 é reescrita como `tempo->ss` no programa 21.3.

21.4 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. Fazemos isso muitas vezes quando prorgramamos para solucionar grandes problemas.

No problema da seção anterior, em que é dado um horário em um formato específico e queremos atualizá-lo em um segundo, usamos um registro como parâmetro de entrada e saída na função `adiciona_segundo`. Por outro lado, 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. Ou seja, teremos então um parâmetro de entrada e um parâmetro de saída que são registros distintos. Dessa forma, os programas 21.2 e 21.3 poderiam ser reescritos como no programa 21.4.

Observe que a interface da função `adiciona_segundo` é bastante diferente das definições anteriores no programa 21.4. O tipo do valor devolvido é o tipo `struct marca` – uma etiqueta de um registro –, indicando que a função devolverá um registro. O corpo dessa função também é diferente das funções anteriores e há a declaração de uma variável `atualizado` do tipo `struct marca` 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 `adiciona_segundo` também é diferente, já que é descrita do lado direito de uma instrução de atribuição, indicando que o registro devolvido pela função será armazenado no registro apresentado do lado esquerdo da atribuição: `novo = adiciona_segundo(agora);`.

Programa 21.4: Um programa com uma função que devolve um registro.

```
#include <stdio.h>

struct marca {
   int hh;
   int mm;
   int ss;
};

/* Recebe um registro marca que contém uma medida de tempo no formato
   hh:mm:ss e devolve um registro com tempo atualizado em 1 segundo */
struct marca adiciona_segundo(struct marca tempo)
{
   struct marca 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;
}

/* Recebe uma medida de tempo no formato hh:mm:ss
   e mostra esse tempo atualizado em 1 segundo */
int main(void)
{
   struct marca agora, novo;

   printf("Informe um horario (hh:mm:ss): ");
   scanf("%d:%d:%d", &agora.hh, &agora.mm, &agora.ss);
   novo = adiciona_segundo(agora);
   printf("Novo horário %02d:%02d:%02d\n", novo.hh, novo.mm, novo.ss);

   return 0;
}
```

Exercícios

21.1 (a) Escreva uma função com a seguinte interface:

```
int bissexto(struct data d)
```

que receba uma data no formato **dd/mm/aaaa** 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 dias_mes(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 dia_seguinte(struct data d)
```

que receba uma data no formato **dd/mm/aaaa** 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 no formato **dd/mm/aaaa** e imprima a data que representa o dia seguinte. Use as funções dos itens anteriores.

21.2 Reescreva a função do item (c) do exercício 21.1, de modo que a função `dia_seguinte` tenha o registro `d` como um parâmetro de entrada e saída. Isto é, a interface da função deve ser:

```
void dia_seguinte(struct data *d)
```

21.3 (a) Escreva uma função com a seguinte interface:

```
struct tempo tempo_decorr(struct tempo t1, struct tempo t2)
```

que 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 no formato **hh:mm:ss** e calcule o tempo decorrido entre esses dois horários. Use a função anterior.

AULA 22

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. A aula é baseada nas referências [17, 16].

22.1 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 memória 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 omitido neste ponto. 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 com o qualificador de tipo `const`, como por exemplo a auto-documentação e a verificação pelo compilador de tentativas de alteração desse objeto.

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 para 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.

22.2 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. */
```

22.3 Arquivos-cabeçalhos da biblioteca padrão

22.3.1 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;
```

22.3.2 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.

22.3.3 Erros

A biblioteca de monitoramento de erros pode ser 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_FAILURE);
```

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`.

22.3.4 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.

22.3.5 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.

Constantes

Podemos usar uma das macros a seguir para especificar uma categoria:

LC_COLLATE

Afeta o comportamento de duas funções de comparação de cadeias de caractere: **strcoll** e **strxfrm**, ambas declaradas em **string.h**;

LC_CTYPE

Afeta o comportamento das funções em **ctype.h**, a menos de **isdigit** e **isxdigit**.

LC_MONETARY

Afeta a informação de formato monetário devolvida pela função **localeconv**, descrita na próxima seção;

LC_NUMERIC

Afeta o caractere separador de decimais usado nas funções de formatação de entrada e saída e as funções de conversão numérica;

LC_TIME

Afeta o comportamento da função **strftime** declarada em **time.h**, que converte um horário em uma cadeia de caracteres.

Funções

Duas funções são declaradas nesta biblioteca:

char *setlocale(int category, const char *locale)

Altera o local atual para uma única categoria ou para todas. Se o primeiro argumento é uma das macros definidas na seção anterior, uma chamada a esta função afeta apenas uma categoria. Se o primeiro argumento é **LC_ALL**, a chamada afeta todas as categorias. A linguagem C padrão define apenas dois valores para o segundo argumento: **"C"** e **""**. O primeiro é o padrão e, por exemplo, o separador decimal é um ponto. Se o segundo argumento é especificado então as funções da biblioteca comportam-se como "local nativo" e permitem que o programa adapte seu comportamento para o ambiente local.

struct lconv *localeconv(void)

Devolve um apontador para um registro do tipo **struct lconv**. Os campos desse registro contêm informações detalhadas sobe o local atual.

22.3.6 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.

22.3.7 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.

O tipo **`typedef jmp_buf`** é definido nesta biblioteca. Uma variável desse tipo armazena as informações atuais do "ambiente", para uso posterior nas funções da biblioteca.

As duas funções desta biblioteca são listadas abaixo:

`int setjmp(jmp_buf amb)`

Armazena o ambiente na variável **`amb`**. Se um valor diferente de zero é devolvido, então isso indica que o ponto exato neste código foi alcançado por uma chamada à função **`longjmp`**. Caso contrário, zero é devolvido indicando que o ambiente foi armazenado na variável.

`void longjmp(jmp_buf amb, int valor)`

O ambiente armazenado na variável **`amb`** por uma chamada prévia à função **`setjmp`** é recuperado. Uma salto ao ponto onde **`setjmp`** foi chamada é realizado, e o conteúdo armazenado em **`valor`** é devolvido à **`setjmp`**. Esse valor deve ser um número inteiro diferente de zero.

22.3.8 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.

Constantes e variáveis

O tipo **`sig_atomic_t`**, na verdade um apelido para o tipo **`int`**, é usado para declaração de variáveis que manipulam erros.

As macros com identificadores iniciando com **`SIG_`** são usadas com a função **`signal`**:

`SIG_DFL`

Manipulador de erros padrão;

`SIG_ERR`

Sinal de erro;

`SIG_IGN`

Ignora sinal.

As macros com identificadores iniciando com **`SIG`** são usadas para representar um número de sinal:

`SIGABRT`

Término anormal gerado pela função **`abort`**;

`SIGFPE`

Erro de ponto flutuante, causado por divisão por zero, operação não válida, etc;

`SIGILL`

Operação ilegal;

`SIGINT`

Sinal de atenção interativo, tal como um **`Ctrl-c`**;

`SIGSEGV`

Acesso não válido a uma porção de memória, tal como violação de segmento ou violação de memória;

`SIGTERM`

Requisição de término.

Funções

`void (*signal(int sinal, void (*func)(int)))(int)`

Controla como um sinal é manipulado. O parâmetro **`sinal`** representa o número do sinal compatível com as macros **`SIG`**. **`func`** é a função a ser chamada quando o sinal ocorre. A função deve ter um argumento do tipo **`int`** que representa o número do sinal.

`int raise(int sinal)`

Faz com que o sinal **`sinal`** seja gerado. O argumento **`sinal`** é compatível com as macros **`SIG`**.

22.3.9 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.

22.3.10 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.

22.3.11 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 as aulas teóricas Veja a aula 11 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 11.

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.

22.3.12 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.

22.3.13 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.

22.3.14 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

22.1 (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 sen x, onde x é dado em radianos, através da seguinte série:

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

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.

22.2 (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 23

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, documentação, 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.

23.1 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.

23.2 Primeiro contato

De uma janela de comandos, você pode chamar o depurador GDB, como a seguir:

```
prompt$ 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 "x86_64-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,

```
prompt$ 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 "x86_64-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
prompt$
```

23.3 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`.

23.4 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 9` estabelece um ponto de parada na linha 9 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 18 if ant == prox` estabelece um ponto de parada na linha 18, 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 18. Por outro lado, um *breakpoint* também pode ser criado através do comando `breakpoint identificador`, onde o argumento `identificador` é o nome de uma função.

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. Neste exemplo, a expressão contém apenas uma variável e o fluxo de execução será interrompido sempre que o conteúdo da variável `iguais` for modificado. Observe, no entanto, que a expressão pode ser mais complexa que isso, como por exemplo uma expressão aritmética `a*b + c/d`.

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, conforme vimos nas descrições acima, mostramos a seguir as informações de todos os pontos de parada que adicionamos à execução de um programa:

```
(gdb) info break
Num Type           Disp Enb Address            What
1   breakpoint     keep y   0x0000000000400575 in main at consecutivos.c:9
    breakpoint already hit 1 time
2   breakpoint     keep y   0x0000000000400630 in main at consecutivos.c:18
    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, res-

pectivamente. 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`, através do comando `step`, que pode ser abreviado por `s`, ou através do comando `next`, que pode ser abreviado por `n`. 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. O comando `next` muito se assemelha ao comando `step`. A diferença se dá na ocorrência ou não de uma função descrita pelo(a) programador(a) na sentença sendo executada: em caso positivo, o comando `step` abandona a função atual, desviando o fluxo de execução, e inicia a execução da outra função; o comando `next` evita que isso aconteça e a próxima sentença da mesma função é executada.

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` ou `next` nessa área suspeita, examinando o conteúdo das variáveis envolvidas, até que o problema se revele.

23.5 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 1,40`. Neste caso, as primeiras 40 linhas do código fonte do programa carregado no GDB serão mostradas.

```
(gdb) l 1,40
1      #include <stdio.h>
2
3      /* Recebe um número inteiro positivo e verifica se
4         esse número tem dois dígitos consecutivos iguais */
5      int main(void)
6      {
7          int num, ant, prox, iguais;
8
9          printf("\nInforme um número: ");
10         scanf("%d", &num);
11
12         prox = num % 10;
13         iguais = 0;
14         while (num != 0 && !iguais) {
15            num = num / 10;
16            ant = prox;
17            prox = num % 10;
18            if (ant == prox)
19               iguais = 1;
20         }
21
22         if (iguais)
23            printf("Número tem dois dígitos consecutivos iguais\n");
24         else
25            printf("Número não tem dois dígitos consecutivos iguais\n");
26
27         return 0;
28      }
(gdb)
```

23.6 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`.

23.7 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 conteúdo 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.

23.8 Resumo dos comandos

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
`next [n]`	Executa a próxima sentença do programa ou as próximas *n* sentenças, desconsiderando chamadas de funções do(a) programador(a)
`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*

Esta é uma relação dos comandos que aprendemos nesta aula, 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 outros comandos interessantes, mas que infelizmente não puderam ser cobertos neste texto. O(A) leitor(a) interessado(a) deve consultar as referências desta aula para mais informações.

23.9 Exemplos de execução

Vamos usar o GDB sobre o programa 23.1 que soluciona o exercício 7.3. No exercício 7.3 temos de verificar se um número inteiro positivo fornecido como entrada contém dois dígitos consecutivos e iguais.

Programa 23.1: Solução do exercício 7.3.

```
#include <stdio.h>

/* Recebe um número inteiro positivo e verifica se
   esse número tem dois dígitos consecutivos iguais */
int main(void)
{
   int num, ant, prox, iguais;

   printf("\nInforme um número: ");
   scanf("%d", &num);

   prox = num % 10;
   iguais = 0;
   while (num != 0 && !iguais) {
      num = num / 10;
      ant = prox;
      prox = num % 10;
      if (ant == prox)
         iguais = 1;
   }

   if (iguais)
      printf("Número tem dois dígitos consecutivos iguais\n");
   else
      printf("Número não tem dois dígitos consecutivos iguais\n");

   return 0;
}
```

Iniciamos o processo compilando o programa 23.1 adequadamente e carregando o programa executável gerado no GDB:

```
prompt$ gcc consecutivos.c -o consecutivos -Wall -ansi -pedantic -g
prompt$ 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 "x86_64-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) l 1,14
1       #include <stdio.h>
2
3       /* Recebe um número inteiro positivo e verifica se
4          esse número tem dois dígitos consecutivos iguais */
5       int main(void)
6       {
7           int num, ant, prox, iguais;
8
9           printf("\nInforme um número: ");
10          scanf("%d", &num);
11
12          prox = num % 10;
13          iguais = 0;
14          while (num != 0 && !iguais) {
(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/api/2010/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 9
Breakpoint 1 at 0x400575: file consecutivos.c, line 9.
(gdb)
```

Note que não iniciamos a execução do programa ainda. Então,

```
(gdb) run
Starting program:  /ensino/disciplinas/api/programas/consecutivos

Breakpoint 1, main () at consecutivos.c:9
9        printf("\nInforme um número: ");
(gdb)
```

O ponto de parada estabelecido na linha 9 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 = 4195456
ant = 0
prox = -12976
iguais = 32767
(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 9 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 18 if ant == prox
Breakpoint 2 at 0x400630: file consecutivos.c, line 18.
(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   0x0000000000400575 in main at consecutivos.c:9
    breakpoint already hit 1 time
2   breakpoint     keep y   0x0000000000400630 in main at consecutivos.c:18
    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 = 32767
New value = 0
0x00000000004005d3 in main () at consecutivos.c:13
13        iguais = 0;
(gdb)
```

Observe que na linha 13 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:18
18        if (ant == prox)
(gdb)
```

O GDB então interrompeu a execução do programa devido ao *breakpoint* 2, na linha 18 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:14
14          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 19 do programa. Assim, a próxima linha a ser executada é a linha 14, 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.
0x00007fd256015934 in exit () from /lib/libc.so.6
(gdb)
```

Mais uma análise detalhada e mais alguns testes sobre o programa 23.1 nos permitem afirmar com alguma convicção que este programa está correto e que soluciona o exercício 7.3. O fato novo é que essa afirmação foi feita também com o GDB nos auxiliando. Para provar formalmente que o programa 23.1 está correto, devemos descrever algum invariante do processo iterativo das linhas 14 até 20 do programa e então provar por indução a sua validade. Essa técnica de demonstração de correção vale para programas que possuem uma ou mais estruturas de repetição, como é o caso do programa 23.1. Aprenderemos mais sobre demonstração de correção de programas e algoritmos em disciplinas posteriores do nosso curso.

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 23.2 se propõe a implementar a ordenação por seleção.

Programa 23.2: Uma versão de um programa que deveria realizar a ordenação por seleção.

```
#include <stdio.h>

#define MAX 10

/* Recebe um número inteiro n > 0 e uma seqüência de n núme-
   ros inteiros, e mostra essa seqüência em ordem crescente */
int main(void)
{
   int i, j, n, menor, indice, aux, A[MAX];

   scanf("%d", &n);
   for (i = 0; i < n; i++)
      scanf("%d", &A[i]);

   for (i = 0; i < n-1; i++) {
      menor = A[i];
      indice = i;
      for (j = n-1; j > i; j++) {
         if (A[j] < menor) {
            menor = A[j];
            indice = j;
         }
      }
      aux = A[i];
      A[i] = A[indice];
      A[indice] = aux;
   }

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

   return 0;
}
```

Em uma tentativa de execução do programa 23.2, ocorre o seguinte erro:

```
prompt$ ./ord-selecao
8
7 5 4 1 8 6 2 3
Falha de segmentação
prompt$
```

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 é 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 23.2.

```
prompt$ gcc ord-selecao.c -o ord-selecao -Wall -ansi -pedantic -g
prompt$ 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 "x86_64-linux-gnu"...
(gdb) list 15,22
15        for (i = 0; i < n-1; i++) {
16           menor = A[i];
17           indice = i;
18           for (j = n-1; j > i; j++) {
19              if (A[j] < menor) {
20                 menor = A[j];
21                 indice = j;
22              }
(gdb) break 15
Breakpoint 1 at 0x4005bb: file ord-selecao.c, line 15.
(gdb) run
Starting program:  /ensino/disciplinas/api/programas/ord-selecao
8
7 5 4 1 8 6 2 3

Breakpoint 1, main () at ord-selecao.c:15
15        for(i = 0; i < n-1; i++)

(gdb) step 4
19              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:21
21                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:21
21                  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:21
21                  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:21
21                  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.
0x00007fd256015934 in exit () from /lib/libc.so.6
(gdb)
```

Encontramos um erro na linha 18 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.

Um último exemplo será apresentado a seguir. Neste exemplo, temos uma função a mais além da função principal. Veremos que, com o uso de alguns comandos do GDB, podemos interromper o fluxo de execução de uma função, desviá-lo para uma outra função, executar então esta função e retornar para o mesmo ponto da função onde ocorreu aquela chamada. Considere então o programa 23.3 a seguir, que soluciona o exercício 17.7.

A execução do programa dentro do GDB é apresentada a seguir.

```
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 "x86_64-linux-gnu"...
(gdb) list
19          return suf;
20       }
21
22       /* Recebe um número inteiro n > 0 e n pares de números inteiros
23          e verifica, para cada par, se um é subseqüência do outro */
24       int main(void)
25       {
26          int n, a, b, x, y, aux, subseq;
27
28          scanf("%d", &n);
(gdb) list
29          while (n > 0) {
30             scanf("%d%d", &a, &b);
31             x = a;
32             y = b;
33             if (x < y) {
34                aux = x;
35                x = y;
36                y = aux;
37             }
38             else {
(gdb) list
39                aux = a;
40                a = b;
41                b = aux;
42             }
43             subseq = FALSO;
44             while (x >= y && !subseq)
45                if (sufixo(x, y))
46                   subseq = VERDADEIRO;
47                else
48                   x = x / 10;
(gdb) break 44
Breakpoint 1 at 0x4006fc: file subsequencia.c, line 44.
(gdb)
(gdb) run
Starting program: /disciplinas/api/aulas/praticas/subsequencia
1
55678 567

Breakpoint 1, main () at subsequencia.c:44
44             while (x >= y && !subseq)
(gdb) watch subseq
Hardware watchpoint 2: subseq
(gdb) info break
Num  Type           Disp Enb Address            What
1    breakpoint     keep y   0x00000000004006fc in main at subsequencia.c:44
     breakpoint already hit 1 time
2    hw watchpoint  keep y                      subseq
(gdb)
```

```
(gdb) step
45                if (sufixo(x, y))
(gdb) next
48                   x = x / 10;
(gdb) list 44,50
44             while (x >= y && !subseq)
45                if (sufixo(x, y))
46                   subseq = VERDADEIRO;
47                else
48                   x = x / 10;
49             if (subseq)
50                printf("%d S %d\n", a, b);
(gdb) step

Breakpoint 1, main () at subsequencia.c:44
44             while (x >= y && !subseq)
(gdb) step
45                if (sufixo(x, y))
(gdb) info locals
n = 1
a = 567
b = 55678
x = 5567
y = 567
aux = 55678
subseq = 0
(gdb) step
sufixo (a=5567, b=567) at subsequencia.c:11
11          suf = VERDADEIRO;
(gdb) list 6,20
6           /* Recebe números inteiros a e b e devolve 1 se b é sufixo de a */
7           int sufixo(int a, int b)
8           {
9              int suf;
10
11              suf = VERDADEIRO;
12              while (b != 0 && suf)
13                 if (a % 10 != b % 10)
14                    suf = FALSO;
15                 else {
16                    a = a / 10;
17                    b = b / 10;
18                 }
19              return suf;
20          }
(gdb) info locals
suf = 1
(gdb) print a
$1 = 5567
(gdb) print b
$2 = 567
(gdb) break 19
Breakpoint 3 at 0x400644: file subsequencia.c, line 19.
(gdb)
```

```
(gdb) continue
Continuing.

Breakpoint 3, sufixo (a=5, b=0) at subsequencia.c:19
19              return suf;
(gdb) print suf
$3 = 1
(gdb) step
20          }
(gdb) step
Hardware watchpoint 2 deleted because the program has left the block
in which its expression is valid.
main () at subsequencia.c:46
46                      subseq = VERDADEIRO;
(gdb) step

Breakpoint 1, main () at subsequencia.c:44
44              while (x >= y && !subseq)
(gdb) step
49              if (subseq)
(gdb) step
50                 printf("%d S %d\n", a, b);
(gdb) step
567 S 55678
53              n-;
(gdb) step
29           while (n > 0) {
(gdb) step
55           return 0;
(gdb) step
56        }
(gdb) step
0x00007fdc477561a6 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

Programa 23.3: Solução do exercício 17.7.

```
#include <stdio.h>

#define VERDADEIRO 1
#define FALSO      0

/* Recebe dois números inteiros a e b e devolve 1 se b é sufixo de a */
int sufixo(int a, int b)
{
   int suf;

   suf = VERDADEIRO;
   while (b != 0 && suf)
      if (a % 10 != b % 10)
         suf = FALSO;
      else {
         a = a / 10;
         b = b / 10;
      }
   return suf;
}

/* Recebe um número inteiro n > 0 e uma lista de n pares de números in-
   teiros e verifica, para cada par, se um é subseqüência do outro */
int main(void)
{
   int n, a, b, x, y, aux, subseq;

   scanf("%d", &n);
   while (n > 0) {
      scanf("%d%d", &a, &b);
      x = a;
      y = b;
      if (x < y) {
         aux = x;
         x = y;
         y = aux;
      }
      else {
         aux = a;
         a = b;
         b = aux;
      }
      subseq = FALSO;
      while (x >= y && !subseq)
         if (sufixo(x, y))
            subseq = VERDADEIRO;
         else
            x = x / 10;
      if (subseq)
         printf("%d S %d\n", a, b);
      else
         printf("N\n");
      n--;
   }
   return 0;
}
```

AULA 24

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. Esta aula é baseada na referência [17].

24.1 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 24.1 ilustra o papel do pré-processador durante o processo de compilação.

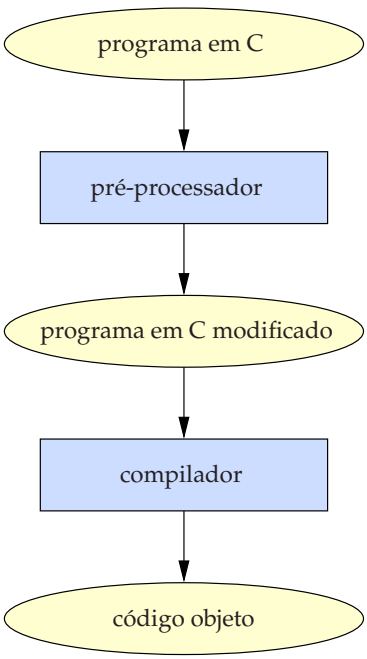


Figura 24.1: O papel do pré-processador durante a compilação.

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.

24.2 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 MAXIMO 100     /* Dimensão dos vetores */
```

24.3 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 nessas 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.

24.4 Inclusão de arquivos-cabeçalhos

Vamos recordar a aula 22. 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.

Programa 24.1: Um programa curioso.

```
#include <stdio.h>

#define PAR       0
#define IMPAR     1
#define Algoritmo main
#define inicio    {
#define fim       }
#define escreva   printf
#define leia      scanf
#define se        if
#define senao     else
#define enquanto  while
#define devolva   return

/* Recebe um número inteiro positivo e verifica sua paridade */
int par_impar(int numero)
inicio
   int paridade;

   se (numero % 2 == 0)
      paridade = PAR;
   senao
      paridade = IMPAR;

   devolva paridade;
fim

/* Recebe um número inteiro n > 0 e uma seqüência de n nú-
   meros inteiros, e imprime a soma dos pares e ímpares */
int Algoritmo(void)
inicio
   int i, n, numero, s_par, s_impar;

   escreva("Informe a quantidade de elementos: ");
   leia("%d", &n);

   s_par = 0;
   s_impar = 0;
   i = 1;
   enquanto (i <= n)
   inicio
      escreva("Informe um número: ");
      leia("%d", &numero);
      se (par_impar(numero) == PAR)
         somapar = somapar + numero;
      senao
         somaimpar = somaimpar + numero;
      i++;
   fim
   escreva("\nSoma dos números pares: %d\n", s_par);
   escreva("Soma dos números ímpares: %d\n", s_impar);

   devolva 0;
fim
```

Futuramente, 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.

24.5 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 o mesmo `#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 0 (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 0 (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 outros usos como na construção de programas que podem executados em computadores diferentes, em sistemas operacionais diferentes, que podem ser compilados por compiladores diferentes, no suporte à definições padronizadas de macros, entre outros.

24.6 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
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] "Computer History – from B.C. to today."
http://www.computerhope.com/history/.
último acesso em 10 de março de 2010. 1

[2] R. X. Cringely, "A History of the Computer."
http://www.pbs.org/nerds/timeline/index.html.
último acesso em 10 de março de 2010. 1

[3] J. Kopplin, "An Illustrated History of Computers."
http://www.computersciencelab.com/ComputerHistory/History.htm.
último acesso em 10 de março de 2010. 1

[4] A. Sapounov, E. Rosen, and J. Shaw, "Computer History Museum."
http://www.computerhistory.org/.
último acesso em 10 de março de 2010. 1

[5] "Wikipedia – the Free Encyclopedia."
http://en.wikipedia.org/wiki/Main_Page.
último acesso em 10 de março de 2010. 1

[6] P. Breton, História da Informática. Editora da UNESP, 1991. Tradução de Elcio Fernandes do original *Histoire de L'informatique*, 1987, Editions La Decouvert. 1

[7] R. L. Shackelford, Introduction to Computing and Algorithms. Addison Wesley, 1997. 1

[8] T. Kowaltowski, "John von Neumann: suas contribuições à Computação," Revista de Estudos Avançados, vol. 10, Jan/Abr 1996. Instituto de Estudos Avançados da Universidade de São Paulo. 2

[9] "Departamento de Ciência da Computação – IME/USP, listas de exercícios – Introdução à Computação." http://www.ime.usp.br/~macmulti/.
último acesso em 10 de março de 2010. 2, 6

[10] M. F. Siqueira, "Algoritmos e Estruturas de Dados I." Notas de aula, 1998. (ex-professor do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul (DCT/UFMS), atualmente professor do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte (DIMAp/UFRN)). 2

[11] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," Proceedings of the London Mathematical Society, vol. 42, no. 2, pp. 230–265, 1936. 2.1, 2.4

[12] K. Zuse, "Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen." Patentanmeldung Z 23 139 – GMD Nr. 005/021, 1936. 2.1

[13] H. Lukoff, From Dits to Bits... a Personal History of the Electronic Computer. Hawley Books, 1979. 2.1

[14] J. von Neumann, "First draft of a report on the EDVAC," tech. rep., Moore School of Electrical Engineering – University of Pennsylvania, 1945. 2.1, 2.3

[15] P. Feofiloff, Algoritmos em linguagem C. Editora Campus/Elsevier, 2009. 4.5, 4.5, 5, 5.3.1, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21

[16] K. N. King, C Programming – A Modern Approach. W. W. Norton & Company, Inc., 2nd ed., 2008. 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22

[17] E. Huss, "The C Library Reference Guide." `http://www.acm.uiuc.edu/webmonkeys/book/c_guide/`, 1997. último acesso em 10 de março de 2010. 22, 24

[18] S. A. Cook, "The complexity of theorem-proving procedures," in Proceedings of the Third Annual ACM Symposium on Theory of Computing, (Shaker Heights, Ohio, United States), pp. 151–158, SIGACT: ACM Special Interest Group on Algorithms and Computation Theory, ACM, 1971.

[19] B. W. Kernighan and R. Pike, The Practice of Programming. Addison-Wesley Professional, 1999.

[20] B. W. Kernighan and D. M. Ritchie, C Programming Language. Prentice Hall, 2nd ed., 1988.

[21] D. E. Knuth, The Art of Computer Programming, vol. 1–7. Addison-Wesley, 3rd ed., 1997.

[22] S. G. Kochan, Unix Shell Programming. Sams Publishing, 3rd ed., 2003.

[23] K. A. Lambert, D. W. Nance, and T. L. Naps, Introduction to Computer Science with C++. West Publishing Company, 1996.

[24] L. A. Levin, "Universal sequential search problems," Problemy Peredachi Informatsii, vol. 9, no. 3, pp. 115–116, 1973.

[25] G. J. Pothering and T. L. Naps, Introduction to Data Structures and Algorithm Analysis with C++. West Publishing Company, 1995.

[26] S. S. Skiena and M. Revilla, Programming Challenges. Springer, 2003.

[27] A. Tucker, A. P. Bernat, W. J. Bradley, R. D. Cupper, and G. W. Scragg, Fundamentals of Computing I – Logic, Problem Solving, Programs, and Computers – C++ Edition. McGraw-Hill, 1995.

[28] "The past is the future at Bletchley Park." `http://news.bbc.co.uk/2/hi/technology/6895759.stm`. Mark Ward, Technology Correspondent, BBC News website, julho de 2007, último acesso em 10 de março de 2010.