LÓGICA DE PROGRAMAÇÃO

AUTOR
FABIANO DOS SANTOS

LÓGICA DE PROGRAMAÇÃO

AUTOR
FABIANO DOS SANTOS

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2015

Estácio

Conselho editorial REGIANE BURGER; ROBERTO PAES; GLADIS LINHARES

Autor do original FABIANO DOS SANTOS

Projeto editorial ROBERTO PAES

Coordenação de produção GLADIS LINHARES

Projeto gráfico PAULO VITOR BASTOS

Diagramação BFS MEDIA

Revisão linguística BFS MEDIA

Revisão de conteúdo SIMONE MARKENSON

Imagem de capa ALEXALDO | DREAMSTIME.COM

Dados Internacionais de Catalogação na Publicação (CIP)

S237L SANTOS, FABIANO DOS

Lógica de programação / Fabiano dos Santos

Rio de Janeiro: SESES, 2015.

192 P. : IL.

ISBN: 978-85-5548-154-3

1. Memórias. 2. Linguagem. 3. Vetores. I. SESES. II. Estácio.

CDD 005.133

Diretoria de Ensino — Fábrica de Conhecimento

Rua do Bispo, 83, bloco F, Campus João Uchôa

Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio

Prezados(as) alunos(as),

Se estivéssemos escrevendo este texto há alguns anos provavelmente seria em uma sala, com uma máquina de datilografar e cercado por livros e papéis com anotações e lembretes para colocar no texto.

Hoje fazemos com um computador, conectado na internet o tempo todo (e ai dela se cair!) e conseguimos acessar links e mais links de referências e salvar as anotações na "nuvem".

Steve Jobs, que dispensa apresentação, disse que "Todos neste país deveriam aprender a programar um computador pois isto ensina a pensar". E é exatamente esta a maior vantagem em saber programar um computador: desenvolver o raciocínio e pensar abstratamente. Não estamos falando que você deve se tornar o ás da computação, a ideia aqui é saber as principais estruturas e usá-las não apenas para programar, mas também para poder desenvolver competências de poder resolver problemas de uma maneira mais rápida e eficiente.

E acredite, programar não é difícil! Intimida? Talvez, mas "o que não intimida quando não se sabe?", como pergunta o jogador da NBA Chris Bosh.

Saber programar vai abrir várias oportunidades, não só de empregos, vai abrir fronteiras de resolução de problemas, de organização de informações, de criação de novos negócios entre outras coisas.

E é este convite que nós fazemos a você. Vamos experimentar! Você vai gostar porque é divertido! E muito útil! Contamos com você!

Quer saber a opinião de alguns famosos sobre programação?

Clique aqui :-) (use seu celular com leitor de QR Code)

Bons estudos!

1

Introdução à Lógica de Programação

Quando você estiver navegando na internet, use os mecanismos de pesquisa para perceber quantos sites ensinam as pessoas a aprender a programação de computadores gratuitamente.

A programação de computadores está fundamentada em um assunto chamado lógica de programação o qual pode ser aplicado em qualquer área do conhecimento, como por exemplo, um engenheiro pode explicar para seus colegas a descrição de um processo industrial por meio de um algoritmo, um bioinformata pode discutir um determinado assunto no sequenciamento de genes usando um algoritmo específico e vários outros exemplos. O que aprendemos na lógica de programação nos ajuda a pensar abstratamente e entender o mundo real por meio de comandos e procedimentos encadeados.

Além disso, estamos cada vez mais conectados na internet, não é? Os celulares estão ficando mais acessíveis, assim como os planos de dados das operadoras facilitam o acesso, na verdade estamos cercados! E conhecer o funcionamento dos programas vai facilitar cada vez mais conhecer como os aplicativos de celulares funcionam.

Nosso trabalho aqui é aprender sobre algoritmos. Saber o que é, como funciona, onde e como se aplica. E usar uma linguagem de programação como suporte, o C/C++, que é uma linguagem muito usada atualmente e base para muitas linguagens atuais.

OBJETIVOS

- Identificar os componentes básicos de um computador
- Reconhecer as diferentes representações da lógica de programação.
- Interpretar uma sequência lógica em diferentes representações.

1.1 Histórico

A história dos algoritmos se confunde com a história da computação. O computador na verdade é uma máquina, coitada, sem qualquer tipo de inteligência. Mas o que as tornam tão poderosas, rápidas e temidas? Sua espantosa velocidade de fazer contas matemáticas.

Há alguns anos, um famoso enxadrista chamado Gary Kasparov desafiou um computador da IBM chamado Deep Blue para um *match*[1]. Foi uma árdua batalha, mas Kasparov conseguiu ganhar de uma máquina que foi especialmente preparada para jogar xadrez. A máquina não tinha inteligência, porém possuía uma programação que aproveitava o excepcional *hardware* da máquina para analisar milhares de jogadas ao mesmo tempo e escolher a melhor delas. Porém, quem fez a programação desta máquina foram pessoas.

Houve uma revanche. A equipe da IBM fez um *upgrade* no Deep Blue e o tornou especialista em partidas de xadrez de Kasparov, ou seja, ela foi treinada para jogar xadrez e mais: jogar contra Kasparov. Quer dizer, os programadores da IBM tiveram que refazer os algoritmos para incluir jogadas de vários outros mestres em xadrez para enfim, conseguir derrotá-lo. Ou seja, o computador venceu por dois grandes motivos: primeiro, tinha um *hardware* extremamente rápido e específico para a tarefa, segundo, o computador foi preparado com jogadas de outros mestres e até do próprio Kasparov para saber quais seriam suas jogadas no futuro. Isso ocorreu em 1997.

CONEXÃO

Assista a um clipe sobre o Deep Blue aqui: https://goo.gl/A6IRYa. Embora esteja em inglês, é possível compreender o que o narrador diz e o seu contexto.

1. Match, no xadrez, é um conjunto de partidas com regras bem definidas. Possuem um esquema de melhor de 3, 5 ou 7 partidas a fim de não haver empate

Este exemplo serve para mostrar que por trás de qualquer máquina, há a intervenção fundamental de um humano. O computador sempre foi uma máquina e para operá-la sempre foi necessário um humano e um conjunto de regras e procedimentos para operar a máquina corretamente. E há quem diga que o que derrotou Kasparov foi um erro, um bug no programa do Deep Blue. O erro desestabilizou Kasparov emocionalmente e o fez perder o jogo.

CONEXÃO

Veja o artigo sobre o bug. Disponível em: <goo.gl/tecTSI>.

A história da programação de computadores começa há muito tempo e muitos autores citam que na Babilônia antiga, por volta de 3500 AC foi criada a escrita cuneiforme[2].

Os sumérios, criadores deste tipo de escrita, utilizavam tábuas de argila, veja a figura 1.1 para escrever sendo que os textos eram compostos basicamente de poemas e histórias, contratos e acordos, texto sobre astronomia e matemática.

O descobrimento destas tábuas de argila possibilitou perceber que os babilônios utilizavam um sistema de numeração com base 60 (nós usamos a base 10) para poder representar número de ponto flutuante. A numeração babilônia influencia nossos dias, pois ainda temos alguns vestígios como a divisão da hora em 60 minutos, os graus de uma circunferência, os ângulos internos de um triangulo equilátero.

A escrita cuneiforme também influenciou os algoritmos, pois com ela eram escritos vários procedimentos usando linguagem natural, como por exemplo:

"Para calcular o volume da cisterna, se o seu raio for 2 e sua altura 7, então o volume é 3,1 vezes 2 vezes 2 vezes 7".

2 Cuneiforme: forma de cunha.

Figura 1.1 – Tábua de argila com escrita cuneiforme. Fonte: Wikipedia.

É um algoritmo bem arcaico e os números eram usados no lugar das variáveis.

Portanto, a necessidade de documentar processos e transformá-los em instruções para serem repetidas é muito antiga. Temos que lembrar que os povos da antiguidade cresciam separadamente e conforme a mistura de culturas foi aumentando, várias áreas do conhecimento foram sendo desenvolvidas.

Como exemplo, vamos contar um caso ocorrido em Bagdá, onde havia um matemático, astrônomo, astrólogo, geógrafo e autor persa chamado Abu'Abd Allah Muhammad ibn Musa al-Khwarizmi (780 a 850 dC) , ou melhor, al-Khwarizmi, que apresentou várias contribuições para a álgebra, equações lineares e quadráticas. A palavra algoritmo vem da raiz latina do seu nome, *algoritmi*.

A forma de apresentação dos seus trabalhos era feita usando linguagem natural, porém simbólica que tem certa semelhança com os algoritmos que conhecemos atualmente. Um importante estudo foi também na área de numeração, pois os números indo-arábicos, como sabemos, são a base da nossa numeração atual.

O livro original deste trabalho foi perdido, mas a tradução para o latim seria Algorthmi de número Indorum, este trabalho é um dos primeiros relatos que temos sobre algoritmos.

Existem outras contribuições da antiguidade para os atuais algoritmos, como por exemplo:

• Euclides (300 AC), o pai da Geometria, descreveu um algoritmo para calcular o máximo divisor comum (MDC), chamado de Algoritmo de Euclides (veja a animação do algoritmo em: http://pt.wikipedia.org/wiki/Algoritmo_de_Euclides)

• Alexander de Villa Dei (1175-1240) foi um autor, professor e poeta francês que escreveu textos de gramática e aritmética em latim, todos em verso. Alexander escreveu Carmen de Algorismo, um poema sobre aritmética o qual ensinava os seus alunos a arte do cálculo.

É claro que não havia um método formal para descrever os algoritmos. Como percebemos, eram usados a linguagem natural, poemas e até música.

A descrição dos algoritmos começou a sofrer maiores influências com os trabalhos de Ada Augusta Byron King, Condessa de Lovelace (1815-1852), ou Ada Lovelace. Ela foi uma matemática e autora inglesa que é reconhecida atualmente por ter escrito o primeiro algoritmo para ser processado por uma máquina: a máquina analítica de Charles Babbage.

A máquina analítica de Charles Babbage (1791-1871) foi um dos primeiros computadores que conhecemos. Tratava-se de uma máquina conceitual de uso geral que podia somar, subtrair, dividir e multiplicar de maneira sequencial na razão de 60 operações por segundo. Ela tinha muita semelhança em conceito, com os atuais computadores. Charles Babbage foi um cientista, matemático, engenheiro mecânico e inventor inglês o qual morreu antes de ver sua máquina totalmente construída. Veja a máquina na figura 1.2.

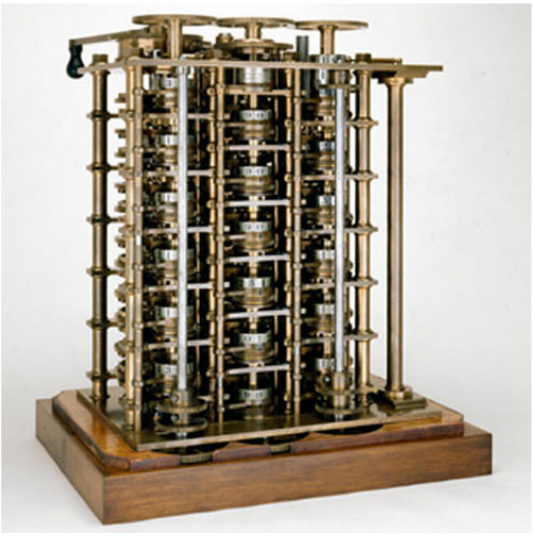


Figura 1.2 – A máquina analítica de Babbage (Fonte: computerhistory.org)

A contribuição de Ada se deu porque Charles Babbage ministrou uma palestra sobre sua máquina na Universidade de Turim, na Itália. Um engenheiro italiano, ouvinte da palestra, publicou esta em francês a qual foi republicada em uma revista científica da Universidade de Geneva, na Suíça. Babbage pediu então para Ada traduzir esta publicação para o inglês e adicionar as anotações de Ada sobre o tema. A tradução de Ada com suas anotações foram publicadas com suas iniciais "AAL", pois as mulheres não possuíam educação formal naquela época, tampouco em matemática. Ada tinha aulas particulares. Casou com 19 anos e teve três filhos seguindo a tradição feminina da época.

As notas de Ada foram republicadas em 1953, mais de cem anos após sua morte. Nesta ocasião, a máquina de Babbage foi reconhecida por ser um computador e as notas de Ada como um *software*.

Há um algoritmo para calcular a Sequência de Bernoulli, porém nunca chegou a ser testado, pois a máquina de Babbage ainda não tinha sido construída, mas de qualquer forma, é considerado o primeiro programa e Ada é o nome de uma linguagem de programação, dado como homenagem.

Em relação às linguagens de programação, Konrad Zuse (1910-1955) teve uma grande importância. Ele foi o inventor do primeiro computador eletromecânico o qual fazia cálculos e apresentava os resultados em fita perfurada. A primeira versão deste computador, chamada Z1, foi construída na sala de sua casa.

O computador na verdade era um ábaco mecânico controlado por pinos de metal e correias e programado por fitas perfuradas. Em 1945, Zuse desenvolveu o projeto de uma linguagem de programação chamada Plankalkul.

Esta linguagem possuía várias contribuições, as quais possibilitaram Zuse criar vários programas como ordenação, busca, análise sintática etc.

- Atribuição, expressões aritméticas, subscritos
- Uso do bit como tipo primitivo e tipos mais complexos como inteiro, real, vetores, etc
- Utilização de laços e comandos condicionais, além de sub-rotinas

Veja um exemplo desta linguagem. O exemplo é de um programa que calcula o valor máximo de 3 variáveis usando a função max definida no programa:

```
P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) => R0[:8.0]
max(V0[:8.0],V1[:8.0]) => Z1[:8.0]
max(Z1[:8.0],V2[:8.0]) => R0[:8.0]
END
P2 max (V0[:8.0],V1[:8.0]) => R0[:8.0]
V0[:8.0] => Z1[:8.0]
(Z1[:8.0] < V1[:8.0]) -> V1[:8.0] => Z1[:8.0]
Z1[:8.0] => R0[:8.0]
END
```

Como podemos perceber, é uma linguagem que não influenciou outras, dada sua complexidade, porém ainda assim é considerada de alto nível. Ela só foi amplamente publicada em 1972 e seu compilador em 1998.

Lembre-se que o termo "alto nível" usado para linguagens significa que a linguagem de programação é escrita de uma forma que seja compreensível para o programador. "Baixo nível" é usado para linguagens de máquina.

Temos que lembrar novamente da tecnologia disponível na época: não havia teclado, monitor e *mouse* para operar um computador. Um computador era uma máquina que ocupava espaços físicos muito grandes e era basicamente composta de fios e válvulas. Programar na verdade era trocar fios de lugar e quando era disponível, furar cartões. Era uma atividade que durava muito tempo e não podia haver erros (um erro resultava em uma nova perfuração de cartões, imagine então para descobrir onde estava o erro!).

Em 1949, John Mauchly propôs a linguagem Short Code. Era uma linguagem de alto nível que representava expressões matemáticas de uma maneira possível de ser entendida.

Em 1950, Alick Glennie desenvolveu a Autocode, a qual usava um compilador para converter a linguagem em linguagem de máquina. Esta linguagem usava o computador Mark I da Universidade de Manchester.

Outra linguagem de programação precursora foi a Flow Matic, desenvolvida por Grace Hooper para o computador UNIVAC entre 1955 e 1959. Esta linguagem foi uma tentativa de evitar que usuários de processamento de dados lidassem com tanta matemática na programação de computadores e assim foi escrita uma especificação para a linguagem e implementado um protótipo no final dos anos 50. A Flow Matic foi forte influência para o aparecimento da linguagem COBOL a qual é usada até hoje.

A linguagem Fortran foi desenvolvida pela IBM na metade dos anos 50 e foi a primeira linguagem de alto nível amplamente utilizada, inclusive até hoje para previsão do tempo, dinâmica de fluidos, etc..

O projeto da linguagem na IBM foi liderado por John Backus e o que era para ter demorado seis meses, demorou dois anos. Com a linguagem Fortran foi possível diminuir o número de erros dos programadores e além disso possuía um compilador que gerava um código de qualidade.

Outras linguagens que apareceram nesta época e ainda são usadas até hoje são LISP (1958) e COBOL (1959). A Algol 68 foi uma linguagem especial a qual possuía muitas características que influenciou Niklaus Wirth a desenvolver a linguagem Pascal em 1969 e publicada em 1970. A linguagem BCPL que foi base para a criação da linguagem C foi criada em 1967. A linguagem BASIC, muito famosa e usada na década de 80, foi criada em 1964.

A década de 80 foi a época de consolidação das chamadas linguagens dos paradigmas imperativos, veja o box.

Existem vários paradigmas nas linguagens de programação:

- Paradigmas imperativos: compõem as linguagens que facilitam a computação por meio de mudanças de estado. Temos os paradigmas:
  - Procedural: os programas são executados por meio de chamadas sucessivas a procedimentos separados. Fortran e Basic são exemplos deste paradigma.
  - Estrutura de blocos: a maior característica deste paradigma é o aninhamento de escopos como ocorre nas linguagens Algol 60, Pascal e C.
  - Orientação a objetos: é o paradigma que suporta a criação de classes e objetos. Como exemplo temos o C++, Java, Python, Ruby, C# e outras
  - Computação distribuída: neste paradigma, uma rotina pode ser dividida em partes que podem ser executadas independentemente. Exemplo: linguagem Ada.
- Paradigmas declarativos: os programas são declarados como uma relação ou função.
  - Paradigma funcional: um programa é um conjunto de funções como em Lisp, Scheme e Haskell
  - Programação Lógica cujo maior exemplo é a linguagem Pro

A década de 60 e 70 foi muito produtiva em relação ao aparecimento de novas linguagens. A década de 80, ao invés de continuar nessa produção de novos paradigmas, foi uma época na qual as características já desenvolvidas foram melhoradas como é o caso da linguagem C++, que veio da linguagem C e adicionou os conceitos de orientação a objetos.

Ainda nesta década houve um grande avanço na programação de sistemas em larga escala e em especial com a adoção de módulos ao invés de programar todo o sistema. Linguagens como Modula, Ada e ML foram bastante usadas porque tratavam os módulos de uma maneira muito eficiente para a época.

Algumas linguagens que foram desenvolvidas neste período incluem: C++ (1980), Ada (1983), Matlab (1984), Objective-C (1986), Perl (1987), TCL (1988) e outras.

A década de 90 é conhecida como a era das linguagens da internet. E não podia ser diferente: a internet cresceu rapidamente principalmente no meio dos anos 90 abrindo uma nova plataforma para desenvolvimento de aplicativos. Linguagens como Javascript (1995), Python (1991), Visual Basic (1991), Object Pascal (1995), Java (1995), PHP (1995) e outras apareceram nesta década.

Todas estas linguagens trazem consigo o paradigma da orientação a objetos e características como o desenvolvimento rápido de aplicações (RAD), além de possibilidade de desenvolvimento de *scripts* próprios para Internet. A linguagem Java teve um grande destaque, pois possibilitava ter o seu código fonte escrito em uma plataforma e ter a aplicação rodando em outras plataformas diferentes.

Após a década de 90 e início dos anos 2000, percebe-se que as linguagens continuam se aperfeiçoando tanto no mercado quanto na pesquisa. Algumas tendências que são encontradas nas atuais linguagens são:

Aumento do suporte à programação funcional em linguagens usadas comercialmente. Isto implica em geração de código mais fácil e maior produtividade

- Aumento do suporte à programação paralela e concorrente
- Mecanismos para adicionar segurança e confiança na linguagem
- Desenvolvimento orientado a componentes
- Aumento do estudo em mobilidade e distribuição de aplicativos
- Integração com bancos de dados e XML
- Aumento do desenvolvimento de ferramentas de código aberto como ocorre nas linguagens Python, Ruby, PHP e outras
- Outras

1.2 Organização de computadores

A organização de computadores é uma área da computação que estuda os componentes físicos de um computador, ou também chamado de *hardware*. É importante estudar a organização dos computadores para entender como um programa é executado, armazenado e como ele se comunica com as outras partes do computador.

A ciência da computação não é uma área tão velha assim. Considerando o tópico 1.1 onde vimos que os primeiros computadores datam de 1600, temos que o computador tem 500 anos e se considerarmos os primeiros computadores transistorizados (década de 50), temos apenas 60 anos de evolução.

Mas um elemento permanece fiel às primeiras gerações: sua arquitetura interna. Embora existam evoluções, a arquitetura inicialmente concebida por John Von Neumann foi adaptada aos nossos dias mas ainda assim é utilizada.

CONEXÃO

Conheça um pouco mais sobre Von Neumann neste link da Wikipedia: http://pt.wikipedia.org/wiki/John_von_Neumann

Von Neumann sugeriu uma forma de organizar o computador por meio dos seguintes elementos:

- Uma memória, dividida em primária e secundária
- Uma unidade de controle
- Uma unidade lógica e aritmética (ALU)
- Dispositivos de entrada e saída

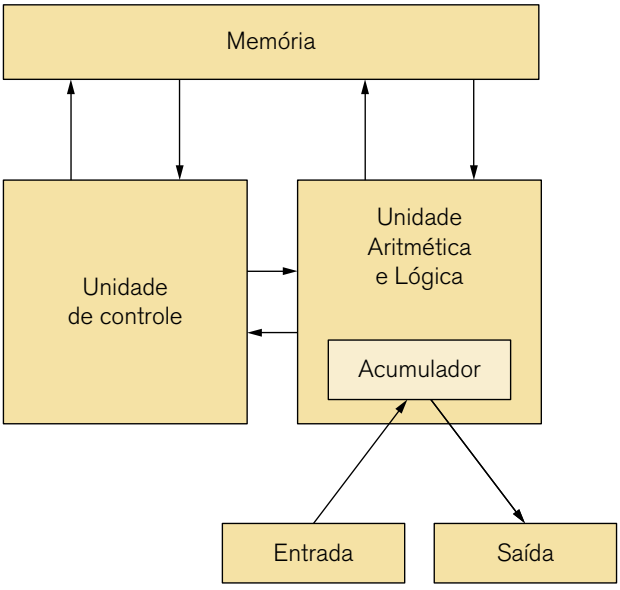


Figura 1.3 – Diagrama da arquitetura de Von Neumann (Fonte: http://pt.wikipedia.org/wiki/Arquitetura_de_von_Neumann)

Atualmente os computadores ainda possuem variações desta organização. Basicamente a arquitetura permite que o computador funcione em ciclos onde em cada ciclo ocorre a busca de novas instruções, a decodificação da instrução e a sua execução. Os ciclos são controlados e promovidos pela CPU (Unidade Central de Processamento), composta pela Unidade de Controle e Unidade Lógica Aritmética e tem como principal função executar os programas que estão armazenados na memória principal, buscar as instruções dos programas, decodificá-las e executá-las sequencialmente, veja a figura 1.4.

Portanto, percebemos que a CPU na verdade não é um componente unitário. É um componente que é dividido em várias partes entre elas a Unidade de controle, a ULA, o contador de programa e os registradores, que podem ser especiais ou de uso geral, cache de memória e outros componentes mais modernos. Todos esses elementos ficam encapsulados em um único chip chamado de processador. Na figura 1.4 também são mostrados o aspecto externo de um processador e uma foto amplificada da sua arquitetura interna.

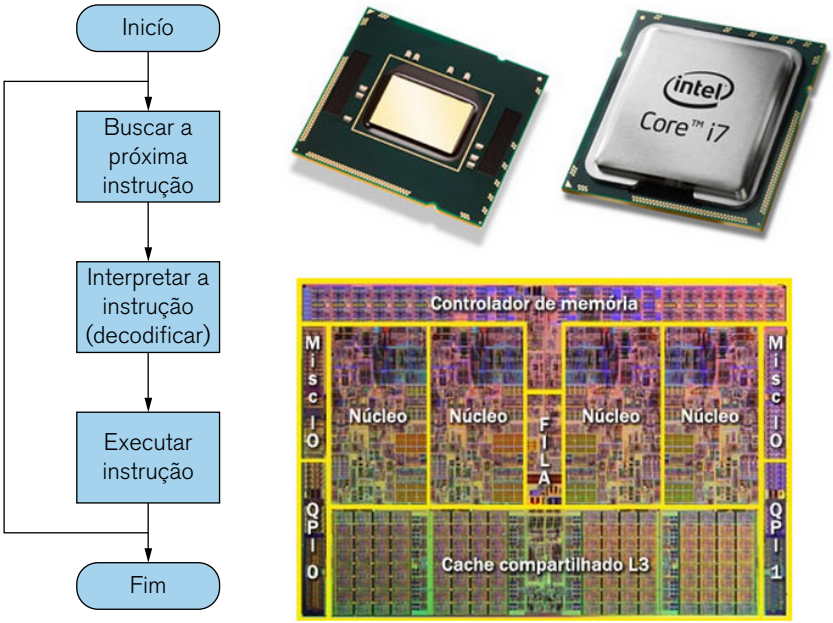


Figura 1.4 – Ciclo da arquitetura de Von Neumann, um processador Intel I7 e a sua arquitetura e organização interna (Fonte: http://tecnologia.hsw.uol.com.br/core-i71.htm)

A memória é o local onde os dados e programas ficam armazenados para serem executados. Existem vários tipos e classificações de memória, entre elas:

- Voláteis e não voláteis
- Quanto à sua forma de operação e acesso (Read Only Memory (ROM) – memória somente de leitura, Random Access Memory (RAM) – memória de acesso aleatório)
- Velocidade de operação

1.2.1 As memórias secundárias

As memórias secundárias servem para auxiliar a memória RAM quanto ao armazenamento de dados. Elas também são voláteis e podem ser escritas e lidas.

Possuem custo mais baixo que as memórias RAM e normalmente podem armazenar mais informação, chegando a Terabytes atualmente e por isso são usadas principalmente para armazenamento de programas não ativos (que não estão sendo executados no momento).

Os maiores exemplos deste tipo de memória são os pendrive (flash drives), discos rígidos (hard disk – HD), cartões SD, MD, etc...

1.2.2 A MEMÓRIA RAM

A memória RAM é do tipo que pode ser lida e gravada, portanto considerada volátil. Nela são carregados os programas que estão sendo executados naquele momento pela CPU. Quando a energia cessa, os dados presentes neste tipo de memória são apagados.

Antes o custo era muito alto, porém com a evolução dos componentes eletrônicos, a memória RAM tem diminuído de preço.

Ela possui alta velocidade, pois suas características eletrônicas e o barramento que elas utilizam as aproximam da CPU e tornam a comunicação com os outros componentes mais fácil e mais rápida.

A figura 1.5 mostra vários tipos de memória RAM e sua evolução ao longo do tempo. Atualmente as memórias RAM mais procuradas para os computadores pessoais são do tipo DDR3.

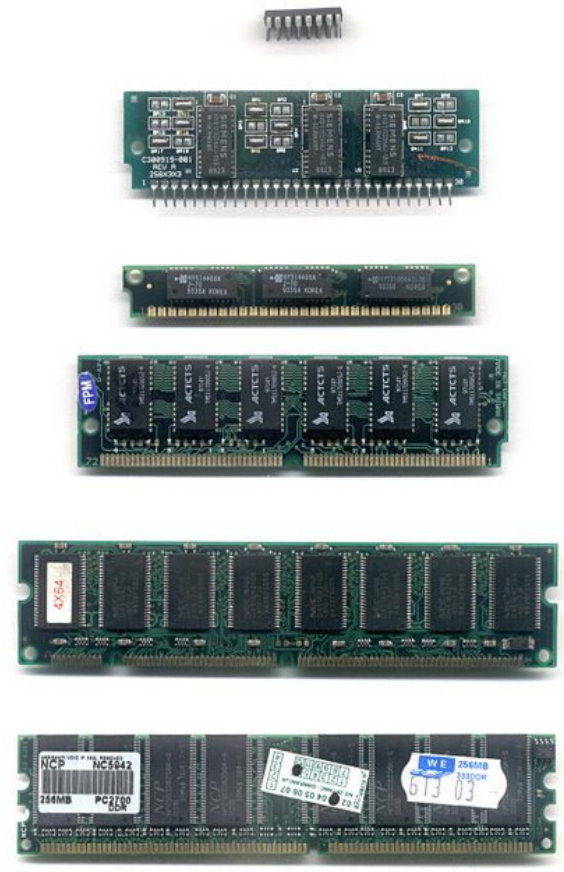


Figura 1.5 – Vários tipos de memória RAM (Fonte: http://pt.wikipedia.org/wiki/RAM)

Ao longo do tempo as memórias RAM foram evoluindo e sofrendo modificações em sua arquitetura interna. Essas modificações ocorreram para acompanhar a arquitetura das placas-mãe e para aumentar a capacidade de armazenamento de dados e a taxa de transferência. A DDR3 é a versão mais nova de uma família de memórias RAM, chamadas de SDRAM que possui um ciclo de trabalho que é sincronizado com o *clock* do processador.

1.2.3 Caches e registradores

São componentes que apareceram para aumentar a velocidade e eficiência de todo o *hardware* do computador. Também são variações da memória RAM e fisicamente estão localizados perto do processador para poder ser usadas como armazenamento temporário de operações de dados feitas pelo processador com mais frequência.

Por serem variações da RAM, são voláteis e podem ser escritas e lidas.

Possuem desempenho muito superior em relação a velocidade de trocas de dados mas não possuem grande capacidade de armazenamento.

1.2.4 As memórias somente leitura (ROM)

São representadas atualmente pelos CDs e DVDs (que não sejam regraváveis). Neste tipo de memória, a informação é gravada apenas uma vez e não pode ser sobrescrita guardando assim informações por bastante tempo.

Em relação a preço, é bem mais barata que os outros tipos de memória que já vimos, porém são muito mais lentas.

1.2.5 A placa mãe

A placa mãe (*mother board* ou *main board*) é o componente que conecta todos os outros elementos do computador anteriores. Além disso ela contém os barramentos por onde passam os dados de informações e controle e também fornece energia elétrica para os componentes.

A figura 1.6 mostra uma arquitetura típica de uma placa mãe. Percebemos que ela contém vários "setores" onde são conectados elementos de acordo com o seu tipo.

Basicamente os principais componentes que todo programador deve saber sobre um computador são esses. Na verdade, qualquer equipamento microprocessado atualmente possui estes componentes como por exemplo um aparelho para *home theater*, receptores de TV a cabo, os *smartphones* e tantos outros que encontramos no nosso dia a dia.

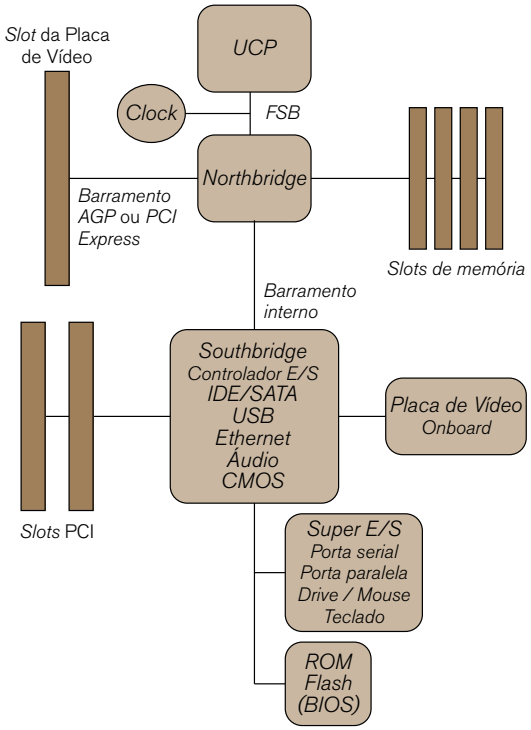


Figura 1.6 – Arquitetura típica de uma placa mãe

1.3 Lógica e lógica de programação

O estudo da lógica é bem abrangente pois desde a antiguidade os povos tem refletido e estudado sobre a lógica. Basicamente, a lógica tem duas definições principais: o estudo do uso do raciocínio em alguma atividade ou também pode ser entendida nos campos das exatas, ciências e matemática, ou também chamada de lógica matemática.

Basicamente, a lógica é a ciência que estuda as leis e critérios de validade que regem o pensamento e a demonstração, ou seja, a ciência dos princípios formais do raciocínio.

A lógica é usada pelos profissionais em geral, especialmente os das ciências exatas, porque estes possuem como objetivo solucionar problemas e atingir as metas com eficiência e eficácia usando recursos computacionais ou outros. O conhecimento na forma de lidar com problemas administrativos, de controle,

de planejamento e de estratégia necessita de grande atenção e desempenho de conhecimento do nosso raciocínio.

A lógica está presente no nosso dia a dia. Veja o exemplo:

```
A gaveta está fechada.
A agenda está na gaveta.
Logo, para pegar a agenda, preciso abrir a gaveta.
```

Outro exemplo:

```
Pedro é mais velho que João.
João é mais velho que Tiago.
Logo, Pedro é o mais velho e Tiago é o mais novo.
```

Lógico, não?

```
A lógica na verdade é a ciência que nos ajuda a colocar ordem no pensamento.
```

Diferença entre eficiência e eficácia: existem algumas definições mais técnicas sobre essas duas palavras. Vamos adotar uma mais prática: eficiência é quando um sujeito atinge os seus objetivos seguindo as normas e procedimentos já pré-definidos, ou seja, é fazer o certo. A eficácia é quando o sujeito também consegue atingir os objetivos, porém seguindo os procedimentos estabelecidos de uma maneira parcial e usa alguns elementos como a criatividade para se chegar no objetivo, ou seja, é a coisa certa. Para quem gosta de futebol, eficiência é jogar bonito e ganhar o jogo, eficácia é ganhar o jogo com um gol "de bico" aos 45 minutos do segundo tempo.

A lógica de programação é um campo específico da lógica matemática que envolve o uso da lógica na resolução de problemas computacionais, especialmente na área de desenvolvimento de algoritmos.

Muitos recursos são usados pelos profissionais para representar o seu raciocínio lógico como os fluxogramas, os diagramas de bloco e até mesmo a linguagem natural e variações da língua nativa do profissional (no nosso caso, o português). Por meio desses recursos, como por exemplo o fluxograma, é

possível representar graficamente a sequência de operações a serem feitas por um programa.

Desta forma, de posse do desenho do raciocínio, é possível codificar o desenho em alguma linguagem de programação por outra pessoa, desde que sejam adotados padrões para o desenho.

Alguns autores afirmam que a técnica mais importante no projeto da lógica de programação é a chamada programação estruturada, que possui como meta:

- Agilizar o desenvolvimento do código fonte
- Facilitar a depuração do programa
- Verificar possíveis erros
- Facilitar a manutenção dos programas

Além disso, a programação estruturada possui quatro etapas essenciais:

1. Criar as instruções e comandos em sequências interligadas apenas por estruturas sequenciais, repetitivas ou de seleção
2. Criar instruções em blocos pequenos e combiná-las
3. Compartilhar os módulos do programa entre os vários participantes da equipe, sob a supervisão de um programador mais experiente
4. Revisar o trabalho em reuniões frequentes e previamente programadas.

A seguir vamos apresentar algumas formas de representar os algoritmos e relacionar os diagramas com a programação estruturada.

1.4 Algoritmos e formas de representação

Basicamente um algoritmo é uma sequência de passos que devem ser executados ordenadamente para cumprir um determinado objetivo. É como uma receita de bolo: se a receita foi bem escrita e você segui-la direitinho, seu bolo vai sair conforme o desejado. É claro que alguma experiência é indicada na produção de um bolo, mas uma boa receita dará as dicas e demais "pegadinhas" para que o bolo seja apreciado por todos.

Outro exemplo é de um robô. O robô faz exatamente as tarefas que foi programada, nem a mais, nem a menos. Imagine que você vai programá-lo a sair de São Paulo e ir para o Rio de Janeiro de carro. Muitos vão dizer: "é só pegar a via Dutra!" e não é bem assim. E os pedágios? E a questão dos radares de velocidade? E mais outros tantos detalhes.

Um algoritmo pode ser representado por muitas maneiras, desde figuras até textos escritos em português. O que une todas essas formas de representação é o significado que eles possuem para quem está lendo e interpretando.

O algoritmo, independente da forma pela qual esteja representado, tem que ser descrito de uma forma clara e fácil de ser seguida. Desta forma, no caso de uma depuração, ele será melhor rastreado e terá seus eventuais erros reparados mais facilmente.

1.4.1 Descrição Narrativa

A forma mais intuitiva é escrever um algoritmo por meio de um texto narrativo. Uma receita de bolo, como temos usado como exemplo, está escrita na forma narrativa. Porém, já percebemos que esta forma pode trazer problemas: já comentamos que as receitas de bolo podem ser imprecisas e esconder detalhes que vão resultar em pessoas insatisfeitas.

A descrição narrativa consiste em entender o problema e propor sua solução por meio de uma escrita em linguagem natural. A maior vantagem da descrição narrativa é que não é necessário aprender novas técnicas e conceitos, consiste somente em usar a linguagem nativa.

Esta é uma desvantagem da narrativa: ser imprecisa e pode gerar ambiguidades (vários tipos de interpretação). Isto acarreta futuramente problemas na forma de passar o algoritmo para uma linguagem de programação.

Porém, a narrativa é forte quando usada como comentário de um determinado trecho de um algoritmo explicando aquela parte para quem for codificá-la.

Vamos estudar um exemplo clássico: “trocar um pneu” na forma de descrição narrativa.

```
Afrouxe um pouco os parafusos
Levante o carro
Retire os parafusos
Retire o pneu
Coloque o pneu reserva
Aperte os parafusos
Abaixe o carro
Aperte os parafusos completamente
```

Quais são os problemas que você percebe no Algoritmo 1? Se você for um trocador de pneu com certa experiência vai entendê-lo perfeitamente, porém se você nunca trocou um pneu de carro na vida vai encontrar alguns pontos complicados: “como vou levantar um carro?”. É simples: “pegue o macaco!”. “Que macaco???”. Usando a receita novamente como exemplo, se você nunca fez um pudim de leite condensado vai ficar com muita dúvida na parte da receita onde pede para coloca-lo em “banho maria”! “Quem é essa Maria?”.

O algoritmo está correto, mas gera muitas dúvidas para quem não entende do assunto. A descrição narrativa é uma boa alternativa para a elaboração de algoritmos, mas gera imprecisão e falta de confiabilidade no entendimento do algoritmo. Além disso, temos muito texto para descrever coisas simples.

1.4.2 Fluxogramas

O fluxograma, ou *flowdraw*, é uma forma gráfica de representar algoritmos. O fluxograma foi muito utilizado para representar algoritmos por muito tempo. Ainda hoje muitas pessoas usam o fluxograma para desenhar a forma de execução de um processo.

O fluxograma possui algumas características fundamentais:

• Possui símbolos padronizados: cada símbolo representa uma ação ou funcionalidade e são universais, ou seja, os mesmos símbolos usados no Brasil são usados internacionalmente.

• Os fluxogramas possuem uma sintaxe e uma semântica bem definidas. A semântica de um fluxograma corresponde ao que cada símbolo significa. Ou seja, cada instrução possui um símbolo específico e o comando escrito dentro do símbolo tem que ser claro o suficiente. Sobre a sintaxe, ela corresponde ao uso correto dos elementos do fluxograma: cada símbolo permite que um conjunto de expressões próprias seja usado e as expressões mostram as operações a serem realizadas com os dados.

• É fácil de traduzir para qualquer linguagem de programação: devido aos padrões adotados nos símbolos, é possível fazer uma correspondência direta entre os símbolos e os comandos encontrados nas linguagens de programação, até mesmo as mais atuais, desde que esteja sendo considerada uma sequência lógica.

Um fluxograma possui algumas regras para que possa representar corretamente um determinado processo:

• O fluxograma precisa ser claro e muito bem escrito de forma que sua leitura não leve a gerar dupla interpretação de um determinado comando.

• Procure criar desenhos e quebrá-los em vários níveis. Deixe os níveis iniciais para o contexto principal e para as principais ideias. Os diagramas posteriores conterão o detalhamento na profundidade necessária.

• A direção de leitura de um fluxograma sempre é de cima para baixo, seguindo as setas. Quando o espaço físico não permitir, é possível recomeçar o desenho na coluna adjacente à esquerda do fluxo atual.

• É incorreto e "proibido" cruzar linhas de fluxo de dados

Observe a figura 1.7. Ela mostra um fluxograma simples representando o processo de trocar o pneu de um carro.

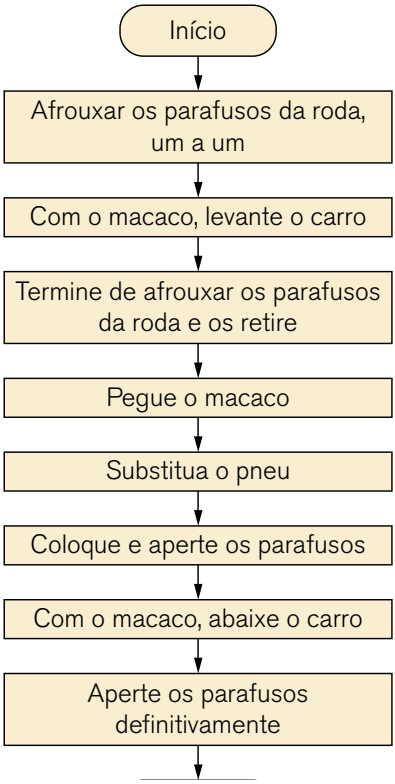


Figura 1.7 – Fluxograma representando o processo de trocar pneu.

O fluxograma da figura 1.7 é “bem comportado”, ou seja, percebemos que ele segue uma estrutura sequencial sem muitos desvios. Existem fluxogramas bem mais complexos do que o apresentado e isso varia de acordo com a situação que está sendo representada.

Como já vimos, cada figura em um fluxograma possui um significado. A tabela 1.1 mostra os principais elementos de um fluxograma e seu significado.

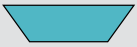
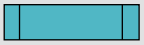
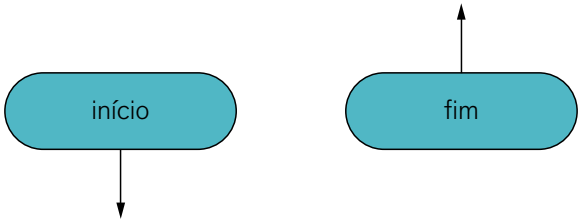
SÍMBOLO	NOME	FUNÇÃO
	Terminador	Representa o início e o fim de um processo.
	Fluxo	Mostra o sentido de execução do processo.
	Conector	MOstra ligações com outras partes do fluxograma.
	Atribuição	Armazena temporariamente o resultado de um processamento
	Processo	Faz o cálculo de expressões e/ou executa funções.
	Leitura de dados	Faz a entrada manual de dados, na execução do algoritmo
	Exibição	Apresenta os resultados de um processamento
	Decisão	Avalia uma expressão lógica ou relacional
	Subprograma ou processo pré-definido	Processa chamada de funções ou procedimentos

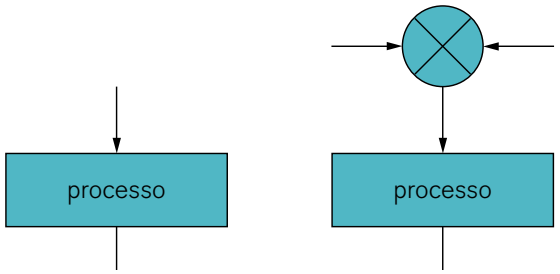
Tabela 1.1 – Símbolos e funções de um fluxograma

Continuando com as regras gerais de um fluxograma, observe as regras a seguir:

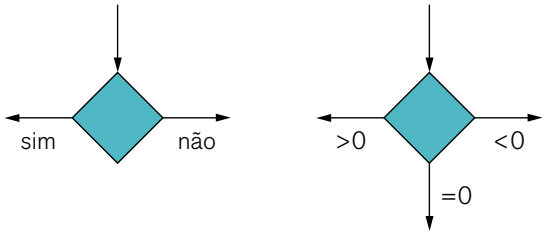
- Somente uma linha de fluxo deve sair ou chegar a um terminador:



• O processo permite apenas uma linha de saída de fluxo. Essa regra é bem desobedecida. Quando o programador achar que existem duas saídas, é importante que ele reveja o fluxograma a fim de considerar uma decisão e não um processo:



• A decisão permite que apenas uma linha de entrada chegue, mas permite que duas ou três linhas de saída sejam possíveis:



Ainda:

• O texto de que fica dentro de cada elemento deve estar adequado à instrução a ser executada

• Os conectores são usados para reduzir o número de linhas de um fluxograma

• Evite cruzar linhas a fim de deixar o fluxograma claro

• Normalmente a validação de um fluxograma é feito por um conjunto de testes chamados de "teste de mesa"

Usando os elementos apresentados, vamos mostrar um fluxograma representando um algoritmo para ler dois números, diferentes de zero, e calcular sua média.

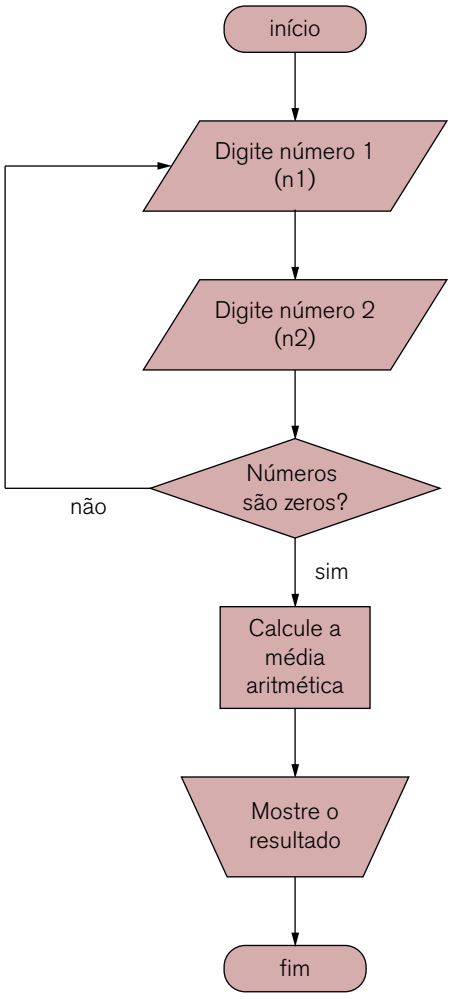


Figura 1.8 – Fluxograma – cálculo da média entre dois números.

Vamos analisar o fluxograma da figura 1.8:

1. O fluxograma começa com um símbolo indicando o início do processo. Observe que sai apenas 1 fluxo dele.

2. O primeiro paralelogramo pede para que o usuário digite (faça a entrada de dados) o primeiro número o qual será chamado de n1.

3. O segundo paralelogramo faz o mesmo, chamando o segundo número de n2.

4. O losango representa a decisão. Neste momento o programa terá que avaliar se algum dos números é igual a zero. Poderíamos ter melhorado a descrição deste símbolo, mas da forma como está fica claro que será feito um teste nos números.

5. Se algum dos números for zero, o fluxo do programa volta para o início da entrada de dados e faz a leitura novamente.

6. Se nenhum for zero, o fluxo avança para o processamento no retângulo onde será calculada a média aritmética entre n1 e n2.

7. O trapézio invertido representa a exibição do resultado. Em algumas notações de fluxograma, quando a exibição é feita em papel, como um relatório, é usado outro símbolo, mas basicamente para representar a exibição de dados usamos o trapézio como está mostrado aqui.

8. Seguindo o fluxo, o programa finaliza.

Agora vamos analisar outro exemplo e verificar que os fluxogramas podem ficar mais complicados a medida que o problema se torna mais complexo. O exemplo tem como objetivo avaliar e calcular uma equação do segundo grau lembrando que a forma desta equação é $ax^2 + bx + c = 0$. O programa deve calcular as raízes x1 e x2 da equação quando possível. Isto varia de acordo com a raiz de delta, onde o delta = $b^2 - 4*a*c$. Veja o fluxograma da figura 1.9.

O programa de cálculo das raízes de uma equação do segundo grau é mais complexo que o da média aritmética e percebemos que o desenho visualmente ocupa um espaço físico maior. Já deu para perceber que complexos maiores vão ocupar mais espaço e como normalmente esses desenhos são impressos, veja que isto pode ser um ponto negativo desta representação de algoritmo. Porém, observar um processo por meio de um desenho é muito eficiente para entender o processo como um todo.

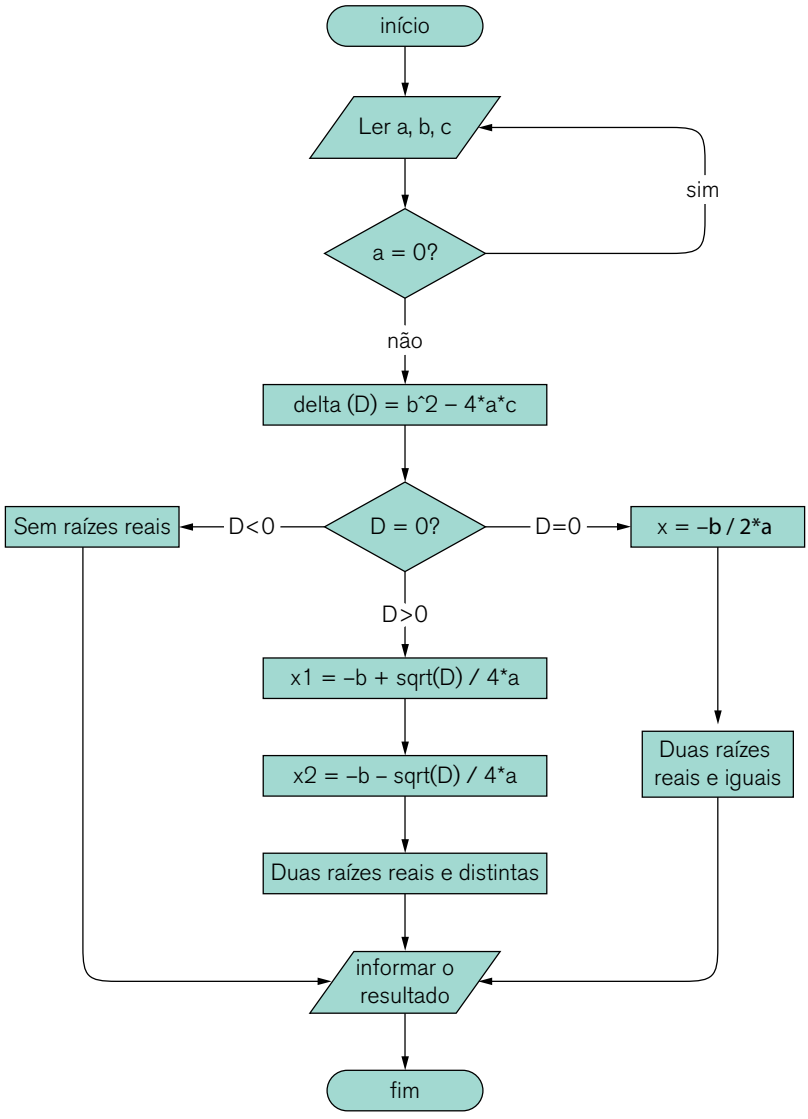


Figura 1.9 – Fluxograma - cálculo das raízes de uma equação do segundo grau.

Vamos analisá-lo:

1. Após o início do programa, é feita a leitura das 3 variáveis a, b, e c da equação.
2. Em seguida é necessário fazer uma avaliação do valor de a. Se for zero, a equação não é do segundo grau e assim o fluxo volta para o início, para fazer a leitura de novos dados.
3. Se a variável a não for zero, é feito o cálculo de delta, que foi chamado D na nossa representação. O sinal do acento circunflexo em b^2 significa uma exponenciação, neste caso, b elevado ao quadrado.
4. Após o cálculo de delta, ele é analisado pois pode assumir três valores: igual, menor ou maior que zero.
5. Se delta for menor que zero, a equação não terá raízes reais e assim o programa é levado para o final, informando o resultado para o usuário.
6. Se delta for maior que zero, a equação terá duas raízes reais e diferentes. O programa prossegue para o cálculo das duas raízes (x1 e x2), informa o resultado e finaliza. Observe que neste processo, usamos a função sqrt() a qual faz o cálculo da raiz quadrada do valor de delta (D). Vamos explicar o uso de funções em outro capítulo.
7. Se delta for igual a zero, o fluxo prossegue para o cálculo das duas raízes reais e iguais. O programa calcula as raízes, informa o resultado e finaliza.

1.4.3 Programação estruturada

Os fluxogramas são bastante utilizados para um paradigma de programação que já vimos um pouco chamado programação estruturada. Como o próprio nome sugere, este tipo de paradigma está baseado em três estruturas principais: as estruturas sequenciais, as estruturas de decisão e as estruturas de repetição. Toda linguagem de programação que está baseada neste paradigma possui estas estruturas.

Estas estruturas tem um ponto em comum: possui um ponto de entrada e um único ponto de saída e isto pode ser representado por um fluxograma. De fato, a programação estruturada tem muita compatibilidade com os fluxogramas, por isso é que fluxogramas são usados até hoje: porque as linguagens principais que são usadas ainda possuem elementos estruturados.

Vamos examinar cada uma dessas estruturas a seguir, porém teremos um capítulo dedicado para cada uma delas mais adiante.

1.4.3.1 Estrutura sequencial

Como o nome sugere, trata-se de uma sequência ou seja, o fluxo de execução de um programa é executado linearmente com os comandos sendo executados um após o outro. Neste caso, deve existir apenas um caminho possível no conjunto de instruções de um algoritmo.

Vamos examinar um exemplo de uma estrutura sequencial: o objetivo é determinar o valor do saldo no final do 3º mês de uma aplicação financeira, com investimento inicial de R$100,00, juros de 1% ao mês.

> Você lembra das aulas de física quando tinha que calcular o valor da velocidade de um carro, da distância percorrida e etc.? Lembra que você usava fórmulas para isso? Por exemplo, a fórmula da velocidade média é Vmédia = dt/dS, onde dt é a variação do tempo e dS é a variação da distância percorrida. Ou seja, esses elementos são chamados de variáveis, ou valores que podem variar em um programa ou, no caso, em uma equação.

No nosso exemplo teremos algumas variáveis:

- i: valor do investimento inicial
- j: a taxa de juros
- p: saldo ao final do 1º mês
- s: saldo ao final do 2º mês
- t: saldo ao final do 3º mês

O fluxograma para o exemplo é mostrado na figura 1.10. Veja o quanto o fluxograma colabora e nos mostra visualmente o fluxo do processo. Olhando a figura é muito fácil perceber que a estrutura sequencial é caracterizada por uma sequência ordenada de comandos. Não há desvios e nem repetições, é uma sequência de comandos: um após o outro, até o fim do processo.

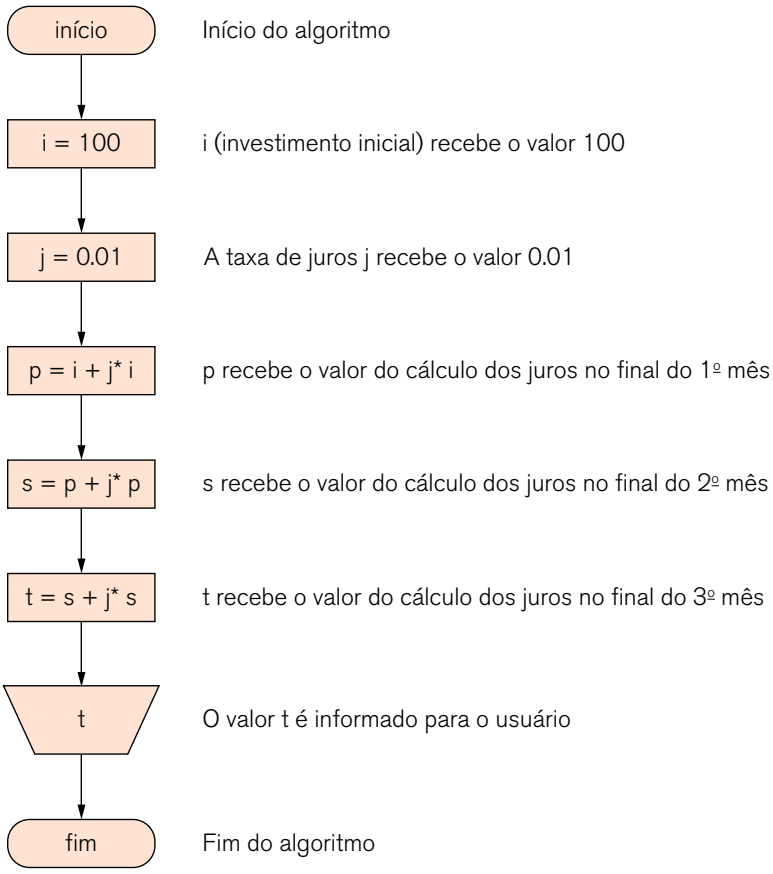


Figura 1.10 – Fluxograma da estrutura sequencial

1.4.3.2 A estrutura de seleção

Existem situações nas quais o processo precisa fazer uma decisão para poder continuar. Por exemplo, em uma linha de produção de suco de laranja existe um sensor que separa as boas das ruins por meio de uma análise de imagem. O sensor verifica a imagem e baseado em um padrão de uma fruta de boa qualidade analisa a amostra e deixa-a na esteira ou a retira de produção. Ou seja, é feita uma decisão, uma seleção. Esta é a característica principal da estrutura de seleção. Existem muitas situações parecidas com esta do exemplo.

Como exemplo de um fluxograma que usa esta estrutura, vamos estudar o cálculo do resto da divisão inteira entre dois números inteiros positivos.

Novamente vamos usar algumas variáveis:

- a é o valor do dividendo
- b é o valor do divisor
- q é o valor do quociente
- r é o valor do resto

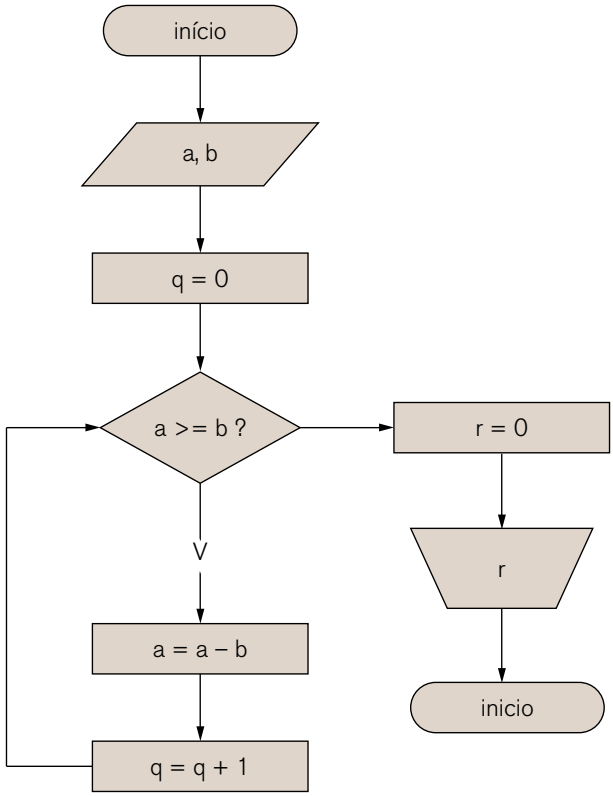


Figura 1.11 – Fluxograma com estrutura de seleção

Assim como a figura 1.10, a figura 1.11 mostra visualmente de uma maneira muito eficiente como é organizada a estrutura de seleção, ou também chamada de estrutura de decisão.

O símbolo desta estrutura em um fluxograma é o losango, e como vemos na figura 1.11, é feita uma comparação entre os valores de a e b. Caso o valor seja positivo, o fluxo toma um determinado caminho, tornando a condição verdadeira. Caso a condição não seja satisfeita, o que a torna falsa, o fluxo segue por um caminho diferente.

1.4.4 A estrutura de repetição

Como o nome diz, este tipo de estrutura serve para repetir alguma ação até que ou enquanto uma condição é satisfeita. Veremos adiante que existem alguns detalhes quanto ao controle da repetição.

Basicamente temos as seguintes formas:

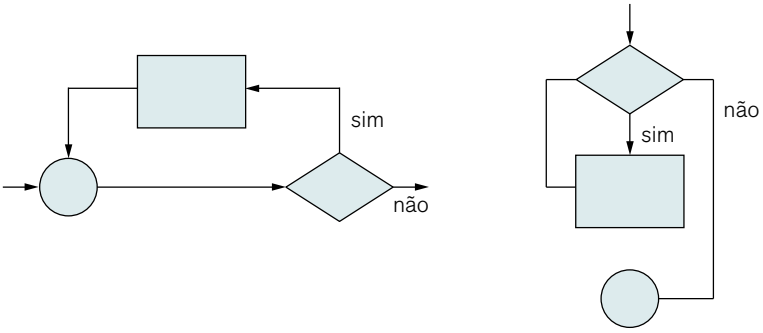


Figura 12 – Estrutura de repetição "faça ... enquanto".

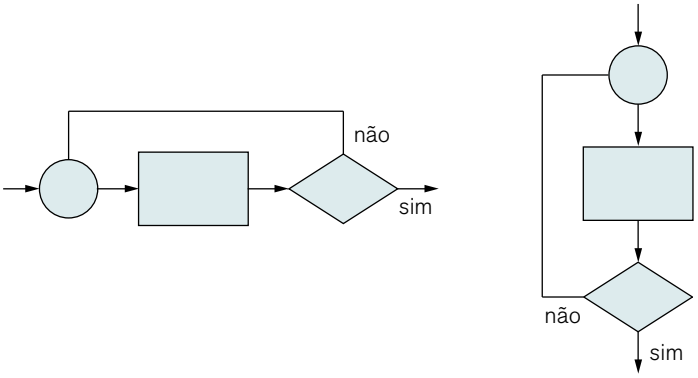


Figura 13 – Estrutura de repetição "faça ... até que".

A figura 1.12 e a figura 1.13 são bem parecidas. Porém existe uma diferença fundamental nos diagramas. Veja a posição do “sim” e “não” de cada tipo e perceba também a forma do desenho. Cada um desses tipos é usado em situações específicas e mostram o quanto um fluxograma pode contribuir para o processo de descrição do algoritmo.

Basicamente, a repetição funciona com o conhecimento ou não da quantidade de vezes que a ação será repetida ou se o teste da condição é feito no início ou no fim da repetição.

1.4.5 Português estruturado

Outra forma de representação de algoritmos é o chamado Português Estruturado. Esta forma é a que mais se aproxima da definição de algoritmos como conhecemos.

O português estruturado é uma simplificação da nossa linguagem natural na qual usamos frases simples e estruturas que possuem um significado muito bem definido. Apesar disso, a aparência do português estruturado é muito semelhante a uma linguagem de programação tradicional. Possui um conjunto de palavras e regras que formam as sentenças válidas e compõem a sintaxe da linguagem.

O português estruturado é baseado em uma Program Design Language (PDL). A PDL é uma forma de notação muito parecida com a linguagem Pascal e foi escrita originalmente em inglês. Ela é usada como uma referência genérica para uma linguagem de projeto de programação e ser usada como referência para uma implementação em alguma linguagem computacional. Veja o Algoritmo 3.

O algoritmo a seguir tem como objetivo ler 4 notas do usuário, somar as notas, fazer a média aritmética e apresentar o resultado. Perceba que a forma de escrever o algoritmo é bem parecida com um programa.

A diferença entre uma linguagem de programação de alto nível (como o C) e uma PDL é que a PDL não pode ser compilada em um computador. Porém, o programa mostrado no algoritmo[Campo] foi feito usando um programa chamado Visualg o qual possui finalidades didáticas e pode executar os algoritmos e disponibilizar várias ferramentas para o estudante compreender como o algoritmo está sendo executado.

```
algoritmo "média"
var
    nome_aluno : caracter
    n1,n2,n3,n4 : real
    soma : real
    media : real
```

```
inicio
    escreva("Digite o Nome do Aluno: ")
    leia(nome_aluno)
    escreva("Digite a primeira nota: ")
    leia(n1)
    escreva("Digite a segunda nota: ")
    leia(n2)
    escreva("Digite a terceira nota: ")
    leia(n3)
    escreva("Digite a quarta nota: ")
    leia(n4)
    soma <-(n1+n2+n3+n4)
    media<-(soma/4)
    escreva(media)
fimalgoritmo
```

1.5 Linguagens de programação

A forma que o programador tem para se comunicar com o computador é por meio de uma linguagem de programação. Dizemos que é a maneira "alto nível" de linguagem.

Sabemos que atualmente existem muitas linguagens de programação diferentes e para o iniciante chega a ser difícil escolher alguma linguagem para poder aprender e se especializar. Já foi citado que o mais importante é conhecer a lógica e saber as estruturas existentes para poder construir programas, a linguagem chega a ser um fator secundário.

A figura 1.14 e a figura 1.15 mostram um *ranking* de linguagens e sua evolução ao longo dos anos de acordo com a empresa Tiobe. Esta empresa lida basicamente com questões relacionadas à qualidade de *software* e mensalmente divulga o *ranking* das figuras. O *ranking* é obtido por meio da coleta de informações na internet envolvendo o que foi publicado, número de pesquisas feitas nos mecanismos de busca e mostra apenas a quantidade obtida, não trata de esclarecer qual a melhor linguagem de programação. O *ranking* serve principalmente para os programadores verificarem o quanto a sua linguagem de uso está sendo buscada e acessada pela internet. Os dados da figura foram obtidos

em 2015 e podemos perceber que as linguagens derivadas do C ainda são muito comentadas pela internet. Ou seja, estudá-las é uma boa oportunidade de ter um bom suporte do mercado.

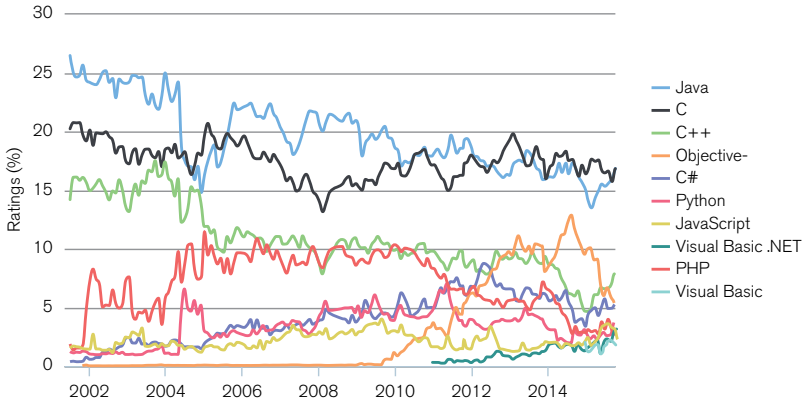


Figura 1.14 – Ranking da TIOBE (Fonte: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

MAY 2015	MAY 2014	CHANGE	PROGRAMMING LANGUAGE	RATINGS	CHANGE
1	2	^	Java	16.869%	–0.04%
2	1	v	C	16.847%	–0.08%
3	4	^	C++	7.875%	+1.89
4	3	v	Objective-C	5.393%	–6.40%
5	6	^	C#	5.264%	+1.52%
6	8	^	Python	3.725%	+0.67%
7	9	^	Javascript	3.127%	+1.34%
8	11	^	Visual Basic .Net	2.968%	+1.70%
9	7	v	PHP	2.720%	–0.67%
10	–	^^	Visual Basic	1.893%	+1.89%
11	10	v	Perl	1.820%	+0.35%
12	33	^^	R	1.444%	+1.06%
13	15	^	Delphi/Object Pascal	1.302%	+0.33%
14	19	^^	MATLAB	1.283%	+0.57%
15	12	v	Ruby	1.273%	+0.03%
16	27	^^	COBOL	1.169%	+0.58%
17	22	^^	PL/SQL	1.127%	+0.49%
18	–	^^	Swift	1.115%	+1.12%
19	18	v	Pascal	0.960%	+0.21%
20	37	^^	ABAP	0.887%	+0.57%

Figura 1.15 – Ranking das linguagens, referente à figura 1.14

Para quem tem mais experiência na área de informática, não deixará de notar que linguagens como Matlab e R (sim, isso mesmo, o nome da linguagem é "R") aparece no *ranking* de uma maneira destacada e ascensão. Particularmente, R e Matlab estão subindo no *ranking* porque são muito usadas na área de bioinformática e como esta área é cada vez mais crescente, essas linguagens crescem junto. Até mesmo o Cobol, que foi muito usada nas décadas de 70 e 80, ainda aparece em crescimento. Outra linguagem que vem se destacando segundo o índice da Tiobe é o Python, que está sendo usada por grandes empresas como Facebook e Google nos seus sistemas.

As linguagens de programação possuem um pouco de semelhança com as linguagens escritas e faladas por nós. As linguagens possuem regras sintáticas e semânticas que devem ser respeitadas para a comunicação ser estabelecida e é assim, e de uma maneira muito forte, com as linguagens de programação. Quando estamos conversando se erramos uma palavra ou outra, até mesmo na concordância de verbos, é provável que a outra pessoa entenda, mas o computador não, as regras precisam ser respeitadas sem nenhum tipo de erro.

Assim como os idiomas, as linguagens de programação possuem os lexemas, que são as unidades básicas de palavras que compõem as regras sintáticas e semânticas de qualquer linguagem. Os lexemas são classificados em tokens os quais compõem o código fonte do programa. O código fonte na verdade é o programa propriamente dito.

Observe o seguinte exemplo. Trata-se de um programa muito simples escrito na linguagem Pascal o qual faz uma contagem de 1 a 3 (e nem mostra na tela, só faz a conta).

```
program p;
    var x: integer;
begin
    x:=1;
    while (x<3) do
           x:=x+1;
end.
```

O programa possui palavras em inglês, esta devidamente formatado de um modo tal que qualquer pessoa consegue ler o programa. Por isto que é dito estar em alto nível, pois embora tenha códigos estranhos na primeira vista, ele é

possível de ser lido. As outras linguagens modernas mostradas na figura 1.14 e figura 1.15 possuem uma aparência semelhante ao código fonte do exemplo.

Mas o computador não entende o programa desta forma. É necessário ser convertido para o único idioma no qual o computador entende: o código de máquina. E é ai que entra o compilador: é ele quem faz esta conversão.

O compilador também faz a seguinte análise: ele varre o programa desde o início para poder encontrar os tokens e os converter devidamente na linguagem de máquina da plataforma na qual ele está sendo compilado. Se o compilador estiver rodando em um PC com Windows, o programa será convertido para esta plataforma. Se estiver rodando em um Mac Book, com IOS, será compilado para esta plataforma. Veja a figura 1.16.

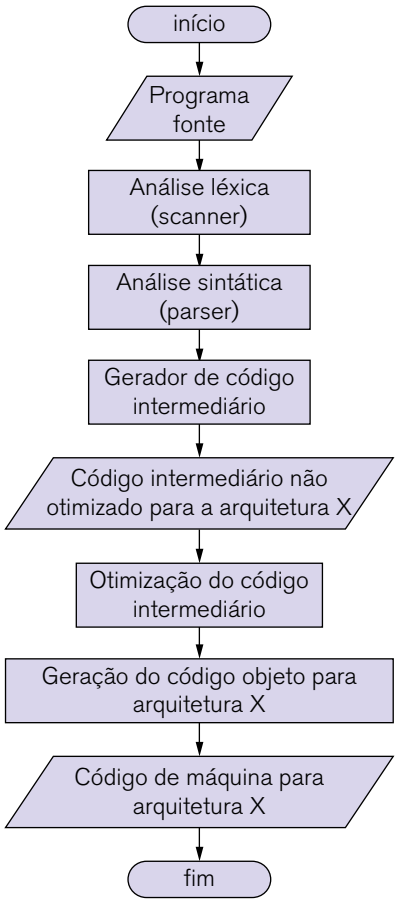


Figura 1.16 – Processo de compilação.

A figura 1.16 mostra como funciona um compilador de uma maneira geral. Basicamente o processo consiste em transformar o código fonte em um código de máquina para a arquitetura na qual desejamos executar o programa e para isso o programa fonte deve passar por algumas fases.

Percebemos que o programa fonte passa por uma análise léxica onde os tokens são identificados e separados. Nesta fase os comentários do código, espaços em branco e demais caracteres são descartados do próximo passo, a análise sintática.

A análise sintática é a parte principal do compilador. Nesta fase o compilador verifica se o que o programador escreveu "bate", ou seja, é compatível com a gramática da linguagem. Por exemplo, se o programador escreveu o código fonte na linguagem C, o analisador sintático verifica se as palavras, comandos, instruções e demais tokens correspondem às regras gramaticais e sintáticas da linguagem C.

A análise semântica verifica se o que foi gerado na análise sintática consegue gerar resultados executáveis e ser avaliados. Nas fases de análise, o compilador informa ao programador os possíveis erros de modo que possam ser corrigidos e a compilação ser reiniciada.

A partir da análise sintática e semântica é feita a geração do código intermediário e depois o código final.

Algumas linguagens são interpretadas. Nestas linguagens não temos um compilador e sim um interpretador. A diferença básica nestes dois métodos é que o interpretador lê o código fonte linha por linha e converte em código objeto (ou também chamado de bytecode) à medida que o código é lido. Outros interpretadores, dependendo da implementação, leem todo o código para depois executá-lo.

No nosso caso, vamos usar a linguagem C como ferramenta de estudo. Vamos estudar os algoritmos e depois convertê-los para C de forma que você entenda as duas partes: o projeto e a execução.

Vamos usar um programa chamado DevC++. É gratuito e roda em ambiente Windows e é facilmente encontrado na internet para *download*.

LEITURA

Algumas sugestões de leituras estão em inglês.

• Computer History Museum: excelente site. Vale muito a visita.

Site 1: http://www.computerhistory.org/

• Museu do computador: outro site muito bacana que mostra a história do computador e o futuro da tecnologia

Site 2: http://museudocomputador.org.br/

• A história da computação e o caminho da tecnologia: livro *online* de Cléuzio Fonseca Filho. Muito bom. Mostra a computação por outro enfoque, diferente do tradicional.

Site 3: http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf

• KNUTH, D. E. The Art of Computer Programming: Fundamental Algorithms. Vol 1, 2ª Edição. Addison-Wesley, 1973. É um clássico sobre algoritmos. Está em inglês porém é um livro excelente.

• Animações de algoritmos: site do livro Algorithm in C++ com vários algoritmos animados em Java.

Site 4: http://www.ansatt.hig.no/frodeh/algmet/animate.html

Dois excelentes livros para se aprofundar no assunto:

- WIRTH, N. Algoritmos e estruturas de dados. Rio de Janeiro: LTC, 1989.
- ZIVIANI, N. Projeto de algoritmos com implementações em Java e C++. São Paulo: Thomson Learning, 2007.

REFLEXÃO

No início da computação, a programação era uma arte como diziam. Porém com o passar do tempo foi mostrado que desenvolver sistemas e programar sem um prévio planejamento não era possível. Os algoritmos fazem parte de um projeto de sistema e também de vários outros tipos de projetos. Com a crescente valorização dos processos nas empresas e a necessidade de controlá-los, os elementos vistos neste capítulo podem ser aliados muito importantes nestes tipos de atividade.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos de programação de computadores**. São Paulo: Pearson Education, 2008.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. **Algoritmos**. São Paulo: McGraw Hill, 2009.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Campus, 2008.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. D. **Algoritmos**: lógica para desenvolvimento de programação. 9ª. ed. São Paulo: Érica, 1996.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Education, 2003.

2

Estrutura Sequencial

Qualquer que seja a linguagem de programação, desde a mais simples até a mais moderna terá um sequencia linear de operações.
Há alguns anos apareceu um conceito chamado de programação orientada a eventos onde linguagens como o Object Pascal (e seu famoso ambiente Delphi) e Visual Basic fizeram com que o mercado de desenvolvimento de aplicações mudasse para aplicações voltadas ao Windows e ambientes de janelas. Basicamente, com poucos movimentos de arrastar e soltar do *mouse* era possível ter uma aplicação bem completa funcionando.
Mesmo com toda essa funcionalidade e produtividade, as estruturas básicas de programação e principalmente a estrutura sequencial sempre esteve presente. Os primeiros programas de qualquer iniciante serão com este tipo de estrutura e vamos começar com ela.
Bons estudos!

OBJETIVOS

Após o término deste capítulo você estará apto a:

- Escrever um programa simples com entradas e saída em C++
- Identificar e usar os tipos de variáveis usados na linguagem C++
- Identificar e usar os operadores matemáticos e lógicos existentes na linguagem C++
- Criar entradas e saídas em C++

2.1 Analisando um programa com início e fim

Nosso primeiro programa tem um objetivo muito simples: calcular a média aritmética entre quatro notas de um aluno.

Isto é feito em C++ de acordo com o código da Listagem 1 a seguir. Para criar este programa e executá-lo vamos usar o software DevC++. Ele é gratuito, fácil de ser encontrado para *download* na internet e roda na plataforma Windows. Recomendamos que você digite este programa e o execute.

```
1   int main(int argc, char** argv) {
2           float nota1;
3           float nota2;
4           float nota3;
5           float nota4;
6           float media;
7
8           nota1 = 7;
9           nota2 = 6;
10          nota3 = 9;
11          nota4 = 5;
12
13          media = (nota1+nota2+nota3+nota4)/4;
14
15          return 0;
16  }
```

Listagem 1: Programa em C++

> É importante que você obedeça exatamente o que está escrito no programa. Os recuos nas linhas 4 até 17 podem ser feitos usando a tecla TAB. Outro detalhe: maiúsculas e minúsculas são interpretadas de maneira diferente pelo C++!

Após você ter digitado o programa, o que aconteceu? O que apareceu na tela? Apareceu alguma janela diferente?

Não tenha medo, leia o programa e tente entender o que aconteceu.

Complicado? Nem tanto. Lembre-se que aprender uma linguagem de programação é como aprender um idioma: precisamos conhecer a estrutura da linguagem, sua sintaxe, suas regras e até mesmo suas "manhas". A frase "Quero

aprender a programar computadores" em inglês é "I want to learn how to program computers", em espanhol é "Quiero aprender a programar computadoras", em francês é "Je veux apprendre à programmer les ordinateurs", em alemão: "Ich möchte lernen , wie man Computer zu programmieren" e em grego é "Θέλω να μάθω πώς να προγραμματίσετε υπολογιστές" ! (Ainda bem que temos tradutores *online*!). Ou seja, conhecendo as palavras da linguagem e a forma de usá-las de acordo com as regras, trata-se de uma questão de tradução.

Portanto, por mais que um algoritmo seja simples, a conversão dele para uma linguagem de programação vai depender da linguagem alvo: tem algumas que vamos escrever mais e outras menos, assim como traduzir uma frase em português para outros idiomas. Algumas vezes a tradução é bem fácil e parecida (veja a tradução da frase em português para espanhol) e outras vezes mais difícil e mais longa (veja a tradução para o alemão). E outras mais bonitas (veja a tradução para o francês!).

Vamos estudar o que o programa faz linha a linha para você poder entender o programa do início ao fim.

```
1   int main(int argc, char** argv) {
```

A linha 1 contém qual palavra que nos indica que ela inicia um programa? Se você pensou em main, acertou. Main significa "principal" e, portanto esta linha indica que estamos iniciando o programa principal. Mas e as outras palavras que aparecem na linha, o que significam? Por enquanto não é hora de tratarmos delas. Apenas vamos considerar que para começar um programa em C++ precisamos escrever o programa principal iniciando tal como está na linha 1.

Antes de continuarmos a explicação linha a linha do programa, observe o "visual" da listagem 1. Veja que temos uma parte com uma cor mais clara dentro de outra mais escura. A parte escura define todo o programa principal como já vimos. Esta delimitação do programa principal é feita entre os caracteres { e } (presentes nas linhas 1 e 16. Na digitação do programa, se você esquecer algum destes dois caracteres, o programa não compilará.

```
1   int main(int argc, char** argv) {
2           float nota1;
3           float nota2;
4           float nota3;
```

```
5           float nota4;
6           float media;
7
8           nota1 = 7;
9           nota2 = 6;
10          nota3 = 9;
11          nota4 = 5;
12
13          media = (nota1+nota2+nota3+nota4)/4;
14
15          return 0;
16  }
```

Portanto, tudo que estiver após o abre-chave "{" e antes do fecha-chave "}" compreenderá um bloco de programação e neste caso, será o bloco do programa principal.

Continuando, temos as linhas:

```
2   float nota1;
3   float nota2;
4   float nota3;
5   float nota4;
6   float media;
```

Observe que as linhas são bem parecidas. O que você acha que elas fazem? Bem, temos que existem 4 variáveis chamadas "nota" (de 1 a 4) e uma variável media as quais nos levam a pensar que elas estão relacionadas com os valores das notas e médias do aluno.

E além disso, elas possuem a palavra *float* acompanhando cada uma delas. Em uma situação real, o valor de uma nota bimestral de um aluno pode ser um valor inteiro como por exemplo nota 9, nota 5 ou valores decimais como 7,5 ou 4,25 e em algumas escolas temos até 6,3, 3,8 ou seja, valores diferentes de múltiplos de 5.

Não esqueça! Em C++, a separação do campo decimal é feita com o ponto "." e não com vírgula. Portanto, como exemplo, a constate PI que conhecemos da trigonometria terá a representação como 3.141592... (observe o ponto no lugar da vírgula!)

Portanto, como podemos trabalhar com valores de notas que contém decimais, vamos usar a palavra *float* para as variáveis. Além disso, mesmo que os valores das notas sejam inteiros, a variável média precisa ser de algum tipo decimal, pois a divisão que será feita para calcular a média aritmética pode gerar um número decimal. Ainda neste capítulo vamos estudar com mais detalhes os tipos existentes na linguagem C++.

Temos aqui uma importante lição sobre a linguagem C++: a criação de variáveis. Em C++ toda e qualquer variável que for usada em um programa precisa ser declarada. O compilador acusará um erro caso uma variável for usada no programa e não ter sido declarada anteriormente. E mais, além de ser declarada, ela precisa ser de um determinado tipo primitivo da linguagem, ou, de um tipo criado pelo usuário.

Continuando com o programa, o que você imagina que está acontecendo nas próximas linhas:

```
8   nota1 = 7;
9   nota2 = 6;
10  nota3 = 9;
11  nota4 = 5;
```

Não é difícil responder, certo? Estas quatro linhas colocam os valores 7, 6, 9 e 5 nas variáveis nota1, nota2, nota3 e nota4 respectivamente. A média desses valores será (7+6+9+5)/4 resultando em 6.75 (viu como pode ser decimal?). Caso você deseje alterar o valor de qualquer uma das notas é possível fazer diretamente em alguma dessas linhas.

O cálculo da média é feito na linha 13:

```
13  media = (nota1+nota2+nota3+nota4)/4;
```

O resultado é armazenado na variável média. Observe que a atribuição de valores sempre é feita com o sinal "=" e além de valores podemos colocar expressões matemáticas, como é o caso deste cálculo.

```
15  return 0;
```

A linha 15 pode parecer um pouco estranha. Ela será devidamente explicada no Capítulo 5 onde trataremos de funções. Por enquanto vamos usar este comando em todos os programas como sendo a última instrução do programa. Esta linha faz com que o valor 0 seja devolvido para quem executou o programa, no caso, o sistema operacional, por meio do compilador. Não se preocupe com isso agora, vamos explicar melhor esta questão mais adiante.

Explicamos anteriormente o papel da IDE. Como estamos trabalhando com o DevC++ e este é um ambiente integrado de desenvolvimento, ele possui recursos que além de compilar o programa para o usuário, executa o programa também com a intenção de testá-lo. Neste caso, a IDE tem um relacionamento com o sistema operacional (no caso, o Windows) que executa o programa.

E nunca se esqueça de fechar o programa principal, como é feito na linha 16:

```
16  }
```

Lembre-se: estamos fechando o bloco de programa que se refere ao programa principal e por isso usamos o }.

Neste momento você pode estar pensando sobre o programa:

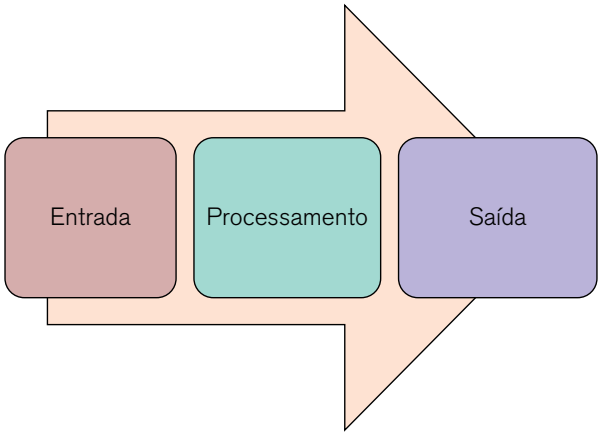
- "ele não tem nenhuma saída na tela"
    - Exatamente. Não tem ainda. Nosso objetivo aqui foi estudar um programa do início ao fim e o programa cumpre o que promete. Vamos estudar sobre entradas e saídas de dados mais adiante.
- "e se eu quiser mudar os valores das notas?"
    - Neste caso, a única forma de fazer isso é alterar os valores diretamente nas variáveis nas linhas 8 a 11 e compilar o programa novamente.
- "e se eu quiser repetir o programa?"
    - Neste caso temos que executar o programa manualmente pelo próprio ambiente DevC++.

Como você deve ter percebido, o programa possui algumas linhas em branco e tabulações. As tabulações são chamadas endentações e servem para melhorar a leitura do código por outra pessoa e ser mais fácil de verificar os blocos de programas. É altamente recomendado que você mantenha um padrão para as endentações e espaços no programa para que desta forma o seu programa, além de executar corretamente, seja fácil de ser editado e mantido.

2.2 Entendendo a organização de um programa

Basicamente a estrutura de um programa em C++ é:

- declaração de variáveis,
- depois o início do programa com as instruções e comandos e
- finalização, apresentando os resultados



Todo algoritmo ou programa tem um objetivo. Este objetivo é alcançado por meio de uma entrada de dados a qual será processada e resultará em uma saída a qual será avaliada.

O problema é que se a entrada for ruim, serão processados da maneira correta e a saída será ruim (e a culpa cairá sobre o sistema, ou o computador, como sempre!).

Portanto, fazer a correta entrada de dados é fundamental. O processo de entrada-processamento-saída ocorre o tempo todo. Todo fluxograma obedece a

esse processo, o processo de compilação também, assim como o processo de execução.

A entrada e a saída podem ocorrer de diversas formas em um computador. Podemos ter uma leitura de um cartão de banco em um caixa eletrônico, a digitação da senha, a informação que a senha digitada foi errada como saída, e várias outras formas.

Vamos estudar a organização de um programa começando pela declaração de variáveis, porém antes disso, vamos entrar em ...

> As reticências são propositais, é para dar a ideia de emendar com o tópico abaixo

2.2.1 Alguns tópicos importantes

No capítulo anterior começamos o estudo contando sobre o Deep Blue. Devido a ele ser um computador exclusivamente para cálculos obviamente sabemos que ele resolvia milhares de equações rapidamente.

Outro exemplo: quando fazemos um cadastro na internet normalmente inserimos nossos dados pessoais e digitamos vários tipos de caracteres: só letras para o nome, só números para o CEP e assim por diante.

Quando assistimos um filme em forma de desenho animado ou qualquer outro que envolva efeitos especiais sabemos que existem computadores que ajudam os criadores a desenharem e animarem esses efeitos.

Ou seja, para cada tipo de situação temos uma informação diferente. O computador, portanto, é um equipamento, uma ferramenta que resolve problemas que lidam com informações e essas informações são divididas de uma maneira muito geral em dois tipos: dados e instruções.

2.2.1.1 Tipos de dados

Os dados são representados pelas informações a serem tratadas por um computador. Normalmente usamos três tipos dados: os dados numéricos, que podem ser números inteiros e reais, os dados alfanuméricos e os dados lógicos. Estes tipos são chamados de tipos primitivos:

- **Inteiros**: quando lidamos com números que não possuem casa decimal, incluindo os negativos. Exemplos: -65, 0, 44, 2030, etc.

• **Reais**: números que possuem casas decimais, incluindo os negativos. Exemplos: -33.5, -128.3, 255.6, 32767.4, etc.

• **Caracteres**: são os dados contendo sequências de letras, números e símbolos especiais. Este tipo de dado sempre deve estar entre aspas como, por exemplo: "Brasil", "Fone: 2001-2002", "Rua dos Bobos nº 0", "R2-D2", "4" (o caractere "4" é diferente do inteiro 4! Não é possível somar "4" + "4", veremos isso mais tarde). Este tipo de dado também pode ser conhecido como alfanumérico, string (quando houver mais de 2 caracteres), literal ou cadeia.

• **Lógicos**: estes tipos de dados só possuem dois valores: verdadeiro (ou true) e falso (ou false). Nunca poderão assumir os dois valores simultaneamente. Este tipo também é chamado de booleano devido ao matemático inglês George Boole ter feito vários estudos sobre lógica matemática.

Todas as linguagens de programação possuem um conjunto de tipos primitivos da linguagem parecidos com os que foram citados. Em C++ temos os seguintes tipos de dados:

• **char**: de apenas 1 byte, capaz de armazenar apenas 1 caractere

• **int**: capaz de armazenar um inteiro

• **float**: armazena números de ponto flutuante de precisão simples

• **double**: armazena números de ponto flutuante de precisão dupla

• **void**: em inglês, void significa vazio. É um tipo que informa ao compilador que a variável não terá um tipo definido

Em C++, além dos tipos temos os modificadores de tipo. Existem quatro modificadores de tipo na linguagem C: signed, unsigned, long e short. A Tabela 5 mostra os tipos e modificadores de tipo existentes na linguagem C++. Além disso, a tabela mostra o intervalo de alcance de cada um dos tipos bem como o número de bits que o tipo ocupa na memória.

Veja que o modificador *unsigned* retira o sinal do tipo e dobra a capacidade do tipo no intervalo positivo. Como exemplo, vamos usar o tipo int: originalmente ele compreende a faixa existente entre os números -32768 e 32767 e portanto possui 65536 números dentro do intervalo. Tirando a parte negativa com o modificador *unsigned*, a capacidade do int sobe para 65535 porém começando de 0 (sem a parte negativa). Este mesmo raciocínio serve para os outros tipos que possuem o modificador *unsigned*.

TIPO	NUM DE BITS	INTERVALO	
		INÍCIO	FIM
CHAR	8	-128	127
UNSIGNED CHAR	8	0	255
SIGNED CHAR	8	-128	127
INT	16	-32.768	32.767
UNSIGNED INT	16	0	65.535
SIGNED INT	16	-32.768	32.767
SHORT INT	16	-32.768	32.767
UNSIGNED SHORT INT	16	0	65.535
SIGNED SHORT INT	16	-32.768	32.767
LONG	32	-2.147.483.648	2.147.483.647
UNSIGNED LONG INT	32	-2.147.483.648	2.147.483.647
SIGNED LONG INT	32	0	4.294.967.295
FLOAT	32	3,4E-38	3,4E+38
DOUBLE	64	1,7E-308	1,7E+308
LONG DOUBLE	80	3.4E-4.932	3.4E+4.932

Tabela 2.1 – Tipos da linguagem C++.

2.2.1.2 Declaração e inicialização de variáveis

Em C++ é obrigatório que toda variável usada no programa seja declarada. Em C++ a sintaxe da declaração de variáveis é:

```
tipo_da_variável    lista_de_variáveis
```

Ou seja, toda vez que uma variável for declarada em C++, é preciso que o seu tipo venha em primeiro lugar e depois a variável desejada, ou uma lista de variáveis, separada por vírgula.

Portanto, as declarações abaixo são válidas:

```
int a;
char a, b, c;
unsigned int x1, x2, media;
```

Alguns detalhes sobre a linguagem C++:

C++ é case sensitive: ou seja, uma letra maiúscula é diferente de uma letra minúscula. Portanto a variável "a" é diferente da variável "A". Assim:

```
int a, A;
float soma, SOMA, Soma, sOMa, somA;
```

a e A são variáveis diferentes, assim como as variáveis *float* declaradas no exemplo

• Uma vez que C++ é case sensitive, as palavras chave da linguagem sempre serão escritas em letras minúsculas

• Os comentários na linguagem C devem começar com /* e terminar com */ e servem tanto para uma linha quanto para um bloco de linhas. No C original as barras "//" que normalmente são usadas para comentar apenas uma linha, não existem, porém alguns compiladores aceitam esta forma.

• A linguagem C++ possui as seguintes palavras reservadas. Logo, não é possível criar variáveis com estas palavras:

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

• Os nomes das variáveis na linguagem C++ podem começar com uma letra ou um sublinhado "_". Os outros caracteres podem ser letras, números ou sublinhado. Não é permitido usar caracteres especiais ou outros em nomes de variáveis.

• Em relação a nomes de variáveis, é bom sempre usar letras minúsculas para obedecer a um padrão internacionalmente usado para variáveis

Quando criarmos constantes, estas devem ser escritas com todas as letras em maiúsculas

2.3 Adicionando entradas e saídas

É muito importante criar formas de orientar o usuário com relação ao que o programa necessita para poder funcionar adequadamente. Ou seja, é importante mostrar mensagens na tela e receber informações do usuário de uma maneira eficiente.

Por exemplo, no programa da Listagem 1 seria mais legal exibir mensagens como:

"Digite a primeira nota"

"Digite a segunda nota"

"A média para este aluno é ____"

Isso é possível em C++ e qualquer outra linguagem de programação.

Em C++ o comando que exibe uma mensagem na tela, ou seja faz a saída de dados, é o *cout*.

Este comando, por incrível que pareça, não faz parte das palavras-chave da linguagem e sendo assim o compilador não o reconhece. Ele fica definido em um outro arquivo, chamado *iostream*, e para ser usado, precisamos incluir (include) a biblioteca e definir um espaço de nomes (namespace) para isso. Não se preocupe com o *namespace*, apenas use-o nos seus programas para poder usar as bibliotecas incluídas.

Sendo assim, as duas primeiras linhas do programa ficarão assim:

```
1   #include <iostream>
2   using namespace std;
```

Assim como temos a saída de dados, temos a entrada de dados que é feita com o comando cin. Para você lembrar com mais facilidade, in significa entrada e out, saída.

Vamos retomar o programa da Listagem 1 e torná-lo mais interativo por meio de mensagens de entrada e saída de dados. Veja agora como ficou a parte de atribuição de valores às variáveis:

```
11  cout<<"Digite a primeira nota: "<<endl;
12  cin>>nota1;
13  cout<<"Digite a segunda nota: "<<endl;
14  cin>>nota2;
```

```
15  cout<<"Digite a terceira nota: "<<endl;
16  cin>>nota3;
17  cout<<"Digite a quarta nota: "<<endl;
18  cin>>nota4;
```

Observe as linhas 11, 13, 15 e 17 e o uso do *cout*. Veja que após o comando temos o operador "<<". Tudo que vier após o "<<" será impresso na tela. No caso das linhas citadas, será impresso a *string* entre aspas.

Para finalizar, as linhas terminam com "<<endl;". Isso diz ao compilador para pular uma linha após cada impressão na tela. Se não houvesse esse comando, todo o texto ficaria em uma linha só deixando o texto difícil de ser lido.

Portanto a linha abaixo:

```
cout<<"Digite a primeira nota: "<<endl;
```

Significa:

- "Compilador: escreva "Digite a primeira nota:" na tela e depois pule uma linha"

As linhas 12, 14, 16 e 18 fazem a entrada de dados, ou seja, fazem a leitura do que o usuário digitar pelo teclado. O objeto de entrada de dados (cin) lê um stream de dados é usado da seguinte forma:

cin>>_____;

Portanto a linha abaixo:

```
cin>>nota1;
```

Significa:

- "Compilador: pegue o que o usuário digitar no teclado e coloque o valor na variável nota1"

Veja que também é possível combinar algumas coisas:

```
20  media = (nota1+nota2+nota3+nota4)/4;
21  cout<<"A media do aluno eh: "<<media<<endl;
```

Calculamos a média na linha 20 e combinamos o resultado com uma *string* na linha 21. Veja que é possível combinar valores de variáveis e *strings* usando o operador "<<".

O programa da Listagem 1 reescrito com as entradas e saídas ficará assim:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5           float nota1;
6           float nota2;
7           float nota3;
8           float nota4;
9           float media;
10
11          cout<<"Digite a primeira nota: "<<endl;
12          cin>>nota1;
13          cout<<"Digite a segunda nota: "<<endl;
14          cin>>nota2;
15          cout<<"Digite a terceira nota: "<<endl;
16          cin>>nota3;
17          cout<<"Digite a quarta nota: "<<endl;
18          cin>>nota4;
19
20          media = (nota1+nota2+nota3+nota4)/4;
21          cout<<"A media do aluno eh: "<<media<<endl;
22          return 0;
23  }
```

E sua execução resultará na seguinte tela:

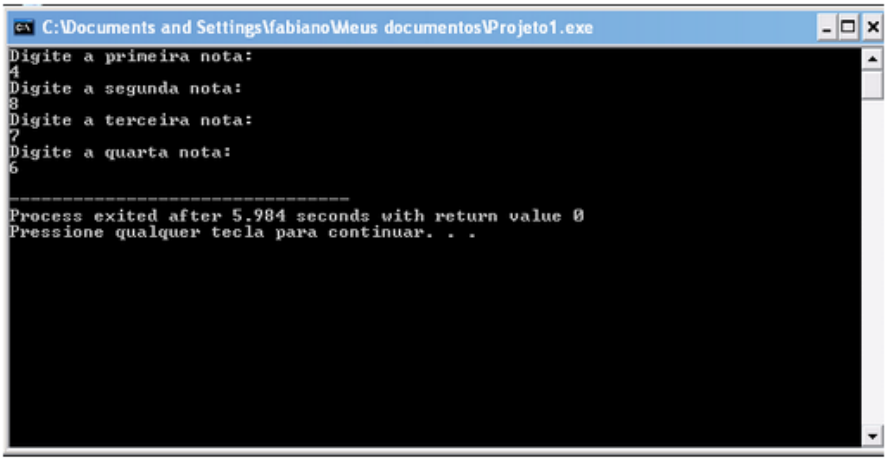


Figura 2.1 – Execução do programa

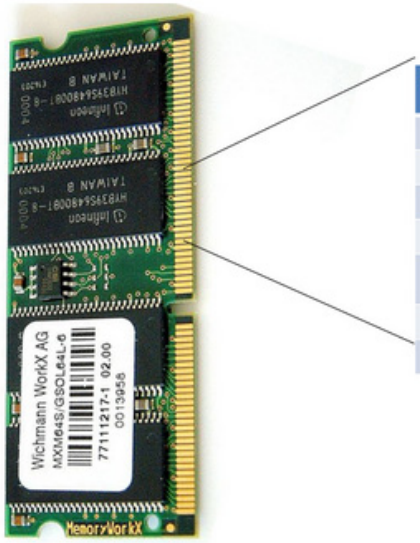
2.4 Entendendo como os dados são armazenados

As variáveis são parte fundamental de qualquer programa. Tudo aquilo que é sujeito a alguma variação durante a execução de um algoritmo, é uma variável. Usando novamente o exemplo do Deep Blue, a cada nova jogada, milhões de possibilidades eram analisadas para prever as próximas jogadas, sendo assim, o computador tinha que processar várias variáveis simultaneamente.

As variáveis são armazenadas na memória RAM do computador e devem ter seu tipo identificado antes do armazenamento. Uma vez armazenado, ele pode ser usado e manipulado a qualquer momento. Veja a figura 2.2.

Esta figura mostra resumidamente como seria a situação da RAM em um determinado momento, após a criação de algumas variáveis e atribuição de valores a elas.

A RAM da figura é modesta, tem apenas 64Mbytes de capacidade de armazenamento. Toda posição na memória possui um endereço em formato hexadecimal e na figura nota-se que eles variam de 0000 a FFFF (é apenas um exemplo, em uma memória real, existem outros valores e formas de endereçamento).



Endereço de memória	Nome da variável	Conteúdo
0001	pi	3.1415
0002	nome	"Zé"
0003	valido	True
0004	R2d2	"robô"
0005	endereço	"Rua dos Bobos nº 0"
...	...	...
FFFF		

Figura 2.2 – Representação da memória RAM

Neste caso criamos algumas variáveis:

- **pi** : que é do tipo real, possui o valor 3.1415 e está na posição 0001
- **nome**: que é do tipo caractere, possui o valor "Zé" e está na posição 0002
- **valido**: que é do tipo booleano, possui o valor true e está na posição 0003
- **R2d2**: que é do tipo caractere, possui o valor "robô" e está na posição 0004
- **Endereço**: que também é do tipo caractere, possui o valor "Rua dos Bobos nº 0" e está na posição 0005
- As outras posições da memória podem ou não estar vazias

Todos os valores que estão armazenados na memória podem sofrer variações ao longo da execução do programa que usa estas variáveis. Por exemplo, a variável valido pode ter um valor false ao longo da execução. A variável endereço pode assumir outro valor como "Rua Zero, nº 123" e assim por diante. Inclusive a variável pi. Embora saibamos que ela é uma constante, ela pode ser alterada durante o programa. Quem vai definir que ela é uma constante e não pode ser alterada é o programador, durante a elaboração do programa e o compilador por sua vez, não vai permitir que outro valor seja atribuído a essa variável.

O nome da variável é usado para identificá-la durante todo o programa. Esta identificação possui algumas regras que devem ser observadas na sua criação:

- Os nomes das variáveis podem ter um ou mais caracteres. Por exemplo: a, a1, aa1, aaa1 são nomes válidos

• O primeiro caractere do nome da variável nunca poderá ser um número ou um caractere especial.

• Não é permitido criar variáveis com espaços em branco. Porém podemos usar outros caracteres para "simular" o espaço. Por exemplo: codigo_aluno, endereco_cliente, nomeAluno, inicioPeriodoFerias, etc..

• Não é possível criar variáveis com o mesmo nome de palavras reservadas da linguagem. No caso de português estruturado, não poderemos criar uma variável cujo nome é inicio, fim, enquanto, etc..

Uma variável pode ter seu valor mantido fixo durante todo o programa (isso perde um pouco o sentido de ser variável) e neste caso, temos uma constante. Por exemplo, podemos criar uma variável chamada pi do tipo real e atribuir a ela o valor 3.141592. Por questões de padronização, toda vez que usarmos constantes é comum escrevê-las com todas as letras em maiúsculo. Neste caso, teremos PI = 3.131592.

2.5 Utilizando o computador para fazer conta

Como já vimos, o computador é uma grande calculadora. E para isso precisamos dos operadores aritméticos para poder realizar os cálculos matemáticos necessários. Os operadores matemáticos podem ser de dois tipos: unário e binário. O operador unário é usado quando temos apenas um operando na expressão, por exemplo, uma inversão de sinal. O binário possui dois operandos como por exemplo na adição, subtração etc. Veja a tabela 2.2.

OPERADOR	OPERAÇÃO	TIPO	PRIORIDADE	EXEMPLO
+	Manutenção de sinal	Unário	1	+(+1) = +1
–	Inversão de sinal	Unário	1	–(+1) = +1
^	Exponenciação	Binário	2	2 ^ 3 = 8
/	Divisão	Binário	3	4/2 = 2
*	Multiplicação	Binário	3	4*2 = 8
+	Adição	Binário	4	1 + 1 = 2
–	Subtração	Binário	4	2 – 1 = 1

Tabela 2.2 – Operadores aritméticos

A tabela 2.2 mostra os operadores aritméticos que são usados nos algoritmos, o tipo dos operadores e sua prioridade em uma avaliação da expressão matemática. Além disso, nota-se que as expressões matemáticas computacionais serão escritas de uma maneira diferente da forma como escrevemos em papel.

Considere os exemplos da tabela 2.3:

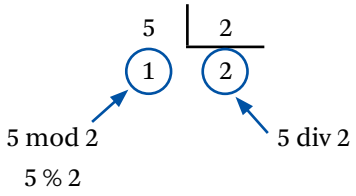
FÓRMULA	FORMA ESCRITA	FORMA COMPUTACIONAL
Área do triângulo	$S = \frac{base * altura}{2}$	S < – (base * altura) / 2
Área do círculo	$S = \pi * raio^2$	S < – PI * raio ^ 2 ou S < – PI * raio * raio *Considerando PI uma constante*
Expressão matemática	$x = \left\{43 * \left[\frac{55}{30+2}\right]\right\}$	X < – (43 * (55 / (30 + 2)))

Tabela 2.3 – Exemplos de formas escritas e computacionais

Note que na escrita computacional não é possível escrever em 2 linhas, temos que escrever qualquer expressão matemática em 1 linha apenas. Outro detalhe, o sinal de atribuição, o qual normalmente escrevemos "=" é substituído por uma "seta", composta pelos caracteres "<" e "-" juntos.

Também é comum usar em algoritmos e programas alguns operadores não convencionais como:

- Mod: resto da divisão, denotado pelo operador "%".
- Div: quociente da divisão inteira



2.5.1.1 Operadores aritméticos e de atribuição

Assim como nos algoritmos temos operadores aritméticos na linguagem C++ para desenvolver operações matemáticas. A seguir na tabela 1.4 apresentamos a lista dos operadores aritméticos do C++.

OPERADOR	AÇÃO
+	Soma (inteira e ponto flutuante)
–	Subtração ou troca de sinal (inteira ou ponto flutuante)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
– –	Decremento (inteiro e ponto flutuante)

Tabela 1.4 – Operadores aritméticos na linguagem C++.

A linguagem C++ também possui operadores unários e binários. Os unários atuam sobre apenas uma variável, modificando ou não o seu valor e retornam o seu valor final. Os operadores "++" e "--" são unários e servem para incrementar e decrementar a variável que está sendo usada. Assim x++ é equivalente a x=x+1. Idem para x – –.

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

Por exemplo:

```
int a=5, b;
b=a++;
```

Qual será o valor de a e b após a execução?
Resposta: a = 6, b = 5

E agora, qual o valor de a e b?

```
int a=5, b;
b=++a;
```

Resposta: a = 6, b = 6

Quando o operador é pós-fixado, o valor da variável é utilizado no comando e após a conclusão desse comando o valor será atualizado. Quando o operador é pré-fixado, primeiro a variável é atualizada para então utilizar o seu valor.

2.5.1.2 Funções matemáticas

Além das operações aritméticas, é comum usarmos nos programas outras funções já existentes na linguagem C++ como por exemplo:

- sen(x): seno do ângulo x
- cos(x): cosseno do ângulo x
- tg(x): tangente de x
- demais funções trigonométricas
- abs(x): valor absoluto de x
- int(x): parte inteira de um número fracionário
- frac(x): parte fracionária de x
- e outras

Lembrando que x nas funções acima pode ser uma variável, um número, uma expressão aritmética ou outra função matemática.

2.5.1.3 Expressões lógicas e operadores relacionais

As expressões lógicas são aquelas que envolvem operadores lógicos e/ou relacionais e nas quais os operandos são relações e/ou variáveis e/ou constantes booleanas.

Os operadores relacionais são usados para fazer comparações entre dois operandos do mesmo tipo primitivo. Esses valores são representados por constantes, variáveis ou expressões aritméticas.

OPERADOR	DESCRIÇÃO
==	Igual
>	Maior que
<	Menor que
!=	Diferente
>=	Maior ou igual que
<=	Maior ou igual que

Tabela 1.5 – Operadores relacionais.

A tabela 1.5 mostra os operadores relacionais. Nota-se que no caso do igual e diferente os operadores tem diferenças quando escritos em algoritmos (português estruturado) e na linguagem C. O resultado obtido de uma relação é sempre um valor lógico.

Alguns exemplos:
2 vezes 4 é igual a 24 dividido por 3 ?
Usando o português estruturado:
2 * 4 = 24 /3
8 = 8
Verdadeiro !

O resto da divisão de 15 por 4 é menor que o resto da divisão de 19 por 6?
15 % 4 < 19 % 6
3 < 1
Falso!

3 * 5 div 4 <= 3^2/0.5
15 div 4 <= 9/0.5
3 <= 18
Verdadeiro!

2 + 8%7 >= 3*6-15
2+1 >= 18-15
3 >= 3
Verdadeiro!

2.5.1.4 Operadores lógicos

Assim como existem operadores aritméticos que lidam com expressões matemáticas, existem também os operadores lógicos, que lidam com expressões booleanas e com a lógica proposicional.

A álgebra de Boole ou álgebra booleana, é uma variação da álgebra convencional. Basicamente ela tem três pontos diferentes:

- Nos valores que as variáveis podem assumir, sendo binárias ou seja, só podem assumir dois valores: 0 ou 1 (falso ou verdadeiro)
- Nas operações que se aplicam a esses valores
- Nas propriedades dessas operações, ou seja, às leis que elas obedecem.

Na álgebra de Boole temos dois princípios fundamentais:

- Princípio da não contradição: uma proposição não pode ser simultaneamente verdadeira e falsa
- Princípio do tereiro excluído: Uma proposição só pode tomar um dos valores possíveis: ou é verdadeira ou é falsa, não sendo possível uma terceira hipótese.

Basicamente temos os seguintes operadores: e, ou e não, e algumas variações.

Para poder resumir, basicamente a álgebra e Boole funciona da seguinte maneira:

Temos uma proposição como: "Amanhã vai chover?". Vamos atribuir essa proposição a uma variável, chamada A. Sendo assim, teremos:

A = "Amanhã vai chover?"

Note que é uma pergunta que pode ter várias respostas: "Sim", "não", "talvez", "depende", etc.. Logo, A não é uma variável que faz parte da álgebra de Boole porque não é uma variável que pode assumir apenas um dos dois valores "sim" ou "não".

Porém,

A = "Brasília é a capital do Brasil"

B = "Buenos Aires é um país da Europa"

Percebeu a diferença? Aqui temos somente duas respostas possíveis: "sim" ou "não". Logo, nesse caso, A é uma variável boolena, assim como B. Aqui podemos associar a A o valor lógico verdadeiro (true) e a B o valor lógico falso (false) e são duas proposições.

Logo, "Brasília é a capital do Brasil e Buenos Aires não é um país da Europa" também é uma proposição e podemos associar o valor verdadeiro a ela.

Na álgebra de Boole usamos um recurso chamado tabela verdade para mostrar os valores possíveis que as variáveis lógicas podem assumir. A tabela verdade mostra todas as possibilidades combinatórias entre os valores de diversas variáveis lógicas que são encontradas em somente duas situações e um conjunto de operadores lógicos. Nas tabelas verdade a seguir, A e B são proposições.

Em C++ os operadores lógicos E e OU são representados pelos símbolos "&&" e "||" respectivamente. Veja suas tabelas verdades:

A	B	A && B
F	F	F
F	V	F
V	F	F
V	V	V

A	B	A || B
F	F	F
F	V	V
V	F	V
V	V	V

Exemplos:
Proposição A = "Se fizer sol"
Proposição B = "o dia está quente"
"Se fizer sol E o dia está quente, eu irei à praia"
Quando eu irei à praia?

Pela tabela verdade do operador E, a proposição verificada só será verdadeira (ou seja, o sujeito só irá à praia) quando as proposições A e B forem verdadeiras. Qualquer outra situação, a proposição será falsa (observe a tabela verdade do E).

Proposição A = "Se fizer sol"
Proposição B = "o dia está quente"
"Se fizer sol OU o dia está quente, eu irei à praia"
Quando eu irei à praia?

Pela tabela verdade do operador OU, as chances de o sujeito ir à praia irão aumentar bastante pois pela tabela verdade do OU a proposição será verdadeira em três situações: somente sol, somente dia quente e dia com sol e quente. O sujeito não irá à praia se não estiver fazendo sol nem o dia estiver quente (observe a tabela verdade do OU).

Vamos terminar o capítulo com outro programa um pouco mais complicado: calcular raízes de uma equação do 2º grau.

Com um projeto criado e um arquivo já definido no DevC++, vamos começar a escrever.

Antes de partirmos para o programa, por mais que seja simples calcular as raízes de uma equação, precisamos entender o que consiste uma equação deste tipo. O pleno entendimento do problema nos dá uma segurança para desenvolver um bom algoritmo e consequentemente um bom programa.

Uma equação do segundo grau é uma expressão matemática que consiste de incógnitas, coeficientes, expoentes e uma igualdade que pode ser reduzida ao seguinte formato:

$ax^2 + bx + c = 0$

Onde a, b e c são os coeficientes, x é a incógnita e 2 é o expoente. Além disso, a tem que ser diferente de zero.

A forma de resolver uma equação do segundo grau consiste em descobrir os valores reais para a incógnita e torne a sentença verdadeira. A fórmula para esta resolução é chamada de Fórmula de Bhaskara, apresentada a seguir:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

O valor $b^2 - 4ac$ é chamado de discriminante e representado pela letra grega Δ (delta). Logo, delta = $b^2 - 4ac$. Para simplificar o algoritmo, vamos supor que estamos lidando com equações do segundo grau completa e com raízes reais.

Então sendo assim, do que consiste em o programa?

Basicamente em ler os valores a, b e c do usuário e fazer o cálculo das raízes, correto? Vamos chamar as raízes de x1 e x2.

Vamos lá?

Começamos nosso programa incluindo as bibliotecas que serão usadas. Isto é feito nas linhas 1 a 4.

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <cmath>
4   using namespace std;
```

Quando começamos a escrever o programa, é bom pensar nas possíveis variáveis que iremos usar. No caso da resolução da equação do segundo grau, quais as variáveis serão utilizadas? Consegue pensar nelas? Quais os possíveis valores que serão usados no programa? Para isso é importante conhecer bem o problema que estamos lidando. Lembrando da fórmula de Bhaskara, temos que serão usados alguns valores que podem ser variáveis como a, b, c, delta, x1 e x2. Vamos por hora declarar estas variáveis:

```
7   float x1,x2;
8   float a,b,c;
9   float delta;
```

Observe que as variáveis podem ser declaradas todas em uma mesma linha ou separadamente. As duas formas estão corretas e você deve usar a que deixar o código mais legível. Como são variáveis que vão usar valores de ponto flutuante, todas serão do tipo *float*. Nós também poderíamos propor que os coeficientes a,b e c fossem inteiros porém vamos deixar como *float* a fim de simplificação.

Quando declaramos uma variável que vai ser informada pelo usuário, vamos por enquanto admitir que o usuário será "legal" e digitar um valor do tipo que o algoritmo espera. Por exemplo, se a variável de leitura for inteira, o usuário tem a total liberdade de digitar a letra "a" por exemplo. "a" não é um valor inteiro e isso obviamente vai gerar um erro no algoritmo e no programa. No nosso caso por enquanto, o usuário sempre digitará o valor correto. Esta verificação do valor lido é chamada de validação dos dados.

Se houver outras variáveis que não foram previstas, estas podem ser declaradas posteriormente na seção específica.

Definidas as variáveis iniciais vamos à próxima etapa: colocar a sequência de comandos em ordem para poder fazer o cálculo das raízes. Não esqueça: estamos considerando neste programa que as raízes são reais e inteiras (para simplificar).

Em primeiro lugar, precisamos conhecer os coeficientes a, b e c.

```
11  cout<<"Digite o coeficiente a"<<endl;
12  cin>>a;
13  cout<<"Digite o coeficiente b"<<endl;
```

```
14  cin>>b;
15  cout<<"Digite o coeficiente c"<<endl;
16  cin>>c;
```

Conhecendo os coeficientes, podemos proceder com os cálculos do delta (veja a linha 18:

```
18  delta = b*b-4*a*c;
19  //calculo das raízes
```

A linha 19 contém uma informação especial. As barras inclinadas "//" colocadas no início da linha indicam que esta linha é um comentário.

> Um comentário é um texto que explica uma determinada parte do programa ou algoritmo. Ele é muito usado e é uma boa prática de programação. Colocar comentários nos seus códigos faz com que outras pessoas, ao lerem o que você codificou entendam melhor o seu raciocínio ao elaborar o programa ou o algoritmo. As barras duplas "//" caracterizam comentários de uma linha: tudo que for escrito após estas barras até o fim da linha será ignorado pelo algoritmo ou pelo compilador.

Com o delta calculado, vamos para o cálculo das raízes da equação. Lembrando que temos duas raízes:

$$x1 = \frac{-b + \sqrt{\Delta}}{2a} \qquad x2 = \frac{-b - \sqrt{\Delta}}{2a}$$

Porém não conseguimos escrever desta forma e temos que usar a forma computacional:

$$x1 = (-b + \sqrt{delta}) / (2 * a)$$
$$x2 = (-b - \sqrt{delta}) / (2 * a)$$

A forma computacional ainda apresenta um problema: a raiz quadrada. Não temos como inserir a raiz quadrada na forma de um caractere como é feito com letras e números. A raiz quadrada é outro recurso computacional chamado função o qual pode ser usado em situações nas quais temos que abstrair outras

tarefas. Por exemplo, neste algoritmo não é necessário saber como uma raiz quadrada é calculada, só precisamos saber para que ela é necessária neste caso: retornar o valor calculado. Já vimos um pouco sobre isto no tópico 2.1.4.

Para fazer o cálculo da raiz quadrada precisamos usar uma função que não faz parte da biblioteca padrão do C++. Para isso tivemos que incluir a biblioteca cmath definida no cabeçalho do programa. A função a ser usada é a sqrt(x), onde x é o argumento (valor desejado) da função.

Chamamos as duas raízes da equação de x1 e x2. Vamos proceder ao cálculo. Na listagem a seguir é mostrado o cálculo e a forma computacional das expressões.

```
20  x1=(-b+sqrt(delta))/2*a;
21  x1=(-b-sqrt(delta))/2*a;
```

E por fim, é bom mostrarmos para o usuário o resultado das variáveis que representam as raízes calculadas da equação ([Campo]linhas 23 e 24):

```
23  cout<<"x1 = "<<x1<<endl;
24  cout<<"x2 = "<<x2<<endl;
```

A execução do programa está mostrada na Figura 22. Neste caso a equação, considerando os coeficientes a=1, b=-7 e c=6, é a seguinte:

$x^2 - 7x + 6 = 0$

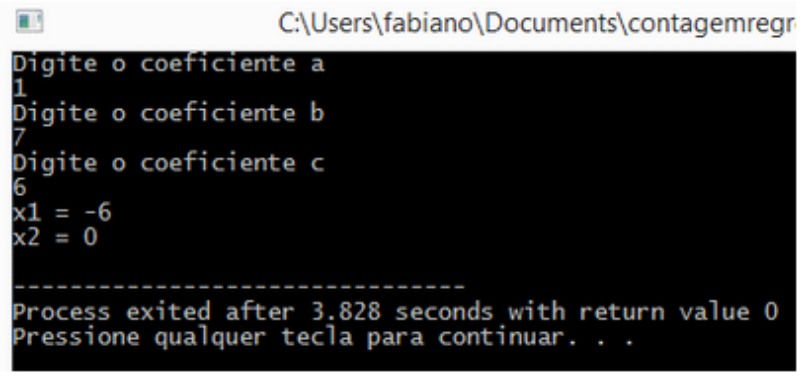


Figura 2.3 – Execução do algoritmo da equação do segundo grau

O programa completo está mostrado na Listagem 1:

```
1    #include <iostream>
2    #include <cstdlib>
3    #include <cmath>
4    using namespace std;
5
6    int main(void) {
7            float x1,x2;
8            float a,b,c;
9            float delta;
10
11           cout<<"Digite o coeficiente a"<<endl;
12           cin>>a;
13           cout<<"Digite o coeficiente b"<<endl;
14           cin>>b;
15           cout<<"Digite o coeficiente c"<<endl;
16           cin>>c;
17
18           delta = b*b-4*a*c;
19           //calculo das raízes
20           x1=(-b+sqrt(delta))/2*a;
21           x1=(-b-sqrt(delta))/2*a;
22
23           cout<<"x1 = "<<x1<<endl;
24           cout<<"x2 = "<<x2<<endl;
25           return 0;
26   }
```

Listagem 2

LEITURA

Algumas sugestões de leituras estão em inglês.

- Computer History Museum: excelente site. Vale muito a visita.

Site 1 – http://www.computerhistory.org/

- Museu do computador: outro site muito bacana que mostra a história do computador e o futuro da tecnologia

Site 2 – http://museudocomputador.org.br/

- A história da computação e o caminho da tecnologia: livro *online* de Cléuzio Fonseca Filho. Muito bom. Mostra a computação por outro enfoque, diferente do tradicional.

Site 3 – http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf

- KNUTH, D. E. The Art of Computer Programming: Fundamental Algorithms. Vol 1, 2ª Edição. Addison-Wesley, 1973. É um clássico sobre algoritmos. Está em inglês porém é um livro excelente.
- Animações de algoritmos: site do livro Algorithm in C++ com vários algoritmos animados.

Site 4 – http://www.ansatt.hig.no/frodeh/algmet/animate.html

Dois excelentes livros para se aprofundar no assunto:

- WIRTH, N. Algoritmos e estruturas de dados. Rio de Janeiro: LTC, 1989.
- ZIVIANI, N. Projeto de algoritmos com implementações em Java e C++. São Paulo: Thomson Learning, 2007.

REFLEXÃO

Iniciamos o aprendizado da linguagem C. Pode parecer difícil, mas não é! É como aprender um novo idioma, acredite. É questão de "caligrafia" ou seja, assim como uma criança aprende a escrever usando cadernos de caligrafia, programação a gente aprende programando. A linguagem C é legal porque muitas coisas do mercado usam esta linguagem como base para os seus produtos como o Arduino, microcontroladores e até mesmo para criar pequenos robôs.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos de programação de computadores**. São Paulo: Pearson Education, 2008.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. **Algoritmos**. São Paulo: McGraw Hill, 2009.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Campus, 2008.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. D. **Algoritmos**: lógica para desenvolvimento de programação. 9ª. ed. São Paulo: Érica, 1996.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Education, 2003.

3

Estruturas de Decisão

“Nada é mais difícil e, portanto, tão precioso, do que ser capaz de decidir.”

Napoleão Bonaparte

“É nos momentos de decisão que o seu destino é traçado.”

Anthony Robbins

Como percebemos, tomar decisão é muito importante e fazemos isso toda hora, todo dia, não é? Se estiver chovendo, trocamos o caminho, se estiver nublado avaliamos a possibilidade de chover ou não para poder pegar um guarda-chuva. E por aí vai.

Um algoritmo também tem momentos nos quais tem que parar e avaliar as condições para tomar uma determinada direção. Imagine o robô Curiosity lá em Marte fazendo suas explorações: ao encontrar um determinado tipo de material ele precisa avaliar e tomar alguma decisão. Enfim, a estrutura de decisão é outra fundamental e presente em qualquer linguagem de programação.
E vamos estudá-la neste capítulo. Bom trabalho!

OBJETIVOS

No final deste capítulo você estará apto a:

- Identificar quando é necessário usar uma estrutura de repetição
- Aplicar os operadores relacionais
- Desenvolver com estruturas compostas

3.1 Introdução

Até agora vimos os seguintes elementos de programação:

- bloco de programação
- entrada e saída de dados
- variáveis
- constantes
- atribuições
- expressões lógicas
- expressões relacionais
- expressões artiméticas
- comandos sequenciais

Já é um bom repertório para nosso estudo porém para ele ficar mais completo precisamos estudar alguns comandos que desviam o fluxo sequencial e natural de um algoritmo e de um programa assim como ocorre na estrada da figura 3.1.

Figura 3.1 - Que caminho seguir?

Muitos programas se comportam como a estrada acima: é necessário em certa hora tomar uma decisão baseada em uma condição. E toda decisão leva a uma consequência, como na vida real.

Vamos estudar outro exemplo típico de estrutura de decisão. Você vai ao banco e quer fazer um saque de R$100,00. O caixa eletrônico possui um programa com vários módulos, certo? Um programa para extrato, outro para depósito, outro para pagamento e um para saque. Ele trabalha com uma simples decisão: se houver saldo suficiente, então o saque é realizado, senão é mostrada uma

mensagem na tela para o cliente informando saldo insuficiente. Já recebeu uma mensagem "Transação não autorizada" ao tentar pagar com o seu cartão de débito? É baseada em uma estrutura de decisão.

Quer mais um exemplo para esclarecer? Você quer ler as notícias mais atuais da sua rede social digital favorita. Mas na tela inicial você tem que informar o seu usuário e senha. Se o seu usuário estiver correto e sua senha estiver correta, então você poderá fazer o *log in* no sistema, senão a tela inicial volta a ser apresentada para você fazer uma nova tentativa. Se for a terceira tentativa, então bloqueia a entrada, senão permite nova tentativa.

> Quer assistir um vídeo para ilustrar uma situação de decisão? Que tal esse: https://www.youtube.com/watch?v=vYXiFMLJ8XU. O piloto percebe que não vai conseguir pousar o avião na curta pista do aeroporto Santos Dumont no Rio de Janeiro e tomou a decisão de contornar e fazer outra tentativa. Esta manobra é chamada "arremetida" e você vai encontrar vários vídeos no Youtube sobre este assunto.

Percebeu como temos vários exemplos do nosso dia a dia que envolvem decisões?

Você deve lembrar-se do programa que estudamos no Capítulo 2, certo? Aquele que calculava a média do aluno. Ele nos ensinou bastante, porém ainda é um pouco sem graça. Não seria mais interessante se ele mostrar um resultado para o usuário informando se o aluno foi aprovado de acordo com sua média?

Tente falar em voz alta como seria o programa:

- "Tenho que ler as duas notas do aluno"
- "Calculo a média"
- "SE a nota for maior ou igual a 6 ENTÃO o aluno está aprovado"

3.2 Desvio condicional simples

O programa do cálculo da média agora vai ter o seguinte código:

```
1   #include <iostream>
2   using namespace std;
3
```

```
4   int main(int argc, char** argv) {
5   float nota1;
6   float nota2;
7   float media;
8
9   cout<<"Digite a primeira nota: "<<endl;
10  cin>>nota1;
11  cout<<"Digite a segunda nota: "<<endl;
12  cin>>nota2;
13  media = (nota1+nota2)/2;
14
15  if (media>=6){
16  cout<<"Aprovado"<<endl;
17  }
18  cout<<"Execute novamente para calcular outra media"<<endl;
19  return 0;
20  }
```

Antes de explicar o programa, tente entender o que foi feito. Você vai se surpreender que não é tão difícil quanto parece. Basta lembrar que "se" em inglês é "if" e outro fator importante para ser lembrado é que todo bloco de programação começa com { e termina com }.

Vamos relembrar alguns pontos do Capítulo 2 e explicar o programa parte por parte.

O problema consiste em calcular a média aritmética entre duas notas bimestrais de um aluno. Estas notas serão informadas pelo usuário e podem ter valores inteiros como 8, 9, 7 e valores decimais como 8.5, 9.1, 3.5, etc.

Portanto, a pergunta inicial é: quais variáveis temos que definir para este problema?

Teremos 2 entradas de dados, portanto, temos 2 variáveis e podemos chamá-las de nota1, representando a primeira nota e nota2 representando a segunda. Como as notas podem ser decimais, vamos usar o tipo *float*. Além disso, temos que calcular a média aritmética das 2 notas. Quando somamos 2 *floats*, o resultado será um *float*. Então vamos precisar de outra variável *float* para a média.

```
1   #include <iostream>
2   using namespace std;
```

As linhas 1 e 2 importam a biblioteca *iostream* para podermos usar os comandos de entrada e saída que aprendemos. Se não colocarmos esta linha, os comandos cin e cout não serão reconhecidos pelo compilador e o programa não funcionará.

Não esqueça que podemos pular quantas linhas quisermos no código. As linhas em branco não serão compiladas e servem como um fator estético e de legibilidade para o código ficar mais organizado. É o que ocorre na linha 3.

A linha 4 inicia o programa principal.

Nas linhas 5, 6 e 7 temos a declaração das variáveis que vamos utilizar no programa.

Com as variáveis prontas, vamos passar para a próxima etapa: ler os dados do usuário:

```
9   cout<<"Digite a primeira nota: "<<endl;
10  cin>>nota1;
11  cout<<"Digite a segunda nota: "<<endl;
12 cin>>nota2;
```

Agora as variáveis nota1 e nota2 conterão os valores informados pelo usuário. A próxima etapa é calcular a média:

```
13  media = (nota1+nota2)/2;
```

Agora entra a parte nova: fazer a avaliação da variável média. Embora seja um programa simples, esta etapa é muito importante, pois se errarmos na condição poderemos comprometer a saída dos resultados.

No nosso caso, a palavra “Aprovado” vai ser mostrada quando a média for maior ou igual a 6 e sendo assim precisamos usar o operador relacional correto. Desta forma:

```
15  if (media>=6){
16  cout<<"Aprovado"<<endl;
17  }
```

Na linha 15 é feito o teste e a avaliação da variável média. O operador relacional que usamos é o “>=”. Logo, comparamos o valor da variável média com 6 e se for maior ou igual, a linha 16 será executada e a palavra “Aprovado” será mostrada na tela.

Um detalhe: veja que só teremos 1 (UM e somente UM) comando após a linha 15. Quando isso ocorrer, podemos omitir os caracteres de início e fim de bloco ("{" e "}"). Mas lembre-se: somente quando tiver UM comando. Dois ou mais, deve-se colocar os delimitadores de bloco. Portanto, poderíamos ter escrito assim:

```
15  if (media>=6)
16  cout<<"Aprovado"<<endl;
```

E daria o mesmo resultado. É recomendável que os delimitadores de bloco sejam sempre usados, para deixar o código mais legível. Preze por isso sempre!

Lembre-se sempre da organização de um programa: entrada-processamento-saída. Até agora fizemos a entrada e o processamento, resta terminar o programa com a saída dos dados:

```
18  cout<<"Execute novamente para calcular outra media"<<endl;
```

A linha 16 também faz parte da saída dos dados afinal, ela mostra o resultado do cálculo da média.

As linhas restantes finalizam o programa. Não esqueça delas!

A figura 3.2 mostra a execução do programa considerando duas hipóteses: quando a condição da linha 15 é verdadeira e quando ela é falsa.

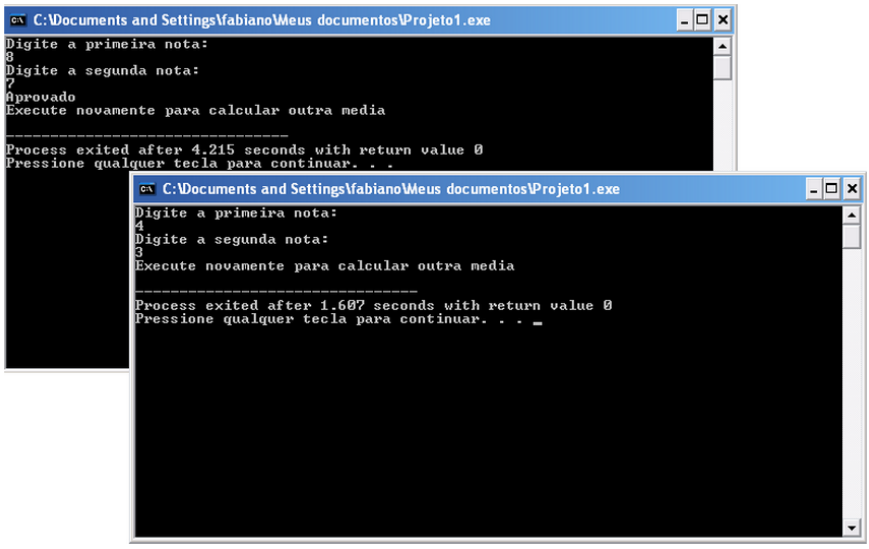


Figura 3.2 –Resultado da execução do programa

Sendo assim, quantos "if" tem no programa? Um só não é? Logo temos aqui a estrutura de decisão mais simples que existe: o desvio condicional simples. Esta estrutura portanto consiste na avaliação de somente uma condição. Se esta for verdadeira, o comando seguinte é executado.

A estrutura possui o diagrama da figura 3.3.

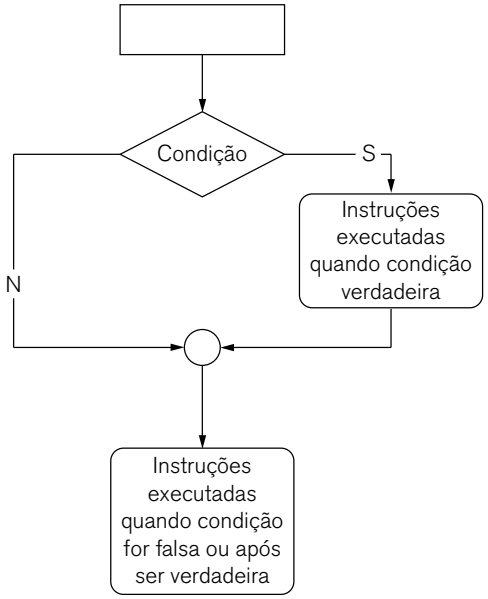


Figura 3.3 – Fluxograma da estrutura de decisão simples

Tenha muita atenção na condição! No exemplo da figura 3.2, a condição é que a média seja MAIOR OU IGUAL a 6. Lembra das tabelas verdade do Capítulo 1? Elas serão muito utilizadas daqui para frente.

O diagrama da figura 3.3 mostra um desvio de fluxo simples. A decisão é representada pelo losango e dele partem dois fluxos: um com a letra "N" (não) que corresponde a uma condição falsa e um "S" (sim) que corresponde a condição ser satisfeita e, portanto verdadeira e assim, o próximo comando será executado.

O diagrama finaliza no processo que representa a condição não ser satisfeita (ser falsa) que é o mesmo processo que é executado após o comando da condição verdadeira ser executado.

Em C++, a estrutura do comando será:

```
if (<condição>) {
<comandos quando a condição for verdadeira>
}
```

3.3 Operadores relacionais

Você vai perceber que existem decisões complexas as quais necessitam de recursos para representar esta complexidade. Estamos nos referindo aos operadores e expressões relacionais.

As expressões relacionais na sua maioria são expressões que comparam duas partes. No exemplo que estudamos, a comparação é feita entre a variável média e o número 6 (media>=6). As expressões relacionais serão usadas também na estrutura de repetição, que veremos no próximo capítulo.

Para nos auxiliar nas expressões relacionais precisamos lembrar os operadores relacionais que estão mostrados na tabela 3.1:

SÍMBOLO	DESCRIÇÃO
=	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a

Tabela 3.1 – Operadores relacionais

3.4 Desvio condicional composto

Vamos dificultar só um pouco. A estrutura que estudamos no tópico anterior é muito simples e existem situações que temos que avaliar a condição e dependendo do resultado, executar o conjunto de comandos que correspondem à condição verdadeira ou senão executar os comandos que correspondem à condição falsa.

Já demos alguns exemplos desta situação. Observe o diagrama representado na figura 3.4:

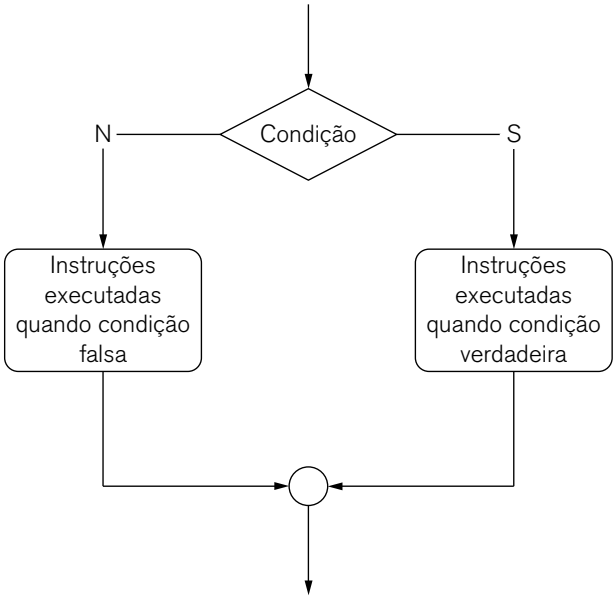


Figura 3.4 – Fluxograma do "se ... então ... senão"

Compare a figura 3.3 com a figura 3.4. Qual a principal diferença?

A diferença reside que na figura 3.4, caso a condição seja falsa, o fluxo do algoritmo segue por outro caminho e na figura 3.3 somente a condição verdadeira é desviada para outro fluxo. Portanto agora temos uma "bifurcação" no código e um tipo de ação diferente para cada resultado da avaliação da condição. Sendo assim trata-se de um desvio condicional composto.

Em C++, o comando fica desta forma:

```
if (<condição>) {
    <instruções executadas quando condição é verdadeira>
}
else {
    <instruções executadas quando condição é falsa>
}
```

Agora, vamos praticar um pouco. O próximo exemplo vai reescrever o programa do cálculo da média e ensinar outros fundamentos da linguagem C++, portanto preste atenção!

Exemplo 1

Leia duas notas bimestrais de um aluno. Calcule a média das notas. Se a nota for maior ou igual a 6, o aluno estará "aprovado", senão estará "reprovado". O programa deve ainda mostrar qual foi a média do aluno.

Antes de começar a escrever o programa, lembre-se que já fizemos uma análise anterior sobre este problema. Já sabemos quantas e quais variáveis serão usadas. Portanto, podemos começar o programa da mesma forma que o programa anterior:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   float nota1;
6   float nota2;
7   float media;
8
9   cout<<"Digite a primeira nota: "<<endl;
10  cin>>nota1;
11  cout<<"Digite a segunda nota: "<<endl;
12  cin>>nota2;
13  media = (nota1+nota2)/2;
```

Agora vem a parte nova: avaliar a condição e mostrar a mensagem correspondente:

```
15  if (media>=6){
16  cout<<"Media = "<<media<<" - Aprovado"<<endl;
17  }
18  else {
19  cout<<"Media = "<<media<<" - Reprovado"<<endl;
20  }
```

As linhas 15 a 17 são familiares para nós, pois é o mesmo código estudado anteriormente. Na linha 15 a condição é verificada e se for verdadeira, o bloco entre as linhas 15 e 17 será executado. Após a execução deste bloco o código passará a ser executado a partir da linha 21. Tudo que fizer parte do else, entre as linhas 18 e 20 será ignorado, pois a condição do if já foi verificada como verdadeira.

Caso a condição da linha 15 seja falsa, ou seja a média é um número menor que 6, o bloco do if será ignorado e o que será executado é o bloco do else que neste caso está entre as linhas 18 e 20.

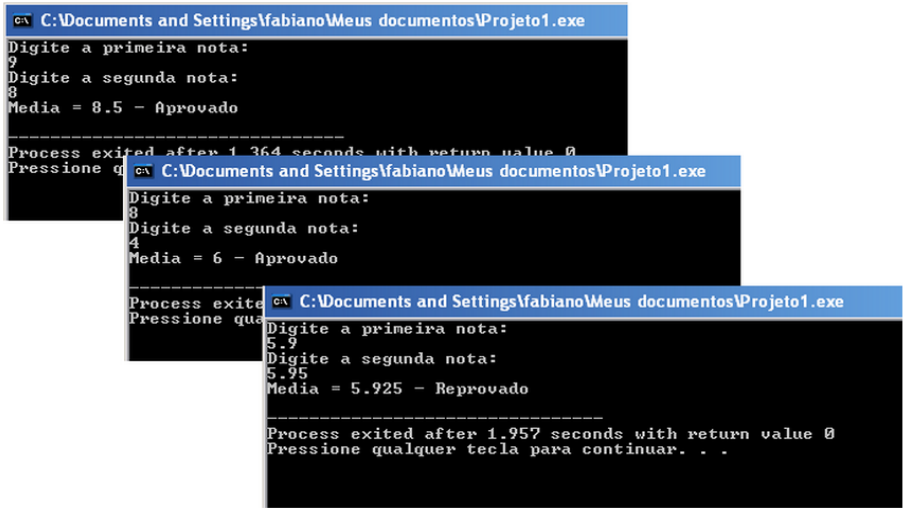


Figura 3.5 – Execução do programa

Na figura 3.5 apresentamos 3 execuções do programa para testar alguns valores e possíveis respostas geradas. Veja a tabela 3.2. Veja que no terceiro teste inserimos valores bem próximos de 6 e como a média ainda será menor que 6, o aluno estará reprovado.

NOTA1	NOTA2	MÉDIA	RESULTADO
9	8	8,5	Aprovado
8	4	6	Aprovado
5,9	5,95	5,925	Reprovado

Tabela 3.2 – Possíveis resultados

3.5 O operador ternário

Em algumas situações é possível substituir o if-then-else da linguagem C por um tipo de operador que envolve três partes: o operador ternário.

Veja como ele funciona:

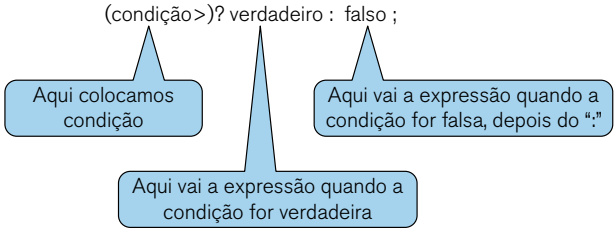


Figura 3.6 – O operador ternário.

É melhor exemplificar. O programa da Listagem 4 é muito simples. O programa pede para o usuário digitar um número. Se o número for positivo ele será incrementado ou se for negativo será decrementado. Feito isso será mostrada uma mensagem para o usuário informando o novo valor.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int numero;
6   cout<<"Digite um numero:"<<endl;
7   cin>>numero;
8   (numero>=0)? numero++ : numero--;
9   cout<<"O novo valor do numero eh:"<<numero;
10  return 0;
11  }
```

Listagem 1 – Operador ternário

A execução do programa será conforme a figura 3.7:

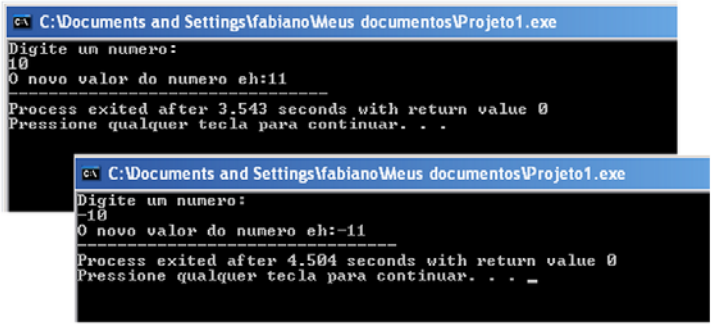


Figura 3.7 – Execução mostrando o operador ternário.

O programa é bem simples e exceto a linha 8, as outras já foram estudadas.

A linha 8 mostra o exemplo do uso do operador ternário. A condição a ser testada é "(numero>=10)". A condição pode ser lida assim: "O valor da variável numero é maior ou igual a 10?". Veja que até temos o ponto de interrogação (?) no comando. O "?" termina a condição.

A próxima parte do comando ("numero++") é o comando a ser executado quando a condição for verdadeira. Não é possível aqui colocar um bloco de comandos. O operador ternário só funciona com 1 comando de cada vez.

A seguir, os ":" limitam até onde vai o comando quando a condição for verdadeira e o início do comando quando a condição for falsa. Neste caso, o comando da parte falsa decrementa o valor da variável número ("numero--"). E terminamos o comando com o ";" usual. Veja a figura 3.8 para melhor entendimento.

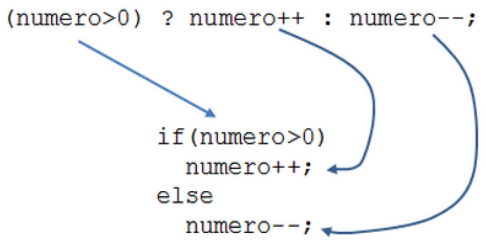


Figura 3.8 – Representação do operador ternário

3.6 Desvios condicionais encadeados

Nem sempre as coisas são tão simples assim. Existem situações que vários "se" devem ser avaliados. Até mesmo no nosso dia a dia precisamos avaliar vários "se" durante o dia:

"Se o caminho habitual para o trabalho estiver muito cheio, então eu pego o atalho. Se o atalho também estiver cheio e eu estiver atrasado então começo a chorar. Se eu não estiver atrasado e o atalho estiver com trânsito bom, ligo o rádio e escuto as músicas". E segue assim durante todo o dia.

É claro que é uma situação diária e corriqueira, mas computacionalmente também vamos encontrar situações semelhantes. Lembra do exemplo do Deep Blue do capítulo inicial? Imagine quantos "se" existem numa partida de xadrez.

Podemos então encadear, ou aninhar os "se". Ou seja, podemos montar um "se" dentro de outro. Isto é chamado de aninhamento ou encadeamento. A figura 3.9 mostra este conceito.

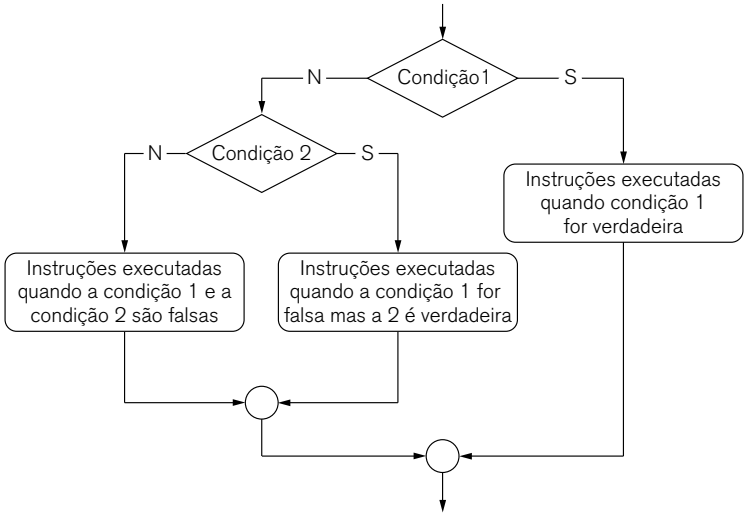


Figura 3.9 – Estrutura condicional composta ou encadeada

Vamos estudar como isso pode ser implementado em C++.

Exemplo 2

Faça um programa que calcule o reajuste do salário de um empregado. O empregado deve receber um reajuste de 15% caso seu salário seja menor que 500, se o salário for maior ou igual a 500, mas menor ou igual a 1.000, o reajuste será de 10%, caso seja maior que 1000, o reajuste deverá ser de 5%.

Antes de começar a codificar, vamos entender o problema:

- Se o salário < 500 [Símbolo] reajuste de 15%
- Se o salário estiver entre 500 e 1000, ou seja salário >=500 e salário <=1000 [Símbolo] reajuste de 10%
- Se o salário > 1.000 [Símbolo] reajuste de 5%

Para podermos entender melhor o problema, vamos fazer o fluxograma para visualizar o encadeamento.

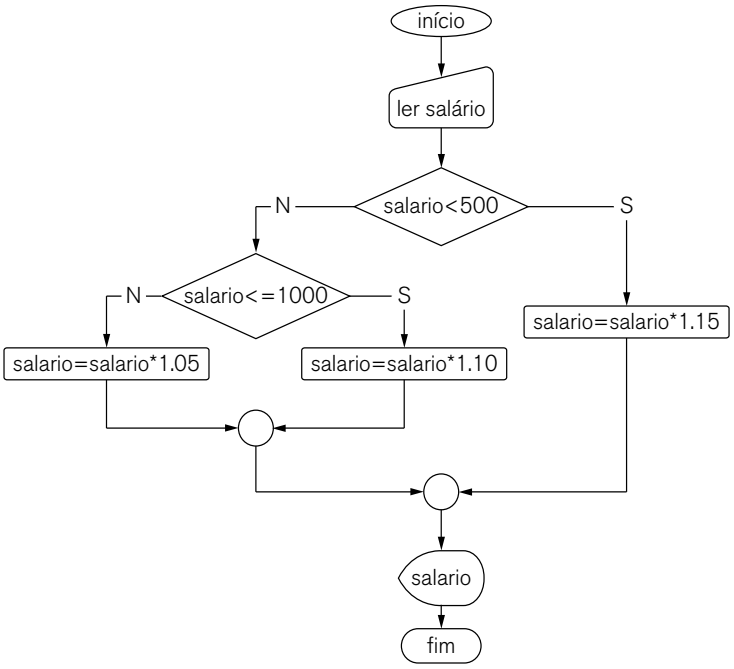


Figura 3.10 – Fluxograma dos ifs aninhados

Esta é a lógica do problema. E as variáveis? Quais poderemos prever que serão usadas? Basicamente temos por enquanto: salário e reajuste. E seus tipos? Neste caso, como vamos trabalhar com salário e reajuste é bom usar um tipo que permita uma grande variação de números decimais e para este caso o tipo *float* é adequado.

Vamos começar a escrever o programa:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   float salario,reajuste;
6
7   cout<<"Digite o salario:"<<endl;
8   cin>>salario;
```

Como podemos perceber, nenhuma novidade até agora: declaramos as variáveis previstas e fizemos a entrada de dados assim como está no fluxograma.

A parte nova começa agora:

O primeiro teste que temos que fazer é verificar se o salário é menor que 500. Se esta condição for verdadeira, devemos aplicar um reajuste de 15% sobre o salario (multiplicar o valor do salário por 1,15).

```
10  if (salario<500){
11  salario = salario*1.15;
12  }
```

Se a condição for falsa, vamos verificar se o salário é menor que 1000. Perceba que isto faz o teste do salário para verificar se ele está na faixa entre 500 e 1000. Se o salário for menor que 1000, o reajuste será de 10%.

```
13  else {
14  if (salario<=1000){
15  salario=salario*1.10;
16  }
```

Senão (salário > 1000), o reajuste será de 5%:

```
17  else {
18  salario=salario*1.05;
19  }
20  }
```

E assim terminamos a parte principal do programa. O código final está mostrado a seguir.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   float salario,reajuste;
6
7   cout<<"Digite o salario:"<<endl;
8   cin>>salario;
9
10  if (salario<500){
11  salario = salario*1.15;
12  }
```

```
13  else {
14  if (salario<=1000){
15  salario=salario*1.10;
16  }
17  else {
18  salario=salario*1.05;
19  }
20  }
21  cout<<"O novo salario e igual a R$"<<salario<<endl;
22  return 0;
23  }
```

Listagem 2 – Ifs aninhados

A figura 3.11 mostra 3 execuções do programa com diferentes valores para demonstrar o funcionamento.

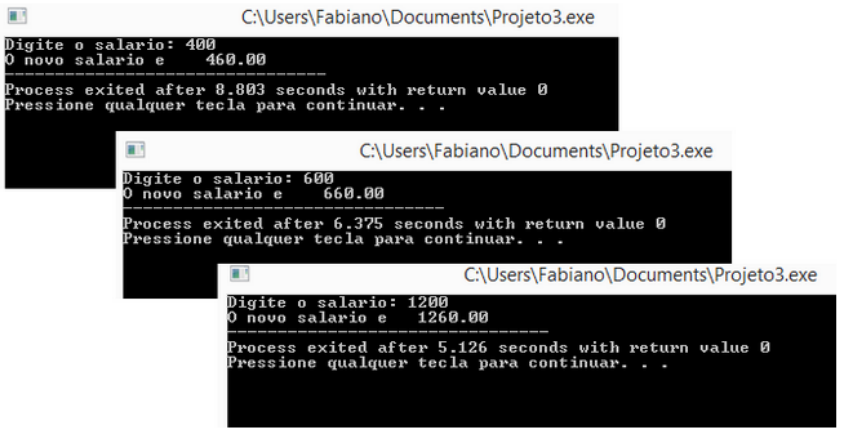


Figura 3.11 – Execução do programa de ifs aninhados

Preste muita atenção com os sinais de fechamento de bloco (“}”). É uma boa ideia fechá-los assim que você abrir um. Desta forma o risco de esquecer o fechamento é menor. Mais uma vez, observe a organização do código com os recuos e endentações. Imagine um programa bem maior e note o quão importante é organizar o código que você está escrevendo e outra pessoa poder dar manutenção posteriormente.

3.7 O comando switch

Quando se trata de uma estrutura de decisão, sem dúvida o comando if que estudamos é o mais importante e o mais usado em qualquer linguagem de programação.

Mas existe outro comando que é muito útil e pode ser usado quando temos vários ifs aninhados, trata-se do comando *switch*. Este comando é interessante ser usado para substituir vários ifs e deixar o código mais legível. E também é bastante usado em estruturas de menu. Porém temos uma limitação aqui: o comando não pode ser usado para avaliar expressões relacionais. Ele só funciona quando comparamos com um valor constante e quando a comparação é verdadeira, um bloco de comandos é executado.

O *switch* tem a seguinte sintaxe:

```
switch (variável){
    case constante1:
           Instruções;
    break;

    case constante2:
           Instruções;
    break;

    default
           Instruções;
}
```

Veja alguns exemplos:

Exemplo 3

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int numero;
6
7   cout<<"Digite um numero de 1 a 7:"<<endl;
8   cin>>numero;
9
```

```
10  switch (numero){
11  case 1 :
12  cout<<"Domingo"<<endl;
13  break;
14  case 2 :
15  cout<<"Segunda"<<endl;
16  break;
17  case 3 :
18  cout<<"Terça"<<endl;
19  break;
20  case 4 :
21  cout<<"Quarta"<<endl;
22  break;
23  case 5 :
24  cout<<"Quinta"<<endl;
25  break;
26  case 6 :
27  cout<<"Sexta"<<endl;
28  break;
29  case 7 :
30  cout<<"Sabado"<<endl;
31  break;
32  default :
33   cout<<"Digitou outra coisa!!"<<endl;
34  }
35  return 0;
36 }
```

O *switch* compreende as linhas 10 a 34. O valor a ser testado é o conteúdo da variável numero. Na linha 10, o valor de numero é avaliado.

Caso o valor seja igual a 1 (linha 11), será impressa na tela a mensagem "Domingo" (linha 12) e o *switch* será finalizado (linha 13).

Caso o valor seja igual a 2 (linha 14), será impressa na tela a mensagem "Segunda" (linha 15) e o *switch* será finalizado (linha 16). E assim por diante.

Porém se o usuário for "legal" e digitar um valor diferente de 1 a 7, o *switch* entre na cláusula *default* na linha 32 e escreve uma mensagem na tela. É muito importante que você use a cláusula *default* neste comando.

Veja a execução do programa:

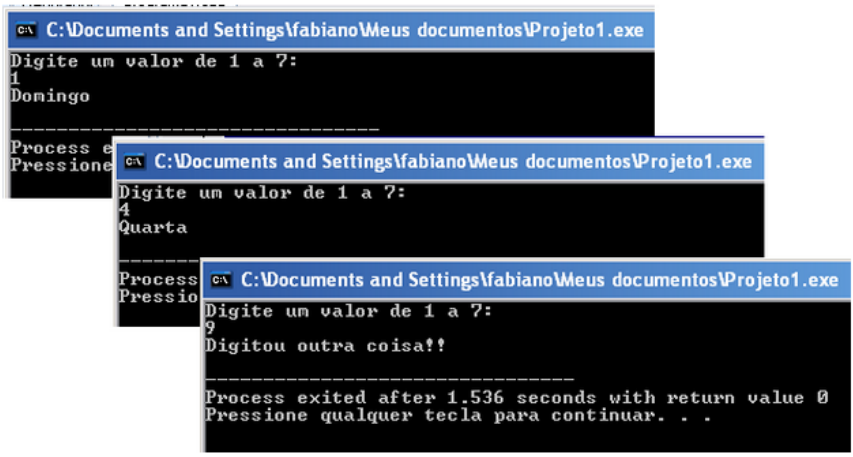


Figura 3.12 – Uso do comando switch:

O programa do Exemplo 3 é o mesmo programa que o descrito na Listagem 3. Quanta diferença na legibilidade não é?

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int numero;
6
7   cout<<"Digite um valor de 1 a 7: "<<endl
8   cin>>numero;
9
10  if (numero == 1)
11  cout<<"Domingo"<<endl;
12  else
13  if (numero == 2)
14  cout<<"Segunda"<<endl;
15  else
16  if (numero == 3)
17  cout<<"Terça"<<endl;
18  else
19  if (numero == 4)
```

```
20  cout<<"Quarta"<<endl;
21  else
22  if (numero == 5)
23  cout<<"Quinta"<<endl;
24  else
25  if (numero == 6)
26  cout<<"Sexta"<<endl;
27  else
28  if (numero == 7)
29  cout<<"Sabado"<<endl;
30  else
31  cout<<"Valor invalido!"<<endl;
32  return 0;
33  }
```

Listagem 3

Vamos dar uma complicadinha no próximo exemplo. “Só que não!”. Só é necessário um pouco mais de atenção:

Exemplo 4

O programa a seguir simula uma calculadora bem simples. O usuário vai digitar dois números e depois escolher uma dentre as quatro operações aritméticas para ter o resultado.

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int a, b, c, operacao;
6
7   cout <<"Digite o 1o valor: ";
8   cin >> a;
9
10  cout <<"Digite o 2o valor: ";
11  cin >> b;
12
```

```
13  cout <<"Que operacao deseja realizar?\n1. Adicao\t2.Subtracao\
t3.Multiplicacao\t4.Divisao\n\n=>";
14  cin >> operacao;
15
16  system ("cls");
17  switch (operacao){
18  case 1:
19  c=a+b;
20  cout <<a<<" + "<<b<<" = "<<c<<"\n\n";
21  break;
22  case 2:
23  c=a-b;
24  cout <<a<<" - "<<b<<" = "<<c<<"\n\n";
25  break;
26  case 3:
27  c=a*b;
28  cout <<a<<" * "<<b<<" = "<<c<<"\n\n";
29  break;
30  case 4:
31  c=a/b;
32  cout <<a<<" / "<<b<<" = "<<c<<"\n\n";
33  break;
34  }
35  return 0;
36  }
```

Neste programa temos algumas novidades:

• Na linha 16 usamos uma função chamada system("cls") para apagar o conteúdo da tela em execução. Vamos aprender sobre as funções no Capítulo 5.

• Usamos alguns caracteres especiais como o "\n" e o "\t" para pular uma linha e para tabular uma *string*, na linha 13. Estes caracteres são usados apenas para formatar a saída de dados.

Agora vamos dar uma pausa no exemplo e aprender algumas coisas:

O "\n" é um tipo especial de caractere chamado de sequência de escape. Este caractere, assim como outros, é usado nas linguagens de programação para representar algum caractere ou uma sequência de caracteres que são difíceis de serem representados.

Por exemplo, o "\n" significa "nova linha". Toda vez que o compilador encontra este caractere em uma *string* ele não vai interpretar a barra "\" e depois o "n" como dois caracteres separados. Eles serão um só e neste caso, vai inserir uma quebra de linha na *string*. Assim, a *string*:

```
"Olá\npessoal\ntudo\nbem?"
```

Será impressa assim:

```
Olá
pessoal
tudo
bem?
```

Existem outros caracteres de escape em C++. Veja outro exemplo: Até agora temos escrito nossas *strings* assim:

```
cout<<"Olá mundo";
cout<<"Este é um exemplo de uma string";
```

Mas e se fosse pedido para escrever a seguinte *string*:

```
Castro Alves era conhecido como o "Poeta dos Escravos".
```

Fácil! Poderíamos fazer com o *cout* sem problema!

```
cout<<“Castro Alves era conhecido como o “Poeta dos Escravos”.”);
```

Temos um problema! Observe as aspas nas *strings*. Sabemos que todo o texto delimitado pelas aspas será impresso na tela e as aspas não são impressas, mas agora precisamos que as aspas sejam impressas! E agora?

Vamos verificar o que está ocorrendo:

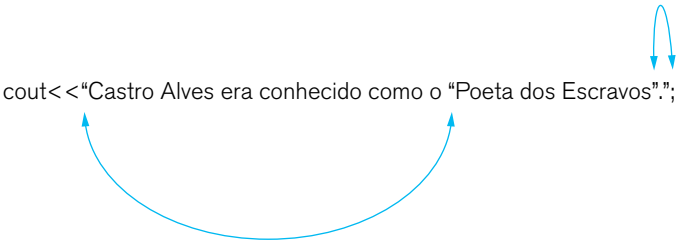


Figura 3.13 – Conferência das aspas na string

De acordo com o que sabemos sobre *strings* no comando cout, se deixarmos desta forma mostrada na figura 3.13 será impresso:

```
Castro Alves era conhecido como o
```

E teríamos um erro, pois o cout não vai aceitar o final da *string*.

Para que a *string* final seja impressa com as aspas, precisamos novamente de um caractere de escape. Neste caso o caractere é “\x22”. Desta forma, a *string* final terá que ser escrita assim:

```
cout<<“Castro Alves era conhecido como o \x22Poeta dos Escravos\
x22.”;
```

Ainda no caso das aspas, podemos usar outro caractere de escape: “\”. Esta forma é a mais usada inclusive em outras linguagens. Portanto, podemos escrever a string também de acordo com a forma a seguir:

```
cout<<“Castro Alves era conhecido como o \“Poeta dos Escravos\”.”;
```

Veja o Box para outros caracteres de escape:

A seguir apresentamos as principais sequências de escape na linguagem C:

ESCAPE	FUNÇÃO
	Caracter Nulo
\a	Alarme
\b	Retrocesso
\f	Avançar Página
\n	Quebra de Linha
\t	Tab Horizontal
\v	Tab Vertical
\\	Barra Invertida
\'	Aspa Simples
\"	Aspa Dupla

Voltando ao nosso exemplo ...

Veja que nas linhas 18 a 21, assim como os outros case usamos mais do que um comando dentro do case. Isto é possível. Mas não se esqueça de terminar o *case* com um *break*.

```
18  case 1:
19  c=a+b;
20  cout <<a<<" + "<<b<<" = "<<c<<"\n\n";
21  break;
```

Qual o problema que você encontra neste programa?

Resposta: Não temos a cláusula *default*. Se o usuário for "legal" novamente e digitar 5, o programa vai executar incorretamente.

3.8 Operadores lógicos

Embora já tenhamos visto, vamos explorar um pouco mais os operadores lógicos, que são os operadores responsáveis por estabelecer o relacionamento de duas ou mais condições ao mesmo tempo na mesma instrução e assim possibilitar a realização de vários testes simultâneos.

Em C temos três operadores: o operador E, representado pelos caracteres &&, o operador OU, escrito como || em C e o operador de negação, representado pelo caractere ! (ponto de exclamação).

Vamos estudar estes operadores por meio de exemplos:

Exemplo 5

Faça um programa que verifique se um número informado pelo usuário está no intervalo entre 50 e 100.

Para resolver este problema temos que considerar que o número a ser informado tem que ser maior ou igual a 50 e ser menor ou igual a 100. Portanto, temos que avaliar duas expressões simultaneamente, observe:

```
Numero>=50 E Numero<=100, em C:  (numero>=50)&&(numero<=100)
```

Lembre-se que neste caso, a expressão será verdadeira somente se cada condição for verdadeira, de acordo com a tabela verdade:

CONDIÇÃO 1	CONDIÇÃO 2	RESULTADO
Falsa	Falsa	Falsa
Falsa	Verdadeira	Falsa
Verdadeira	Falsa	Falsa
Verdadeira	Verdadeira	Verdadeira

Tabela 3.3 – Tabela verdade para o operador E

Vamos ao programa. Por enquanto, sem novidades:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int numero;
6
```

```
7   cout<<"Digite um numero:"<<endl;
8   cin>>numero;
9
```

Agora o teste com as duas expressões:

```
10  if ((numero>=50) && (numero<=100)){
11  cout<<"O numero "<<numero<<" esta dentro do intervalo";
12  }
13  else {
14  cout<<"O numero "<<numero<<" NAO esta dentro do intervalo";
15  }
```

Perceba que cada expressão está dentro de parênteses e além disso, toda a linha fica dentro de parênteses. A linguagem C++ não permite que um if seja avaliado sem que a expressão toda esteja dentro dos parênteses.

Com o teste feito, partimos para a saída dos dados e finalização do programa. A listagem completa está logo abaixo:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int numero;
6
7   cout<<"Digite um numero:"<<endl;
8   cin>>numero;
9
10  if ((numero>=50) && (numero<=100)){
11  cout<<"O numero "<<numero<<" esta dentro do intervalo";
12  }
13  else {
14  cout<<"O numero "<<numero<<" NAO esta dentro do intervalo";
15  }
16  return 0;
17  }
```

Listagem 4 – Uso do operador &&

A execução do programa está mostrada na figura 3.14.

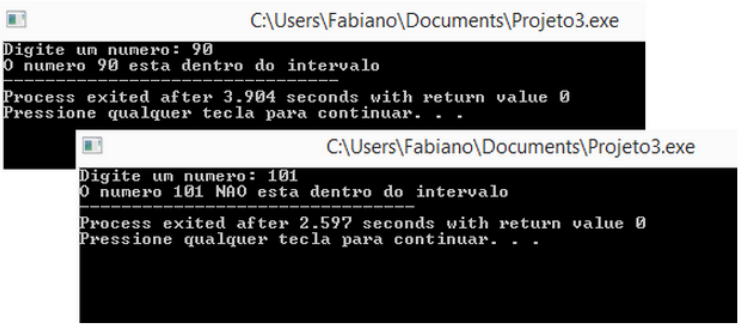


Figura 3.14

Exemplo 6

O próximo exemplo é sobre o operador OU (||). O objetivo do programa é responder a uma pergunta corretamente e para isso duas condições serão avaliadas. De acordo com a tabela verdade para o operador OU, o resultado da avaliação da expressão será verdadeiro quando uma das condições for verdadeira.

CONDIÇÃO1	CONDIÇÃO2	RESULTADO
Falsa	Falsa	Falsa
Falsa	Verdadeira	Verdadeira
Verdadeira	Falsa	Verdadeira
Verdadeira	Verdadeira	Verdadeira

Tabela 3.4 – Tabela verdade para o operador OU

Vamos ao programa:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   char resposta;
6
7   cout<<"Cadastro de Clientes - Sexo (m) ou (f)?"<<endl;
8   cin>>resposta;
9
10  if ((resposta=='m')||(resposta=='f')){
11  cout<<"Campo validado!"<<endl;
```

```
12  }
13  else {
14  cout<<"Voce nao digitou corretamente"<<endl;
15  }
16  return 0;
17  }
```

Neste programa usamos uma variável do tipo char. Este tipo admite um conteúdo de apenas 1 caractere. Um nome, ou um conteúdo contendo mais de 1 caractere já é considerado uma string e há outra forma de trabalhar com strings em C. Veremos isso adiante.

Na linha 10 ocorre a comparação com as duas expressões. Note que o operador "==" é usado. Este operador é do tipo relacional e serve para comparar se um determinado valor é igual a outro. Como estamos usando caracteres, fazemos a comparação entre a resposta e os caracteres 'm' ou 'f'. Note que quando estamos nos referindo a um caractere apenas, usamos aspas simples.

A figura 3.15 mostra a execução do programa.

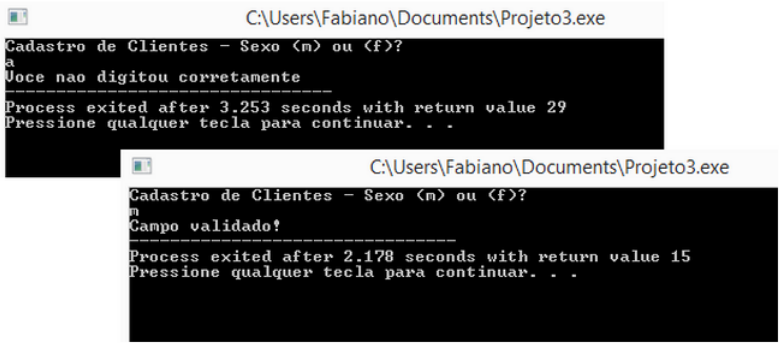


Figura 3.15 – Execução do programa

E para terminar, vamos ver um exemplo com o operador de negação.

Exemplo 7

Este programa é bem simples. O objetivo dele é realizar o cálculo c=(a+b)*x somente se o valor da variável x não for maior que 5. Qualquer valor menor que 5 fará o cálculo c = (a-b)*x.

Programas que envolvem relações lógicas precisam ser muito bem entendidos pois os operadores lógicos as vezes podem confundir. A dica é ler muito bem a especificação do problema.

Como já estamos acostumados, quais variáveis serão usadas no programa? Como serão lidos 3 valores inteiros, vamos precisar de 3 variáveis inteiras: a, b e x.

Começando o programa sem novidades:

```
1   #include <iostream>
2
3   int main(int argc, char** argv) {
4   int a,b,c,x;
5
6   cout<<"Digite o valor de a:"<<endl;
7   cin>>a;
8
9   cout<<"Digite o valor de b:"<<endl;
10  cin>>b;
11
12  cout<<"Digite o valor de x:"<<endl;
13  cin>>x;
```

Os valores foram lidos e agora vem a parte importante. Antes de codificar, vamos testar alguns valores para poder entender melhor o problema. Observe a tabela 3.5:

A	B	X	X>5?	CÁLCULO A SER FEITO	RESULTADO
5	1	2	Não	c=(a+b)*x	12
5	1	6	Sim	c=(a–b)*x	24

Tabela 3.5 – Testes para o exemplo

Se atribuirmos os valores 5, 1 e 2 para a, b e x respectivamente, de acordo com o enunciado do problema, vamos verificar que o valor de x não é maior que 5. Então neste caso, o cálculo a ser efetuado é c=(a+b)*x e o resultado de c será 12.

No outro exemplo se atribuirmos os valores 5, 1 e 6 para a,b e x, o cálculo será o c=(a-b)*x e o resultado será 24.

Podemos então usar o operador de negação neste caso. Vamos ver como isto pode ser feito:

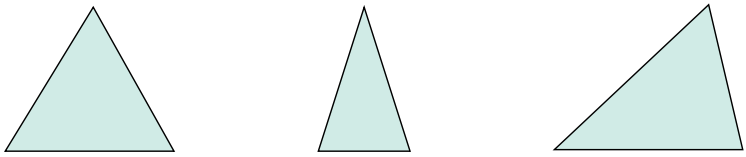
```
15  if (!(x>5)) {
16  c = (a+b)*x;
17  }
18  else {
19  c=(a-b)*x;
20  }
```

A condição será (x>5), lembre-se sempre dos parênteses, e como queremos que o cálculo com a soma entre a e b seja feito quando o x não é maior que 5, usamos a negação (!) para isso, e novamente colocamos parênteses, como mostra a linha 15.

O else da linha 18 será executado quando x for maior que 5.

Se você percebeu, daria para fazer este programa de outra forma sem usar a negação. Porém, o objetivo deste exemplo foi de mostrar como o operador é usado e o tipo de situação que ele pode ser utilizado.

Para terminar, vamos elaborar um programa final, mais abrangente e com uma aplicação mais prática. O objetivo do programa é identificar se um dado triângulo é equilátero, isósceles ou escaleno. Lembrando das aulas de geometria: um triângulo equilátero possui os 3 lados iguais, o isósceles possui 2 lados iguais e o escaleno não possui lados iguais.



Equilátero Isósceles Escaleno

Continuando com a recordação das aulas de geometria, um triângulo é uma forma geométrica composta por 3 lados onde cada lado é menor que a soma

dos outros dois lados. Isto não pode ser discutido pois trata da definição de um triângulo e temos que obedecer a esta regra.

Então, sendo os lados de um triângulo a, b e c, temos o seguinte:

- a < b + c
- b < a+ c
- c < a + b

Agora temos que criar as formas de resolver o problema.

- Para ser um triângulo equilátero, temos que: a = b, b = c e c = a.
- Para ser um isósceles, a=b ou a=c ou b=c
- Para ser escaleno, a!=b, b!=c (lembre-se que "!=" é o símbolo para "diferente de")

Quais variáveis vamos precisar para o problema? Certamente que os lados a, b e c precisarão ser lidos e serão nossas primeiras variáveis. Para facilitar, vamos considerar apenas valores inteiros.

Começando o programa, vamos proceder com a leitura dos dados.

```
1   #include <stdio.h>
2
3   int main(int argc, char** argv) {
4   int a,b,c;
5
6   cout<<"Digite o valor de a:"<<endl;
7   cin>>a;
8
9   cout<<"Digite o valor de b:"<<endl;
10 cin>>b;
11
12  cout<<"Digite o valor de c:"<<endl;
13  cin>>c;
```

Com os dados lidos, precisamos em primeiro lugar verificar se os dados satisfazem a condição de existência do triângulo. Isto é feito na linha 15.

```
15   if ((a<b+c) && (b<a+c) && (c<a+b)){
     ...
27  }
28  else {
29  cout<<"Os dados nao formam um triangulo");
30  }
```

Veja que neste caso já colocamos a parte do else para tratar a condição falsa. Sobre a condição, veja que temos 3 expressões que são avaliadas juntas. Se uma das condições for falsa, toda a expressão será falsa e não será caracterizado que estamos lidando com um triângulo.

Vamos agora escrever o código para quando a expressão da linha 15 é verdadeira. A linha 16 avalia a possibilidade de ser um triângulo equilátero. Para que isso ocorra, a=b=c:

```
16  if ((a==b) && (b==c)){
17  cout<<"Triangulo equilatero");
18  }
19  else {
20
26  }
```

Caso os lados não sejam iguais, temos a possibilidade de estar lidando com um triângulo isósceles ou escaleno. Neste caso, o teste será feito comparando dois lados. Se dois lados forem iguais, o triângulo é isósceles, em caso negativo, o triângulo só pode ser o escaleno.

```
20  if ((a==b) || (a==c) || (c==b)){
21  cout<<"Triangulo isosceles");
22  }
23  else {
24  cout<<"Triangulo escaleno");
25  }
```

E sendo assim, finalizamos o programa. O código final do programa está mostrado na Listagem 7:

```
1   #include <iostream>
2
3   int main(int argc, char** argv) {
4   int a,b,c;
5
6   cout<<"Digite o valor de a:”<<endl;
7   cin>>a;
8
9   cout<<"Digite o valor de b:”<<endl;
10  cin>>b;
11
12  cout<<"Digite o valor de c:”<<endl;
13  cin>>c;
14
15  if ((a<b+c) && (b<a+c) && (c<a+b)){
16  if ((a==b) && (b==c)){
17  cout<<"Triangulo equilatero");
18  }
19  else {
20  if ((a==b) || (a==c) || (c==b)){
21  cout<<"Triangulo isosceles");
22  }
23  else {
24  cout<<"Triangulo escaleno");
25  }
26  }
27  }
28  else {
29  cout<<"Os dados nao formam um triangulo");
30  }
31  return 0;
32  }
```

Listagem 5 – Código final do programa classificador de triângulos

Figura 3.16 – Execução do programa que classifica triângulos

Com este exemplo finalizamos o segundo tipo de estrutura usados na programação de computadores: a estrutura de decisão. Não deixe de praticar para podermos avançar para o próximo tipo de estrutura: a de repetição.

LEITURA

Apresentamos alguns livros sobre a linguagem C os quais podem complementar o seu aprendizado. O destaque fica para o livro de Kernighan e Ritchie, pois esta obra é um dos clássicos sobre a linguagem C++.

- FEOFILOFF, P. Algoritmos em linguagem C. São Paulo: Elsevier, 2008.
- MIZRAHI, V. V. Treinamento em linguagem C. São Paulo: Prentice Hall Brasil, 2008.
- KERNIGHAN, B. W.; RITCHIE, D. M. The C programming language. Prentice Hall, 1989.

REFERÊNCIAS BIBLIOGRÁFICAS

O melhor aprendizado na área de programação é programar sempre. Já citamos que aprender uma linguagem de programação é como aprender um novo idioma: quanto mais praticar mais vai aprender.

As estruturas de decisão são muito importantes para a programação em geral e o que vimos aqui é apenas o começo do seu aprendizado. Você consegue pensar em exemplos de uso prático deste tipo de estrutura?

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos de programação de computadores**. São Paulo: Pearson Education, 2008.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. **Algoritmos**. São Paulo: McGraw Hill, 2009.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Campus, 2008.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. D. **Algoritmos**: lógica para desenvolvimento de programação. 9ª. ed. São Paulo: Érica, 1996.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Education, 2003.

4

Estrutura de Repetição

"O oposto da vida não é a morte: é a repetição."

Geraldo Eustáquio

"Há repetição em todos lugares, e nada é encontrado apenas uma vez no mundo."

Goethe

"Meu maior medo: repetição."

Max Frisch

A última estrutura de programação que veremos é a estrutura de repetição. Existem vários nomes para as repetições: *loopings*, laços ou malhas de repetição. Assim como as outras estruturas, as repetições são encontradas e essenciais em qualquer linguagem de programação e são muito utilizadas. E na vida prática também: cálculo de folha de pagamento, impressão de boletos de cobrança, jogos de computador, basicamente em todos os programas vamos encontrar algum tipo de repetição.

Os *loopings* podem ser de vários tipos e vamos estuda-los neste capítulo. Existem loopings que não sabemos quantas vezes serão executados, outros conseguimos estabelecer este número, outros são controlados por meio de algum resultado de outra expressão, enfim, existem vários. Vamos estudá-los a partir de já. Bons estudos! E não esqueça: pratique, pratique e quando tiver um tempinho, pratique!

OBJETIVOS

Ao final deste capítulo você estará apto a:

- Identificar a necessidade de uso de uma estrutura de repetição
- Aplicar os comandos while, do-while e for em diferentes contextos
- Construir programas que atendam a diferentes situações agregando seleção e repetição
- Saber como depurar um programa em C++

4.1 Por que repetir?

Você deve se lembrar das aulas de matemática e do conceito de fatorial de um número n, certo? O fatorial de n é representado por n! e consiste de se encontrar o produto de todos os números positivos diferentes de zero e menor que o número. O fatorial é importante para expressões estatísticas e de análise combinatória.

Como exemplo, vamos fazer um programa em C++ para calcular o fatorial de 6, ou 6!. O programa é bem simples, observe:

```
1   #include <iostream>
2
3   int main() {
4   int fat;
5
6   fat = 6*6;
7   fat = fat*5;
8   fat = fat*4;
9   fat = fat*3;
10  fat = fat*2;
11  fat = fat*1;
12
13  cout<<"Fatorial de 6 = "<<fat<<endl;
14 return 0;
15 }
```

O resultado do programa mostrado na linha 13 é 4.320.

Bem, o resultado está correto, o programa também está, mas ficou simples porque 6 é um número pequeno. Mas e se fosse para calcular o fatorial de 600? Teríamos que escrever 600 multiplicações? Seria muito trabalhoso!

Imagine uma pessoa do setor de recursos humanos de uma faculdade. Uma faculdade é uma organização onde existem vários tipos de profissionais: aqueles que ganham por hora, outros por aula, outros possuem um salário fixo e ainda outros. Suponha que esta faculdade possua 500 funcionários. Já pensou o funcionário do RH na época de fechamento da folha de pagamento? Fazer o cálculo de um a um dos funcionários? Que trabalho hein?

E a urna eletrônica? Por mais polêmica que ela seja, já pensou como seria contar voto a voto manualmente em uma eleição para presidente do Brasil? Atualmente o resultado sai em poucas horas mas até bem pouco tempo levava alguns dias para isso.

O que você consegue encontrar de comum entre essas situações? Não são trabalhos repetitivos sendo que a cada repetição temos que fazer algum tipo de processamento? Não é melhor usar o computador para fazer esse trabalho para nós? Nós ficamos cansados, o computador não! Ainda bem que existem algumas estruturas de programação que quebram esse galho para nós. E na linguagem C++ temos algumas possibilidades de se fazer isso.

Figura 4.1 – Loopings.

4.2 Repetir até quando? Analisando entradas e saídas

Existem vários tipos de repetição que podemos usar nos nossos programas. Cada uma delas possui peculiaridades que devem ser observadas a fim de criarmos programas mais eficientes.

Em qualquer tipo, a repetição será comandada por uma expressão que sempre será avaliada.

Se a expressão for verdadeira, a repetição continua.

Se for falsa, a repetição para.

A condição da expressão é alterada dentro do bloco de código e se o bloco não for bem montado e a condição mudada, a repetição não terminará e isso gerará um termo chamado *loop* infinito. E isso é tudo que não queremos!

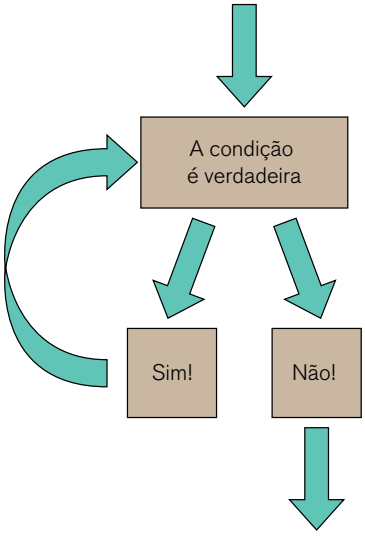


Figura 4.2 – Esquema de uma repetição.

Portanto, o que é importante em uma estrutura de repetição é controlar muito bem o início da repetição e isto é feito por meio da entrada de dados, e o fim da repetição.

Durante a repetição, alguns processamentos vão ser executados e estes processamentos vão definir a continuidade da repetição. Logo, a saída de dados vai controlar também a repetição, como já foi dito.

Novamente nos deparamos com aquele conceito de: entrada, processamento e saída. Mas agora, a saída vai controlar a entrada, por exemplo:

Se você estiver em um caixa eletrônico, você vai seguir os seguintes passos:

1. Passar o seu cartão magnético (entrada de dados)
2. Digitar sua senha (entrada de dados)
3. A máquina vai verificar se sua senha é válida (processamento)
4. Se for válida, vai apresentar o menu de opções (saída)
5. Se não for, volta ao passo 1 (saída controlando a entrada)

Atualmente os bancos possuem um mecanismo de segurança que novamente usa os conceitos de repetição para garantir que este processo não se repita por mais de três vezes. Na terceira vez se o cliente erra a senha, o cartão é bloqueado. Ou seja, analisar a entrada e a saída de dados é fundamental para a continuação da repetição.

Vamos passar agora ao estudo dos tipos de repetição.

Laços, *loops* ou *loopings* são formas sinônimas para repetição. Assista este vídeo no YouTube. Trata-se de um vídeo com estruturas de repetição usando um programa gráfico de aprendizado. Disponível em: < https://www.youtube.com/watch?v=7pZYh3f9nuE >.

4.3 Estilos de repetição

4.3.1 Repetição controlada por contador

Como o nome sugere, este tipo de estrutura de repetição usa um tipo de variável para contar o número de repetições que um determinado laço terá. Esta variável é chamada de contador e é usado o tipo int para poder implementá-la.

Veremos durante este capítulo que podemos usar basicamente três comandos na linguagem C++ para fazer repetições:

- O comando for
- O comando while
- O comando do-while

Todos estes comandos usam contadores para controlar suas repetições. Vamos começar aprendendo o comando for.

A estrutura que permite controlar o número de repetições é chamada de para ou pelo nome do seu comando, for.

A sintaxe geral do for é

```
for(inicialização ; condição ; incremento) <comandos>
```

O comando for em C++ é composto por três partes:

• A inicialização: onde declaramos uma variável do tipo inteiro (sempre) e atribuímos um valor inicial

• A condição: é uma expressão relacional que determina quando a repetição acaba. Quando a condição se tornar falsa, a execução do programa continua no comando seguinte ao for.

• O incremento: esta parte configura como a variável de controle da repetição varia cada vez que a repetição é executada. O incremento pode ser positivo ou negativo.

Exemplificando:

```
#include <iostream>
1.  using namespace std;
2
3   int main(int argc, char** argv){
4   int x;
5
6   for (x=1; x<=10; x++)
7   cout<<x;
8   }
```

Você consegue imaginar o que este programa faz? Analise a sintaxe do for, o programa e o que você já aprendeu da linguagem C++ até aqui. A execução está mostrada na figura 4.3.

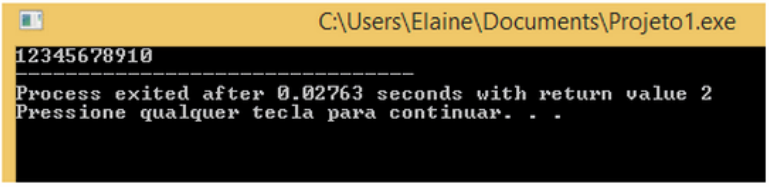


Figura 4.3 – Execução do comando for

Alguns detalhes sobre o programa:

• Como só existe 1 comando dentro do *looping*, não é necessário colocar o abre e fecha chaves (“{“ e “}”)

• Lembre-se sempre do valor inicial da repetição e da condição. Se ao invés do “<=10” fosse escrito somente “<10”, o resultado seria outro.

• Veja que usamos a forma abreviada no incremento. Acostume-se a usar o incremento da forma que está exemplificada aqui.

Portanto, o programa imprime na tela o valor de x, sem pular linha, 10 vezes, variando de 1 (valor inicial), somando 1 ao x (x++), enquanto o x for menor ou igual a 10, repetindo a impressão de uma maneira controlada e configurada pelo programador.

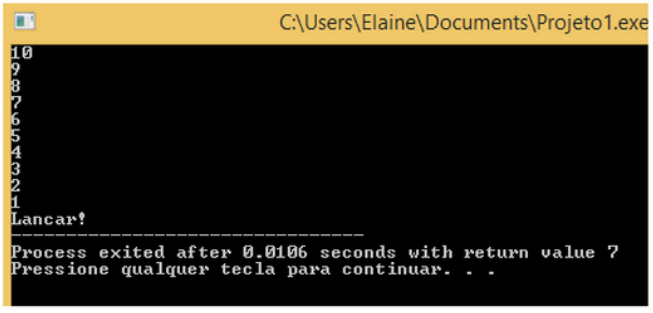
Veja outro exemplo:

Exemplo 1

O programa a seguir poderia ser usado na NASA para lançar foguetes:

```
#include <iostream>
1    using namespace std;
2
3    int main(int argc, char** argv){
4    int x;
5    for (x=10; x>0; x--)
6    cout<<x<<endl;
7    cout<<"Lançar!";
8    }
```

Este exemplo é bem parecido com o anterior, porém neste caso fizemos uma contagem decrescente, usando x—como “incremento”.



4.3.2 Repetição com limite do contador determinado pelo usuário

Este tipo de *loop* é caracterizado por uma estrutura que faz um teste lógico no *looping*. Portanto, a expressão é avaliada e caso ela seja falsa, todo o bloco que seria repetido não é executado.

Caso a condição seja verdadeira, o bloco é executado e após a execução, a condição é avaliada novamente.

Se ela for verdadeira, o bloco é novamente executado e se for falsa, a repetição para. E assim por diante.

Esta estrutura é chamada de enquanto-faça e é bem simples de ser entendida: "Enquanto a condição for verdadeira, faça o que está dentro do bloco". Em C++ o comando apropriado para esta estrutura é o while e vamos usar um exemplo para estudá-lo.

A sintaxe do comando while em C++ é:

```
while (<condição>) {
  <bloco de comandos>
}
```

O fluxograma para este tipo de repetição é mostrado na figura 4.4.

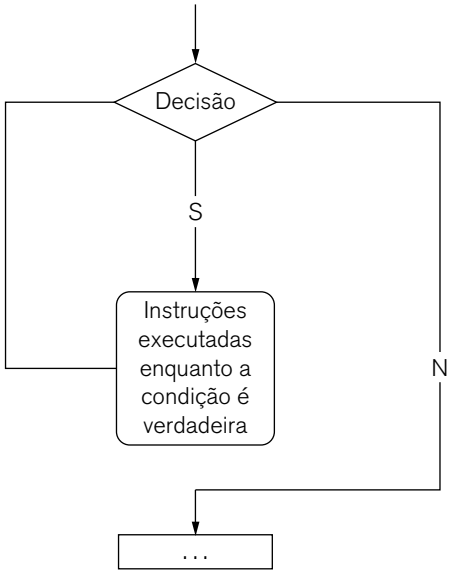


Figura 4.4 – Enquanto-faça

Exemplo 2: Enquanto-faça

Podemos usar como exemplo um programa que mostre a tabuada de qualquer número, informado pelo usuário.

Vamos começar com o que sempre fazemos: quais são as variáveis que poderão ser usadas pelo programa? Vamos ter um número digitado pelo usuário que será a tabuada que queremos mostrar, logo temos nossa primeira variável, numero.

Nas estruturas de repetição usaremos o contador. Esta variável será a que vai controlar nossa estrutura de repetição. Ela será incrementada até que a condição se torne falsa e termine o programa. Você verá que esta variável vai ser usada muitas vezes nos programas que envolvem repetição e é claro, ela pode assumir qualquer nome, desde que faça sentido e que fique claro que ela é o contador da repetição.

E como estamos tratando de tabuada, podemos ter uma variável chamada resultado. Esta variável guarda o resultado da multiplicação.

Antes de tratar do código, vamos entender o que vai acontecer.

Quando estudamos tabuada no colégio, a professora nos ensinava assim (exemplo com a tabuada do 2):

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

Com base no que você já aprendeu de programação e também com um pouco de intuição, observe a tabuada do 2 e tente responder às perguntas:

- Qual é o número que está fixo na tabuada?
- Por que ele está fixo?
- Como ele apareceu?
- O que está repetindo?
- Quantas vezes ocorrem as repetições?

- Por que ocorre esse número de repetições?
- O que acontece em cada repetição?

Agora veja as possíveis respostas para as perguntas na figura 4.5.

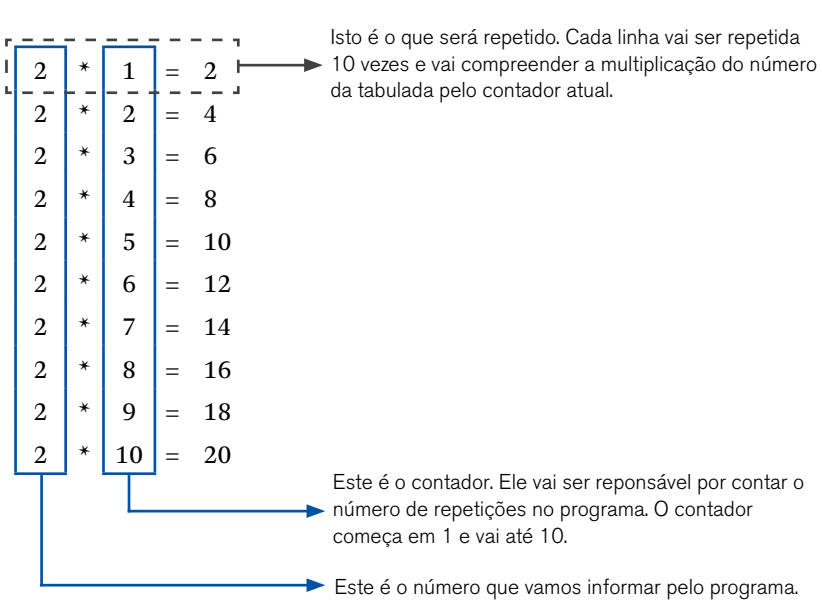


Figura 4.5 – Explicação da tabuada do 2.

O importante para o programador iniciante é conseguir enxergar o que será repetido. No nosso exemplo, a repetição será cada linha da tabuada ou seja, uma conta bem simples: 2 vezes o valor do contador. O "2" será lido pelo programa. E vamos repetir 10 vezes porque normalmente aprendemos a tabuada de 1 a 10.

Vamos ao código? Iniciando pela declaração de variáveis:

```
1   #include <iostream>
2
3   int main(int argc, char** argv) {
4   int numero;
5   int contador=1;
6   int resultado;
```

Sem novidades não é? Exceto pela linha 5. Até agora vimos que apenas declaramos as variáveis sem inicializa-las, como ocorre nas linhas 4 e 6. Mas na linha 5 usamos um recurso da linguagem C++, e de outras também, que inicia a variável com um valor e assim quando formos usar a variável pela primeira vez, ela já terá um valor inicial definido, portanto escrever:

```
int contador = 1;
```

É o mesmo que escrever:

```
int contador;
contador = 1;
```

Nas linhas 8 e 10 perguntamos ao usuário qual a tabuada que gostaria de ser mostrada:

```
8   cout<<"Qual a tabuada?"<<endl;
9   cin>>numero;
```

E nas linhas 11 a 15 entra a parte nova:

```
11  while (contador<=10){
12  resultado = numero*contador;
13  cout<<numero<<" * "<<contador<<" = "<<resultado<<endl;
14  contador = contador+1;
15  }
```

Na linha 11 está o tipo de repetição que será usado. De acordo com a sintaxe do while, a condição deve estar entre parênteses assim como está na linha 11. Observe que o valor atual de contador é 1, sendo assim, a condição é satisfeita e o *looping* começa.

Na linha 12 é feita a conta para ser mostrada na tela. Número é a variável que foi lida na linha 9 e contador é a variável de repetição. Inicialmente temos os valores 2 e 1 respectivamente, supondo que o usuário escolheu a tabuada do 2 para ser mostrada.

A linha 14 é fundamental. Sem ela entraríamos em um *loop* infinito. O resultado de um *loop* infinito muitas vezes é o que chamamos de "computador travado" pois a impressão que temos é que nada está sendo processado naquele momento e dependendo do tamanho da tela, teremos a impressão que o computador está "travado".

O *loop* infinito também gerará consumo da memória e do processamento do processador e dependendo do tamanho do bloco de comandos dentro do looping e o que está sendo processado, o computador poderá ficar sem memória e as consequências desse deslize podem comprometer coisas maiores.

Observe na linha 14 que a variável contador é incrementada de 1. Desta forma teremos a progressão da tabuada de 1 em 1. Se o incremento fosse feito em 2 unidades, teríamos outros resultados na tela.

Portanto o programa final está mostrado na Listagem 1.

```
#include <iostream>
1    using namespace std;
2
3    int main(int argc, char** argv) {
4    int numero;
5    int contador=1;
6    int resultado;
7
8    cout<<"Qual a tabuada?"<<endl;
9    cin>>numero;
10
11   while (contador<=10){
12   resultado = numero*contador;
13   cout<<numero<<" * "<<contador<<" = "<<resultado;
14   contador = contador+1;
15   }
16   cout<<"*** FIM ***";
17   return 0;
18   }
```

Listagem 1 – Programa da tabuada.

A figura 4.6 mostra a execução do programa, tendo como numero o valor 2.

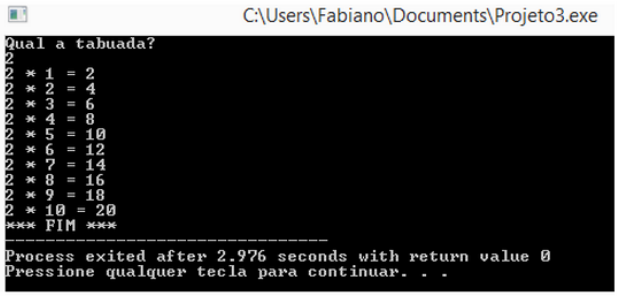


Figura 4.6 – Execução do programa da tabuada.

4.3.3 Repetição controlada pelo resultado de uma operação

Um problema que atualmente é fácil de ser resolvido porque a tecnologia permite ter processadores rápidos e memórias minúsculas é o cálculo da raiz quadrada de um número.

Há alguns anos este era de fato um problema pois existem algumas formas diferentes de se obter o resultado. Entre os métodos, o mais usado é o de aproximações sucessivas de Newton-Raphson. Basicamente ele diz o seguinte:

- A primeira aproximação para a raiz quadrada de um número y é $x_1 = \frac{y}{2}$.
- As demais aproximações são calculadas por uma fórmula de recorrência $x_{i+1} = x_i - \frac{x_i^2 - y}{2x_i}$.

Em uma situação real, estas fórmulas e métodos seriam passados a você, não se preocupe com a matemática deste exemplo.

Porém observe que o "i" que existe na fórmula de recorrência trata-se de uma repetição e é aí que você entra!

Para o cálculo da raiz quadrada poderíamos usar um número razoável de repetições como 20 ou 30. Mas para ele ficar mais aproximado, temos que usar uma variável auxiliar.

Veja o programa a seguir:

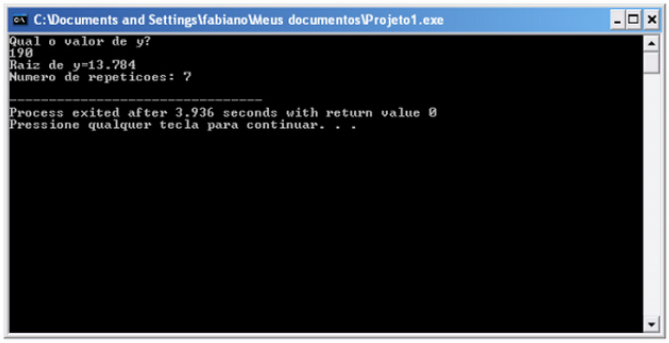
```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int i;
6   float y,x,aux;
7
8   i=0;
9   cout<<"Qual o valor de y? "<<endl;
10  cin>>y;
11
12  if (y==0){
13  cout<<"Raiz de y=0"<<endl;
14  }
```

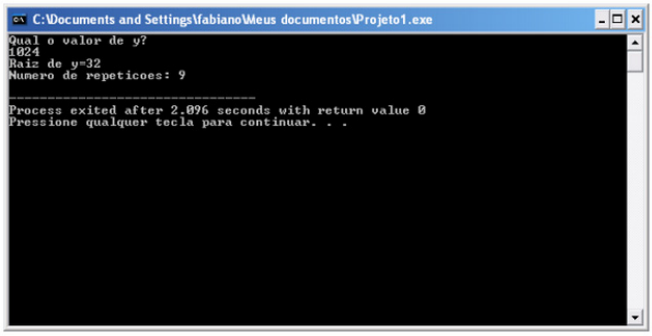
```
15  else {
16  if (y<0){
17  cout<<"Nao e' possivel fazer o calculo!"<<endl;
18  }
19  else {
20  x=y/2;
21  aux=0;
22  while (x!=aux){
23  aux=x;
24  x=x-(x*x-y)/(2*x);
25  i++;
26  }
27  cout<<"Raiz de y="<<x<<endl;
28  cout<<"Numero de repeticoes: "<<i<<endl;
29  }
30  }
31  return 0;
32  }
```

Colocamos algumas restrições: o valor de y não pode ser 0 ou negativo. Porém se for positivo, ele entra em um laço presente na linha 22 e vai até a 26.

Veja que quem vai controlar a quantidade de repetições é a variável aux, que é atualizada dentro do laço. Este é um exemplo, dentre vários outros, que mostram que o *loop* pode ser controlado por uma expressão matemática.

Veja o resultado da execução do programa:





4.3.4 Repetição controlada pelo valor da entrada de dados

Você conhece a série de Fibonacci? Não, não é uma série de TV! Ela ficou muito famosa e popular no livro e filme "O código Da Vinci". Na matemática ela é representada por uma sequência de números inteiros começando por 0 ou 1 na qual cada próximo termo da série equivale a soma dos dois anteriores. Os números de Fibonacci compõem a sequência:

1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,etc..

Para que ela serve?

- Análise de mercados financeiros,
- Ciência da computação
- Teoria dos jogos
- Funções bioestatísticas
- Afinação de instrumentos musicais

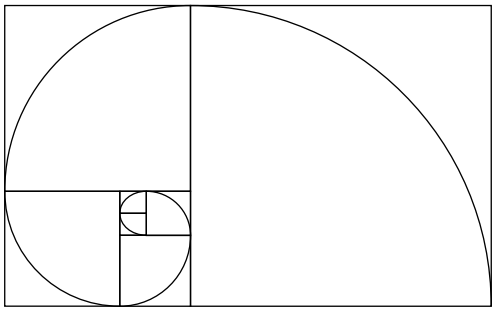


Figura 4.7 – Aplicação da sequência de Fibonacci na arquitetura. Esta figura também é conhecida como "assinatura de Deus" pois as conchas dos Nautilus seguem a proporção dos números da sequência de Fibonacci.

Chega de matemática!

Vamos supor que você precise gerar a sequência para alguém a qual irá te passar quantos termos da sequência serão necessário.

Por exemplo, se o usuário pedir 5 termos, a sequência será: 1,1,2,3,5. Se pedir 8 termos: 1,1,2,3,5,8,13,21. E assim por diante.

O programa para este problema está representado a seguir:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   int a,b,auxiliar,i,n;
6
7   a=0;
8   b=1;
9
10  cout<<"Quantos termos da sequencia?"<<endl;
11  cin>>n;
12  cout<<"----------"<<endl;
13  cout<<"Serie de Fibonacci com "<<n<<" termos:";
14
15  for (i=0; i<n; i++){
16  auxiliar = a+b;
17  a=b;
18  b=auxiliar;
19  cout<<auxiliar<<" ";
20  }
21  return 0;
22  }
```

O laço de repetição está entre as linhas 15 e 20.

Observe que o que determinou quantas vezes o laço será repetido é uma variável que recebeu um valor o qual o usuário escolheu: a variável n.

Este é apenas um dos vários exemplos nos quais o usuário determina a quantidade de repetições de acordo com a entrada que foi feita. A saída do programa para 9 repetições está mostrada a seguir:

Figura 4.8 – Série de Fibonacci.

4.3.5 Repetição controlada pela resposta do usuário

Todos os nossos programas até agora executavam uma vez e terminavam não é? Não é mais interessante que fosse feita uma pergunta ao usuário solicitando se ele quer continuar ou não? É claro que sim, isso é natural de qualquer programa.

Mas como fazer isso? Com o conceito de repetição e refletindo um pouco você terá uma resposta.

Vamos tomar o exemplo da tabuada. No final da execução o programa poderia perguntar: "Deseja continuar (s/n)?". Se o usuário digitar "s" o programa repete e "n", o programa termina.

Basicamente o que temos que fazer é:

1. Criar uma variável para ser usada como resposta
2. Enquanto a resposta for "s", executar as linhas de 8 a 15 da Listagem 1.
3. Quando a resposta for diferente de "s", terminar o programa.

Vamos ao código. Começamos como o programa da Listagem 1:

```
#include <iostream>
1    using namespace std;
2
3    int main(int argc, char** argv) {
4    int numero;
5    int contador=1;
6    int resultado;
```

Agora começa a parte nova. Lembre-se que queremos repetir a execução do programa enquanto a resposta do usuário sobre a repetição for 's' ("sim"). Então temos que ter outra variável, a qual vamos chamar resposta e iremos usá-la como controle da repetição de execução do programa. Vamos assumir

que ela inicialmente já terá o valor 's' (lembre-se que quando usamos char, usamos aspas simples).

```
7   char resposta ='s';
```

Lembrando que podemos escrever a linha 7 em duas linhas: uma para a declaração, outra para a inicialização do valor.

A parte do cálculo da tabuada já foi feito no Exemplo1. Esta parte não muda. O que temos que fazer agora é inserir um *looping* externo a este para controlar a execução do programa. Teremos aqui então um *loop* aninhado, assim como tivemos com os if's, lembra?

O *looping* que controlará a repetição ficará da seguinte forma:

```
9   while (resposta=='s'){
10  contador = 1;
11  //Aqui entra o programa do Exemplo 1
19
20  cout<<"Deseja continuar (s/n)?"<<endl;
21  cin>>resposta;
22  }
```

Basicamente a forma do while é bem parecida com o while do Exemplo 1. Porém neste caso, não temos um contador para controlar este *looping*, temos aqui uma variável que receberá um valor a cada iteração do *looping*. Esta variável será a resposta que será lida na linha 21. Lembre-se que para a primeira execução do *looping*, a variável já recebeu o valor inicial para poder entrar no looping da linha 9. Daqui a pouco comentaremos sobre a variável contador presente na linha 10.

O código final do programa está mostrado na Listagem 2.

#include <iostream>

```
1   using namespace std;
2
3   int main(int argc, char** argv) {
4   int numero;
5   int contador;
6   int resultado;
7   char resposta ='s';
8
```

```
9   while (resposta=='s'){
10  contador = 1;
11  cout<<"Qual a tabuada? "<<endl;
12  cin>>numero;
13
14  while (contador<=10){
15  resultado = numero*contador;
16              cout<<numero<<" * "<<contador<<" = "<<resultado;
17              contador = contador+1;
18          }
19
20          cout<<"Deseja continuar (s/n)?"<<endl;
21          cin>>resposta;
22      }
23      cout<<"*** FIM ***";
24      return 0;
25  }
```

Listagem 2 – Repetição controlada pelo usuário

Observe que agora temos um *looping* (entre as linhas 14 e 18) dentro de outro (entre as linhas 9 e 22). Vamos estudar esta situação com um pouco mais de detalhe no tópico 4.3.7.

Preste muita atenção agora: vamos supor que seja a primeira execução do programa. O código final da Listagem 2 é um pouquinho diferente do programa da Listagem 1, verifique!

Sendo assim, qual é o valor da variável contador agora? Não dá para saber, certo? Declaramos a variável na linha 5 e tiramos a inicialização na declaração.

O programa inicia com a variável resposta valendo 's' e entra no *looping* da linha 9. Na linha 10 temos a variável contador sendo inicializada. Novamente vamos deixar pra lá por enquanto, apenas guarde que ela contém o valor "1" nesta linha.

O programa prossegue nas linhas 11 e 12 perguntando ao usuário qual será a tabuada a ser mostrada.

As linhas 14 a 18 já foram estudadas durante o Exemplo 1, certo? Porém agora a pergunta é: "qual o valor da variável contador ao final do *looping* da tabuada? Ou melhor, qual o valor da variável contador quando o programa chegar na linha 19? Se você respondeu 11, acertou.

O programa prossegue nas linhas 19 e 20 perguntando ao usuário se ele quer continuar a execução do programa. Se o usuário responder 's', a variável resposta receberá este valor, o *looping* chegará na linha 22 e voltará para a linha 9 onde a variável resposta será avaliada novamente. Como ela terá o valor 's', o *looping* voltará a ser iniciado, o usuário informará a tabuada e etc.

Agora vamos responder o motivo do contador receber o valor "1" na linha 10. Como acabamos de citar, se nada for feito, o valor de contador será 11 e assim numa próxima execução do *looping*, nunca entrará na parte da tabuada (porque a condição do *looping* da tabuada é (contador <= 10). Se ele valer 11, a condição nunca será verdadeira). Por isso, tiramos a inicialização da declaração da variável como está na Listagem 1 e colocamos na linha 9 desta forma, corrigindo o código para funcionar do jeito que planejamos.

Entendendo as repetições aninhadas

Você sabia que a palavra tabuada tem relação com tábua? Sim, isso mesmo, a tábua de madeira que conhecemos. Esta palavra é usada porque o matemático Pitágoras usou uma tabela escrita em uma tábua para poder explicar a multiplicação.

A tabuada de Pitágoras tinha aproximadamente as seguintes informações:

*	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

E agora, como desenvolver essa tábua em C++? Já vimos anteriormente como repetir números de 1 a 10, ou 1 ao 9, mas como fazer isso para aplicar em duas dimensões? A resposta é: temos que fazer uma repetição dentro da outra, ou trabalhar com uma repetição aninhada. Veja o seguinte programa:

```
1   #include <iostream>
2   using namespace std;
3
4   int main(int argc, char** argv) {
5   cout<<"Tabuada de Multiplicacao"<<endl;
6   cout<<"\t1\t2\t3\t4\t5\t6\t7\t8\t9"<<endl;
```

```
7   cout<<"\t-\t-\t-\t-\t-\t-\t-\t-"<<endl;
8   for (int linha=1; linha<=9; linha++){
9   cout<<linha<<"\t";
10  for(int coluna=1; coluna<=9; coluna++){
11  cout<<linha*coluna<<"\t";
12  }
13  cout<<"\n";
14  }
15  return 0;
16  }
```

Listagem 3

Veja que temos um "for dentro do outro" nas linhas 8 e 10. Outro detalhe: lembra dos caracteres de escape, principalmente o de tabulação "\t" e o nova linha "\n"? Estamos usando eles neste exemplo para poder formatar a saída de dados.

O for da linha 8 se encarrega de representar as linhas da tabela de tabuada e o for da linha 10 se encarrega das colunas.

Desta forma, podemos aninhar os dois laços e o resultado não fica muito bonito, mas representa corretamente a tabela de tabuada do Pitágoras:

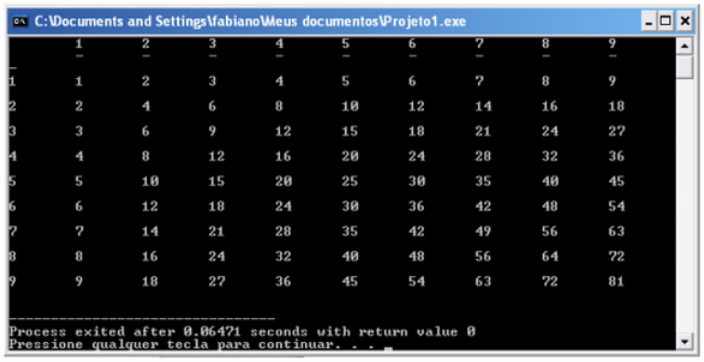


Figura 4.9 – Tábua de Pitágoras.

Exemplo 3

Há alguns anos, um famoso apresentador da TV tinha um programa chamado "Domingo no Parque". Neste programa, o apresentador chamava as crianças para um desafio no qual elas tinham que contar de 1 a 40, mas nos números

que faziam parte da tabuada do 4, ao invés de falar o número, a criança falava "PIM" e continuava assim, até 40. Quem chegava até o 40 ganhava um prêmio.

Portanto, para ganhar o prêmio a criança teria que falar assim: "1, 2, 3, PIM, 5, 6, 7, PIM, 9, 10, 11, PIM, 13, 14, 15, PIM, 17, 18, 29, PIM, 21, 22, 23, PIM, 25, 26, 27, PIM, 29, 30, 31, PIM, 33, 34, 35, PIM, 37, 38, 39, PIM". Tente fazer isso, é bem fácil de confundir.

Como você faria um programa para fazer a mesma coisa em C++?

Veja que é uma repetição controlada pois vai de 1 a 40. Além disso, durante a contagem de 1 a 40, se o número for da tabuada do 4, ele não será impresso, será impresso "PIM". Mas, como saber se um número é da tabuada do 4? Perceba que muitas vezes o problema não é o algoritmo em sim, e sim saber outros assuntos de uma maneira bem detalhada para poder resolver o problema.

Voltando ao problema, um número é de uma tabuada qualquer quando ele for divisível pelo número da tabuada. Ou seja, neste caso, divisível por 4. E como saber se o número é divisível? Simples! Quando o resto da divisão, no caso, por 4 for zero. Lembre-se que temos um operador em C que faz este trabalho para você. Está vendo como uma coisa puxa a outra?

Vamos lá!

Neste programa também vamos dar a possibilidade para o usuário rodar o programa mais uma vez, sendo assim, o do-while pergunta se ele quer continuar será utilizado.

A parte principal é o *looping* que vai de 1 até 40, com incremento de 1 e a cada repetição o número, na verdade é o próprio contador, é impresso na tela. Vamos usar um for isso pois sabemos o número de repetições.

A cada repetição, antes do número ser impresso na tela, verificamos se o contador é divisível por 4, se for, imprimimos "PIM" no lugar do contador. Veja a Listagem 4 e as linhas de 10 a 15.

```
#include <iostream>
1   using namespace std;
2
3   int main(int argc, char** argv){
4       int numero;
5       int contador;
6       int resultado;
7       char resposta ='s';
8
```

```
9       do {
10          for (contador=1; contador<=40; contador++){
11          if (contador%4==0)
12           cout<<"PIM ! "<<endl;
13           else
14               cout<<contador<<endl;
15          }
16
17          cout<<"Deseja continuar (s/n)?"<<endl;
18          cin>>resposta;
19      } while (resposta=='s');
20      return 0;
21  }
```

Listagem 4 – Programa do "PIM"

4.4 Comparação entre as estruturas de repetição

Existem alguns “postulados” a respeito das estruturas de repetição:

• Toda estrutura feita com while pode ser convertida para do-while e vice-versa

• Toda estrutura for pode ser convertida em while, mas nem toda estrutura while pode ser convertida em for

A tabela 4.1 faz uma comparação entre as estruturas de repetição de acordo com suas características.

Estrutura	Condição	Quantidade de execuções	Condição de existência
while	Início	Indefinido	Condição verdadeira
do-while	Fim	Mínimo 1	Condição falsa
for	Não tem	De acordo com o programador	Valor inicial < Valor final

Tabela 4.1 – Comparação entre estruturas de repetição.

Sabe quem está explicando sobre estruturas de repetição neste vídeo? Mark Zuckenberg, o criador do Facebook. Assista o vídeo e relaxe um pouco:
https://www.youtube.com/watch?v=BlXtMr7ge9Q

4.5 Depuração de programas

Muitas vezes é necessário executar o programa passo a passo para verificar erros e corrigir possíveis falhas durante o desenvolvimento. Os exemplos que temos estudado são simples e curtos, além disso, assim como outras IDE's, o DevC++ aponta o número da linha com o erro que ocorreu. Isto já ajuda mas não é suficiente.

Em muitas situações será necessário pausar o programa, verificar o valor de uma determinada variável, entre outras necessidades. Isto é chamado de depuração ou debug. Várias IDE's oferecem esse recurso e deve ser muito bem explorado pelos programadores.

Quando os computadores ainda ocupavam grandes espaços físicos e funcionavam com válvulas, a execução de um programa e sua depuração eram atividades extremamente complicadas. Certa vez, um determinado programa não estava rodando e os programadores não descobriam o motivo. Foram verificar e encontraram um inseto (bug, em inglês) dentro dos circuitos e este estava impedindo o computador funcionar. Daí surgiu o termo bug de computador, usado quando existe algum tipo de erro e consequentemente o debug, que é o processo de depuração.

No caso do DevC++ o depurador é muito simples de ser usado.

A tela inicial do DevC++ é mostrada na figura 4.10, observe a flecha que indica onde o depurador é acionado. Vamos estudar o depurador com o programa da Listagem 5.

A figura 4.11 mostra os detalhes da tela do Depurador após o acionamento do botão “Depurador” na tela inicial.

Para o Depurador ser executado de fato, o programa precisa ser recompilado para que códigos de depuração sejam anexados ao programa a ser gerado. Note que na tela da figura 4.11 há um botão chamado “Depurar”. Quando o usuário clica neste botão, se for a primeira vez que estiver sendo usado no projeto, vai aparecer um diálogo dizendo que o projeto não possui códigos de depuração e pergunta ao usuário se ele deseja depurar o programa. Confirmando esta opção, o programa será compilado novamente com os novos códigos e executado.

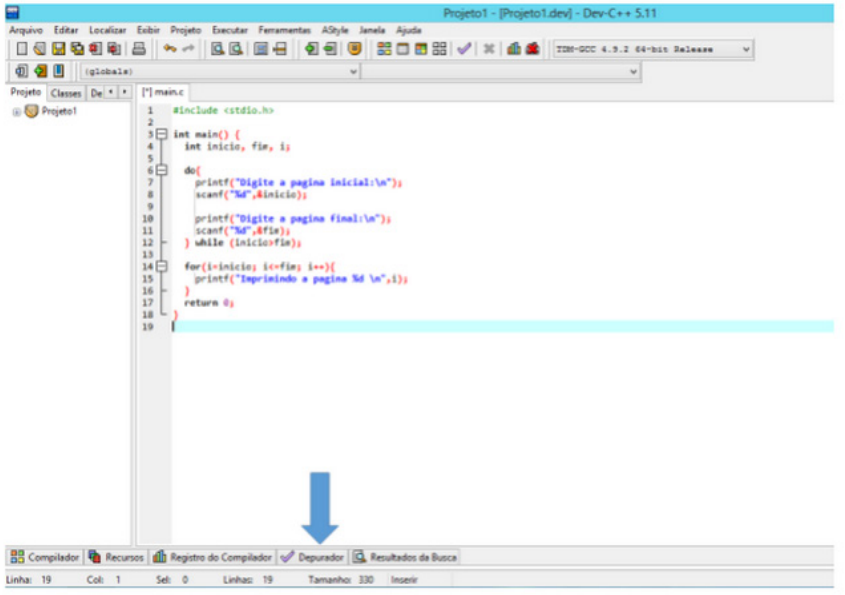


Figura 4.10 – Tela inicial do DevC++.

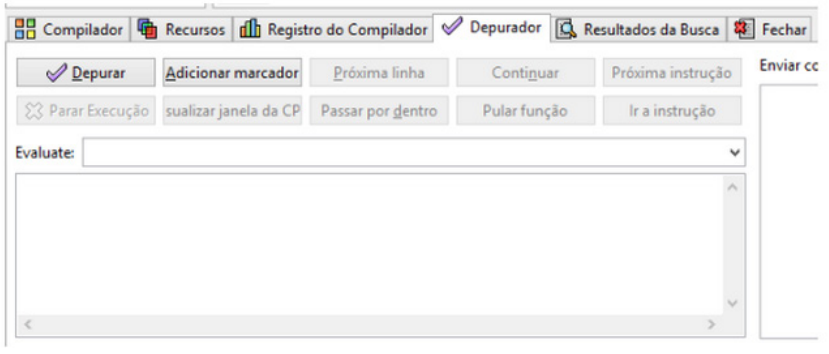


Figura 4.11 – Tela do depurador.

Observe na tela da figura 4.11 que existe um botão chamado “Adicionar marcador”. Os outros botões ainda estão desabilitados porque apesar do programa estar compilado agora com os códigos de depuração, ainda precisamos habilitar algumas funções de depuração.

No programa da Listagem 5 existem 3 variáveis declaradas: inicio, fim e i. A depuração é muito útil para verificar a execução de *loopings* passo a passo. Vamos criar 3 marcadores, um para cada variável.Veja o resultado da adição desses marcadores na figura 4.12.

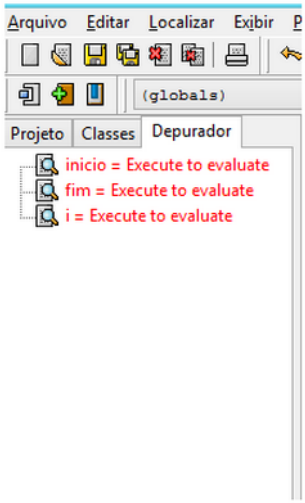


Figura 4.12 – Aba dos marcadores adicionados ao projeto.

Agora temos condições de executar o programa passo a passo e a cada passo poderemos verificar o estado de cada uma dessas variáveis. Em projetos maiores, o uso dos marcadores e da depuração serão extremamente importantes para o bom desenvolvimento do *software*.

Falta um pequeno detalhe para iniciarmos a depuração: no programa digitado, o programador precisa selecionar a linha na qual deseja iniciar a execução passo a passo e marcá-la como *breakpoint*. Como o nome sugere, estamos inserindo um ponto de parada. Portanto, o depurador vai executar o programa e ao encontrar o *breakpoint*, vai parar e esperar o usuário tomar alguma ação. Quando marcamos uma linha como *breakpoint*, a linha toda fica destacada em vermelho.

Neste momento estamos prontos para acionar o botão “Depurar” e verificar o funcionamento do programa passo a passo. Veja como fica a tela do programa na figura 4.13. Preste atenção na aba do Depurador onde encontramos as variáveis inicio, com valor 1, fim, também com valor 0 e i com valor 0. Acabamos de executar o programa pela primeira vez, então é natural esperar estes valores na primeira execução. Observe também que os outros botões que estavam desabilitados, agora estão habilitados.

Na tela do programa, a linha a ser executada está marcada em azul com uma pequena flecha indicando que ao acionarmos o botão “Próxima linha”, ela é que será executada.

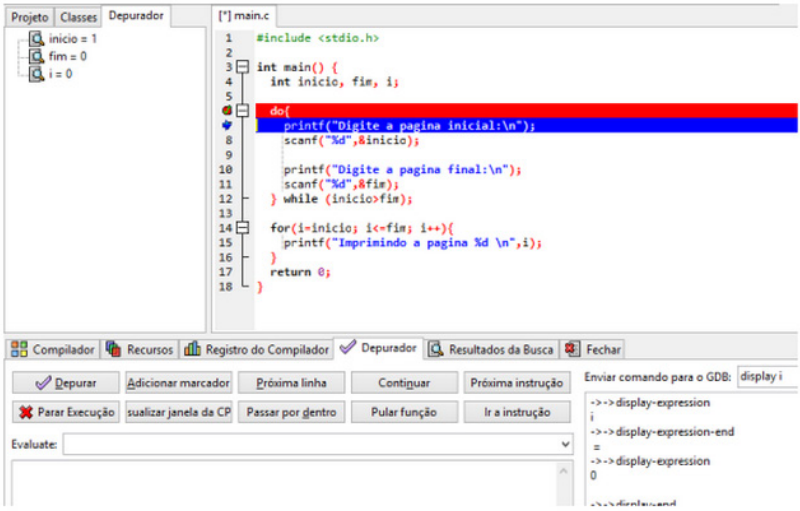


Figura 4.13 – Depurando o programa

A depuração consiste em executar passo a passo e verificar o estado e conteúdo das variáveis. Em algumas IDE’s o depurador oferece ferramentas mais detalhes as quais permitem verificar o estado da memória e registradores. No caso do DevC++ é possível acessar uma tela para ver o estado da CPU durante a depuração conforme mostra a figura 4.14.

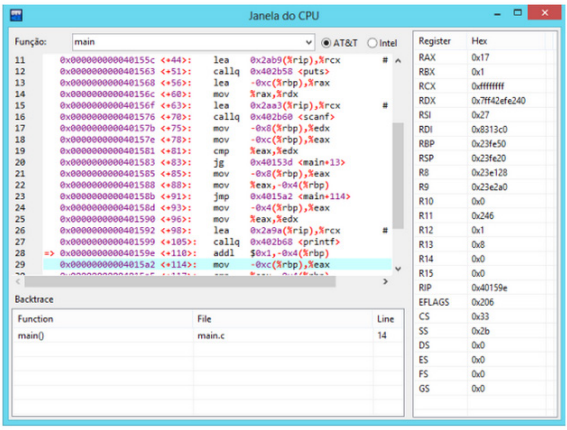


Figura 4.14 – Visualização da CPU

É interessante usar o botão "Próxima linha". Desta forma, o programa será executado passo a passo. Veja como ficou o estado das variáveis na figura 4.15 após inserir o valor 40 para a variável início.

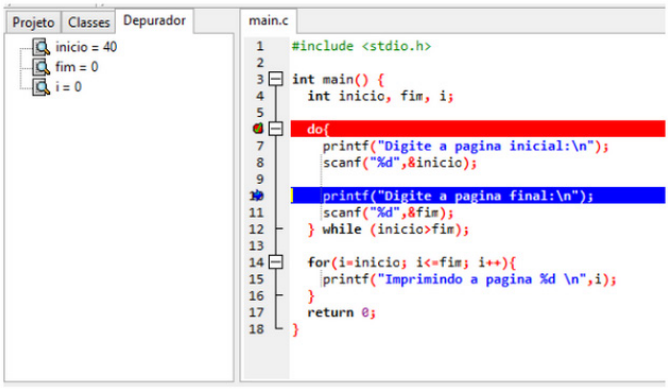


Figura 4.15 – Depurando o programa

Após algumas execuções passo a passo e depois de entrar no *looping* do for, temos a seguinte configuração das variáveis, mostrada na figura 4.15. Observe o valor da variável i e como ela varia conforme o *looping* é executado.

Enquanto isso, na tela de execução do programa, as mensagens de impressão na tela continuam normalmente. Veja a figura 4.16.

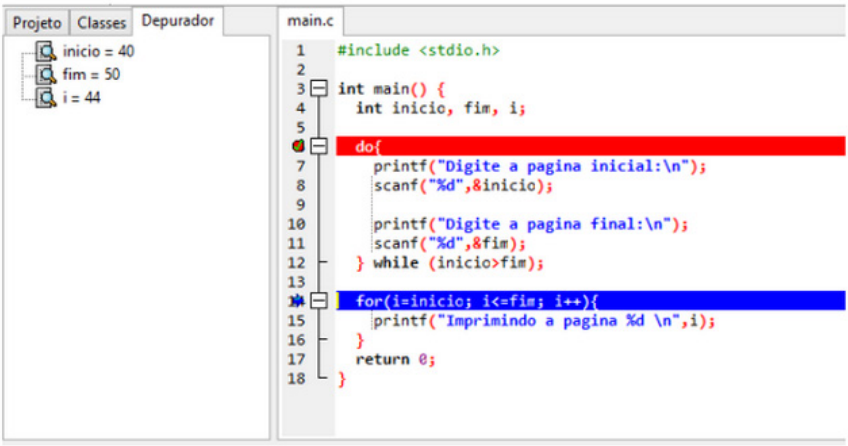


Figura 4.16 – Depurando o programa.

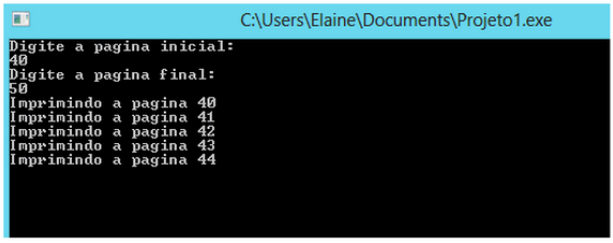


Figura 4.17 – Tela de execução do programa enquanto está em depuração.

E assim o programa prossegue até o final. A depuração não deve ser feita rapidamente. Como é um trabalho de investigação para saber onde que o erro está ocorrendo, é um trabalho que deve ser feito com cuidado e detalhadamente.

O grande segredo é escolher os melhores marcadores para avaliar as variáveis e colocar os *breakpoints* em locais estratégicos do código. É possível inserir vários *breakpoints* no programa para melhorar a depuração.

No caso de o erro já ter sido descoberto, ou pelo menos encontrado um ponto suspeito, é possível clicar no botão "Continuar" para terminar a depuração.

Os outros botões encontrados no Depurador também são encontrados em outras IDE's com pequenas variações no nome.

• Próxima linha: executa a próxima linha em relação ao cursor. O cursor de depuração é caracterizado por uma seleção azul na linha atual e uma pequena seta ao lado da linha.

• Próxima instrução:

• Parar execução: para a execução do depurador

• Visualizar janela da CPU: este botão abre uma tela contendo o estado atual da CPU. Mostra a alocação do programa na memória e o estado de alguns registradores

• Passar por dentro:

• Pular função:

• Ir a instrução:

• Continuar: este botão faz com que o depurador avance até o próximo *breakpoint*.

REFLEXÃO

Neste capítulo vimos a última estrutura de controle de fluxo para a programação. Com estes elementos já é possível programar em muitas linguagens de programação. Mas ainda existem alguns elementos que vamos estudar no próximo capítulo.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos de programação de computadores**. São Paulo: Pearson Education, 2008.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. **Algoritmos**. São Paulo: McGraw Hill, 2009.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Campus, 2008.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. D. **Algoritmos**: lógica para desenvolvimento de programação. 9ª. ed. São Paulo: Érica, 1996.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Education, 2003.

5

Módulos e Vetores

Até agora vimos as situações mais comuns que serão encontradas na programação básica de computadores. A linguagem C é muito poderosa e nosso objetivo geral da disciplina é introduzi-lo na programação desta grande linguagem.

Porém, se pensarmos um pouco, ainda faltam alguns recursos que poderiam facilitar ainda mais a programação. Por exemplo, quando tratamos uma variável, tratamos de uma maneira unitária e falta as vezes a possibilidade de tratar um conjunto de variáveis do mesmo tipo. Ou seja, será que não seria possível usar um conjunto de números inteiros para guardar os meses do ano, por exemplo? Isso facilitaria muito em algumas ocasiões como no cálculo de uma folha de pagamento, etc.

Nossos programas e exemplos possuem efeitos didáticos, mas na vida real os programas são bem maiores. Quando tratamos sobre o que é um projeto no capítulo 2 citamos que um programa de grande porte é composto de módulos e outras partes, além dos programas fonte. Porém, precisamos aprender como dividir os programas em partes, assim, cada parte pode ser reaproveitada em outros programas.

Uma vez que você já sabe programar com as estruturas de controle, neste capítulo vamos introduzir dois assuntos importantes: os vetores e os módulos, muito usados na linguagem C e outras linguagens também.
Vamos lá?

OBJETIVOS

Ao final deste capítulo você estará apto a:

- Desenvolver programas utilizando vetores
- Implementar as operações básica em vetores
- Elaborar programas complexos utilizando funções.

5.1 Introdução aos vetores

Figura 5.1 – Um arquivo

Você já deve ter visto uma gaveta de um arquivo de aço como está mostrado na figura 5.1. Vamos tentar descreve-la?

Trata-se de uma gaveta, contendo várias fichas (de um cliente, aluno, funcionário, contribuinte, etc) numeradas de acordo com um índice. Embora seja possível, a gaveta contém fichas do mesmo tipo: pessoas. Se aparecer uma ficha de um imóvel, um carro, ou outra coisa, este elemento está errado e não pertence a este conjunto. Resumindo, temos elementos (fichas) do mesmo tipo, ordenadas ou não, porém indexadas. Basicamente isso é o mesmo que um vetor!

Um vetor, ou array, ou variável composta unidimensional é uma estrutura de dados contendo um conjunto de dados, todos do mesmo tipo, indexados por um valor. Graficamente podemos representa-lo assim:

	0	1	2	3	4	5	6	7	8
x →	'a'	'l'	'g'	'o'	'r'	'i'	't'	'm'	'o'

Figura 5.2 – Um típico vetor

A figura 5.2 mostra um típico vetor: todos os elementos são do mesmo tipo (char), é indexado por números (começando do 0 e vai até o 7) e possui um nome (x).

Em C, para se declarar um vetor usamos a seguinte sintaxe:

```
tipo nome_vetor[tamanho]
```

Então, o vetor da figura 5.2 é declarado assim:

```
char v[8];
```

Preste atenção! Anteriormente usamos a mesma sintaxe para criar um *string*! Agora faz sentido! Quer dizer que uma *string* é um vetor de caracteres? Na linguagem C é. Em outras linguagens isto é diferente, mas em C e outras linguagens como C++, Java e outras, uma string é um vetor de caracteres!

Portanto, como vimos no box, a *string* "linguagem C" é um vetor de caracteres com 11 posições (não esqueça do espaço em branco, pois o espaço também é um caractere).

Outro detalhe que você tem que tomar cuidado é que na linguagem C, todo vetor começa com um índice 0 e se você declarar um vetor assim:

```
int x[10];
```

teremos um vetor de inteiros com 10 posições, com índices que variam de 0 a 9.

5.1.1 Manipulação

É possível fazer várias operações com vetores e estas operações são chamadas de manipulações.

Da mesma forma que criamos e declaramos outros tipos de variáveis como `int`, `double`, `float`, etc, criamos e declaramos um vetor. Portanto, eles têm um identificador (um nome).

A posição de um elemento dentro de um vetor é feita por meio de uma constante numérica chamada índice. Lembre-se da figura do fichário no início desta seção, é exatamente a mesma ideia e concepção.

Então em um vetor abc de inteiros com 10 posições, para que o valor 10 seja atribuído na posição 5, faremos o seguinte:

abc[4] = 10;//lembre-se que o vetor começa no índice 0, por isso a posição 3 está no índice 2

O vetor ficaria assim:

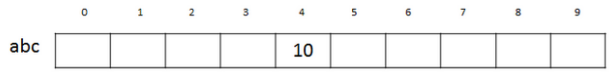


Figura 55

Tome cuidado! O vetor abc tem 10 posições e os índices vão de 0 a 9! Se você tentar colocar um elemento na posição cujo índice é 11 (abc[11]), o compilador gerará um erro cuja mensagem é "Index Out of Bounds" que significa que uma tentativa de acessar um índice inexistente foi feita.

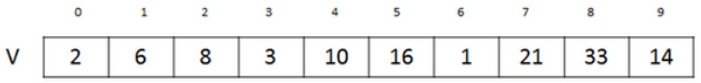
Uma vez que podemos atribuir valores às posições dos vetores, como você faria para preencher um vetor de inteiros com 100 posições e todas as posições com o número 1? Use um loop:

```
1   #include <stdio.h>
2
3   int main() {
4     int vetor[100];
5     int i;
6
7     for (i=0; i<100; i++){
8      vetor[i]=1;
9     }
10    return 0;
11  }
```

Listagem 1 – Atribuição de valores em um vetor

A Listagem 1 mostra um programa simples o qual nos ensina bastante. Na linha 4 temos a declaração e criação do vetor. Nas linhas 7 a 9 criamos um *loop* que percorre o vetor posição por posição e para cada uma, insere o valor 1. Isto é feito usando o índice com o contador do *loop*. Este recurso é muito usado em vários programas.

Também podemos fazer operações aritméticas com os índices e valores, por exemplo:



- v[2] + v[3]: o valor de v[2] é 8, o valor de v[3] é 3, sendo assim v[2]+v[3]=11
- v[2+3] = v[5] = 16
- 2*v[7]: o valor de v[7] é 21, logo 2*v[7] = 2*21 = 42

Vamos fazer um exemplo prático:

Exemplo 1: Aplicação de vetor

O objetivo do programa a seguir é calcular a média aritmética geral de uma classe com 10 alunos. O programa deve imprimir a quantidade de notas acima da média.

Vamos analisar o problema como sempre fazemos antes de começar a programar. Inicialmente temos que prever quais variáveis serão usadas no programa. Como estamos craques em calcular média, já sabemos que vamos usar as variáveis soma e media.

Agora temos que analisar o que vai acontecer com a lógica do problema:

- Temos que criar um *loop* para ler a média de cada aluno. Cada valor será atribuído à uma variável e o total será atribuído na variável soma. Depois disso, teremos o total das notas
- Fazer a média é fácil. É simplesmente calcular a expressão media=soma/10.
- Agora falta a parte nova no problema: saber quantas notas estão acima da média. Para isto teremos que recuperar o valor de cada nota e verificar uma a uma qual delas está acima da média. Mas como vamos recuperar? Lembre-se que estamos aprendendo uma estrutura de dados que pode guardar estes valores para serem usados posteriormente.

Um vetor pode ser usado nesta situação. Ele pode ser declarado com 10 posições para que cada uma guarde a média de cada aluno. Ele deve ser percorrido para fazer a soma das médias inicialmente. Vamos chamar o vetor de notas.

Depois temos que percorrer o vetor posição por posição e verificar se o valor lido da posição é maior que a média. Se for, teremos que criar uma variável para guardar e acumular quantas notas são maiores que a média. Vamos chamar esta variável de notasAcima.

A variável notasAcima está escrita em uma forma de notação chamada CamelCase. Esta é uma forma em inglês para escrever palavras compostas ou frases, onde cada palavra é iniciada com maiúsculas e unidas sem espaços. A maioria das linguagens atuais adota esta prática a fim de padronizar o modo de escrever e nomear variáveis.

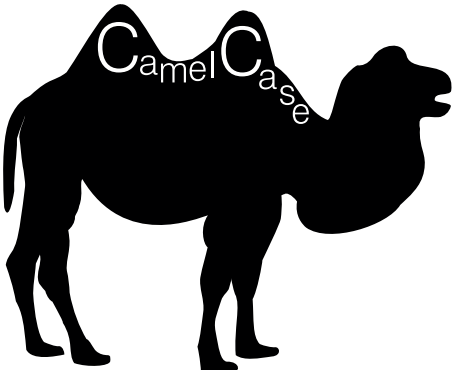


Figura 5.3 – Notação CamelCase (https://pt.wikipedia.org/wiki/CamelCase)

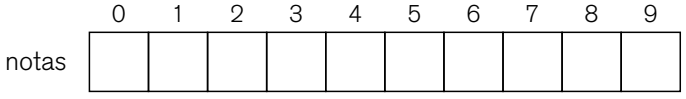


Figura 5.4 – vetor para guardar as médias

A figura 5.4 representa o vetor notas que será usado para a solução deste exemplo. Ele receberá os valores das médias e será percorrido algumas vezes para os outros processamentos necessários no programa.

Vamos começar o código do programa com a declaração de variáveis. A maior diferença em relação aos exemplos que já vimos é a declaração e criação do vetor na linha 8.

```
1   #include <stdio.h>
2
3   int main() {
4     float soma=0;
5     float media;
6     int i;
7     int notasAcima=0;
8     float notas[10];
9
```

Com as variáveis criadas, passamos para a entrada de dados. Como agora vamos usar um vetor para armazenar os valores lidos, vamos usar um *looping* para fazer a entrada de dados. O *looping* consiste em posicionar um "cursor" na primeira posição do vetor, ler o valor, avançar o cursor para a próxima posição, fazer a soma da média e repetir o processo até o fim do vetor. Veja a representação deste processo na figura 5.5.

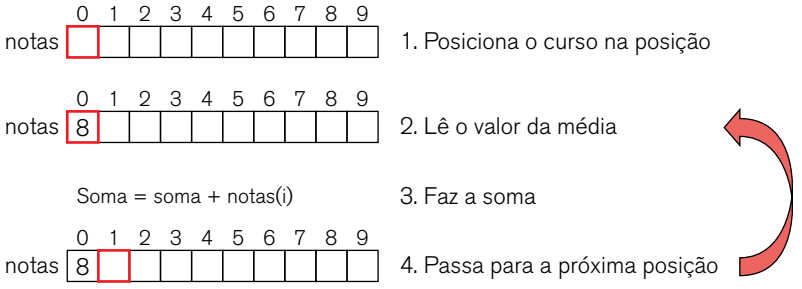


Figura 5.5

Neste caso, como sabemos a quantidade de vezes que o *looping* será repetido, usaremos o for. Após o fim do *looping*, calculamos a média:

```
10    for (i=0;i<10;i++){
11     cout<<Digite a media do aluno <<i+1<<endl);
12     cin>>notas[i];
13     soma = soma + notas[i];
14    }
15
16    media = soma/10;
 7
```

Estamos quase terminando, até agora teremos um vetor carregado com valores e com a média calculada. Como exemplo, podemos ter o vetor da Figura 59 no final da linha 16 e com o valor da variável soma igual a 66 e a media igual a 6,6.

	0	1	2	3	4	5	6	7	8	9
notas	8	7	6,5	9	4	7	4,5	6	8	6

Figura 5.6

Agora para finalizar teremos que percorrer o vetor novamente para verificar quantas notas estão acima do valor da média. A variável notasAcima será usada para fazer a contagem das notas. Veja a figura 5.7.

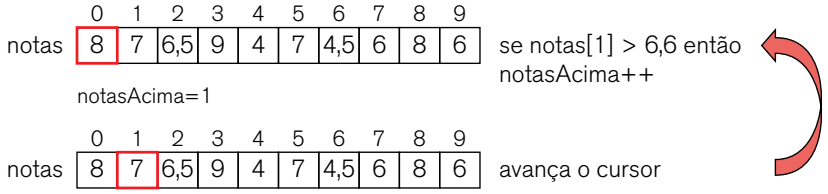


Figura 5.7

```
18    for (i=0; i<10; i++){
19     if (notas[i]>media){
20  notasAcima++;
21    }
22    }
23    cout<<"Existem "<<notasAcima<<" notas acima da media";
24    return 0;
25  }
```

Após a finalização do *looping*, o programa termina com a impressão de uma mensagem na tela. O programa completo está mostrado na Listagem 14 e a execução do programa está mostrada na Figura 61.

```
1   #include <stdio.h>
2
3   int main() {
```

```
4      float soma=0;
5      float media;
6      int i;
7      int notasAcima=0;
8      float notas[10];
9
10     for (i=0;i<10;i++){
11      cout<<"Digite a media do aluno "<<i+1<<endl;
12      cin>>notas[i];
13      soma = soma + notas[i];
14     }
15
16     media = soma/10;
17
18     for (i=0; i<10; i++){
19      if (notas[i]>media){
20      notasAcima++;
21     }
22     }
23     cout<<”Existem”<<notasAcima<<” notas acima da media”;
24     return 0;
25   }
```

Listagem 2

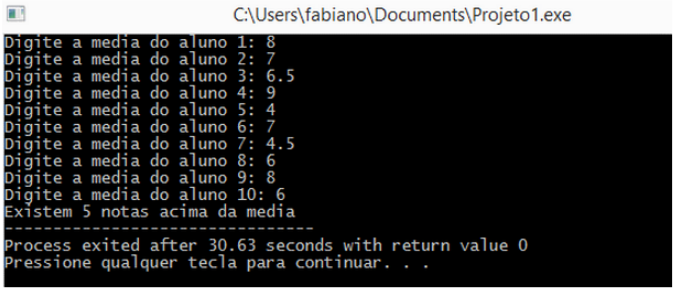


Figura 5.8 – Execução do programa

5.2 Variáveis compostas multidimensionais

Muitas vezes não dá para resolver os problemas com vetores de apenas uma dimensão. Existem várias situações que ficam claras que se tivéssemos um vetor com 2 ou mais dimensões ficaria mais fácil de resolver um problema. Veja a figura 5.9, nossa conhecida.

As fichas formam um conjunto pertencente a uma mesma gaveta em um arquivo de aço. Esta gaveta poderia guardar todas as fichas que começam pelo número "1" por exemplo. As fichas são os vetores que já aprendemos.

Figura 5.9

Porém o arquivo contém várias gavetas. Na figura só aparecem 4, mas poderíamos ter vários arquivos com gavetas numeradas. Portanto, se o arquivo possui gavetas e estas gavetas estão numeradas, temos outro vetor, em outra dimensão, que armazena outros vetores. Esta situação está ilustrada na figura 5.10.

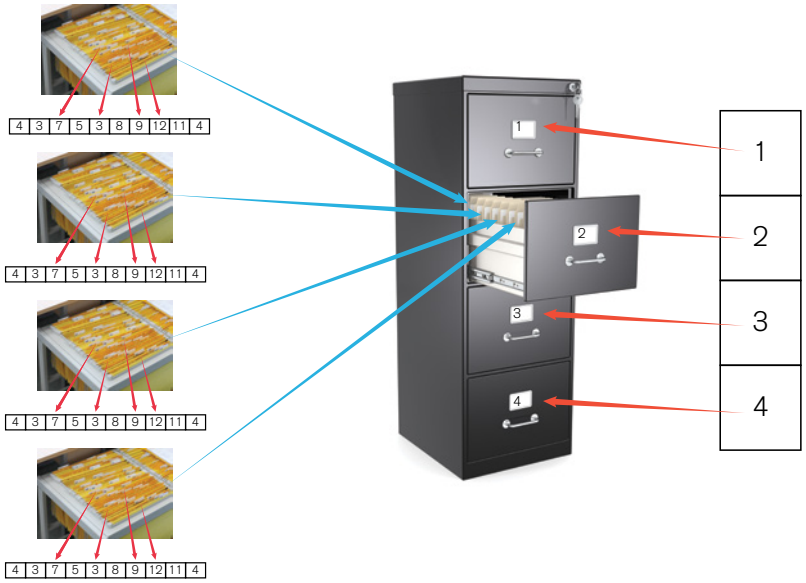


Figura 5.10

Usando um diagrama para ilustrar a Figura 68 teríamos o seguinte:

arquivo	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										

Figura 5.11

Portanto, como podemos perceber, agora teremos 2 índices para poder indexar a estrutura. No exemplo da Figura 5.11 o nome da estrutura, e consequentemente da variável, é arquivo.

Esta estrutura, assim como os vetores possui vários nomes: estrutura composta homogênea multidimensional, matriz, arranjo bidimensional, array bidimensional ou vetor bidimensional. Neste caso temos 2 dimensões mas podemos criar estruturas com mais dimensões de acordo com a necessidade do programa. As matrizes permitem a manipulação de um conjunto de informações de um mesmo tipo primitivo.

Para exemplificar a estrutura com exemplos práticos podemos citar:
Tabelas nas planilhas eletrônicas (como o Microsoft Excel)
O jogo “batalha naval”
O jogo da velha
O teclado do computador, do celular, da calculadora
Jogo de Xadrez, Dama
E outras formas tabulares

Este vídeo é muito interessante e mostra vários exemplos de matrizes. Disponível em: < https://www.youtube.com/watch?v=ks-q6gKoQKs >.

5.2.1 Manipulação

Assim como foi feito com os vetores, ou estruturas compostas homogêneas unidimensionais, é possível manipular os índices e valores das matrizes. Na maioria das linguagens de programação existentes, as notações para matrizes são feitas da seguinte forma:

```
Matriz [dimensão1][dimensão2]...[dimensão n]
```

Desta forma, uma matriz bidimensional poderia ser anotada assim:

```
Matriz[linha][coluna]
```

Veja a figura 5.12. Temos um exemplo de manipulação de índices e valores da mesma forma como foi feito com os vetores unidimensionais.

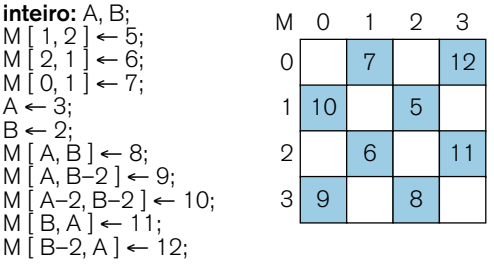


Figura 5.12

Em C, a Figura 70 seria implementada da seguinte forma:

```
int m[4][4];
int a, b;

m[1][2]=5;
m[2][1]=6;
m[0][1]=7;
a=3;
b=2;
m[a][b]=8;
m[a][b-2]=9;
m[a-2][b-2]=10;
m[b][a]=11;
m[b-2][a]=12;
```

É hora de programar um pouco.

Exemplo 2: Loteria esportiva - jogo mais marcado

Espero que este exemplo te ajude a ganhar na loteria!

A Loteca é um tipo de loteria existente no Brasil que é baseada nos resultados de jogos de futebol. A aposta consiste em adivinhar o ganhador (ou empate) de 14 jogos do campeonato. Por exemplo, no jogo entre Palmeiras e Flamengo, o apostador deve adivinhar se é o Palmeiras que será o vencedor, ou o Flamengo, ou irá empatar.

Na figura 5.13 temos um diagrama representando um cartão de aposta da Loteca (da Caixa Econômica Federal) na parte à esquerda. Note que existem 3 quadrados que devem ser pintados para indicar a aposta. Na parte central foi feito um exemplo do que seria um cartão de apostas com os jogos e a correspondência dos espaços para preenchimento. Mais à direita representamos nossa abstração do problema: uma matriz com 14 linhas e 3 colunas onde poderemos

anotar os resultados nesta estrutura de dados. Chamaremos a matriz de loteria.

Ainda, é possível marcar em um mesmo jogo:

- Uma vitória
- Um empate
- Uma vitória E um empate (chamado de duplo)
- As três colunas, chamado de triplo.

Observação: No cartão real da Loteca, existem outras colunas como mostrado na Figura 66 (Colunas D e T). Para simplificar não vamos usá-las.

Jg	Coluna 1	Ept	Coluna 2	Loteria
1	Santos		Corinthians	
2	Flamengo		Fluminense	
3	Palmeiras		São Paulo	
4	Vasco		Botafogo	
5	Portuguesa		XV de Jaú	
6	São Caetano		XV de Piracicaba	
7	Grêmio		Internacional	
8	Havaí		Figueirense	
9	Coritiba		Atlético-PR	
10	Paysandú		Juventude	
11	Atlético-MG		Cruzeiro	
12	Brasiliense		Ponte Preta	
13	Fortaleza		Goiás	
14	Esportivo		Londrina	

Figura 5.13 – Loteca da Caixa Economica Federal.

O objetivo deste exemplo é identificar o jogo mais marcado, ou seja, o programa vai procurar jogo por jogo (linha por linha) e verificar qual é aquele que tem mais marcações (veja a figura 5.14).

	Loteria			
Jogo 1	x			Vitória time 1
Jogo 2		x	x	Duplo: Empate ou Vitória time 2
Jogo 3	x	x	x	Triplo: Vitória time 1 ou Empate ou Vitória time 2
...	...	...	...	
Jogo 14		x		Empate

Mais marcado

Figura 5.14

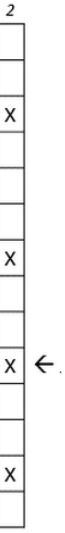
Portanto, da mesma forma que percorremos os vetores, temos que percorrer a matriz para poder procurar o jogo mais marcado. Isto é feito linha a linha, coluna por coluna, ou seja, fixa-se a linha e "varre-se" a coluna para somar as marcações.

Vamos ao código. Iniciamos pela criação das variáveis:

• Uma variável para a matriz, chamada loteria com 14 linhas e 3 colunas

• 2 contadores, chamados i e j para percorrer as linhas e colunas respectivamente

• 3 variáveis para acumular as contagens e tarefas auxiliares maisMarcado (maior número de marcações encontrado), nJogo (número do jogo com mais marcações) e marLin (número de marcações numa linha). Lembre-se do CamelCase.




```
1   #include <stdio.h>
2
3   int main() {
4     char loteria[14][3];
5     int i,
6         j,
7     maisMar=0,
8     nJogo,
9     marLin;
```

Agora temos um problema: como vamos fazer para ler os jogos? Ou seja, para o programa funcionar, temos que ter a matriz preenchida então sendo assim temos uma decisão a tomar: ou pedimos para o usuário informar os dados para preenchermos a matriz pelo programa ou partimos de uma matriz já preenchida.

Vamos optar neste exemplo pelo mais simples: matriz preenchida. E vamos supor que o preenchimento será conforme a figura 5.15 (a seta aponta para o jogo mais marcado). O código à direita mostra o preenchimento da matriz em C.

		0	1	2	
Jogo 1	0	x			
Jogo 2	1		x		
Jogo 3	2		x	*x*	
Jogo 4	3		x		
Jogo 5	4	x			
Jogo 6	5		x		
Jogo 7	6			*x*	
Jogo 8	7		x		
Jogo 9	8	x			
Jogo 10	9	x	x	*x*	
Jogo 11	10	x	x		←
Jogo 12	11		x		
Jogo 13	12			*x*	
Jogo 14	13	x	x		

Figura 5.15 – Exemplo de preenchimento

```
11    loteria[0][0]='x';
12    loteria[1][1]='x';
13    loteria[2][1]='x';
14    loteria[2][2]='x';
15    loteria[3][1]='x';
16    loteria[4][0]='x';
17    loteria[5][1]='x';
18    loteria[6][2]='x';
19    loteria[7][1]='x';
20    loteria[8][0]='x';
21    loteria[9][0]='x';
22    loteria[9][1]='x';
23    loteria[9][2]='x';
24    loteria[10][0]='x';
25    loteria[10][1]='x';
26    loteria[11][1]='x';
27    loteria[12][2]='x';
28    loteria[13][0]='x';
29    loteria[13][1]='x';
```

Variáveis criadas, matriz preenchida, vamos prosseguir com o processamento dos dados. Vamos criar o *looping* externo para percorrer as linhas em primeiro lugar (linha 31 à linha 42) e o *looping* interno (linhas 33 à 41) para percorrer as colunas e acumular a contagem quando o conteúdo da coluna for igual a 'x':

```
31    for (i=0;i<14;i++){
32     marLin = 0;
33     for(j=0;j<3;j++){
34     if (loteria[i][j]=='x'){
35        marLin++;
36  }
37  }
38  if (marLin>maisMar){
39  maisMar = marLin;
40  nJogo=i;
41  }
42    }
```

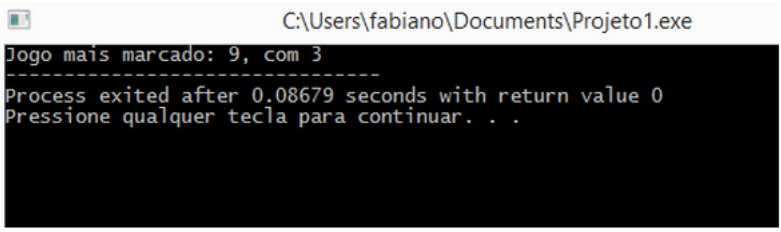
É uma boa ideia usar o depurador do DevC++ para executar o programa passo a passo e verificar como os *loopings* e variáveis se comportam durante o processamento do programa.

O código final do programa está mostrado na Listagem 13. Em seguida mostramos a execução do programa.

```
1    include <stdio.h>
2
3   int main() {
4     char loteria[14][3];
5     int i,j,maisMarcado=0,nJogo,marLin;
6
7     loteria[0][0]='x';
8     loteria[1][1]='x';
9     loteria[2][1]='x';
10    loteria[2][2]='x';
```

```
11    loteria[3][1]='x';
12    loteria[4][0]='x';
13    loteria[5][1]='x';
14    loteria[6][2]='x';
15    loteria[7][1]='x';
16    loteria[8][0]='x';
17    loteria[9][0]='x';
18    loteria[9][1]='x';
19    loteria[9][2]='x';
20    loteria[10][0]='x';
21    loteria[10][1]='x';
22    loteria[11][1]='x';
23    loteria[12][2]='x';
24    loteria[13][0]='x';
25    loteria[13][1]='x';
26
27    for (i=0; i<14; i++){
28      marLin = 70;
29  for(j=0;j<3;j++){
30    if (loteria[i][j]=='x'){
31      marLin++;
32    }
33  }
34  if (marLin>maisMarcado){
35  maisMarcado = marLin;
36  nJogo=i;
37  }
38   }
39    marcado:”<<nJogo<<”,com”<<maisMarcado;
40    return 0;
41  }
```

Listagem 3



Exemplo 3: Loteria esportiva – coluna mais marcada

Neste exemplo vamos usar a loteria esportiva do exemplo anterior como base para poder descobrir qual a coluna mais marcada: se a coluna 1, a coluna dos empates ou a coluna 2.

Desta vez, precisamos descobrir quantas marcações existem em cada coluna e verificar qual dos três valores é o maior. Portanto, é uma inversão do modo de percorrer a matriz em relação ao exemplo anterior: verificamos todas as linhas primeiro e depois as colunas, ou seja, fixamos a coluna e variamos as linhas.

O programa e o modo de resolução é muito parecido com o exemplo anterior com apenas algumas modificações simples. O que muda essencialmente são os algoritmos de repetição para tratar as colunas e linhas.

A figura 5.16 mostra a matriz que iremos considerar neste exemplo. Na verdade, é a mesma matriz do exemplo anterior, porém a figura mostra a coluna que contém o maior número de marcações.

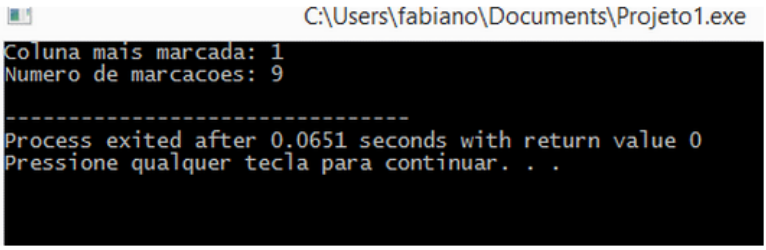
Neste exemplo, iremos substituir as variáveis nlogo e marLin pelas variáveis nColuna (para guardar o número da coluna com mais marcações) e marCol (para guardar o número de marcações numa coluna).

Como já detalhamos o primeiro exemplo de matrizes, vamos apresentar diretamente o código final neste exemplo. Perceba as diferenças e semelhanças entre eles.

Figura 5.16

```
1   #include <stdio.h>
2
3   int main() {
4 char loteria[14][3];
5   int i,j,maisMar=0,nColuna,marCol;
6
7   loteria[0][0]='x';
8   loteria[1][1]='x';
9   loteria[2][1]='x';
10  loteria[2][2]='x';
11  loteria[3][1]='x';
12  loteria[4][0]='x';
13  loteria[5][1]='x';
14  loteria[6][2]='x';
15  loteria[7][1]='x';
16  loteria[8][0]='x';
17  loteria[9][0]='x';
18   loteria[9][1]='x';
19   loteria[9][2]='x';
```

```
20   loteria[10][0]='x';
21   loteria[10][1]='x';
22    loteria[11][1]='x';
23   loteria[12][2]='x';
24   loteria[13][0]='x';
25   loteria[13][1]='x';
26
27   for (j=0; j<3; j++){
28     marCol = 0;
29 for(i=0;i<14;i++){
30   if (loteria[i][j]=='x'){
31      marCol++;
32   }
33 }
34 if (marCol>maisMar){
35    maisMar = marCol;
36   nColuna = j;
37 }
38   }
39   cout<<"Coluna mais marcada: "<<nColuna<<endl;
40   cout<<"Numero de marcacoes: <<maisMar<<endl;
41   return 0;
42  }
```



5.3 módulos: funções

No exemplo da loteria tivemos que escolher entre trabalhar com uma matriz já pré-preenchida ou ler os dados a partir de entradas do usuário. Se a opção fosse por ler os dados do usuário, o programa ficaria maior e mais difícil de ser lido.

Além disso, o programa ficaria mais interessante se pudéssemos imprimir na tela a matriz preenchida, como se fosse um cartão preenchido da vida real, não é?

E se esses processamentos mencionados pudessem ficar em outro arquivo separado ou até mesmo em outra parte do programa? Ficaria mais organizado não é?

Situações como essa ocorrem na maioria dos programas. Conforme os problemas ficam mais complexos e mostram maior variedade de situações, temos que resolver diversos pequenos problemas cujas soluções comporão o conjunto de respostas finais.

Muitas vezes, essa grande quantidade de pequenos problemas pode afetar a legibilidade e clareza do programa, fazendo com que uma consulta ou manutenção futura dessa lógica seja uma atividade difícil de ser feita. Usamos então uma estrutura chamada módulo, que é responsável por evitar essa "confusão da lógica".

Ou seja, modularizar significa quebrar o problema em partes menores as quais serão responsáveis pela realização de uma etapa do problema.

Em C os módulos são chamados de funções. As funções formam uma das mais importantes partes da linguagem C++.

A forma geral de uma função em C/C++ é:

```
Tipo_da_função nome_da_função(lista_de_parâmetros){
  Corpo da função
}
```

• Tipo da função: configura um valor que a função retornará quando terminar. O valor é enviado ao módulo que chamou a função. Caso o tipo da função não seja especificado, o compilador assumirá que o tipo retornado é int.

• Lista de parâmetros: é a relação de variáveis e seus tipos.

Veja o exemplo de uma função bem simples. A figura 5.18 mostra um programa com uma função chamada quadrado sendo usada. A função tem o objetivo de receber um número e elevá-lo ao quadrado. A figura também mostra a tela de saída do programa.

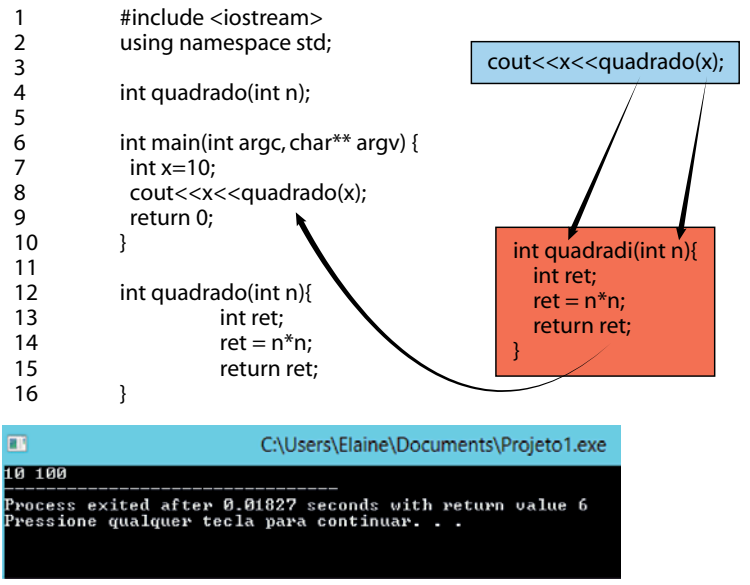


Figura 5.18

Vamos estudar o que está ocorrendo no programa da figura 5.18.

Como sabemos, o objetivo do programa é calcular o quadrado de um número informado pelo próprio programa. O programa principal começa na linha 5 e na linha 6 temos a declaração e a atribuição de um valor para a variável x.

Na linha 8 temos o comando de impressão na tela usual, porém veja que ele vai imprimir o valor x, seguido de um comando chamado quadrado(x). Por mais que seja óbvio que quadrado(x) vá calcular o quadrado de x, quadrado não é um comando conhecido pelo compilador. É uma função que foi definida pelo programador.

A definição da função está entre as linhas 11 e 14. Veja que é um subprograma. A função recebe do programa principal um valor que será copiado para a variável n (veja a figura). A variável n será modificada dentro do subprograma e devolvido para o programa principal na linha 13. A palavra-chave `return` faz com que o valor da variável seja atribuído a quem chamou a função. Porém para

tudo isso funcionar, o protótipo da função precisa ser declarado e isto é feito na linha 3 do programa.

Toda função que for usada no programa tem que ser prototipada inicialmente senão o compilador não irá reconhecê-la ao compilar o resto do programa.

Vamos analisar outros exemplos.

Exemplo 4

Vamos fazer uma função para verificar se um determinado número é primo. Um número é primo quando possui somente dois divisores diferentes: o 1 e ele mesmo. Antes de mais nada, precisamos saber como verificar se um número é primo em C++.

Observe o programa abaixo, suponha que estamos verificando se a variável num é um número primo:

```
23 for(int i = 2; i<=(num/2); i++){
24     if(num%i==0){
25        return false; //não é primo
26      }
27   }
28   return true;   //é primo
```

Este subprograma poderia ser chamado pelo programa principal toda vez que fosse necessário calcular um número primo. Observe que o subprograma fica genérico. Qualquer número que desejarmos verificar se é primo basta substituir o valor do número pela variável num do subprograma acima.

Vamos chamar este subprograma de primo (bem original !), e para ele funcionar é necessário receber do chamador um valor inteiro que chamaremos num. O código ficará assim:

```
22  bool primo(int num){
23    for(int i = 2; i<=(num/2); i++){
24    if(num%i==0){
25        return false;
26      }
27    }
28    return true;
29  }
```

Observe que o resultado desta função é um valor booleano. A função retorna se o valor num é (true) ou não é (false) um número primo.

Vamos inserir esta função em um programa. Veja a linha 13:

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 bool primo(int);
7
8 int main(int argc, char *argv[]){
9  int numero;
10   cout << "Digite um valor inteiro: ";
11   cin >> numero;
12
13    if(primo(numero)){
14      cout << "O numero informado e primo\n" << endl;
15   }
16    else{
17      cout << "O numero informado NAO e primo\n" << endl;
18    }
19   return 0;
20 }
21
22 bool primo(int num){
23   for(int i = 2; i<=(num/2); i++){
24     if(num%i==0){
25       return false;
26     }
27   }
28    return true;
29  }
```

Listagem 4

Observando o programa podemos perceber que existe um caminho de execução o qual é desviado para a função e depois retorna para o programa principal. A figura 5.19 mostra esse fluxo.

Pensando na execução do programa pelo computador, o processador para de executar o programa principal, guarda o estado do programa principal em uma área da memória reservada para isso e chama a função. A função é então executada e depois que termina volta para o programa principal.

Podem existir várias funções em um programa. Os exemplos que estamos usando apresentam apenas uma função por programa, mas é muito comum que em um programa tenha dezenas, centenas de funções.

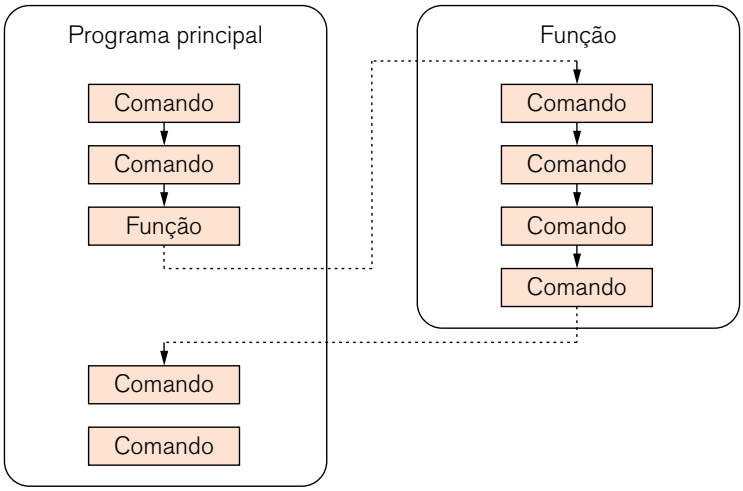


Figura 5.19

5.3.1 Argumentos de funções

Quando uma função usa argumentos, ela deve declarar as variáveis que vão aceitar os valores dos argumentos. Estas variáveis são chamadas de parâmetros formais. Elas são como qualquer outra variável do programa, porém só "existem" dentro da função, o programa que as chamou não tem conhecimento sobre elas. Este conhecimento sobre as variáveis é chamado de escopo de variáveis.

No exemplo do número primo anterior temos um caso de apenas um parâmetro: int num. Uma função pode ter mais de um parâmetro. É importante que os argumentos usados para chamar uma função tenham os mesmos tipos dos seus parâmetros caso contrário, o compilador gerará um erro e o programa não será compilado.

5.3.2 Escopo de variáveis

As variáveis podem ser declaradas de algumas maneiras diferentes: dentro de uma função, fora de uma função, e como parâmetro de uma função. Essas 3 maneiras de declaração definem três tipos de escopo de variáveis: locais, globais ou parâmetros formais.

5.3.2.1 Variáveis Globais

As variáveis globais são visíveis durante toda a execução do programa e podem ser usadas por qualquer função. Elas são declaradas fora de qualquer função, inclusive do main(), e no início de um programa.

5.3.2.2 Variáveis Locais

A variável local só é visível e existente dentro da função ou bloco que a declarou. Outras funções não a reconhecem e só pode ser usada dentro do bloco no qual está declarada. Uma variável local é criada quando a função começa a ser executada e removida no final da execução da função.

5.3.2.3 Parâmetros Formais

Embora já citados anteriormente, os parâmetros formais são variáveis locais em uma função que são inicializadas no momento da chamada desta função e só existem dentro da função onde foram declarados. Mesmo sendo usadas como inicialização da função, elas podem ser usadas como qualquer outra variável local dentro do bloco de função onde estão.

5.3.3 Chamada por valor e chamada por referência

Existem duas formas basicamente para se passar argumentos para uma função. Por meio de uma chamada por valor ou por meio de uma chamada por referência.

A chamada por valor é quando é feita uma cópia do valor de um argumento no parâmetro formal da função. Desta forma, as alterações feitas nos parâmetros dentro do subprograma não terão nenhum efeito nas variáveis usados para chamá-lo.

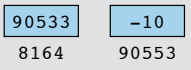
A chamada por referência é diferente. Neste caso, o endereço de memória de um argumento é copiado no parâmetro. Neste caso, todas as alterações que são feitas no parâmetro afetarão a variável que foi usada para chamar o subprograma, pois dentro da função o endereço de memória será usado para acessar o argumento real usado na chamada. A chamada por referência é feita usando um tipo de variável especial chamada ponteiro.

A chamada por valor está exemplificada na Figura 75 (programa do quadrado). Naquele caso, o valor do argumento (10) é copiado no parâmetro n. Quando ocorre a atribuição n=n*n, apenas a variável local n é modificada. A variável x, que foi usada para chamar a função quadrado() permanecerá com o valor 10. Lembre-se que é uma cópia do valor do argumento que é passada para a função. O que ocorre lá dentro da função, fica lá dentro.

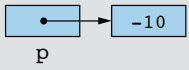
5.3.4 Chamadas por referência

Na passagem de parâmetros, o padrão da linguagem C++ é por valor, mas em algumas situações é preciso fazer uma passagem por referência passando um ponteiro para o argumento.

Lidar com ponteiros e saber usá-los corretamente não é uma tarefa fácil. Não é nosso contexto estudar os ponteiros aqui, porém saiba que ponteiro é um tipo especial de variável que armazena endereços de memória. Um ponteiro pode ter o valor NULL quando não aponta para nada (nenhum endereço) ou outro valor. Quando um ponteiro tem um valor diferente de NULL, então *p é o valor do endereço apontado por p. Por exemplo se x é uma variável e p é igual a &i então *p=i.



Um ponteiro p, armazenado no endereço 8164, contém o endereço de um inteiro



Representação esquemática da situação

```
int main ()   {
    int num = -10;
    int *p;

    p = &num;
    return 0;
}
```

Como fazemos para obter o endereço de memória correspondente ao dado armazenado em uma variável? Usamos o operador *"address-of"* (&).

Observe o programa da Listagem 5. Temos um exemplo de passagem por referência. A função troca() tem como objetivo receber dois parâmetros, x e y, e trocar os seus valores. Até aí tudo bem, o detalhe é que o que for feito dentro da função, alterará os valores das variáveis a e b que foram usadas para chamar a função. Rode este programa no Depurador para você ver como isso ocorre.

Na linha 17, é salvo o valor no endereço x. Na linha 18, o valor de y é colocado em x. E na linha 19 é feita a troca: o x é colocado em y. Na chamada da função, na linha 10, os endereços de a e b são passados como argumentos.

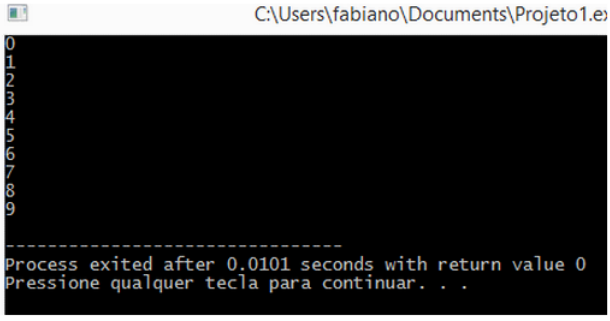
```
1 #include <iostream>
2
3  void troca(int *x, int *y);
4
5  int main(int argc, char** argv) {
6 int a, b;
7
8 a=10;
9 b=20;
10 troca(&a,&b);
11 return 0;
12 }
13
14 void troca(int *x, int *y){
15 int temp;
16
17 temp=*x;
18 *x=*y;
19 *y=temp;
20 }
```

Listagem 5 – Chamada por referência

A passagem por referência também é útil quando uma matriz é usada como argumento para uma função. Neste caso, apenas o endereço da matriz é passada e não uma cópia da matriz inteira.

```
1 #include <iostream>
2 #include<cstdlib>
3
```

```
4 void mostra(int num[10]);
5
6 using namespace std;
7
8 int main(int argc, char** argv) {
9 int a[10], i;
10
11 for(i=0;i<10;i++)
12   a[i]=i;
13 mostra(a);
14 return 0;
15 }
16
17 void mostra(int num[10]){
18 int i;
19
20 for(i=0;i<10;i++)
21   std::cout<<num[i]<<endl;
22 }
```



5.3.5 argc e argv

Você deve ter notado que o programa principal que sempre usamos nos exemplos tem a mesma sintaxe de uma função. E na verdade é uma função! Sendo assim, todo programa em C++ pode receber parâmetros e retornar valor para quem o chamou.

Mas, como fazer isso?

Lembre-se que o processo de compilação completo consiste em transformar um arquivo texto em um arquivo executável. Quando criamos um projeto no DevC++, chamado contagemregressiva.cpp, e a compilação é realizada com sucesso, teremos um arquivo executável chamado contagemregresiva.exe. Sendo assim, poderemos na linha de comando chamar o programa assim:

```
C:\Users\fabiano\Documents>contagemregressiva 20
```

Neste caso, o valor 20 será passado para o parâmetro argv da função main. Se fosse necessário passar outros valores, eles devem ser separados por um espaço (porém isto pode variar de acordo com o compilador ou sistema operacional, mas no Windows é com espaço).

O parâmetro argc contém o número de argumentos da linha de comando e é um inteiro. Sempre será no mínimo 1 porque o próprio nome do programa já é considerado o primeiro argumento.

O parâmetro argv é um ponteiro para uma matriz de ponteiros para caractere. Cada componente desta matriz aponta para argumento da linha de comando, e lembre-se que todos os elementos da linha de comando são *string* e se houver algum número que será usado dentro do main, ele deverá ser convertido para o tipo adequado.

Vamos estudar o programa da Listagem 6. Você vai ver que vamos usar várias funções de bibliotecas (strcmp(), exit(), atoi())da linguagem C++ para nos ajudar com o código. Para saber mais sobre estas e outras funções é preciso estudar a documentação da linguagem.

Você pode encontrar as bibliotecas mais usadas em C++, suas especificações e formas de uso em: http://goo.gl/HQn2fq

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <stdio.h>
4  #include <string.h>
5  using namespace std;
6
7  int main(int argc, char** argv) {
8 int contador;
9  bool mostra;
10
11 if(argc<2){
12   std::cout<<"Voce deve digitar o valor a contar na linha de
comando. Tente de novo"<<endl;
13   exit(1);
14 }
15
16 if (argc==3 && !strcmp(argv[2],"mostra"))
17   mostra=true;
18   else
19  mostra=false;
20
21 for(contador=atoi(argv[1]); contador;--contador){
22   if(mostra)
23     std::cout<<contador<<endl;
24 }
25 std::cout<<"FIM!"<<endl;
26 return 0;
27 }
```

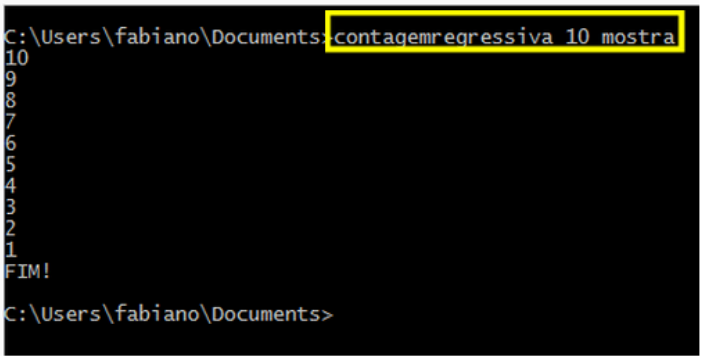
Listagem 6

As linhas de 1 a 5 inserem as bibliotecas mencionadas na compilação do programa. Funções e comandos como strcmp(), atoi() não fazem parte da biblioteca iostream que sempre usamos e para poder usá-los precisamos das bibliotecas auxiliares.

O programa começa na linha 7 e em seguida temos a declaração de variáveis.

Na linha 11 até a linha 14 é feita uma verificação para checar se a contagem de parâmetros na linha de comando é menor que 2. Se for, significa que o usuário cometeu algum erro ao chamar o programa e uma mensagem de alerta é mostrada para ele. A linha 13 usa a função exit() com o argumento 1 para sair do programa e não gerar nenhuma mensagem de erro (cada valor de argumento na função exit() gera um comportamento diferente).

A execução do programa precisa estar na seguinte forma na linha de comando:



se o programa não for chamado desta forma, inclusive com a palavra "mostra", ele não será executado corretamente. Esta checagem é feita nas linhas 16 a 19. O if testa se a quantidade de argumentos é igual a 3 e se o terceiro argumento na chamada é a palavra "mostra". Se for, a variável mostra recebe o valor true, caso contrário, recebe false. Se for false, o programa encerrará nas linhas 25 e 26.

As linhas 21 a 24 se encarregam de fazer o *looping*. Observe o uso da função atoi(). Neste caso ela recebe o valor do segundo argumento da linha de comando (lembre-se que os argumentos são passados para o main em um array de caracteres e sendo assim o primeiro argumento é o nome do programa (índice 0 do vetor) e o próximo, no índice 1, será o segundo argumento) e converte o valor para um número inteiro, que será usado no for e servirá como início da contagem regressiva. Observe na figura 5.20 o que ocorre com outras formas de executar o programa.

Para finalizar, não é obrigatório usar os nomes argc e argv em seus programas. Eles são usados assim por questões tradicionais. Porém eles podem sofrer

variações em alguns compiladores e inclusive ter outros argumentos para o main mas, neste caso, é bom consultar o manual do usuário do compilador.

Figura 5.20

5.3.6 O return e retornos de funções

Na programação em geral, quando tratamos de módulos podemos tratar de duas formas: funções, que estamos vendo neste capítulo e procedimentos, que também é uma função, mas não retorna valor algum para o chamador.

Quando há retorno da função, usamos a palavra-chave return, que pode ter duas funções: sair de uma função forçadamente ou pode ser usado para retornar um valor para o programa chamador.

Observe os códigos que usamos por todo este livro. Observe que todas as funções main() dos nossos programas tem a seguinte forma: int main(...). Como estudamos, o tipo int antes da palavra main significa que há um retorno. E é por isso que usamos return 0 no final dos nossos códigos.

E quando não é necessário fazer um retorno na função (procedimento)? Usamos a palavra-chave void. Podemos inclusive usá-la no programa main da seguinte forma: void main(void) e desta forma o programa principal não terá retorno nem poderá receber nenhum argumento, como o argc e argv.

5.4 Fim do início

Procuramos escrever aqui o que é necessário para você começar a aprender a programar. A linguagem C++ é muito legal e muito usada mundialmente. Porém o que temos aqui é o fim de um início de outros estudos que aprondurão seus conhecimentos e agora é com você.

LEITURA

Você pode aprofundar os seus conhecimentos em C++ de uma maneira muito variada. Na internet existem dezenas de exemplos, tutorias, guias e demais formas de aprender programação. Além disso, o YouTube também possui vídeo aulas e cursos que você pode assistir.

Algumas faculdades internacionais e *sites* oferecem cursos gratuitos de linguagem C++ inclusive com emissão de certificado. Faça uma pesquisa na internet sobre isso. São cursos acessíveis e relativamente fáceis de serem seguidos.

Sobre livros, existem vários também. Sugerimos alguns ao longo dos capítulos. Procure também livros no formato eletrônico. Como a linguagem C++ é muito popular, você encontrará com facilidade.

E bons estudos!

REFLEXÃO

O uso de funções é um grande desafio. Não no sentido técnico, isto é relativamente fácil, pois estudando se aprende, mas é no sentido de arquitetura de *software*. Modularizar programas significa criar mini-programas que podem ser aproveitados em outras situações e economizar tempo e dinheiro em uma equipe de desenvolvimento. O bom arquiteto de *software* vai procurar por novas formas e modelos de tornar o desenvolvimento produtivo e prático para sua equipe.

Neste capítulo vimos a última estrutura de controle de fluxo para a programação. Com estes elementos já é possível programar em muitas linguagens de programação.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos de programação de computadores**. São Paulo: Pearson Education, 2008.

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. **Algoritmos**. São Paulo: McGraw Hill, 2009.

FEOFILOFF, P. **Algoritmos em linguagem C**. Rio de Janeiro: Campus, 2008.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. D. **Algoritmos**: lógica para desenvolvimento de programação. 9ª. ed. São Paulo: Érica, 1996.

PUGA, S.; RISSETTI, G. **Lógica de programação e estrutura de dados**. São Paulo: Pearson Education, 2003.

Apêndice: o Devc++

Vamos começar nossas atividades por meio de um exemplo. Vamos criar um programa para calcular a média de quatro notas bimestrais de um aluno.

Portanto, antes de partirmos para o código, vamos estudar o que foi pedido. Basicamente temos que entender quais são as entradas, o que será processado e a saída do programa.

Quais são as entradas do programa? Bem, para o computador escrever o seu nome, ele deve ser informado sobre o seu nome antes de qualquer coisa, portanto é bom criar uma variável para armazenar isso.

Antes de começarmos a programar, vamos conhecer o nosso ambiente e ferramenta de trabalho, o DevC++.

O DevC++ pode ser encontrado facilmente na internet. O link principal para baixa-lo é: http://goo.gl/5mlHNc

O DevC++ é um *software* gratuito específico para o desenvolvimento de aplicações na linguagem C ou C++. Ele é muito utilizado para o aprendizado devido sua simplicidade e praticidade nas tarefas básicas de desenvolvimento.

O DevC++ faz parte de uma categoria de *softwares* chamados IDEs (*Integrated Development Environment* – Ambiente de desenvolvimento integrado).

Um código fonte em C ou C++ pode ser feito no mais simples editor de texto que você possuir. Porém para compilar um programa em C e depois gerar o seu executável são necessários outros programas para estas tarefas e caberá ao usuário a tarefa de salvar o código fonte, fazer a compilação e depois a linkedição, manualmente e isso pode gerar alguns erros e até mesmo falhas de procedimento, além de ser feito basicamente em linha de comando.

Uma IDE é um programa que faz estas tarefas para o usuário. Além disso, a IDE possui outras ferramentas que podem ajudar o programador a criar o código, depurar o código, criar versões dos programas e outras.

Para ter uma ideia de como é desenvolver com um editor simples e com uma ideia, faça um teste. No Windows, crie um código simples na linguagem C e depois abra um *prompt* de comandos do *DOS* para compilar o programa e gerar o executável. Depois faça a mesma experiência usando o DevC++. Só o fato de editar o texto você verá que é diferente pois as palavras chave ficam em uma cor, as *strings* em outra, as constantes em outra e etc. Desta forma, visualmente o programador fica mais a vontade com o código e mais confortável com o ambiente.

A figura A1 mostra a tela principal do Dev C++.

Na figura percebe-se que existe uma aba chamada *Projeto* e este é um termo que é encontrado em praticamente todas as IDEs existentes, independente da linguagem.

Um projeto é uma forma de organizar um conjunto de arquivos os quais futuramente darão origem a uma aplicação. Até agora vimos somente arquivos. cpp que são os códigos fonte na linguagem C++ que digitamos.

Um programa em C++ de grande escala não possui somente códigos-fonte. Aliás, nem código-fonte uma aplicação final possuirá, ela vai possuir apenas os executáveis e outros arquivos como imagens em formato jpg, png, bmp, ícones com formato .gif, .ico, bibliotecas como as dlls, .lib e vários outros arquivos.

Por exemplo, vamos supor que você está desenvolvendo uma aplicação gráfica, como se fosse o *Paint* do *Windows*. Neste programa é

possível desenhar figuras geométricas e preenche-las com uma determinada cor. Matematicamente falando isso significa calcular a área da figura e preencher esta área com a cor desejada.

Sendo assim, não é mais inteligente criar um programa que armazena as fórmulas matemáticas do cálculo da área do círculo, do retângulo, etc, e deixar este programa disponível para qualquer outro que precise destas fórmulas, como se fosse uma biblioteca? Portanto, essa é uma das grandes vantagens de se utilizar uma IDE!

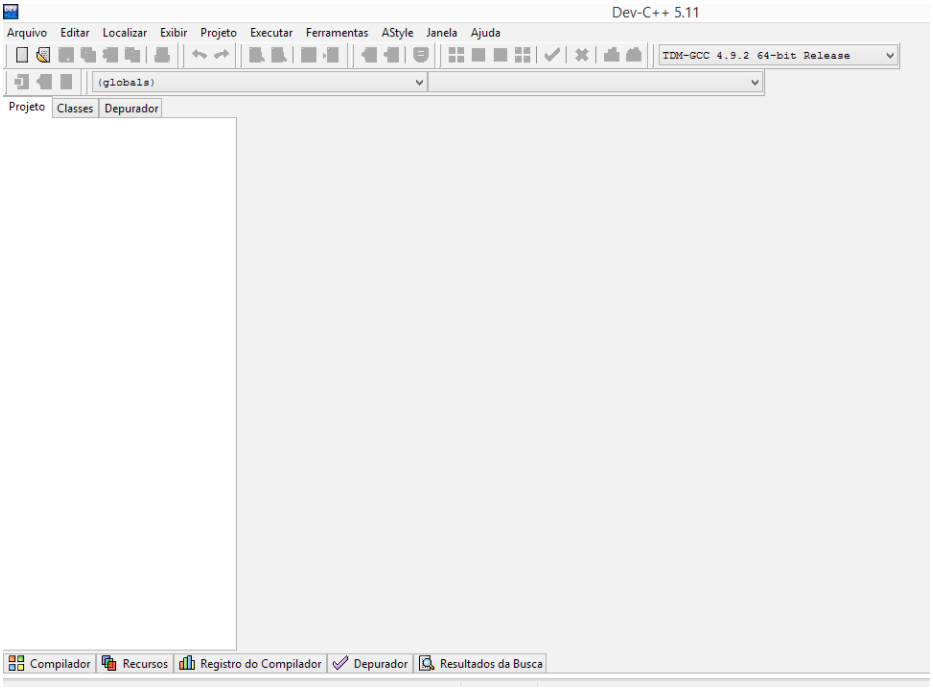


Figura A1 – Tela inicial do DevC++

A instalação do DevC++ é muito fácil e tradicional. É o famoso “next-next-next-finish”. Após a instalação, precisamos ter uma pequena noção do programa para começar a escrever os nossos códigos.

Clique em Arquivo, depois em Novo e Projeto. Irá aparecer a tela da figura A2.

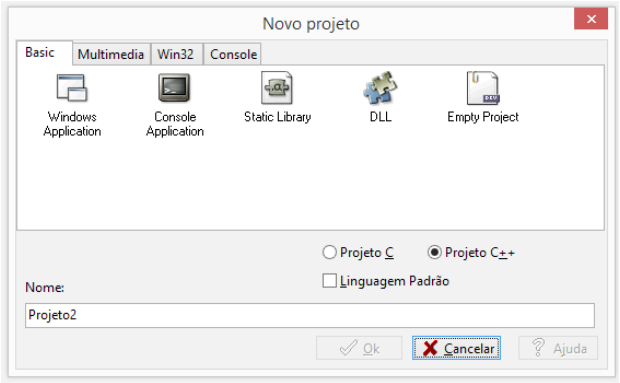


Figura A2 – Tela de criação de um novo projeto

A tela da figura A2 permite que o usuário escolha vários tipos de projetos:

• *Windows Application:* Este tipo de aplicação usa o *Windows* e seu sistema de janelas como plataforma de execução. Neste caso, a IDE é preparada e configurada para receber programas e módulos específicos para usar a biblioteca de interface gráfica. Normalmente é usada para projetos na linguagem C++.

• *Console Application*: Como o nome sugere, este tipo de aplicação é desenvolvida para rodar no prompt do DOS, quando o sistema operacional for *Windows*. Na verdade, uma aplicação para console rodará em um terminal somente texto e neste caso, é possível desenvolver programas para outros sistemas operacionais diferentes do *Windows*.

• *Static library:* neste tipo de projeto é possível desenvolver bibliotecas para serem usadas posteriormente em outros programas. Uma biblioteca, como já vimos pode ser um conjunto de rotinas, funções externas, definições de constantes e variáveis que são usadas na compilação de outros programas.

• *DLL (Dynamic Linked Library):* uma DLL é a implementação da *Microsoft* das *Static Library* citada no item anterior.

• *Empty Project:* neste caso, é criado um projeto vazio, sem nenhum tipo de arquivo ou estrutura previamente criada.

Além disso, a tela da figura A2 permite que o usuário escolha entre criar um projeto na linguagem C ou na linguagem C++. O checkbox para "Linguagem Padrão" serve para indicar se o projeto será criado no padrão ANSI (veremos isso posteriormente, por hora, pode deixar desmarcado).

Estas opções estão na aba *Basic.* Perceba que ainda temos as abas *Multimedia, Win3* e *Consol.* Cada uma dessas abas guarda tipos de projetos agrupados.

No nosso caso, vamos criar uma aplicação para *console* e na linguagem C++. Após colocar um nome, clique no botão Ok para criar o projeto. Em seguida irá aparecer uma tela com a localização da criação do arquivo de projeto (o projeto possui a extensão .dev). Escolha um local de sua preferência e salve o projeto.

> **Dica:** é interessante que você crie uma pasta para os seus projetos. Embora o DevC++ use uma pasta padrão, ter uma pasta específica para os projetos que você desenvolve faz com que um backup seja providenciado futuramente para esta pasta. Ter uma pasta para os projetos e organiza-los é uma boa prática de programação.

A tela da figura A3 mostra a configuração do DevC++ após a criação do projeto. O projeto possui o nome "LogicaDeProgramacao" e ao ser criado, o DevC++ já cria um arquivo chamado *main.cpp* o qual podemos começar a escrever o nosso código.

Dê uma olhada geral na IDE. Veja que temos várias opções para gerenciar nossos programas. Falando nisso, veja que o código fonte que já foi criado está colorido, com cada tipo de elemento possuindo uma cor diferente.

Por enquanto, nosso projeto só tem 1 componente: o arquivo *main.c.* Não faz parte do nosso escopo criar projetos muito grande portanto, quando formos testar algum código, este procedimento pode ser usado para nossos propósitos.

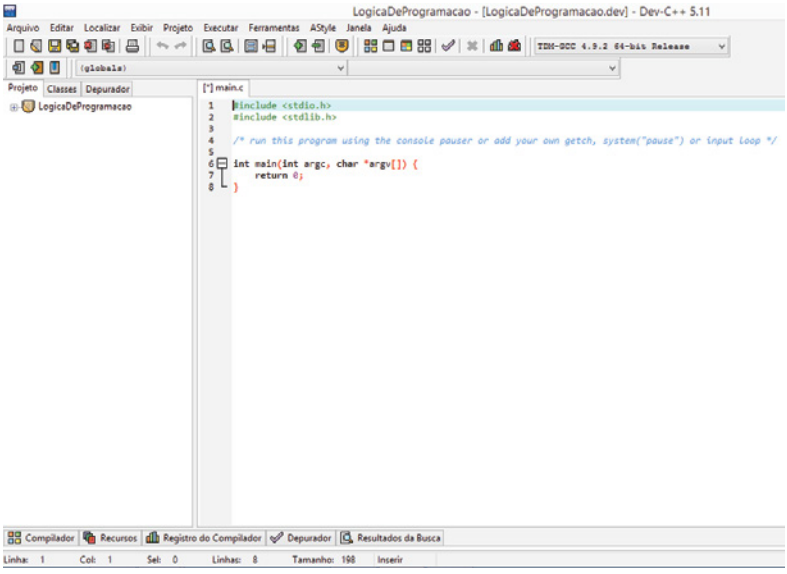


Figura A3 – Projeto criado

Iremos usar a linguagem C++ neste aprendizado. A linguagem C++ foi criada em 1983 por Bjarne Stroustrup nos laboratórios da Bell Labs (famosa empresa de telecomunicações dos EUA). Ela foi criada para implementar uma versão distribuída do núcleo do sistema operacional UNIX.

A linguagem C++ é muito popular e muito utilizada. Qualquer plataforma operacional atual possui compiladores para a linguagem. É uma linguagem que permite fácil acesso à memória do computador e possui facilidades para trabalhar com *hardware*, além de ser orientada a objetos.

Existe uma plataforma de código aberto para desenvolvimento de componentes eletrônicos muito popular chamada Arduino. Ela tem o propósito de ser acessível para qualquer pessoa poder criar os seus componentes. Além da parte eletrônica, o usuário precisa usar uma linguagem de programação para codificar os programas e normalmente a linguagem C ou C++ é usada para isso.

Assista este vídeo a respeito de uma breve história do C++. Trata-se de uma palestra proferida pelo Bjarne Stroustrup. Está em inglês mas com atenção é bem compreensível. https://www.youtube.com/watch?v=86xWVb4XIyE

ISBN 978-85-5548-154-3