Capítulo 1

Introdução

Objetivos do capítulo

- Entender a atividade de programação
- Aprender sobre a arquitetura de computadores
- Aprender sobre linguagens de máquina e linguagens de programação de alto nível
- Familiarizar-se com seu compilador
- Compilar e executar seu primeiro programa em C++
- Reconhecer erros de sintaxe e erros de lógica

Este capítulo contém uma breve introdução à arquitetura de computadores e uma visão geral de linguagens de programação. Você vai aprender sobre a atividade de programação: como escrever e executar seu primeiro programa em C++, como diagnosticar e corrigir erros de programação e como planejar as suas atividades de programação.

Conteúdo do capítulo

1.1 O que é um computador?

Você provavelmente já usou um computador para trabalho ou lazer. Muitas pessoas usam computadores para tarefas cotidianas, como controlar saldos em um talão de cheques ou escrever o texto de um trabalho. Computadores são bons para estas tarefas. Eles podem incumbir-se destas pequenas tarefas repetitivas, tais como totalizar números e colocar palavras em uma página, sem aborrecer-se nem cansar-se. Mais importante, o computador mostra o talão de cheques ou o texto do trabalho na tela e permite que você corrija erros facilmente. Computadores são boas máquinas de jogos por que eles podem mostrar seqüências de sons e imagens, envolvendo o usuário humano no processo.

O que torna tudo isso possível não é somente o computador. O computador deve ser programado para executar estas tarefas. Um programa controla talões de cheques; um outro programa, provavelmente projetado e construído por outra empresa, processa textos; e um terceiro programa joga um jogo. O computador em si é uma máquina que armazena dados (números, palavras, imagens), interage com dispositivos (o monitor, o sistema de som, a impressora) e executa programas. Programas são seqüências de instruções e de decisões que o computador executa para realizar uma tarefa.

Atualmente os programas de computador são tão sofisticados que é difícil acreditar que eles são compostos por operações extremamente primitivas. Uma operação típica pode ser uma das seguintes:

- Colocar um ponto vermelho nesta posição do vídeo.
- Enviar a letra A para a impressora.
- Obter um número desta posição na memória.
- Somar estes dois números.
- Se este valor é negativo, continuar o programa naquela instrução.

O usuário do computador tem a ilusão de uma interação suave porque um programa contém uma enorme quantidade de tais operações e porque o computador pode executá-las a grande velocidade.

A flexibilidade de um computador é realmente um fenômeno interessante. A mesma máquina pode controlar seu talão de cheques, imprimir o texto de seu trabalho e jogar um jogo. Em contraste, outras máquinas realizam um número reduzido de tarefas; um carro anda e uma torradeira tosta. Computadores podem realizar uma grande variedade de tarefas porque eles executam diferentes programas, sendo que cada um deles dirige o computador para trabalhar em uma tarefa específica.

1.2 O que é programação?

Um programa de computador indica ao computador, nos mínimos detalhes, a seqüência de passos necessários para executar uma tarefa. O ato de projetar e implementar estes programas é denominado de programação de computador. Neste livro você vai aprender como programar um computador — isto é, como dirigir o computador para executar tarefas.

Para usar um computador você não necessita fazer nenhuma programação. Quando você escreve um trabalho com um processador de texto, aquele programa foi programado pelo fabricante e está pronto para seu uso. Isto é nada mais que o esperado – você pode dirigir um carro sem ser um mecânico e torrar pão sem ser um eletricista. A maioria das pessoas que usam diariamente computadores nunca necessitará fazer nenhuma programação.

Como você está lendo este livro introdutório à ciência da computação, pode ser que seu objetivo seja tornar-se profissionalmente um cientista da computação ou um engenheiro de *software*. Programação não é a única qualificação exigida de um cientista da computação ou engenheiro de *software*; na verdade, programação não é a única qualificação exigida para criar bons programas de computador. Contudo, a atividade de programação é fundamental em ciência da computação. Também é uma atividade fascinante e agradável, que continua a atrair e motivar estudantes brilhantes. A disciplina de ciência da computação é particularmente afortunada ao fazer desta atividade interessante o fundamento do caminho de aprendizagem.

Escrever um jogo de computador com efeitos de animação e sonoros ou um processador de texto que possua fontes e desenhos atraentes é uma tarefa complexa que exige uma equipe de muitos programadores altamente qualificados. Seus primeiros esforços de programação serão mais tri-

viais. Os conceitos e habilidades que você vai aprender neste livro formam uma base importante e você não deve desapontar-se se os seus primeiros programas não rivalizam com os *softwares* sofisticados que lhe são familiares. Realmente, você verá que mesmo as mais simples tarefas de programação provocam uma imensa vibração. É uma agradável experiência ver que o computador realiza precisamente e rapidamente uma tarefa que você levaria muitas horas para fazer, que fazer pequenas alterações em um programa produz melhorias imediatas e ver o computador tornar-se uma extensão de suas forças mentais.

1.3 A anatomia de um computador

Para entender o processo de programação, você necessita ter um entendimento rudimentar dos componentes que formam um computador. Vamos examinar um computador pessoal. Computadores maiores possuem componentes mais rápidos, maiores ou mais poderosos, mas eles possuem fundamentalmente o mesmo projeto.

No coração do computador fica a *unidade central de processamento (UCP)* (ver Figura 1). Ela consiste de um único *chip*, ou um pequeno número de *chips*. Um *chip* (circuito integrado) de computador é um componente com uma base metálica ou plástica, conectores metálicos e fiação interna feita principalmente de silício. Para um *chip* de UCP, a fiação interna é extremamente complicada. Por exemplo, o *chip* do Pentium (uma UCP popular para computadores pessoais no momento da escrita deste livro) é composto por vários milhões de elementos estruturais denominados *transistores*. A Figura 2 mostra um detalhe ampliado de um *chip* de UCP. A UCP realiza o controle do programa, operações aritméticas e de movimentação de dados. Isto é, a UCP localiza e executa as instruções do programa; ela realiza operações aritméticas como soma, subtração, multiplicação e divisão; ela carrega ou armazena dados da memória externa ou de dispositivos. Todos os dados trafegam através da UCP sempre que são movidos de uma posição para outra. (Existem umas poucas exceções técnicas a esta regra; alguns dispositivos podem interagir diretamente com a memória.)

O computador armazena dados e programas na *memória*. Existem dois tipos de memória. *A memória primária* é rápida porém cara; ela é feita de *chips* de memória (ver Figura 3): sendo denominada de *memória de acesso randômico (RAM- Random-Access Memory)* e de *memória somente de leitura (ROM- Read-Only Memory)*. A memória somente de leitura contém certos programas que devem estar sempre presentes — por exemplo, o código necessário para iniciar o computador. A memória de acesso randômico é mais conhecida como "memória de leitura e escrita", por que a UCP pode ler dados dela e pode escrever dados nela. Isso torna a RAM adequada para conter dados que podem ser alterados e programas que não necessitam estar disponíveis permanentemente. A memória RAM tem duas desvantagens. Ela é comparativamente cara e ela perde seus da-

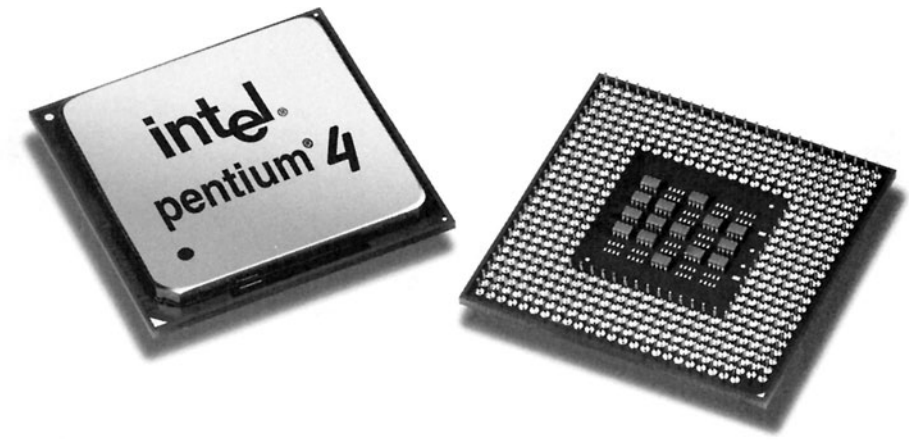

Figura 1
Unidade central de processamento.

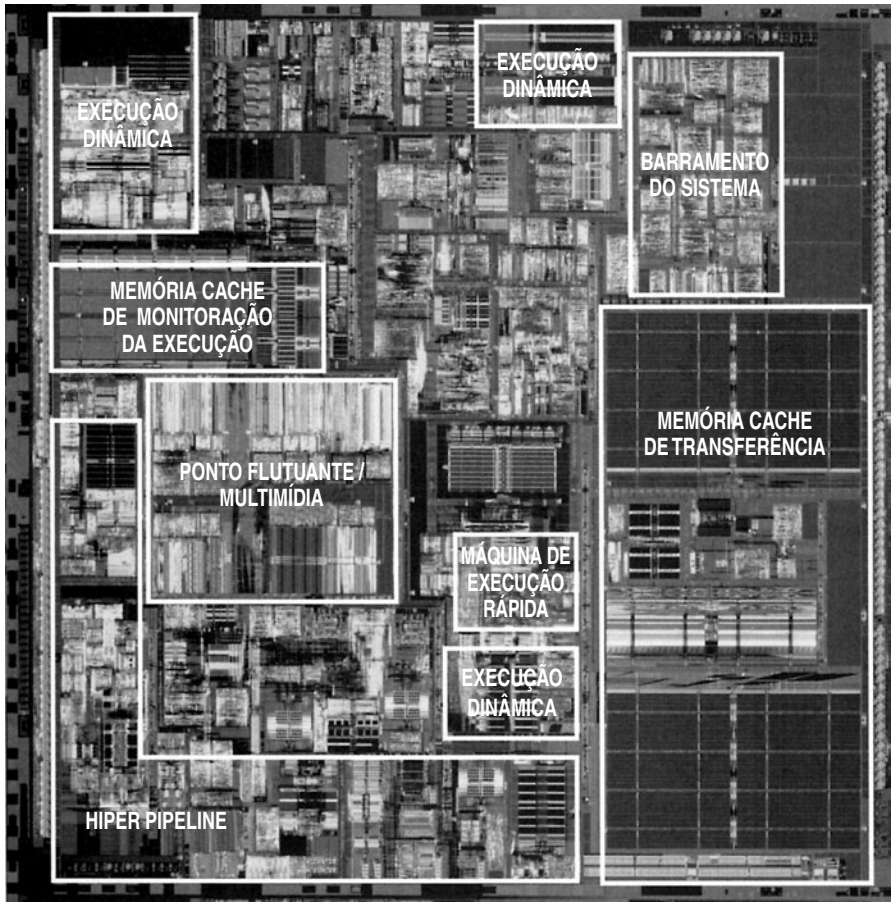

Figura 2

Detalhe de um *chip* de UCP.

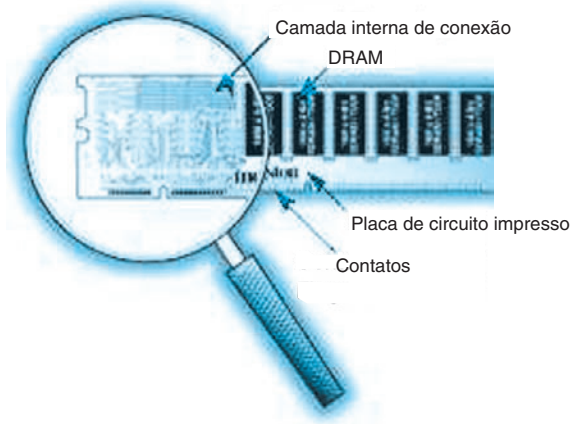

Figura 3

Chips de RAM.

dos quando a energia é desligada. A *memória secundária*, geralmente um *disco rígido* (ver Figura 4), é uma memória de menor custo e que persiste sem eletricidade. Um disco rígido consiste de pratos rotativos, que são recobertos com material magnético, e cabeçotes de leitura/escrita que podem detectar e alterar o fluxo magnético nos pratos rotativos. Esse é exatamente o mesmo processo de armazenamento usado em fitas de áudio ou vídeo. Programas e dados são normalmente armazenados em disco rígido e carregados na RAM quando o programa inicia. O programa então atualiza os dados na RAM e escreve de volta no disco rígido os dados modificados.

A unidade central de processamento, a memória RAM, e a parte eletrônica que controla o disco rígido e outros dispositivos são interconectados através de um conjunto de linhas elétricas denominadas de *barramentos*. Dados trafegam do sistema de memória e dispositivos periféricos para a UCP ao longo dos barramentos e vice-versa. A Figura 5 mostra uma *placa mãe* que contém a UCP, a RAM e encaixes de cartões, através dos quais os cartões que controlam dispositivos periféricos se conectam ao barramento.

Para interagir com o usuário humano, um computador precisa de dispositivos periféricos. O computador transmite informação ao usuário através de telas de vídeo, alto-falantes e impressoras. O usuário pode transmitir informações e fornecer diretivas ao computador usando um teclado ou um dispositivo de apontar como um *mouse*.

Alguns computadores são unidades independentes, enquanto outros são conectados por meio de redes. Através do cabeamento de redes, o computador pode ler dados e programas localizados na memória central ou enviar dados para outros computadores. Para o usuário de um computador em rede nem sempre é óbvio quais dados residem no computador local e quais são transmitidos via rede.

A Figura 6 mostra uma visão geral esquemática da arquitetura de um computador. Instruções do programa e dados (como texto, números, áudio ou vídeo) são armazenados no disco rígido, em um CD-ROM, ou em qualquer lugar da rede. Quando um programa é iniciado, ele é trazido para a memória RAM, onde a UCP pode fazer sua leitura. A UCP lê o programa, uma instrução de cada vez. De acordo com estas instruções, a UCP lê dados, modifica-os e os grava de volta na memória RAM ou no disco rígido. Algumas instruções do programa podem fazer a UCP colocar pontos na tela de vídeo ou impressora ou vibrar o alto-falante. À medida que estas ações ocorrem muitas vezes e a grande velocidade, o usuário humano vai perceber imagens e sons. Algumas instruções de programa lêem a entrada do usuário via teclado ou *mouse*. O programa analisa a natureza destas entradas e então executa a próxima instrução.

Figura 4

Um disco rígido.

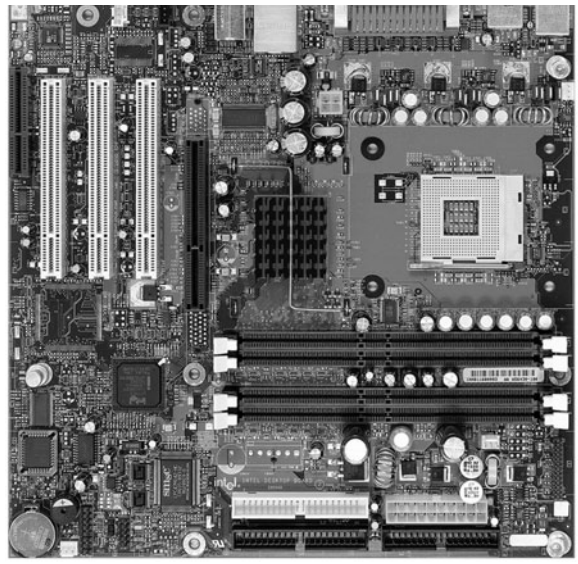

Figura 5
Uma placa-mãe.

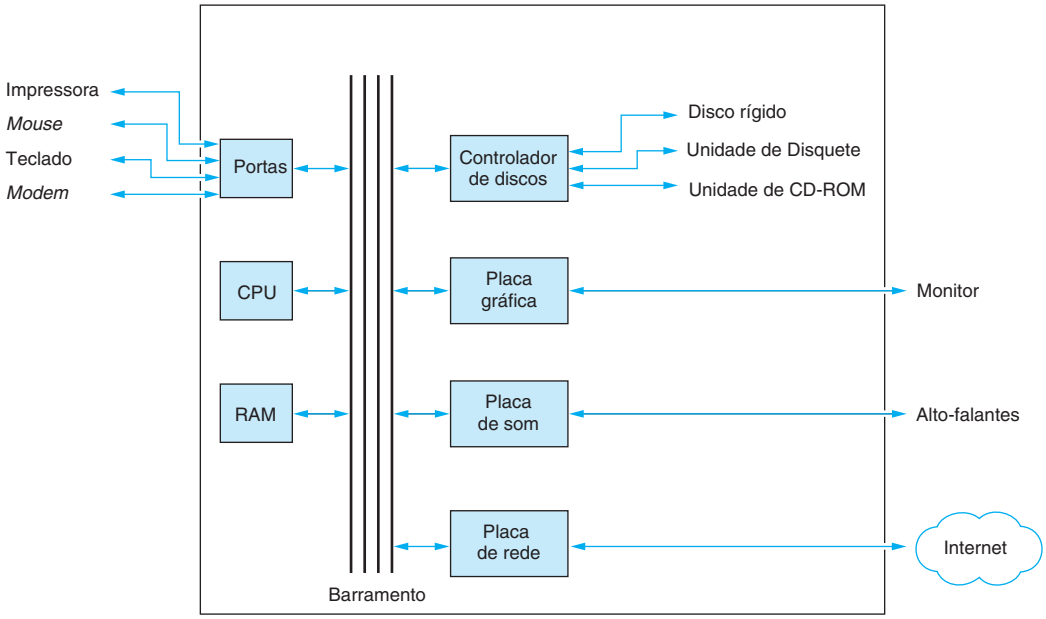

Figura 6
Desenho esquemático de um computador pessoal.

Fato Histórico 1.1

O ENIAC e o Surgimento da Computação

O ENIAC (*Electronic Numerical Integrator and Computer*) foi o primeiro computador eletrônico usável. Ele foi projetado por J. Presper Eckert e John Mauchly na Universidade da Pensilvânia e

foi concluído em 1946 — dois anos antes da invenção dos transistores. O computador foi instalado em uma ampla sala e consistia de vários gabinetes contendo cerca de 18,000 válvulas (ver Figura 7). Diariamente queimavam várias válvulas. Um ajudante com um carrinho de compras cheio de válvulas fazia a ronda e substituía as defeituosas. O computador era programado por conexão de fios em painéis. Cada configuração de fiação instruía o computador para um problema particular. Para fazer o computador trabalhar em outro problema, a fiação deveria ser refeita.

O trabalho no ENIAC foi apoiado pela Marinha dos Estados Unidos, que estava interessada no cálculo de tabelas balísticas que poderiam fornecer a trajetória de um projétil, dependendo da resistência do vento, da velocidade inicial e das condições atmosféricas. Para calcular as trajetórias, era necessário encontrar soluções numéricas de certas equações diferenciais; daí o nome "integrador numérico". Antes do desenvolvimento de máquinas como o ENIAC, as pessoas faziam este tipo de trabalho e até os anos 1950, a palavra "computador" se referia a estas pessoas. O ENIAC foi posteriormente usado para propósitos pacíficos como a tabulação dos dados do censo americano.

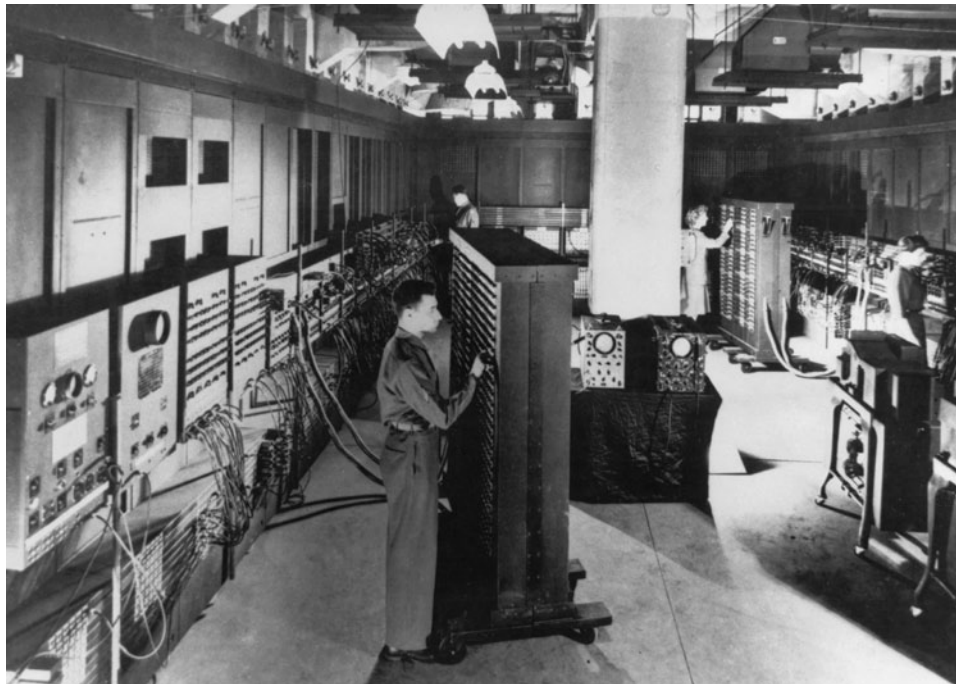

Figura 7

O ENIAC.

1.4 Traduzindo programas legíveis por pessoas para código de máquina

No nível mais básico, instruções de computador são extremamente primitivas. O processador executa *instruções de máquina.* Uma seqüência típica de instruções de máquina é:

1. Mover o conteúdo da posição de memória 40000 para o registrador `eax`. (Um registrador é um elemento de armazenamento da UCP.)
2. Subtrair o valor 100 do registrador `eax`.

3. Se o resultado é positivo, continuar com a instrução que está armazenada na posição de memória 11280.

Na verdade, instruções de máquina são codificadas como números de forma que possam ser armazenadas na memória. Em um processador Intel 80386, esta seqüência de instruções é codificada como uma seqüência de números

161 40000 45 100 127 11280

Em um processador de outro fabricante, a codificação pode ser bem diferente. Quando esse tipo de processador carrega essa seqüência de números, ele os decodifica e executa a seqüência de comandos associada.

Como podemos comunicar a seqüência de comandos ao computador? O método mais simples é colocar os próprios números na memória do computador. Essa era, de fato, a maneira como os primeiros computadores trabalhavam. Entretanto, um programa longo é composto de milhares de comandos individuais, e é uma tarefa tediosa e suscetível a erros procurar os códigos numéricos de todos os comandos e colocar os códigos manualmente na memória. Como dito anteriormente, computadores são realmente bons em atividades tediosas e suscetíveis a erros e não demorou para que programadores de computador percebessem que os próprios computadores poderiam ser aproveitados para auxiliar no processo de programação.

O primeiro passo foi atribuir nomes curtos aos comandos. Por exemplo, `mov` indica "mover", `sub` "subtrair", e `jg` "saltar se maior do que 0". Usando esses comandos, a seqüência de instruções se torna

```
mov 40000, %eax
sub 100, %eax
jg 11280
```

Isso é muito mais fácil para humanos lerem. Contudo, para obter a seqüência de instruções aceitas pelo computador, os nomes devem ser traduzidos para código de máquina. Esta é a tarefa de outro programa de computador: o assim denominado *montador* (*assembler)*. Ele pega a seqüência de caracteres `"mov %eax"` e a traduz para o código de comando 161, e executa operações similares sobre os outros comandos. Montadores possuem outra característica: eles podem associar nomes a *posições de memória,* assim como a instruções. Nossa seqüência de programa poderia ter verificado se alguma taxa de juro era maior que 100%, e se a taxa de juro estava armazenada na posição de memória 40000. Geralmente não é importante onde um valor está armazenado; qualquer posição de memória disponível serve. Ao usar nomes simbólicos ao invés de endereços de memória, o programa se torna ainda mais fácil de ler:

```
mov int_rate, %eax
sub 100, %eax
jg int_erro
```

É tarefa do programa montador encontrar valores numéricos adequados para os nomes simbólicos e colocar estes valores na seqüência de código gerada.

A programação com instruções *assembler* representa um importante avanço sobre a programação em código de máquina puro, mas ela apresenta dois inconvenientes. Ela ainda usa um grande número de instruções para atingir os mais simples objetivos, e a seqüência exata de instruções difere de um processador para outro. Por exemplo, a seqüência de instruções *assembler* acima deve ser reescrita para o processador Sun SPARC, o que impõe um problema real para pessoas que investem bastante tempo e dinheiro produzindo um pacote de *software*. Se um computador se torna obsoleto, o programa deve ser completamente reescrito para ser executado no sistema substituto. Em meados dos anos 1950, linguagens de programação de alto nível começaram a surgir. Nestas linguagens, o programador expressa a idéia sobre a tarefa que necessita ser executada e um programa de computador especial, o assim chamado *compilador*, traduz a descrição de alto nível para instruções de máquina de um processador específico.

Por exemplo, em C++, a linguagem de programação de alto nível que será usada neste livro, você pode ter a seguinte instrução:

```
if (int_rate > 100) message_box("Erro na taxa de juros");
```

Isso significa "Se a taxa de juros é superior a 100, exibir uma mensagem de erro". É então tarefa do programa compilador examinar a seqüência de caracteres `"if (int_rate > 100)"` e traduzi-la para

```
161 40000 45 100 127 11280
```

Compiladores são programas bastante sofisticados. Eles têm que traduzir comandos lógicos como o `if`, para seqüências de computações, testes e saltos e eles devem encontrar posições de memória para variáveis como `int_rate`. Neste livro, geralmente vamos considerar a existência de um compilador como certa. Se você pretende se tornar um cientista da computação profissional, pode aprender mais sobre técnicas de escrita de compiladores em seus estudos posteriores.

Linguagens de alto nível são independentes do *hardware* subjacente. Por exemplo, a instrução `if (int_rate > 100)` não se baseia em nenhuma instrução de máquina particular. De fato, ela será compilada para códigos diferentes em um processador Intel 80386 e em um Sun SPARC.

1.5 Linguagens de programação

Linguagens de programação são independentes de uma arquitetura de computador específica, visto serem criações humanas. Como tais, elas seguem certas convenções. Para facilitar o processo de tradução, estas convenções são mais estritas que aquelas de linguagens humanas. Quando você fala com outra pessoa e mistura ou omite uma palavra ou duas, seu parceiro de conversa irá geralmente entender o que você disse. Os compiladores são menos generosos. Por exemplo, se você omitir a aspa no final da instrução

```
if (int_rate > 100) message_box("Erro na taxa de juros);
```

o compilador C++ ficará bastante confuso e reclamará que não consegue traduzir uma instrução contendo este erro. Isto é, realmente, uma coisa boa. Se o compilador tentasse adivinhar o que você fez errado e tentasse consertar, ele poderia não adivinhar suas intenções corretamente. Neste caso, o programa resultante poderia fazer a coisa errada – bem possivelmente com efeitos desastrosos, se este programa controlasse um dispositivo de cujas funções alguém dependesse para seu bem-estar. Quando um compilador lê instruções de um programa em uma linguagem de programação, ele irá traduzir para código de máquina somente se a entrada obedece exatamente as convenções da linguagem. Assim como existem muitas linguagens humanas, existem muitas linguagens de programação. Considere a instrução

```
if (int_rate > 100) message_box("Erro de taxa de juros");
```

Isso é como você deve formatar a instrução em C++. C++ é uma linguagem de programação bastante popular, e é a que usamos neste livro. Mas em Pascal (outra linguagem de programação comum nos anos 1980) a mesma instrução poderia ser escrita como

```
if int_rate > 100 then message_box('Erro de taxa de juros');
```

Neste caso, as diferenças entre as versões de C++ e Pascal são leves: para outras construções, as diferenças seriam mais substanciais. Compiladores são específicos para linguagens. O compilador C++ irá traduzir somente código C++, enquanto um compilador Pascal irá rejeitar tudo que não seja código válido em Pascal. Por exemplo, se um compilador C++ lê uma instrução `if int_rate > 100 then...`, ele irá reclamar, porque a condição do comando `if` não está cercada por parênteses `( )`, e o compilador não espera a palavra `then`. A escolha do leiaute de uma construção da linguagem como o comando `if` é de certa forma arbitrária. Os projetistas de diferentes linguagens escolhem diferentes balanços entre legibilidade, facilidade de tradução e consistência com outras construções.

1.6 Linguagens de programação: projeto e evolução

Atualmente existem centenas de linguagens de programação. Isso é realmente bastante surpreendente. A idéia norteadora de uma linguagem de programação de alto nível é fornecer um meio para a programação que seja independente de um conjunto de instruções de um processador em particular, de modo que seja possível mover programas de um computador para outro sem rescrita. Mover um programa de uma linguagem de programação para outra é um processo difícil e raramente é feito. Assim, pode parecer haver pouca utilidade para tantas linguagens de programação.

Diferentemente de linguagens humanas, linguagens de programação são criadas com objetivos específicos. Algumas linguagens de programação tornam particularmente fácil expressar tarefas de um domínio particular de problemas. Algumas linguagens se especializam em processamento de bancos de dados; outras em programas de "inteligência artificial" que tentam inferir novos fatos de uma dada base de conhecimento; outras em programação multimídia. A linguagem Pascal foi propositadamente mantida simples por ter sido projetada como uma linguagem de ensino. A linguagem C foi desenvolvida para ser traduzida eficientemente para código de máquina rápido, com um mínimo de sobrecarga de manutenção. C++ foi construída sobre C, adicionando características para "programação orientada a objetos", um estilo de programação que promete uma modelagem mais fácil de objetos do mundo real.

Linguagens de programação de uso específico ocupam seus próprios nichos e não são usadas muito além de sua área de especialização. Pode ser possível escrever um programa multimídia em uma linguagem de bancos de dados, mas provavelmente será um desafio. Em contraste, linguagens como Pascal, C e C++ são linguagens de uso geral. Qualquer tarefa que você gostaria de automatizar pode ser escrita nestas linguagens.

A versão inicial da linguagem C foi projetada por volta de 1972, mas novos recursos foram adicionados a ela ao longo dos anos. Uma vez que vários implementadores de compiladores adicionaram diferentes recursos, a linguagem desenvolveu diversos dialetos. Algumas instruções de programas eram entendidas por um compilador mas rejeitadas por outro. Tal divergência é um obstáculo importante para um programador que deseja mover código de um computador para outro. Esforços foram empregados para resolver as diferenças e culminaram com uma versão padrão de C. O processo de projeto terminou em 1989 com a conclusão do padrão ANSI (*American National Standards Institute*). Neste meio tempo, Bjarne Stroustrup, da AT&T, adicionou a C características da linguagem Simula (uma linguagem orientada a objetos projetada para realizar simulações). A linguagem resultante foi denominada de C++. De 1985 até hoje, C++ tem crescido pela adição de diversos recursos, e um processo de padronização culminou com a publicação do padrão internacional de C++ em 1998.

C e C++ são bons exemplos de linguagens que cresceram de modo incremental. À medida que usuários da linguagem perceberam deficiências, eles adicionaram recursos. Em contraste, linguagens como Pascal foram projetadas de um modo mais ordenado. Um indivíduo, ou um pequeno grupo, estabelecem o projeto de toda a linguagem, tentando antecipar as necessidades de seus futuros usuários. Linguagens assim planejadas possuem uma grande vantagem: uma vez que foram projetadas com uma visão, seus recursos tendem a ser logicamente relacionados entre si e recursos isolados podem ser facilmente combinados. Em contraste, linguagens "crescidas" são geralmente um pouco confusas; diferentes recursos foram projetados por pessoas com diferentes critérios. Uma vez que um recurso passa a fazer parte da linguagem, é difícil removê-lo. Remover um recurso estraga todos os programas existentes que dele fazem uso e seus autores poderiam ficar muito aborrecidos com a perspectiva de ter que rescrevê-los. As linguagens assim estendidas tendem a acumular recursos como uma colcha de retalhos que não necessariamente interagem bem entre si.

Linguagens planejadas são geralmente projetadas com maior meditação. Existe mais atenção à legibilidade e à consistência. Em contraste, um novo recurso em uma linguagem "crescida" é freqüentemente adicionado às pressas, para atender a uma necessidade específica, sem raciocinar sobre as ramificações. Você pode ver um vestígio desse fenômeno nos comandos `if` de Pascal e C++. A versão Pascal

```
if int_rate > 100 then...
```

é mais fácil de ler que a versão C

```
if (int_rate > 100)...
```

porque a palavra chave `then` auxilia a pessoa a ler. É realmente mais fácil de compilar também, porque a palavra chave `then` indica ao compilador onde termina a condição e inicia a ação. Em contraste, C++ necessita parênteses () para separar a condição da ação. Por que essa diferença? O truque com a palavra chave `then` era realmente bem conhecido quando Pascal e C foram projetados. Ela era usada em Algol 60, uma linguagem visionária que influenciou fortemente o projeto de linguagens nos anos subsequentes. (O cientista de computação Tony Hoare disse a respeito de Algol 60: "Aqui está uma linguagem tão além de seu tempo, que não apenas representa uma melhoria sobre suas predecessoras, mas também para quase todas as suas sucessoras". [1]). O projetista de Pascal usou `if...then` por ser uma boa solução. Os projetistas de C não foram tão competentes no projeto da linguagem. Ou eles não conheciam a construção ou não apreciaram seus benefícios. Em vez disso, eles reproduziram o projeto pobre do comando `if` de FORTRAN, outra linguagem de programação antiga. Se eles posteriormente lamentaram sua decisão, era tarde demais. A construção `if (...)` havia sido usada milhões de vezes e ninguém desejaria alterar código existente em funcionamento.

Linguagens que são projetadas por planejadores competentes são geralmente mais fáceis de aprender e usar. Entretanto, linguagens "crescidas" têm o apelo do mercado. Considere, por exemplo, C++. Visto que C++ é simplesmente C com algumas adições, qualquer programa escrito em C irá continuar funcionando sob C++. Entretanto, programadores seriam capazes de tirar proveito dos benefícios das características de orientação a objetos sem ter que descartar seus programas C existentes. Este é um enorme benefício. Em contraste, a linguagem Modula 3 foi projetada desde a sua base para oferecer os benefícios de orientação a objetos. Não existe dúvida que Modula 3 é mais fácil de aprender e usar do que C++, mas para um programador que já conhecia C, o cenário é diferente. Este programador pode facilmente migrar o código C para C++, enquanto que reescrever todo o código em Modula 3 seria doloroso por duas razões. Um programa sério consiste em muitos milhares e mesmo milhões de linhas de código e traduzi-lo linha por linha obviamente consumiria tempo. Além disso, existe mais em uma linguagem de programação do que sua sintaxe e convenções. A linguagem C aproveita um tremendo suporte de ferramentas oferecidas em pacotes de *software* que auxiliam o programador a controlar seus programas em C. Estas ferramentas encontram erros, arquivam código, aumentam a velocidade de programas e auxiliam na combinação de porções de código úteis provenientes de várias fontes. Quando uma nova linguagem como Modula 3 é criada, ela possui apenas um suporte rudimentar de ferramentas, tornando duplamente difícil de adotá-la para um projeto em andamento. Em contraste, ferramentas C podem ser facilmente modificadas para trabalhar com C++.

Atualmente, C++ é a principal linguagem de programação de uso geral. Por essa razão, usamos um subconjunto de C++ neste livro para ensinar você a programar. Isto lhe permitirá se beneficiar de excelentes ferramentas C++ e comunicar-se facilmente com outros programadores, muitos dos quais usam C++ quotidianamente. A desvantagem é que C++ não é assim tão fácil de aprender e possui sua cota de armadilhas e inconvenientes. Não quero dar a você a impressão de que C++ é uma linguagem inferior. Ela foi projetada e refinada por muitas pessoa brilhantes e dedicadas, ela possui uma enorme abrangência de aplicação, que varia desde programas orientados a *hardware* até os mais altos níveis de abstração. Simplesmente existem algumas partes de C++ que exigem mais atenção, especialmente de programadores iniciantes. Irei destacar possíveis ciladas e como você pode evitá-las. O objetivo deste livro não é ensinar tudo de C++, mas sim usar C++ para ensinar a você a arte e a ciência de escrever programas de computador.

Fato Histórico 1.2

Organizações de Padronização

Duas organizações, a American National Standards Institute (ANSI) e a International Organization for Standardization (ISO), desenvolveram em conjunto o padrão definitivo da linguagem C++.

Por que ter padrões? Você se depara com os benefícios da padronização a cada dia. Quando você compra uma lâmpada para uma lanterna, tem a certeza de que ela caberá no soquete sem ter que

medir a lanterna em casa e a lâmpada na loja. De fato, você experimentaria constatar quão dolorosa pode ser a falta de padrões se você adquirisse lâmpadas com bulbos fora do padrão. Bulbos de reposição para tal lanterna seriam caros e difíceis de obter.

As organizações de padronização ANSI e ISO são associações de profissionais da indústria que desenvolvem padrões para tudo, desde pneus de carros e formatos de cartões de crédito até linguagens de programação. Ter um padrão para uma linguagem de programação como C++ significa que você pode levar um programa desenvolvido para um sistema com um compilador de um fabricante para um sistema diferente e ter a certeza de que ele irá continuar a funcionar.

Para saber mais sobre organizações de padronização, consulte os seguintes *sites* da *Web*: `www.ansi.org` e `www.iso.ch`.

1.7 Familiarizando-se com seu computador

Enquanto usa este livro, você bem pode estar usando um computador desconhecido. Você pode despender algum tempo para familiarizar-se com o computador. Uma vez que sistemas de computadores variam bastante, este livro pode somente indicar um roteiro de passos que você deve seguir. Usar um sistema de computador novo e desconhecido pode ser frustrante. Procure por cursos de treinamento que são oferecidos onde você estuda ou apenas peça a um amigo para lhe ensinar um pouco.

Passo 1 Iniciar o sistema

Se você usar seu computador em casa, você não precisa preocupar-se com identificação. Computadores em um laboratório, entretanto, não são normalmente abertos a qualquer um. O acesso é geralmente restrito àqueles que pagam as taxas necessárias e que são confiáveis por não alterarem as configurações. Você provavelmente necessita um número de conta e uma senha para obter acesso ao sistema.

Passo 2 Localizar o compilador C++

Sistema de computadores diferem grandemente neste aspecto. Alguns sistemas deixam você iniciar o compilador selecionando um ícone ou menu. Em outros sistemas você deve usar o teclado para digitar um comando para iniciar o compilador. Em muitos computadores pessoais existe o conhecido *ambiente integrado*, no qual você pode escrever e testar seus programas. Em outros computadores você deve primeiro iniciar um programa que funciona como um processador de texto, no qual você pode inserir suas instruções C++; depois iniciar outro programa para traduzí-las para código de máquina; e então executar o código de máquina resultante.

Passo 3 Entender arquivos e pastas

Como um programador, você irá escrever seus programas C++, testá-los e melhorá-los. Você terá um espaço no computador para armazená-los e deverá saber localizá-los. Programas são armazenados em *arquivos*. Um arquivo C++ é um recipiente de instruções C++. Arquivos possuem nomes e as regras para nomes válidos diferem de um sistema para outro. Em alguns sistemas, os nomes de arquivos não podem possuir mais de oito caracteres. Alguns sistemas permitem espaços em nomes de arquivos; outros não. Alguns distinguem entre letras maiúsculas e minúsculas; outros não. A maioria dos compiladores C++ exige que os arquivos C++ possuam a *extensão* `.cpp` ou `.C`; por exemplo, `teste.cpp`.

Arquivos são armazenados em *pastas* ou *diretórios*. Estes recipientes de arquivos podem ser aninhados. Uma pasta contém arquivos e outras pastas, que por sua vez podem conter mais arquivos e mais pastas (ver Figura 8). Esta hierarquia pode ser bem grande, especialmente em computadores em rede, onde alguns arquivos podem estar em seu disco local, outros em algum lugar da rede. Embora você não precise se preocupar com cada detalhe da hierarquia, deve se familiarizar com seu ambiente local. Sistemas diferentes possuem diferentes maneiras de mostrar arquivos e diretórios. Alguns usam um vídeo gráfico e permitem que você o percorra clicando ícones de pastas com um *mouse*. Em outros sistemas, você deve digitar comandos para visitar ou inspecionar diferentes pastas.

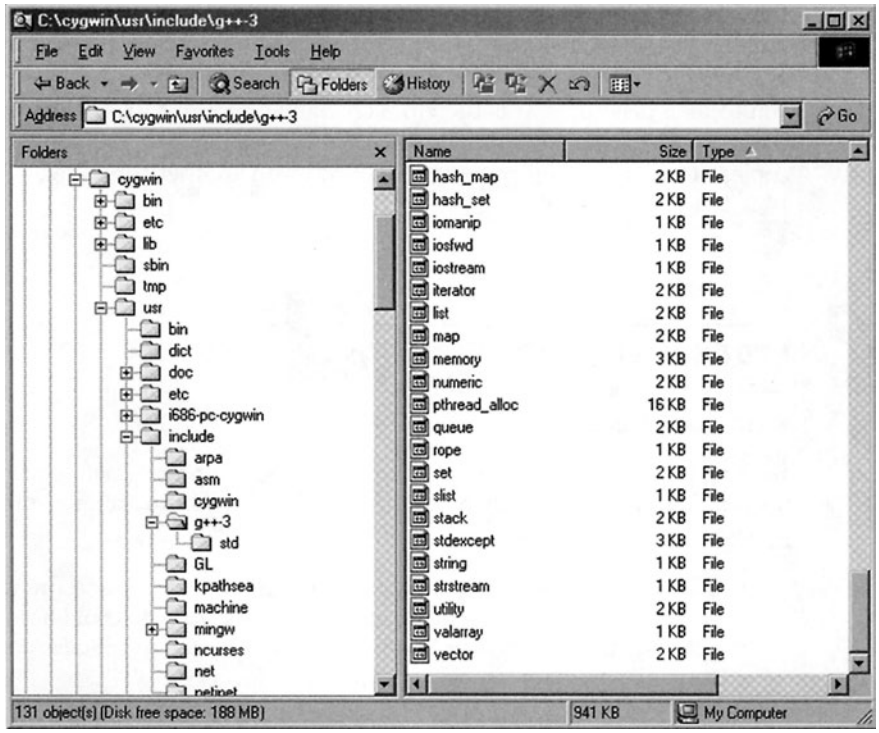

Figura 8

Uma hierarquia de diretório.

Passo 4 Escrever um programa simples

Na próxima seção apresentaremos um programa muito simples. Você irá aprender como digitá-lo, como executá-lo e como corrigir erros.

Passo 5 Salvar seu trabalho

Você irá gastar muitas horas digitando programas C++ e corrigindo-os. Os arquivos de programas resultantes possuem algum valor e você deve tratá-los como trataria outras propriedades importantes. Uma estratégia de segurança cuidadosa é particularmente importante para arquivos de computadores. Eles são mais frágeis do que documentos em papel ou outros objetos mais tangíveis. É fácil eliminar um arquivo por acidente e às vezes arquivos são perdidos devido a um mau funcionamento do computador.

A menos que mantenha outra cópia, possivelmente você terá de redigitar o conteúdo. Visto que dificilmente irá lembrar do arquivo inteiro, possivelmente precisará de tanto tempo quanto usou para fazer da primeira vez o conteúdo e as correções. Este tempo perdido poderá ocasionar a perda de prazos. Então torna-se crucialmente importante que você aprenda como proteger arquivos e adquira o hábito de fazer isto antes que o desastre aconteça. Você pode fazer cópias de segurança ou *backup* de arquivos por salvamento em um disquete ou em outro computador.

Dica de Produtividade 1.1

Cópias de Segurança

Fazer cópias de arquivos em disquetes é o método de armazenamento mais fácil e mais conveniente para a maioria das pessoas. Outra forma de *backup* que está aumentando em popularidade é o armazenamento de arquivos pela Internet. Seguem alguns pontos para manter em mente:

- *Faça* backup *com freqüência*. Fazer um *backup* de um arquivo leva poucos segundos e você irá odiar-se se tiver que gastar muitas horas recriando o trabalho que poderia ter salvo facilmente. Recomendo que você faça um *backup* de seu trabalho uma vez a cada trinta minutos e antes de cada vez que for executar um programa que escreveu.
- *Rotacione* backups. Use mais de um disquete para *backups*, em rodízio. Isto é, primeiro copie em um disquete e coloque-o de lado. Depois copie em um segundo disquete. Então use o terceiro e então retorne ao primeiro. Desta maneira você terá três *backups* recentes. Mesmo se um dos disquetes apresentar defeito, você pode usar os outros.
- *Somente faça* backup *de arquivos fonte.* O compilador traduz os arquivos que você escreveu para arquivos que consistem de código de máquina. Não existe necessidade de fazer *backup* de arquivos de código de máquina, visto que você pode facilmente recriá-los usando um compilador. Concentre sua atividade de *backup* nos arquivos que representam o seu esforço. Dessa maneira seus discos de *backup* não vão se encher de arquivos que você não necessita.
- *Preste atenção na direção do* backup. Fazer *backup* envolve copiar arquivos de um lugar para outro. É importante que você faça isto direito – isto é, copie de seu espaço de trabalho para a posição de *backup*. Se você fizer isto de forma incorreta, você poderá sobrescrever um novo arquivo com uma versão antiga.
- *Confira seus* backups *de vez em quando.* Verifique se seus *backups* estão onde você pensa que estão. Não existe nada mais frustrante que descobrir que seus *backups* não estão lá quando você precisa deles. Isso é particularmente verdadeiro se você usar um programa de *backup* que armazena arquivos em dispositivos desconhecidos (como uma fita de dados) ou em formato compactado.
- *Relaxe, depois restaure.* Quando você perde um arquivo e precisa restaurá-lo a partir do *backup*, possivelmente você estará nervoso e infeliz. Inspire longamente e pense sobre o processo de restauração antes de iniciá-lo. Não é incomum que um usuário agitado apague o último *backup* ao tentar restaurar um arquivo danificado.

1.8 Compilando um programa simples

Agora você está pronto para escrever e executar seu primeiro programa C++. A escolha tradicional para o primeiríssimo programa em uma nova linguagem de programação é um programa que exibe uma simples saudação: "Oi, Mundo!" Nós seguimos esta tradição. Aqui está o programa "Oi, Mundo!" em C++.

Arquivo oi.cpp

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7      cout << "Oi, Mundo!\n";
8      return 0;
9   }
```

Você pode buscar o arquivo deste programa no site da Web associado a este livro. Os números de linhas não fazem parte do programa. Eles são usados para que seu instrutor possa fazer referências a eles durante as aulas.

Vamos explicar o programa em um instante. Por hora, você deve fazer um novo arquivo de programa, e denominá-lo `oi.cpp`. Digite as instruções do programa, compile e execute o programa, seguindo os procedimentos adequados ao seu compilador.

A propósito, C++ é *sensível a maiúsculas e minúsculas*. Você deve digitar as letras maiúsculas e minúsculas exatamente como elas aparecem na listagem do programa. Você não pode digitar `MAIN` ou `Return`. Por outro lado, C++ possui *leiaute livre*. Espaços e quebras de linhas não são importantes. Você pode escrever o programa completo em uma única linha,

```
int main(){cout<<"Oi, Mundo!\n";return 0;}
```

ou escrever cada palavra chave em uma linha separada,

```
int
main()
{
cout
<<
"Oi, Mundo!\n"
;
return
0;
}
```

Entretanto, o bom gosto determina que você formate seus programas de um modo legível e, portanto, você deve seguir o leiaute da listagem.

Quando executar o programa, a mensagem

```
Oi, Mundo!
```

irá aparecer no vídeo. Em alguns sistemas, você pode necessitar mudar para uma janela diferente para encontrar a mensagem.

Agora que você já viu programa funcionando, é hora de entender como ele foi feito. A estrutura básica de um programa C++ é mostrada na Sintaxe 1.1.

A primeira linha,

```
#include <iostream>
```

instrui o compilador a ler o arquivo `iostream`. Esse arquivo contém a definição do pacote *stream input/output*. Seu programa realiza a entrada e saída no vídeo e portanto necessita dos serviços oferecidos por `iostream`. Você deve incluir este arquivo em todos os programas que lêem ou escrevem texto.

A propósito, você verá uma sintaxe ligeiramente diferente, `#include <iostream.h>`, em muitos programas C++. Veja o Tópico Avançado 1.1 para mais informações sobre esse assunto.

A próxima linha,

```
using namespace std;
```

diz ao compilador que todos os nomes que são usados no programa pertencem ao "ambiente de nomes padrão". Em programas grandes, é bastante comum que diferentes programadores usem os mesmos nomes para indicar coisas diferentes. Eles podem evitar conflitos de nomes usando ambientes de nomes separados.

Entretanto, para os programas simples que você escreverá neste livro, ambientes de nomes separados são desnecessários. Você sempre usará o ambiente de nomes padrão e pode simplesmente adicionar a diretiva `using namespace std;` no topo de cada programa que você escrever, logo abaixo das diretivas `#include`. Ambientes de nomes são uma facilidade recente de C++ e seu compilador poderá não suportá-la. O Tópico Avançado 1.1 instrui você a lidar com esta situação.

A construção

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

Sintaxe 1.1: Programa Simples

header files

```
using namespace std;
int main()
{
       statements
       return 0;
}
```

Example:

```
#include <iostream>
using namespace std;
int main()
{
       cout << "Oi, Mundo!\n";
       return 0;
}
```

Objetivo:

Um programa simples, com todas as instruções do programa em uma função `main`.

define uma *função* denominada `main`. Uma função é uma coleção de instruções de programação que realizam uma tarefa em particular. Cada programa C++ deve ter uma função `main`. A maioria dos programas C++ contém outras funções além da `main`, mas vamos demorar até o Capítulo 5 para discutir como escrever outras funções. As instruções ou *comandos* no *corpo* da função `main` — isto é, os comandos dentro das chaves { } — são executados um a um.

Note que cada comando termina com um ponto-e-vírgula.

```
cout << "Oi, Mundo!\n";
return 0;
```

A seqüência de caracteres delimitada por aspas

```
"Oi, Mundo!\n"
```

é chamada de *string*. Você deve colocar o conteúdo do *string* dentro de aspas de forma que o compilador saiba que você literalmente quer dizer `"Oi, Mundo!\n"`. Neste programa curto, realmente não existe a possibilidade de confusão. Suponha, por outro lado, que você quer exibir a palavra *main*. Estando delimitada por aspas, `"main"`, o compilador saberá que você deseja a seqüência de caracteres `m a i n`, e não a função denominada `main`. A regra é que você deve simplesmente colocar todos os textos entre aspas, de modo que o compilador os considere textos puros, e não instruções do programa.

O *string* de texto `"Oi, Mundo!\n"` não deve ser considerado *exatamente* assim. Você não quer que o esquisito `\n` apareça no vídeo. A seqüência de dois caracteres `\n` indica na realidade um caractere único, que não deve ser impresso, chamado de *nova linha*. Quando um caractere de nova linha é enviado para o vídeo, o cursor é movido para a primeira coluna da próxima linha do vídeo. Se você não enviar o caractere de nova linha, então o próximo item exibido simplesmente seguirá o *string* atual na mesma linha. Neste programa somente imprimimos um item, mas em geral queremos imprimir múltiplos itens, e é um bom hábito terminar todas as linhas de entrada com um caractere de nova linha.

O caractere de barra invertida \ é usado como um *caractere de escape*. A barra invertida não indica a si mesma; em vez disso, é usada para codificar outros caracteres que de outra maneira seriam difíceis ou impossíveis de mostrar em comandos do programa. Existem outras poucas combinações

de barra invertida que você encontrará mais adiante. Agora, o que você faz se realmente quiser mostrar uma barra invertida no vídeo? Você deve digitar duas, uma após a outra. Por exemplo,

```
cout << "Oi\\Mundo!\n";
```

imprimiria

```
Oi\Mundo!
```

Finalmente, como você pode exibir um *string* contendo aspas, como em

```
Oi, "Mundo"!
```

Você não pode usar

```
cout << "Oi, "Mundo"!\n";
```

Tão logo o compilador lê `"Oi, "`, ele pensa que o *string* terminou e então fica todo confuso sobre `Mundo` seguido de um segundo *string* `"!\n"`. Compiladores têm uma mente de uma trilha apenas e se uma simples análise da entrada não faz sentido para eles, eles simplesmente se recusam a prosseguir e exibem uma mensagem de erro. Em contraste, um humano provavelmente saberia que a segunda e a terceira aspa devem ser consideradas como parte do *string*. Bem, como nós podemos exibir aspas no vídeo? O caractere de *escape* barra invertida novamente surge para nos salvar. Dentro de um *string* a seqüência `\"` indica o literal aspa e não o final de um *string*. O comando de exibição correto então seria

```
cout << "Oi, \"Mundo\"!\n";
```

Para exibir valores no vídeo, você deve enviá-los para uma entidade chamada `cout`. O operador `<<` indica o comando "enviar para". Você também pode imprimir valores numéricos. Por exemplo, o comando

```
cout << 3 + 4;
```

exibe o número 7.

Finalmente, o comando `return` indica o fim da função `main`. Quando a função `main` termina, o programa termina. O valor zero é um sinal de que o programa foi executado com sucesso. Neste pequeno programa não existe nada que possa dar errado durante a execução. Em outros programas pode haver problemas com a entrada ou com algum dispositivo e então `main` retorna um valor diferente de zero para indicar um erro. A propósito, o `int` em `int main()` indica que `main` retorna um valor inteiro, não um número fracionário ou *string*.

Erro Freqüente 1.1

Omitir Ponto-e-Vírgulas

Em C++, cada comando deve terminar com um ponto-e-vírgula. Esquecer de digitar um ponto-e-vírgula é um erro freqüente. Isso confunde o compilador porque o compilador usa o ponto-e-vírgula para determinar onde termina um comando e inicia o próximo. O compilador não usa o final de linha ou chaves para reconhecer o final de comandos. Por exemplo, o compilador considera

```
cout << "Oi, Mundo!\n"
return 0;
```

um único comando, como se você tivesse escrito

```
cout << "Oi, Mundo!" return 0;
```

e então ele não entende o comando, por que ele não espera a palavra chave `return` no meio de um comando de saída. O remédio é simples. Simplesmente percorrer cada comando buscando por um ponto-e-vírgula terminal, da mesma forma que você verificaria se cada frase em português termina com um ponto.

Tópico Avançado 1.1

Diferenças entre Compiladores

Em algum ponto de um futuro próximo, todos os compiladores estarão aptos a traduzir programas que estão de acordo com o padrão C++. Entretanto, quando este livro estava sendo escrito, muitos compiladores falhavam em estar de acordo com o padrão de uma ou mais maneiras. Se o seu compilador não está plenamente de acordo, você necessitará mudar o código que está impresso neste livro. Existem aqui algumas incompatibilidades comuns. Os arquivos de cabeçalho de compiladores antigos têm uma extensão .h, por exemplo:

```
#include <iostream.h>
```

Se o seu compilador exige que você use `iostream.h` em vez de `iostream`, os programas neste livro possivelmente ainda funcionarão corretamente. Entretanto, simplesmente acrescentar um .h não funciona para todos os arquivos incluídos.

Por exemplo, em C++ padrão, você pode incluir facilidades de manipulação de *strings* com a diretiva:

```
#include <string>
```

Entretanto, a diretiva

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

não inclui os *strings* C++. Em vez disso, inclui os *strings* no estilo de C, que são completamente diferentes e não tão úteis.

Outro arquivo de cabeçalho comum contém as funções matemáticas. Em C++ padrão, você usa a diretiva

```
#include <cmath>
```

Em compiladores antigos, em vez disso você usa:

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

Compiladores antigos não suportam ambientes de nomes. Neste caso, omita a diretiva `using namespace std;`

1.9 Erros

Experimente um pouco com o programa de saudação. O que acontece se você cometer um erro de digitação, tal como:

```
cot << "Oi, Mundo!\n";
cout << "Oi, Mundo!\";
cout << "O, Mundo!\n";
```

No primeiro caso, o compilador irá reclamar. Ele irá dizer que não possui nenhuma pista do que você quer dizer com `cot`. O texto exato da mensagem de erro depende do compilador, mas pode ser algo como "Símbolo cot indefinido" (*Undefined symbol cot*). Esse é um *erro de compilação* ou *erro de sintaxe*. Algo está errado de acordo com as regras da linguagem e o compilador descobriu.

Quando o compilador descobre um ou mais erros, ele não traduz o programa para código de máquina e, em conseqüência, não existe programa para ser executado. Você deve corrigir o erro e compilar novamente. De fato, o compilador é bastante exigente e é comum passar por várias rodadas de correção de erros de compilação antes de conseguir uma primeira compilação com sucesso.

Se o compilador encontra um erro, ele não irá simplesmente parar e desistir. Ele irá tentar reportar tantos erros quantos ele puder encontrar, de modo que você possa corrigi-los todos de uma vez. Algumas vezes, no entanto, um erro o tira de seu caminho. Isso é provável de ocorrer com o erro da segunda linha. O compilador irá perder o fim do *string* por que ele pensa que o `\"` é um caractere aspa embutido. Em tais casos, é comum o compilador emitir mensagens de erros espúrias para as linhas vizinhas. Você pode corrigir somente aqueles erros cujas mensagens fazem sentido e então recompilar.

O erro na terceira linha é de outra espécie. O programa irá compilar e executar, mas sua saída será incorreta. Ele irá imprimir

```
O, Mundo!
```

Este é um *erro de execução* ou *erro de lógica*. O programa está sintaticamente correto e faz algo, mas não aquilo que deveria fazer. O compilador não consegue encontrar o erro, o qual deve ser eliminado quando o programa for executado, através de testes e cuidadosa conferência de sua saída.

Durante o desenvolvimento de um programa, erros são inevitáveis. Sempre que um programa for maior do que algumas poucas linhas, ele requer uma concentração sobre-humana para digitá-lo corretamente sem cometer nenhum deslize. Você vai se descobrir omitindo ponto-e-vírgulas ou apóstrofes com mais freqüência do que gostaria, mas o compilador vai encontrar esses problemas para você.

Erros de lógica são mais problemáticos. O compilador não vai encontrá-los — de fato, o compilador irá carinhosamente traduzir qualquer programa cuja sintaxe esteja correta — mas o programa resultante irá fazer algo errado. É responsabilidade do autor do programa testá-lo e encontrar quaisquer erros de lógica. O teste de programas é um tópico importante que você vai encontrar muitas vezes neste livro. Outro aspecto importante de um bom artesão é a programação defensiva: estruturar programas e processos de desenvolvimento de modo que um erro em um lugar de um programa não provoque uma resposta desastrosa.

Os exemplos de erros que você viu estão longe de serem difíceis de diagnosticar ou corrigir, mas assim que você aprender técnicas de programação mais sofisticadas, vai haver muito mais oportunidades de errar. É um fato desconfortável que localizar todos os erros em um programa é muito difícil. Mesmo que você possa observar que um programa exibe um comportamento errôneo, pode não ser óbvio qual parte do programa o causou e como corrigí-lo. Existem ferramentas de *software* especiais, os *depuradores*, que permitem que você prossiga através do programa para encontrar erros – isto é, erros de lógica. Neste livro você vai aprender como usar efetivamente um depurador.

Observe que todos estes erros são diferentes dos tipos de erros que você costuma fazer em cálculos. Se você totaliza uma coluna de números, pode esquecer de um sinal menos ou acidentalmente esquecer um "vai-um" porque você está aborrecido ou cansado. Computadores não cometem erros desse tipo. Quando um computador adiciona números, ele vai obter a resposta correta. É sabido que computadores podem cometer erros de estouro e de arredondamento, assim como as calculadoras fazem, quando você solicita que façam operações cujos resultados ultrapassem seus limites de representação de números. Um erro de estouro ocorre se um resultado de uma computação é muito grande ou muito pequeno. Por exemplo, a maioria dos computadores e calculadoras provoca estouro quando você tenta calcular 10^{1000}. Um erro de arredondamento ocorre quando um valor não pode ser representado precisamente. Por exemplo, $\frac{1}{3}$ pode ser armazenado no computador como 0.3333333, um valor que é próximo mas não exatamente igual. Se você calcular $1 - 3 \times \frac{1}{3}$, você pode obter 0.0000001, e não 0, como resultado de um erro de arredondamento. Vamos considerar este tipo de erro como erro de lógica, porque o programador poderia ter escolhido um esquema de cálculo mais apropriado que tratasse corretamente estouros e arredondamentos.

Você vai aprender neste livro uma estratégia de tratamento de erros em três partes. Primeiro, vai aprender sobre erros freqüentes e como evitá-los. Então você vai aprender estratégias de programação defensiva para minimizar a possibilidade e o impacto de erros. Finalmente, você vai aprender estratégias de depuração para retirar os erros que permanecerem.

Erro Freqüente 1.2

Erros de Ortografia

Se acidentalmente você erra uma palavra, coisas estranhas podem acontecer, e nem sempre será completamente óbvio o que aconteceu de errado a partir das mensagens de erro. Aqui está um bom exemplo de como simples erros de ortografia podem causar problemas:

```
#include <iostream>
using namespace std;
int Main()
{
    cout << "Oi, Mundo!\n";
    return 0;
}
```

Esse código define uma função chamada `Main`. O compilador não irá considerar que isto seja o mesmo que a função `main`, por que `Main` inicia com letra maiúscula e a linguagem C++ é sensível a maiúsculas e minúsculas. Letras maiúsculas e minúsculas são consideradas completamente diferentes entre si, e, para o compilador, `Main` não coincide com `main`, assim como `rain` não coincidiria. O compilador irá compilar sua função `Main`, mas quando o ligador estiver pronto para construir o arquivo executável, ele irá reclamar sobre a função `main` inexistente e se recusará a ligar o programa. Naturalmente, a mensagem "função `main` inexistente" (*missing* `main` *function*) deve dar a você uma pista sobre onde procurar o erro. Se você receber uma mensagem de erro que parece indicar que o compilador está na pista errada, é uma boa idéia verificar a ortografia e maiúsculas e minúsculas. Todas as palavras chave em C++ e os nomes da maioria das funções usam somente letras minúsculas.

Se você errar o nome de um símbolo, (por exemplo `out` em vez de `cout`), o compilador irá reclamar sobre um "símbolo indefinido" (*undefined symbol*). Esta mensagem de erro é geralmente uma boa pista de que você cometeu um erro de ortografia.

1.10 O processo de compilação

Alguns ambientes de desenvolvimento C++ são bem convenientes de usar. Você apenas entra com o código em uma janela, clica um botão ou menu para compilar, e clica em outro botão ou menu para executar o seu código. Mensagens de erro são mostradas em uma segunda janela, e o programa é executado em uma terceira janela. A Figura 9 mostra o leiaute de tela de um compilador C++ bastante popular, com estas características. Com um ambiente como este você está completamente isolado dos detalhes do processo de compilação. Em outros sistemas você deve incumbir-se de cada passo manualmente.

Mesmo que você use um ambiente C++ conveniente, é útil saber o que ocorre nos bastidores, principalmente porque conhecer o processo auxilia você a resolver problemas quando algo sai errado.

Você primeiro digita os comandos do seu programa em um editor de textos. O editor armazena o texto e dá um nome a ele, tal como `hello.cpp`. Se a janela do editor mostra um nome como `noname.cpp`, você deve trocar o nome. Você deve *salvar* o arquivo em disco freqüentemente, pois o editor somente salva o texto na memória RAM do computador. Se algo errado ocorrer com o computador e você precisar reiniciá-lo, o conteúdo da RAM (incluindo o texto de seu programa) é perdido, mas qualquer coisa armazenada em um disco rígido ou disquete é permanente, mesmo que você necessite reiniciar o computador.

Quando você compila o seu programa, o compilador traduz o *código fonte* C++ (isto é, os comandos que você escreveu) em um *código objeto*. O código objeto consiste de instruções de máquina e informações sobre como carregar o programa na memória antes da execução. O código objeto é armazenado em um arquivo separado, usualmente com a extensão `.obj` ou `.o`. Por exemplo, o código objeto para o programa *hello* pode ser armazenado como `hello.obj`.

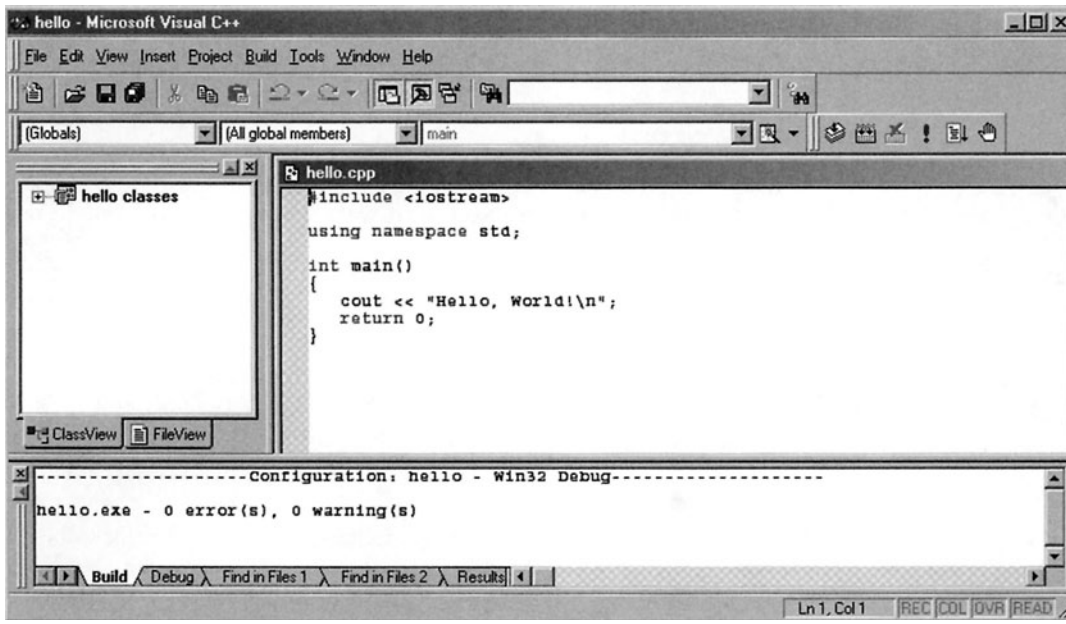

Figura 9

Leiaute de tela de um ambiente integrado C++.

O arquivo objeto contém somente a tradução do código que você escreveu. Isso não é suficiente para realmente executar o programa. Para exibir um *string* em uma janela, uma atividade de baixo nível é necessária. Os autores do pacote `iostream` (que define `cout` e sua funcionalidade), implementaram todas as ações necessárias e colocaram o código de máquina em uma *biblioteca*. Uma biblioteca é uma coleção de códigos que foi programada e traduzida por alguém, exatamente para que você use em seu programa (programas mais complicados são constituídos de mais de um arquivo fonte e de mais de uma biblioteca). Um programa especial denominado *ligador* pega seu arquivo objeto e as partes necessárias da biblioteca `iostream` e constrói um arquivo executável (a Figura 10 mostra uma visão geral destes passos). O arquivo executável é usualmente denominado de `hello.exe` ou `hello`, dependendo de seu sistema de computador. Ele contém todo o código de máquina necessário para executar o programa. Você pode executar o programa digitando `hello` no *prompt* de comando ou clicando no ícone do arquivo, mesmo depois de ter saído do ambiente C++. Você pode colocar o arquivo em um disquete e dá-lo a outro usuário que não possui um compilador C++ ou que pode não saber que existe algo como C++ e essa pessoa pode executar o programa da mesma maneira.

Sua atividade de programação concentra-se nestes arquivos. Você inicia no editor, escrevendo o arquivo fonte. Você compila o programa e olha as mensagens de erro. Você retorna ao editor e

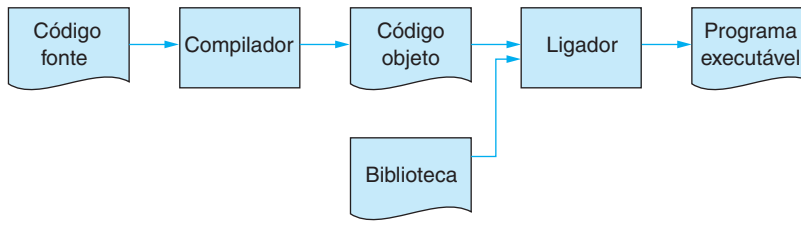

Figura 10

Do código fonte a um programa executável.

corrige os erros de sintaxe. Quando o compilador tem sucesso, o ligador constrói o arquivo executável. Você executa o arquivo executável. Se você encontrar um erro, pode executar o depurador para executar uma linha de cada vez. Uma vez encontrada a causa do erro, você retorna ao editor e corrige o erro. Você compila, liga e executa novamente para ver se o erro foi embora. Se não, você retorna ao editor. Isso é chamado de laço *edita-compila-depura* (ver a Figura 11). Você vai gastar uma quantidade substancial de tempo neste laço nos meses e anos que virão.

1.11 Algoritmos

Você logo vai aprender como programar cálculos e tomadas de decisões em C++. Mas antes de olhar a mecânica de implementação de cálculos no próximo capítulo, vamos examinar o processo de planejamento que antecede a implementação. Você pode já ter visto anúncios que encorajam a pagar por um serviço computadorizado que o(a) coloca em contato com um(a) amável parceiro(a). Vamos agora pensar como isto pode funcionar. Você preenche um formulário e o envia. Outros fazem o mesmo. Os dados são processados por um programa de computador. É razoável assumir que o computador pode realizar a tarefa de encontrar o melhor par para você? Suponha que seu irmão mais novo, e não o computador, tivesse todos os formulários em sua escrivaninha. Que instruções você daria a ele? Você não pode dizer "Encontre a pessoa mais bonita do sexo oposto que gosta de andar de *skate* e navegar pela Internet". Não existe um padrão para boa aparência e a opinião de seu irmão (ou de um programa de computador analisando uma foto digital) provavelmente será diferente da sua. Se você não pode dar instruções escritas para alguém resolver o problema, não há maneira pela qual o computador possa magicamente resolver o problema.

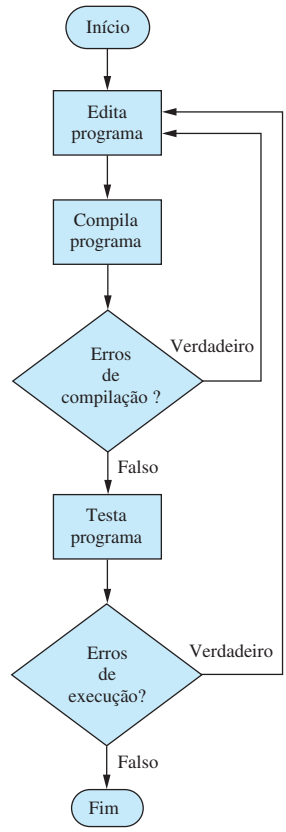

Figura 11

Laço edita-compila-depura.

O computador pode somente fazer aquilo que você diz para ele fazer. Ele somente o faz mais rápido, sem aborrecer-se ou cansar-se.

Agora vamos considerar o seguinte problema de investimento:

> Você coloca $10.000 em uma conta bancária que rende juros de 5% ao ano. Quantos anos são necessários para que o saldo da conta dobre o valor original?

Você poderia resolver esse problema manualmente? Sim, você pode. Você organiza o saldo da conta como segue:

Ano	Saldo
0	$10.000,00
1	$10.500,00 = $10.000,00 × 1,05
2	$11.025,00 = $10.500,00 × 1,05
3	$11.576,25 = $11.025,00 × 1,05
4	$12.155,06 = $11.576,25 × 1,05

Você continua calculando até que o saldo supere $20.000. Então o último número na coluna do ano é a resposta.

Naturalmente, ficar fazendo esses cálculos é extremamente aborrecido. Você pode mandar seu irmão menor fazer isto. Seriamente, o fato de uma computação ser aborrecida e tediosa é irrelevante para o computador. Computadores são muito bons na realização de cálculos repetitivos com rapidez e sem erros. O que é importante para o computador (e o seu irmão mais novo) é a existência de uma abordagem sistemática para encontrar a solução. A resposta pode ser encontrada seguindo uma série de passos que não envolve trabalho de adivinhação. Eis aqui uma série de passos como esta:

Passo 1 **Inicie com a tabela**

Ano	Saldo
0	$10.000,00

Passo 2 **Repita os passos 2a... 2c enquanto o saldo for menor que $20.000.**

Passo 2a **Adicione uma nova linha à tabela.**

Passo 2b **Na coluna 1 da nova linha, adicione mais um ao valor do ano.**

Passo 2c **Na coluna 2 da nova linha, coloque o valor do saldo anterior multiplicado por 1,05 (5%).**

Passo 3 **Use o último número da coluna do ano como o número de anos necessários para dobrar o investimento.**

Naturalmente, esses passos ainda não estão em uma linguagem que o computador possa entender, mas em breve você vai aprender como formular esses passos em C++. O que é importante é que o método descrito seja

- Sem ambigüidade.
- Executável.
- Finito.

O método é não *ambíguo* por que existem instruções precisas sobre o que fazer em cada passo e onde ir a seguir. Não existe margem para adivinhação ou criatividade. O método é *executável* por

que cada passo pode ser realizado na prática. Caso tivéssemos solicitado que fosse usada a taxa de juro real que você receberia nos próximos anos, e não uma taxa fixa de 5% ao ano, nosso método poderia não ser executável, pois não existe forma de alguém saber qual será a taxa. Finalmente, a computação irá em algum momento terminar. Com cada passo, o saldo aumenta pelo menos $500, de modo que em algum momento vai atingir $20,000.

Uma técnica de solução que seja sem ambigüidade, executável e finita é denominada de *algoritmo*. Encontramos um algoritmo para resolver nosso problema de investimento, e assim podemos encontrar uma solução com o computador. A existência de um algoritmo é um pré-requisito essencial para a tarefa de programação. Algumas vezes é muito simples encontrar um algoritmo. Outras vezes é preciso criatividade ou planejamento. Se você não conseguir encontrar um algoritmo, não poderá usar o computador para resolver o seu problema. Você precisa convencer a si mesmo que um algoritmo existe e que entendeu os seus passos, antes de iniciar a programar.

Resumo do capítulo

1. Computadores executam operações muito básicas em rápida sucessão. A seqüência de operações é denominada programa de computador. Diferentes tarefas (tais como controlar um talão de cheques, imprimir uma carta ou jogar um jogo) exigem diferentes programas. Programadores produzem programas de computador para fazer com que o computador realize novas tarefas.
2. A unidade central de processamento (UCP) do computador executa uma operação de cada vez. Cada operação especifica como os dados devem ser processados, como os dados devem ser enviados para a UCP ou como devem ser trazidos da UCP ou qual a próxima operação a ser selecionada.
3. Dados podem ser enviados à UCP, da memória ou de dispositivos de entrada como teclado, *mouse* ou um *link* de comunicação, para serem processados. A informação processada é devolvida pela UCP para a memória ou para dispositivos de saída como um vídeo ou uma impressora.
4. Dispositivos de memória incluem a memória de acesso randômico (RAM) e a memória secundária. A RAM é rápida, porém é cara e perde seu conteúdo quando a energia é desligada. Dispositivos de memória secundária usam tecnologia magnética ou ótica para armazenar informações. O tempo de acesso é mais lento, mas a informação é retida sem a necessidade de energia elétrica.
5. Programas de computador são armazenados como instruções de máquina em um código que depende do tipo do processador. Escrever diretamente códigos de instruções é difícil para programadores humanos. Cientistas de computação encontraram modos de tornar mais fácil esta tarefa, usando linguagens de montagem (*assembler*) ou linguagens de programação de alto nível. O programador escreve os programas em uma destas "linguagens" e um programa especial de computador o traduz para a seqüência equivalente de instruções de máquina. Instruções em linguagem de montagem são atreladas a um tipo particular de processador. Linguagens de alto nível são independentes de processador. O mesmo programa pode ser traduzido para ser executado em muitos processadores diferentes, de diferentes fabricantes.
6. Linguagens de programação são projetadas por cientistas de computação para diversas finalidades. Algumas linguagens são projetadas para finalidades específicas, tais como processamento de bancos de dados. Neste livro usamos C++, uma linguagem de uso geral que é adequada para uma grande variedade de tarefas de programação. C++ é popular por que é baseada na linguagem C, que já era largamente disseminada. Para ser eficiente e compatível com C, a linguagem C++ é menos elegante do que algumas linguagens projetadas desde a sua base, e programadores C++ devem conviver com algumas poucas concessões impróprias. Entretanto, muitas ferramentas excelentes oferecem suporte a C++.

7. Use algum tempo para familiarizar-se com o sistema de computador e com o compilador C++ que você vai usar para seus trabalhos de aula. Adote uma estratégia para manter cópias de segurança de seu trabalho antes que algum desastre ocorra.
8. Cada programa C++ contém diretivas `#include` para acessar os recursos necessários, como os de entrada e saída, e uma função denominada `main`. Em um programa simples, a função `main` somente exibe uma mensagem no vídeo e então retorna com um indicador de sucesso.
9. Erros são um fato da vida para um programador. Erros de sintaxe são construções errôneas que não seguem as regras da linguagem de programação. Eles são detectados pelo compilador, e nenhum programa é gerado. Erros de lógica são construções que podem ser traduzidas em um programa executável, mas o programa resultante não realiza a ação pretendida pelo programador. O programador é responsável por inspecionar e testar o programa para evitar erros de lógica.
10. Programas C++ são traduzidos para código de máquina por um programa denominado compilador. Em um passo separado, um programa denominado ligador constrói seu programa, combinando esse código de máquina com código de máquina previamente traduzido, para realizar entrada e saída e outros serviços.
11. Um algoritmo é uma descrição, sem ambigüidade, executável e finita, de passos para resolver um problema. Isto é, a descrição não dá margem para interpretação, os passos podem ser realizados na prática e o resultado garantidamente pode ser obtido após uma quantidade finita de tempo. Para resolver um problema em um computador, você deve conhecer um algoritmo para encontrar a solução.

Leituras complementares

[1] C. A. R. Hoare, "Hints on Programming Language Design", *Sigact/Sigplan Symposium on Principles of Programming Languages, outubro 1973*. Reimpresso em *Programming Languages, A Grand Tour*, ed. Ellis Horowitz, 3rd ed., Computer Science Press, 1987.

Exercícios de revisão

Exercício R1.1. Explique a diferença entre usar um programa de computador e programar um computador.

Exercício R1.2. Descreva as várias maneiras pelas quais um computador pode ser programado que foram discutidas neste capítulo.

Exercício R1.3. Que partes de um computador podem armazenar código de programa? Quais podem armazenar dados do usuário?

Exercício R1.4. Que partes de um computador servem para fornecer informações ao usuário? Quais partes obtém entradas do usuário?

Exercício R1.5. Classifique os dispositivos de memória que podem ser parte de um sistema de computador por (*a*) velocidade e (*b*) custo.

Exercício R1.6. Descreva a utilidade da rede de computadores no laboratório de seu departamento. A que outros computadores o computador do laboratório se conecta?

Exercício R1.7. Assuma que um computador possui as seguintes instruções de máquina, codificadas como números:

`160` *n*: Mova o conteúdo do registrador A para a posição de memória *n*.
`161` *n*: Mova o conteúdo da posição de memória *n* para o registrador A.
`44` *n*: Adicione o valor *n* ao registrador A.
`45` *n*: Subtraia o valor *n* do registrador A.
`50` *n*: Adicione o conteúdo da posição de memória *n* ao registrador A.

`51` *n*: Subtraia o conteúdo da posição de memória *n* do registrador A.
`52` *n*: Multiplique o registrador A pelo conteúdo da posição de memória *n*.
`53` *n*: Divida o registrador A pelo conteúdo da posição de memória *n*.
`127` *n*: Se o resultado da última computação é positivo, prossiga com a instrução que está armazenada na posição de memória *n*.
`128` *n*: Se o resultado da última computação é zero, prossiga com a instrução que está armazenada na posição de memória *n*.

Suponha que cada uma destas instruções e cada valor de *n* requer uma posição de memória. Escreva um programa em código de máquina para resolver o problema de dobrar o investimento.

Exercício R1.8. Projete instruções mnemônicas para os códigos de máquina do exercício anterior e escreva o programa de dobrar o investimento em código *assembler*, usando seus mnemônicos e um conjunto adequado de nomes simbólicos para variáveis e rótulos de comandos.

Exercício R1.9. Explique dois benefícios de linguagens de programação de alto nível em relação a código *assembler*.

Exercício R1.10. Liste as linguagens de programação mencionadas neste capítulo.

Exercício R1.11. Explique pelo menos duas vantagens e duas desvantagens de C++ em relação a outras linguagens de programação.

Exercício R1.12. Em seu próprio computador ou em um computador de seu laboratório, encontre a localização exata (pasta ou nome de diretório) do

(a) Arquivo do exemplo `hello.cpp`, que você escreveu com o editor
(b) Arquivo de cabeçalho `iostream`
(c) Arquivo de cabeçalho `ccc_time.h`, necessário para alguns programas deste livro

Exercício R1.13. Explique o papel especial do caractere de escape `\` em *strings* de caracteres C++.

Exercício R1.14. Escreva três versões do programa `hello.cpp` com diferentes erros de sintaxe. Escreva uma versão que contenha um erro de lógica.

Exercício R1.15. Como você descobre erros de sintaxe? Como você descobre erros de lógica?

Exercício R1.16. Escreva um algoritmo para resolver a seguinte questão: uma conta bancária inicia com $10.000. O juro é composto mensalmente a 6% por ano (0,5% ao mês). A cada mês, $500 são retirados para cobrir suas despesas escolares. Após quantos anos a conta é exaurida?

Exercício R1.17. Considere a questão do exercício anterior. Suponha que os valores ($10.000, 6%, $500) fossem selecionados pelo usuário. Existem valores para os quais o algoritmo que você desenvolveu não terminaria? Se isto for verdade, altere o algoritmo para assegurar-se que ele sempre termina.

Exercício R1.18. O valor de π pode ser computado segundo a seguinte fórmula:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} -$$

Escreva um algoritmo para computar π. Uma vez que esta é uma série infinita e um algoritmo deve parar após um número finito de passos, você deve parar quando o resultado tiver pelo menos 6 dígitos significativos.

Exercício R1.19. Suponha que você encarregou seu irmão mais novo de fazer cópias de segurança de seus trabalhos. Escreva um conjunto detalhado de instruções para realizar esta tarefa. Explique a freqüência com que ele deve fazer isto, e quais arquivos ele necessita copiar, a partir de qual pasta e para qual disquete. Explique como ele deve verificar se a cópia foi feita corretamente.

Exercícios de programação

Exercício P1.1. Escreva um programa que exibe uma mensagem "Oi, meu nome é Hal!". Então, em uma nova linha, o programa deve imprimir a mensagem "O que você gostaria que eu fizesse?". Então é a vez do usuário digitar uma entrada. Você ainda não aprendeu como fazer isto – apenas use as seguintes linhas de código:

```
string entrada_usuario;
getline(cin, entrada_usuario);
```

Finalmente, o programa deve ignorar a entrada do usuário e imprimir uma mensagem "Sinto muito, eu não posso fazer isto.".

Este programa usa o tipo de dado `string`. Para acessar este mecanismo, você deve colocar a linha

```
#include <string>
```

antes da função `main`.

Aqui está uma execução típica: A entrada do usuário está impressa em cinza.

```
Oi, meu nome é Hal!
O que você gostaria que eu fizesse?
A limpeza do meu quarto.
Sinto muito, eu não posso fazer isto.
```

Ao executar o programa, lembre de pressionar a tecla Enter depois de digitar a última palavra da linha de entrada.

Exercício P1.2. Escreva um programa que imprima uma mensagem "Oi, meu nome é Hal!". Então, em uma nova linha, programa deve imprimir a mensagem "Qual é o seu nome?". Como no Exercício P1.1, apenas use as seguintes linhas de código:

```
string nome_do_usuario;
getline(cin, nome_do_usuario);
```

Finalmente, o programa deve imprimir a mensagem "Oi, *nome do usuário*. Prazer em conhecê-lo!" Para imprimir o nome do usuário, simplesmente use

```
cout << nome_do_usuario;
```

Como no Exercício P1.1, você deve colocar a linha

```
#include <string>
```

antes da função `main`.
Aqui está uma execução do programa típica. A entrada do usuário está impressa em cinza.

```
Oi, meu nome é Hal!
Qual é o seu nome?
Dave
Oi, Dave. Prazer em conhecê-lo.
```

Exercício P1.3. Escreva um programa que calcula a soma dos primeiros dez inteiros positivos $1 + 2 + \cdots + 10$. *Dica*: Escreva um programa no formato

```
int main()
{
   cout <<
   return 0;
}
```

Exercício P1.4. Escreva um programa que calcula o *produto* dos primeiros dez inteiros positivos, $1 \times 2 \times \cdots \times 10$, e a soma de seus inversos $1/1 + 1/2 + \cdots + 1/10$. Isso é mais difícil do que parece. Primeiro, você deve saber que o símbolo `*`, e não um $\times$, é usado para multiplicação em C++. Tente escrever o programa, e confira os resultados usando uma calculadora. Os resultados do programa não serão exatamente corretos. Então escreva os números como números em *ponto flutuante*, `1.0`, `2.0`,..., `10.0`, e execute novamente o programa. Você pode explicar a diferença dos resultados? Vamos explicar este fenômeno no Capítulo 2.

Exercício P1.5. Escreva um programa que exiba seu nome dentro de um retângulo, na tela do terminal, como segue:

```
+------+
| Dave |
+------+
```

Esforce-se para desenhar linhas com caracteres tais como `|` `-` `+`.