

AGRADECIMENTOS

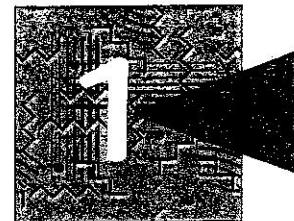
Eu gostaria de agradecer aos seguintes revisores, cujos comentários e conselhos ajudaram-me a melhorar este livro:

J. David Bezek, University of Evansville
Paul Chi, Bowie State University
Jonathan Graham, Norfolk State University
George Harrison, Norfolk State University
Tai-Chi Lee, Saginaw Valley State University
Martin Mansfield, Indiana Institute of Technology
Wilbur Powell, Austin College
Thomas Wolf, University of Pittsburgh
Xudong Yu, Southern Illinois University/Edwardsville

No entanto, o conteúdo final é de minha responsabilidade e eu apreciaria ouvir dos leitores sobre quaisquer falhas ou pontos fortes. Meu endereço de e-mail é drozdek@duq.edu.

Adam Drozdek

Programação Orientada a Objetos Usando C++



■ 1.1 TIPOS ABSTRATOS DE DADOS

Antes de um programa ser escrito, deveríamos ter uma idéia ótima de como realizar a tarefa que está sendo implementada por ele. Por isso, um delineamento do programa contendo seus requisitos deveria preceder o processo de codificação. Quanto maior e mais complexo o projeto, mais detalhada deveria ser a fase de delineamento. Os detalhes de implementação deveriam ser adiados para estágios posteriores do projeto. Em especial, os detalhes das estruturas de dados particulares a serem utilizadas na implementação não deveriam ser especificados no início.

Desde o início, é importante especificar cada tarefa em termos de entrada e de saída. Nos estágios iniciais, não deveríamos nos preocupar tanto em como o programa poderia ou deveria ser feito e sim focar nossa atenção no que ele deveria fazer. O comportamento do programa é mais importante do que as engrenagens do mecanismo que o executa. Por exemplo, se um item é necessário para realizar algumas tarefas, o item é especificado em termos das operações nele realizadas, em vez de em termos de sua estrutura interna. Essas operações podem atuar sobre esse item, por exemplo, modificando-o ou nele procurando alguns detalhes ou armazenando algo. Depois que essas operações são especificadas precisamente, a implementação do programa pode começar. A implementação decide que estrutura de dados deveria ser usada para tornar a execução mais eficiente em relação a tempo e espaço. Um item especificado em termos de operações é chamado de *tipo abstrato de dados*. Um tipo abstrato de dados não é parte de um programa, já que um programa escrito em linguagem de programação exige a definição de uma estrutura de dados, não apenas das operações nessa estrutura. No entanto, uma linguagem orientada a objetos (OO), tal como o C++, tem vínculo direto com os tipos abstratos de dados, implementando-os como uma classe.

■ 1.2 ENCAPSULAMENTO

A programação orientada a objetos (POO) gira ao redor do conceito de um objeto. Os objetos, no entanto, são criados usando-se uma definição de classe. Uma *classe* é um formato, de acordo com o qual os objetos são criados; é um trecho de software que inclui a especificação de dados e as funções que operam sobre esses

dados, e possivelmente sobre os dados que pertencem a outras instâncias de classe. As funções definidas em uma classe são chamadas de *métodos*, *funções-membro* ou *membros de função*, e as variáveis usadas em uma classe, de *membros de dados* (mais apropriadamente, eles deveriam ser chamados de *membros de dado*). Essa combinação de dados e das operações relacionadas é chamada de *encapsulamento* de dados. Um *objeto* é uma instância de uma classe, uma entidade criada usando uma definição de classe.

Em oposição às funções nas linguagens não-orientadas a objetos, os objetos fazem a conexão entre os dados e as funções-membro de maneira muito mais estreita e mais significativa. Em linguagens não-orientadas a objetos, as declarações dos dados e as definições das funções podem ser intercaladas por intermédio do programa inteiro, e somente a documentação do programa indica que há uma conexão entre eles. Em LOOs uma conexão é estabelecida logo no inicio; de fato, o programa é baseado nessa conexão. Um objeto é definido pelos dados e pelas operações relacionadas e, como pode haver muitos objetos usados no mesmo programa, os objetos se comunicam trocando mensagens que revelam um para o outro tão poucos detalhes sobre suas estruturas internas quanto necessário para uma comunicação adequada. A estruturação de programas em termos de objetos nos permite realizar diversos objetivos.

Primeiro, esse forte acoplamento dos dados e das operações pode ser usado muito melhor na modelagem de um fragmento do mundo, o que é enfatizado especialmente pela engenharia de software. Não é surpreendente que a POO tenha suas raízes na simulação, isto é, na modelagem de eventos do mundo real. A primeira LOO foi chamada Síntula e foi desenvolvida na década de 60 na Noruega.

Segundo, os objetos permitem que os erros sejam descobertos mais facilmente, porque as operações são localizadas dentro de seus objetos. Mesmo se efeitos colaterais ocorrem, eles são mais fáceis de se rastrear.

Terceiro, os objetos nos permitem esconder, de outros objetos, certos detalhes de suas operações, de modo que essas operações não podem ser desfavoravelmente afetadas por outros objetos. Isso é conhecido como o princípio de *ocultamento de informação*. Em linguagens não-orientadas a objetos esse princípio pode ser encontrado, até certo ponto, na forma de variáveis locais ou, como em Pascal, em funções ou procedimentos locais, que somente podem ser usados e acessados pela função que os define. No entanto, esse é um ocultamento muito pequeno ou mesmo um não-ocultamento. Algumas vezes acontece de precisarmos usar (novamente, como em Pascal) uma função *f2* definida em *f1* fora de *f1*, mas não podemos. Outras vezes podemos necessitar acessar alguns dados locais a *f1* sem exatamente conhecer a estrutura desses dados, mas não podemos. Por isso, alguma modificação é necessária e é realizada em LOOs.

Um objeto em uma LOO é como um relógio. Como usuários, estamos interessados no que mostram seus ponteiros, mas não em trabalho interno. Sabemos que existem engrenagens e molas dentro do relógio. Por usualmente sabermos muito pouco sobre por que todas essas peças estão em uma configuração particular, não devemos ter acesso a esse mecanismo para não danificá-lo, inadvertida ou propositalmente. Esse mecanismo está oculto de nós, não temos acesso imediato a ele e o relógio está protegido e trabalha melhor do que quando seu mecanismo é aberto por qualquer pessoa para vê-lo.

Um objeto é como uma caixa preta cujo comportamento é muito bem definido, e nós o usamos porque sabemos o que ele faz, não porque temos uma compreensão de como ele faz. Essa opacidade dos objetos é extremamente útil para fazer a sua manutenção independentemente uns dos outros. Se os canais de comunicação entre eles forem bem definidos, as mudanças feitas dentro de um objeto só poderão afetar outros objetos à medida que essas mudanças afetarem os canais de comunicação. Sabendo-se o tipo de informação enviada e recebida por um objeto, este pode ser substituído mais facilmente por um objeto mais adequado em uma situação particular: um novo objeto pode realizar a mesma tarefa diferentemente, mas mais rápido em um certo ambiente de hardware. Um objeto revela somente o necessário para o usuário utilizá-lo. Ele tem uma parte pública que pode ser acessada por qualquer usuário quando este envia uma mensagem que casa com qualquer dos nomes de função-membro revelados pelo objeto. Nessa parte pública, o objeto exibe ao usuário botões que podem ser apertados para invocar as operações do objeto. O usuário conhece somente os nomes dessas operações e o comportamento esperado.

O ocultamento de informação tende a enevoar a linha divisória entre os dados e as operações. Nas linguagens semelhantes ao Pascal a distinção entre dados e funções ou procedimentos é clara e rígida. Eles são definidos diferentemente e seu papéis são bastante distintos. As LOOs poem os dados e os métodos

juntos e, para o usuário do objeto, essa distinção é muito menos notável. Até certo ponto, isso incorpora as características de linguagens funcionais. A LISP, uma das mais antigas linguagens de programação, permite ao usuário tratar os dados e as funções similarmente, já que que a estrutura de ambos é a mesma.

Já fizemos uma distinção entre objetos particulares e tipos ou classes de objeto. Escrevemos funções para serem usadas com diferentes variáveis e, por analogia, não gostamos de ser forçados a escrever tantas declarações de objetos quanto o número de objetos exigidos pelo programa. Certos objetos são do mesmo tipo e gostaríamos somente de usar uma referência a uma especificação geral de objetos. Para variáveis isoladas, fazemos uma distinção entre a declaração de tipo e a declaração de variável. No caso de objetos, temos uma declaração de classe e uma instância de objeto. Por exemplo, na seguinte declaração de classe, C é uma classe e objeto1 até objeto3 são objetos.

```
class C {
public:
    C(char *s = "", int i = 0, double d = 1) {
        strcpy(dadosMembro1,s);
        dadosMembro2 = i;
        dadosMembro3 = d;
    }
    void funcaoMembro1() {
        cout << dadosMembro1 << ' ' << dadosMembro2 << ' '
            << dadosMembro3 << endl;
    }
    void funcaoMembro2(int i, char *s = "desconhecido") {
        dadosMembro2 = i;
        cout << i << " recebido de " << s << endl;
    }
protected:
    char dadosMembro1[20];
    int dadosMembro2;
    double dadosMembro3;
};

C objeto1("objeto1",100,2000), objeto2("objeto2"), objeto3;
```

A *Passagem de Mensagem* é equivalente a uma chamada de função em linguagens tradicionais. No entanto, para acentuar o fato de que em LOOs as funções-membro são relativas aos objetos, esse novo termo é usado. Por exemplo, a chamada à funcaoMembro() pública no objeto1

```
objeto1.funcaoMembro();
```

é vista como a mensagem funcaoMembro() enviada ao objeto1. Ao receber a mensagem, o objeto1 invoca sua função-membro e exibe todas as informações relevantes. As mensagens podem incluir parâmetros, de modo que

```
objeto1.funcaoMembro2(123);
```

é a mensagem funcaoMembro2() com o parâmetro 123 recebida pelo objeto1.

As linhas que contêm essas mensagens podem estar no programa principal, em uma função ou em uma função-membro de um outro objeto. Assim, o receptor da mensagem é identificável, mas não necessariamente o expedidor. Se o objeto1 recebe a mensagem funcaoMembro1(), ele não sabe onde a mensagem é originada. Ele somente responde a ela exibindo a informação que a funcaoMembro1() encapsula. O

mesmo acontece com a funçãoMembro2(). Em consequência, o expedidor pode preferir enviar uma mensagem que também inclua sua identificação, como a seguir:

```
objeto1.funçãoMembro2(123, "objeto12");
```

Uma característica poderosa do C++ é a possibilidade de declarar classes genéricas usando-se parâmetros de tipos nas declarações de classes. Por exemplo, se necessitarmos declarar uma classe que usa uma matriz para armazenar alguns itens, então podemos declarar essa classe como

```
class intClasse {
    int estocagem[50];
    .....
};
```

No entanto, desse modo limitamos a usabilidade dessa classe somente para números inteiros; se necessitarmos uma classe que realize as mesmas operações que intClasse, exceto que opere em números flutuantes, então uma nova declaração é necessária, tal como

```
class flutClasse {
    float estocagem[50];
    .....
};
```

Se estocagem é para conter estruturas ou ponteiros para caracteres, então mais duas classes precisam ser declaradas. É muito melhor declarar uma classe genérica e decidir a que tipo de itens que o objeto está se referindo somente quando se estiver definindo o objeto. Afortunadamente, o C++ nos permite declarar uma classe nessa forma, e a declaração para o exemplo é

```
template<class genTipo>
class genClasse {
    genTipo estocagem[50];
    .....
};
```

Mais tarde, tomamos a decisão de como inicializar genTipo:

```
genClasse<int> intObjeto;
genClasse<float> flutObjeto;
```

Essa classe genérica se manifesta em diferentes formas, dependendo da declaração específica. Uma declaração genérica é suficiente para habilitar tais diferentes formas.

Podemos ir mais adiante, não nos comprometendo com 50 células em estocagem e postergando essa decisão até o estágio de definição de objeto. Por via das dúvidas, podemos deixar um valor default, de modo que a declaração de classe passa a ser

```
template<class genTipo, int tamanho = 50>
class genClasse {
    genTipo estocagem[tamanho];
    .....
};
```

Agora a definição do objeto é

```
genClasse<int> intObjeto1; // usa o tamanho default;
genClasse<int,100> intObjeto2;
genClasse<float,123> flutObjeto;
```

Esse método de usar tipos genéricos não está limitado somente às classes; podemos usá-lo nas declarações de funções. Por exemplo, a operação padrão para se trocar dois valores pode ser definida pela função

```
template<class genTipo>
void troca(genTipo& el1, genTipo& el2) {
    genTipo tmp = el1; el1 = el2; el2 = tmp;
}
```

Esse exemplo indica também a necessidade de adaptar operadores predefinidos a situações específicas. Se genTipo for um número, um caractere ou uma estrutura, o operador de atribuição, =, realiza sua função apropriadamente. Mas se genTipo for uma matriz, podemos esperar problemas em troca(). O problema pode ser resolvido sobrecregendo-se o operador de atribuição, adicionando-lhe a funcionalidade exigida por um tipo de dado específico.

Depois que uma função genérica foi declarada, uma função apropriada pode ser gerada em tempo de compilação. Por exemplo, se o compilador vê duas chamadas,

```
troca(n,m); // troca dois valores inteiros
troca(x,y); // troca dois valores flutuantes
```

ele gera duas funções de troca para serem usadas durante a execução do programa.

1.3 HERANÇA

As LOOs permitem criar uma hierarquia de classes de modo que os objetos não tenham que ser instâncias de uma classe simples. Antes de discutir o problema da herança, considere as seguintes definições de classe:

```
class ClasseBase {
public:
    ClasseBase(){}
    void f(char *s = "desconhecido"){
        cout << "A função f() em ClasseBase chamada de " << s << endl;
        h();
    }
protected:
    void g(char *s = "desconhecido"){
        cout << "A função g() em ClasseBase chamada de " << s << endl;
    }
private:
    void h(){
        cout << "A função h() em ClasseBase\n";
    }
};

class DerivadaNivel1 : public virtual ClasseBase{
public:
    void f(char *s = "desconhecido"){
        cout << "A função f() em DerivadaNivel1 chamada de " << s << endl;
        g("DerivadaNivel1");
        h("DerivadaNivel1");
    }
    void h(char *s = "desconhecido"){

    }
};
```

```

        cout << "A função h() em Derivada1Nivel1 chamada de " << s << endl;
    }

class Derivada2Nivel1 : public virtual ClasseBase {
public:
    void f(char *s = "desconhecido"){
        cout << "A função f() em Derivada2Nivel1 chamada de " << s << endl;
        g("Derivada2Nivel1");
        // h(); //erro: ClasseBase::h() não está acessível
    }
};

class DerivadaNivel2 : public Derivada1Nivel1, public Derivada2Nivel1 {
public:
    void f(char *s = "desconhecido"){
        cout << "A função f() em DerivadaNivel2 chamada de " << s << endl;
        g("DerivadaNivel2");
        Derivada1Nivel1::h("DerivadaNivel2");
        ClasseBase::f("DerivadaNivel2");
    }
};

```

Um programa de amostra é

```

void main() {
    ClasseBase cb;
    Derivada1Nivel1 d11;
    Derivada2Nivel1 d21;
    DerivadaNivel2 d12;
    cb.f("main(1)");
    // cb.g(); // erro: ClasseBase::g() não está acessível
    // cb.h(); // erro: ClasseBase::h() não está acessível
    d11.f("main(2)");
    // d11.g(); // erro: ClasseBase::g() não está acessível
    d11.h("main(3)");
    d21.f("main(4)");
    // d21.g(); // erro: ClasseBase::g() não está acessível
    // d21.h(); // erro: ClasseBase::h() não está acessível
    d12.f("main(5)");
    // d12.g(); // erro: ClasseBase::h() não está acessível
    d12.h();
}

```

Essa amostra produz a seguinte saída:

```

A função f() em ClasseBase, chamada de main(1)
A função h() em ClasseBase
A função f() em Derivada1Nivel1, chamada de main(2)
A função g() em ClasseBase, chamada de Derivada1Nivel1
A função h() em Derivada1Nivel1, chamada de Derivada1Nivel1
A função h() em Derivada1Nivel1, chamada de main(3)

```

```

A função f() em Derivada2Nivel1, chamada de main(4)
A função g() em ClasseBase, chamada de Derivada2Nivel1
A função f() em DerivadaNivel2, chamada de main(5)
A função g() em ClasseBase, chamada de DerivadaNivel2
A função h() em Derivada1Nivel1, chamada de DerivadaNivel2
A função f() em ClasseBase, chamada de DerivadaNivel2
A função h() em ClasseBase
A função h() em Derivada1Nivel1, chamada de desconhecido

```

A classe *ClasseBase* é chamada de *classe base* ou *superclasse*, e outras classes são chamadas de *sub-classes* ou *classes derivadas* porque derivam da superclasse, por poderem usar os membros de dados e as funções-membro especificados na *ClasseBase* como *protected* (protegido) ou *public* (público). Elas herdam todos esses membros a partir de sua classe base, de modo que não tenham que repetir as mesmas definições. No entanto, uma classe derivada pode sobrepor a definição de uma função-membro introduzindo sua própria definição. Assim, tanto a classe base como a classe derivada têm alguma medida de controle sobre suas funções-membro.

A classe base pode decidir quais funções-membro e quais membros de dados podem ser revelados para as classes derivadas, de modo que o princípio de ocultamento de informação se mantenha não só entre a classe base e suas subclasses, mas também entre as classes derivadas. Além disso, a classe derivada pode decidir quais partes das funções-membro e dos membros de dados públicos e protegidos deve reter e usar e quais deve modificar. Por exemplo, tanto *Derivada1Nivel1* como *Derivada2Nivel1* definem suas próprias versões de *f()*. No entanto, o acesso à função-membro com o mesmo nome em qualquer das classes mais altas na hierarquia é ainda possível, precedendo-se a função com o nome da classe e com o operador de escopo, como mostrado na chamada de *ClasseBase::f()* a partir de *f()* em *DerivadaNivel2*.

Uma classe derivada pode adicionar alguns de seus novos membros. Tal classe pode se tornar uma classe base para outras classes que podem ser derivadas a partir dela, de modo que a hierarquia de heranças possa ser deliberadamente estendida. Por exemplo, a classe *Derivada1Nivel1* é derivada de *ClasseBase*, mas, ao mesmo tempo, é a classe base para a *DerivadaNivel2*.

A herança em nossos exemplos está especificada como pública, usando-se a palavra *public* depois do ponto-e-vírgula no cabeçalho de definição de uma classe derivada. A herança pública significa que os membros públicos da classe base são também públicos na classe derivada, e que os membros protegidos são também protegidos. Em caso de herança protegida (com a palavra *protected* no cabeçalho da definição), tanto os membros públicos como os protegidos da classe base se tornam protegidos na classe derivada. Finalmente, para a herança privada, tanto os membros públicos como os protegidos da classe base se tornam privados na classe derivada. Em todos os tipos de herança, os membros privados da classe base são inacessíveis para quaisquer classes derivadas. Por exemplo, uma tentativa de chamar *h()* a partir de *f()* em *Derivada2Nivel1* causa um erro de compilação, "ClasseBase::h() não está acessível". No entanto, uma chamada de *h()* a partir de *f()* em *Derivada1Nivel1* não causa problema porque ela é uma chamada a *h()* definida em *Derivada1Nivel1*.

Os membros protegidos da classe base são acessíveis somente para as classes derivadas, e não para as classes não derivadas. Por essa razão, tanto *Derivada1Nivel1* como *Derivada2Nivel1* podem chamar a função-membro *g()* protegida da *ClasseBase*, mas uma chamada para essa função a partir de *main()* é interpretada como ilegal.

Uma classe derivada não tem que estar limitada somente a uma única classe. Ela pode ser derivada a partir de mais de uma classe base. Por exemplo, *DerivadaNivel2* está definida como uma classe derivada tanto de *Derivada1Nivel1* como de *Derivada2Nivel1*, herdando desse modo, todas as funções-membro de *Derivada1Nivel1* e de *Derivada2Nivel1*. No entanto, *DerivadaNivel2* também herda as mesmas funções-membro da *ClasseBase* duas vezes, porque ambas as classes usadas na definição de *DerivadaNivel2* são derivadas a partir de *ClasseBase*. Isso é redundante no melhor caso e, no pior, pode causar o erro de compilação "membro é ambígua em ClasseBase::g() e ClasseBase::g()". Para

evitar que isso aconteça, as definições das duas classes incluem o modificador `virtual`, que significa que `DerivadaNivel2` contém somente uma cópia de cada função-membro de `ClasseBase`. Um problema similar surge se `f()` em `DerivadaNivel2` chama `h()` sem o precedente operador de escopo e nome de classe, `DerivadaNivel1::h()`. Não importa que `h()` seja privada em `ClasseBase` e inacessível a `DerivadaNivel2`. Um erro será impresso: "membro é ambíguo em `DerivadaNivel1::h()` e `ClasseBase::h()`".

1.4 PONTEIROS

As variáveis usadas em um programa podem ser consideradas como caixas que nunca estão vazias; são preenchidas com algum conteúdo tanto pelo programador como – se não inicializadas – pelo sistema operacional. Tal variável tem pelo menos dois atributos: o conteúdo (ou valor) e a localização da caixa (ou variável) na memória do computador. Esse conteúdo pode ser um número, um caractere ou um item composto como uma estrutura ou uma união. No entanto, esse conteúdo pode ser também a localização de outra variável, e as variáveis com esse conteúdo são chamadas de *ponteiros*. Os ponteiros usualmente são variáveis auxiliares que nos permitem acessar indiretamente os valores de outras variáveis. Um ponteiro é análogo a uma sinalização de estrada que nos leva para um certo local, ou a uma tira de papel na qual um endereço tenha sido anotado. São variáveis que levam a variáveis, humildes auxiliares que apontam para outras variáveis como foco de atenção.

Por exemplo, na declaração

```
int i = 15, j, *p, *q;
```

`i` e `j` são variáveis numéricas e `p` e `q` são ponteiros para números, onde o asterisco à frente de `p` e de `q` indica sua função. Assumindo que os endereços das variáveis `i`, `j`, `p` e `q` sejam 1080, 1082, 1084 e 1086, depois de se atribuir 15 para `i` na declaração, as posições e os valores das variáveis na memória do computador estão como na Figura 1.1a.

Agora, poderíamos fazer a atribuição `p = i`, (ou `p = (int*) i`, se o compilador não aceitar isso), mas a variável `p` foi criada para estocar o endereço de uma variável de número inteiro, não o seu valor. Em consequência, a atribuição apropriada é `p = &i`, onde o E comercial (&) à frente do `i` significa que o endereço de `i` é que se tem em vista, e não o seu conteúdo. A Figura 1.1b ilustra essa situação. Na Figura 1.1c, a seta de `p` para `i` indica que `p` é um ponteiro que contém o endereço de `i`.

Temos que saber distinguir o valor de `p`, que é um endereço, do valor do local cujo endereço o ponteiro contém. Por exemplo, para atribuir 20 à variável apontada por `p`, a declaração de atribuição é

```
*p = 20;
```

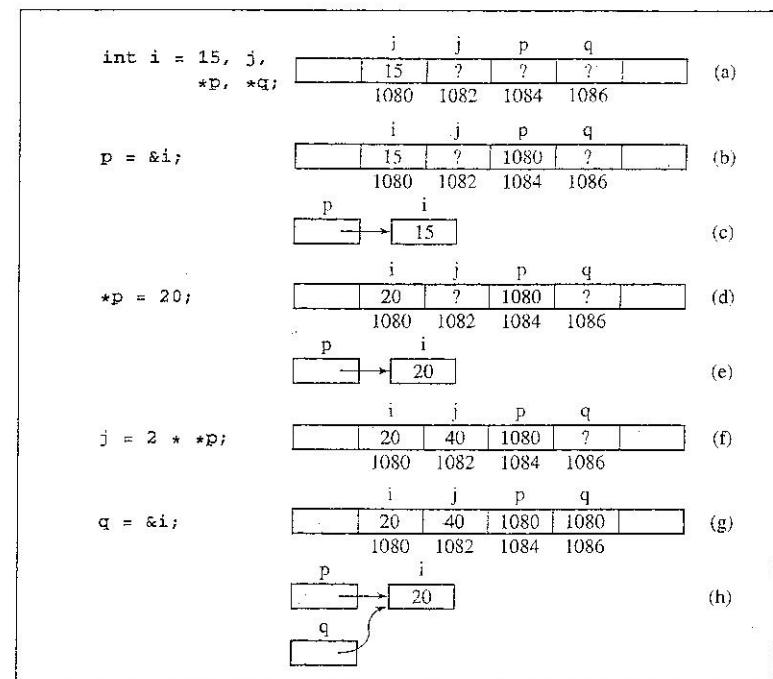
O asterisco (*) aqui é um operador de indireção primeiro que força o sistema a buscar o conteúdo de `p` e então acessar o local cujo endereço acabou de ser recuperado de `p`, e somente depois atribuir 20 a esse local (Figura 1.1d). As Figuras 1.1e até 1.1n fornecem mais exemplos de declarações de atribuição e de como os valores estão estocados na memória do computador.

De fato os ponteiros – como todas as variáveis – também têm dois atributos: um conteúdo e uma localização. Essa localização pode ser estocada em outra variável, que então se torna um ponteiro para um ponteiro.

Na Figura 1.1 os endereços das variáveis foram atribuídos para ponteiros. Os ponteiros podem, no entanto, se referir a localizações anônimas que são acessíveis somente por meio de seus endereços, e não – como as variáveis – por seus nomes. Esses locais precisam ser reservados pelo gerenciador de memória, o que é realizado dinamicamente durante a operação do programa, diferentemente das variáveis, cujas localizações são alocadas no tempo de compilação.

FIGURA 1.1

Mudanças de valores depois que atribuições são efetuadas usando as variáveis de ponteiro. Note que (b) e (c) mostram a mesma situação e assim também (d) e (e), (g) e (h), (i) e (j), (k) e (l) e (m) e (n).



Para se alocar e desalocar memória dinamicamente, duas funções são usadas. Uma função, `new`, toma da memória tanto espaço quanto necessário para estocar um objeto cujo tipo segue a palavra `new`. Por exemplo, com a instrução

```
p = new int;
```

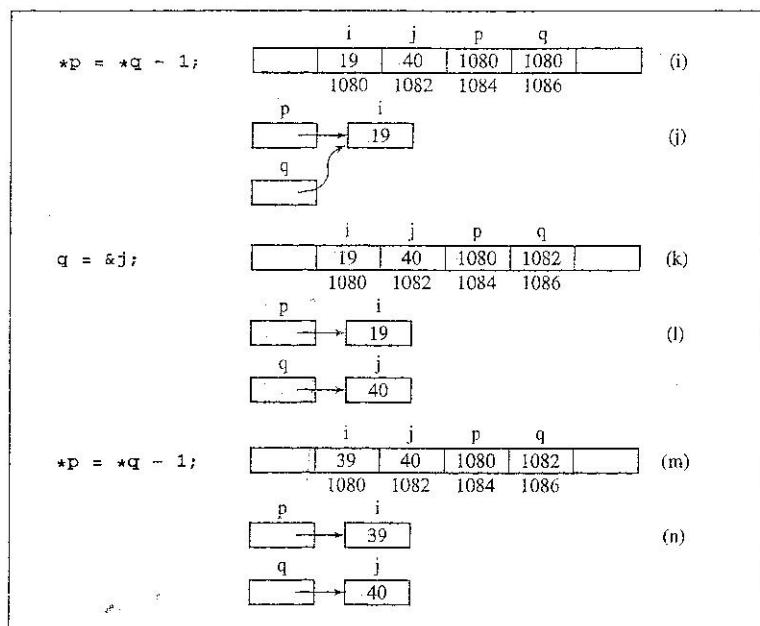
o programa solicita do gerenciador de memória espaço suficiente para estocar um valor inteiro e o endereço dessa porção de memória fica estocado em `p`. Agora os valores podem ser atribuídos ao bloco de memória apontado por `p` apenas indiretamente, por meio de um ponteiro, tanto o ponteiro `p` como qualquer outro ponteiro `q` a que tenha sido atribuído o endereço estocado em `p` com a atribuição `q = p`.

Se o espaço ocupado pelo inteiro acessível a partir de `p` não é mais necessário, ele pode ser retornado à área de posições livres de memória gerenciada pelo sistema operacional, mediante a instrução

```
delete p;
```

No entanto, depois de executar essa instrução, os endereços do bloco de memória liberado estão ainda em `p`, embora o bloco no que diz respeito ao programa não exista mais. É como se tratasse de um endereço de uma casa que tivesse sido demolida, ou seja, o endereço de um local existente. Se usarmos esse endereço para encontrar algo o resultado pode ser facilmente previsto. Similarmente, se depois de emitirmos a declaração `delete` não apagarmos o endereço da variável ponteiro que participa da supressão do bloco de me-

FIGURA 1.1 (continuação)



mória, o resultado é potencialmente perigoso e podemos fazer o programa entrar em colapso quando tentamos acessar locais inexistentes, particularmente para objetos mais complexos do que valores numéricos. Isso é o que se chama de *problema de referência pendente*. Para evitar esse problema, um endereço tem que ser atribuído ao ponteiro; se não puder ser um endereço de algum local, deverá ser um endereço nulo, que é simplesmente 0. Depois da execução da atribuição

```
p = 0;
```

não podemos dizer que p se refere a nulo ou que aponta para nulo, mas que p se torna nulo ou é nulo.

1.4.1 Ponteiros e Matrizes

No último exemplo o ponteiro p se refere a um bloco de memória que contém um valor inteiro. Uma situação mais interessante é quando um ponteiro se refere a uma estrutura de dados que é criada e modificada dinamicamente. Essa é uma situação que gostaríamos de ter para superar as restrições impostas pelas matrizes. As matrizes no C++ e na maioria das linguagens de programação têm que ser declaradas antecipadamente. Em consequência, seus tamanhos devem ser conhecidos antes que o programa inicie. Isso significa que o programador necessita de um conhecimento profundo do problema que está sendo programado, para escolher o tamanho certo da matriz. Se for muito grande a matriz ocupará espaço de memória desnecessariamente, o qual será basicamente desperdiçado. Se o espaço de memória for muito pequeno, a matriz

poderá transbordar com dados e o programa irá abortar. Algumas vezes o tamanho da matriz simplesmente não pode ser previsto. Conseqüentemente, a decisão é adiada até o tempo de execução, e então memória suficiente é alocada para conter a matriz.

O problema é resolvido com o uso de ponteiros. Considerando a Figura 1.1b, observe que o ponteiro p aponta para o local 1080. Mas ele permite também acessar os locais 1082, 1084 e assim por diante, porque os locais são uniformemente espaçados. Por exemplo, para acessar o valor da variável j, que é uma vizinha de i, é suficiente adicionar o tamanho de uma variável do tipo inteiro ao endereço de i estocado em p para acessar o valor de j também de p. Esse é basicamente o modo como o C++ manuseia as matrizes.

Considere as seguintes declarações:

```
int a[5], *p;
```

As declarações especificam que a é um ponteiro para um bloco de memória que pode conter cinco inteiros. O ponteiro é fixo; a deve ser tratada como uma constante, de modo que qualquer tentativa de se atribuir um valor para a, como em

```
a = p;
```

ou em

```
a++;
```

é considerada um erro de compilação. Como a é ponteiro, a notação de ponteiro pode ser usada para acessar as células da matriz a. Por exemplo, a notação de matriz usada no laço que soma todos os números em a,

```
for (soma = a[0], i = 1; i < 5; i++)
    soma += a[i];
```

pode ser substituída pela notação de ponteiro

```
for (soma = *a, i = 1; i < 5; i++)
    soma += *(a+i);
```

ou por

```
for (soma = *a, p = a+1; p < a+5; p++)
    soma += *p;
```

Note que a+1 é o local da próxima célula da matriz a, de modo que a+1 é equivalente a &a[1]. Assim, se a vale 1020, então a+1 não é 1021, mas 1022, porque a aritmética de ponteiros depende do tipo da entidade apontada. Por exemplo, depois das declarações

```
char b[5];
long c[5];
```

e assumindo que b vale 1050 e que c vale 1055, b+1 vale 1051, porque um caractere ocupa um byte, e c+1 vale 1059, porque um número longo ocupa quatro bytes. A razão para esses resultados da aritmética de ponteiros é que a expressão c+1 denota o endereço de memória c+i*sizeof(long).

Nesta discussão a matriz a é declarada estaticamente, especificando-se em sua declaração que ela contém cinco células. O tamanho da matriz é fixo para a duração da operação do programa. Matrizes podem também ser declaradas dinamicamente. Para esse fim as variáveis de ponteiros são usadas. Por exemplo, a atribuição

```
p = new int[n];
```

aloca espaço suficiente para estocar n inteiros. O ponteiro p pode ser tratado como uma variável matriz de modo que a notação de matriz pode ser usada. Por exemplo, a soma dos números da matriz p pode ser obtida com o código que usa a notação de matriz

```
for (soma = p[0], i = 1; i < n; i++)
    soma += p[i];
```

uma notação de ponteiro que é uma versão direta do laço anterior,

```
for (soma = *p, i = 1; i < n; i++)
    soma += *(p+i);
```

ou uma notação de ponteiro que usa dois ponteiros,

```
for (soma = *p, q = p+1; q < p+n; q++)
    soma += *q;
```

Devido a *p* ser uma variável, pode-se atribuir a ela uma nova matriz. Se a matriz correntemente apontada por *p* não for mais necessária, ela deve ser removida pela instrução

```
delete [] p;
```

Note o uso de colchetes vazios nessa instrução. Os colchetes indicam que *p* aponta para uma matriz.

Um tipo muito importante de matrizes são cadeias de caracteres (strings) ou matrizes de caracteres. Existem muitas funções predefinidas que operam sobre cadeias de caracteres. Os nomes dessas funções começam com *str*, como em *strlen(s)*, que encontra o comprimento da cadeia de caracteres *s*, ou *strcpy(s1, s2)*, para copiar a cadeia *s2* para *s1*. É importante lembrar que todas essas funções assumem que as cadeias de caracteres são terminadas com o caractere nulo '\0'. Por exemplo, *strcpy(s1, s2)* copia até encontrar esse caractere em *s2*. Se um programador não inclui esse caractere em *s2*, a cópia termina quando a primeira ocorrência desse caractere é encontrada em algum lugar na memória do computador, depois da localização de *s2*. Isso significa que a cópia é realizada para locais fora de *s1*, o que eventualmente pode levar o programa ao colapso.

1.4.2 Ponteiros e Construtores de Cópias

Alguns problemas podem surgir quando os membros de dados não são manipulados apropriadamente quando se está copiando dados de um objeto para outro. Considere a seguinte definição:

```
//foi mantida a notação inglesa para no (node), para se evitar ambiguidade
struct Node {
    char *nome;
    int idade;
    Node(char *n = "", int a = 0){
        nome = new char[strlen(n)+1];
        strcpy(nome,n);
        idade = a;
    }
};

A intenção das declarações

    Node node1("Roger",20), node2(node1); //ou node2 = node1;
```

é criar o objeto *node1*, atribuir valores aos dois membros de dados em *node1*, e então criar o objeto *node2* e inicializar seus membros de dados para os mesmos valores que em *node1*. Esses objetos existem para ser entidades independentes de modo que atribuindo-se valores a um deles não se deve afetar valores no outro. No entanto, depois das atribuições

```
strcpy(node2.nome,"Wendy");
node2.idade = 30;
```

a declaração de impressão

```
cout<<node1.nome<< ' '<<node1.idade<< ' '<<node2.nome<< ' '<<node2.idade;
gera a saída
```

```
Wendy 30 Wendy 20
```

As idades são diferentes, mas os nomes nos dois objetos são os mesmos. O que aconteceu? O problema é que a definição de *Node* não fornece um construtor de cópia

```
Node(const Node&);
```

necessário para executar a declaração *node2(node1)* para inicializar o *node1*. Se um construtor de cópia do usuário estiver faltando, o construtor é gerado automaticamente pelo compilador. Mas o construtor de cópia gerado pelo compilador realiza a cópia membro a membro. Devido a *nome* ser um ponteiro, o construtor de cópia copia o endereço da cadeia de caracteres *node1.nome* para *node2.nome*, não o conteúdo da cadeia de caracteres, de modo que, logo depois da execução da declaração, a situação é como na Figura 1.2a. Agora, se as atribuições

```
strcpy(node2.nome,"Wendy");
node2.idade = 30;
```

forem executadas, *node2.idade* é atualizada apropriadamente, mas a cadeia de caracteres "Roger" apontada pelo membro *nome* de ambos os objetos é reescrita como "Wendy", que também é apontada pelos dois ponteiros (Figura 1.2b). Para evitar que isso aconteça o usuário precisa definir um construtor apropriado de cópia, como em

```
struct Node{
    char *nome;
    int idade;
    Node(char *n = "", int a = 0){
        nome = new char[strlen(n)+1];
        strcpy(nome,n);
        idade = a;
    }
    Node(const Node& n) { // construtor de cópia
        nome = new char[strlen(n.nome)+1];
        strcpy(nome,n.nome);
        idade = n.idade;
    }
};
```

Com o novo construtor, a declaração *node2(node1)* gera outra cópia de "Roger" apontada por *node2.nome* (Figura 1.2c) e as atribuições aos membros de dados em um objeto não têm efeito nos membros de outro objeto, de modo que depois da execução das atribuições

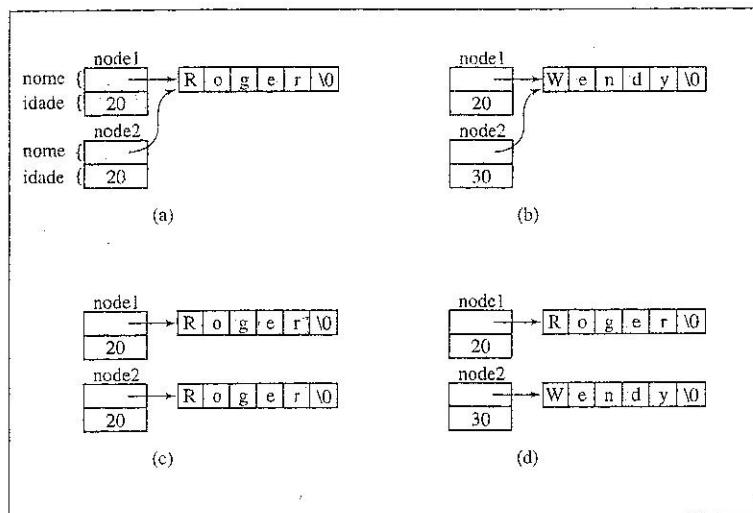
```
strcpy(node2.nome,"Wendy");
node2.idade = 30;
```

o objeto *node1* permanece imutável como ilustrado na Figura 1.2d.

Note que um problema similar é provocado pelo operador de atribuição. Se uma definição do operador de atribuição não é fornecida pelo usuário, uma atribuição

```
node1 = node2;
```

FIGURA 1.2 Ilustrando-se a necessidade de se usar um construtor de cópia para objetos com membros ponteiros.



realiza a cópia membro a membro, que leva ao mesmo problema apresentado na Figura 1.2a-b. Para evitar o problema o operador de atribuição tem que ser sobre carregado pelo usuário. Para `Node` a sobre carga é realizada por

```
Node& operator=(const Node& n){
    if (this != &n){ // sem atribuição para ele próprio;
        if (nome != 0)
            delete [] nome;
        nome = new char[strlen(n.nome)+1];
        strcpy(nome,n.nome);
        idade=n.idade;
    }
    return *this;
}
```

Nesse código um ponteiro especial `this` é usado. Cada objeto pode acessar seu próprio endereço por meio do ponteiro `this`, de modo que `*this` é o próprio objeto.

1.4.3 Ponteiros e Destrutores

O que acontece com os objetos definidos localmente do tipo `Node`? Como todos os itens locais, eles são destruídos, no sentido de que se tornam indisponíveis fora do bloco no qual são definidos, e a memória ocupada por eles é também liberada. Embora a memória ocupada por um objeto do tipo `Node` seja liberada, nem toda memória relacionada com esse objeto se torna disponível. Um dos membros de dados desse ob-

jeto é um ponteiro para uma cadeia de caracteres; em consequência, a memória ocupada pelo membro de dados ponteiro é liberada, mas a memória tomada pela cadeia de caractere não é. Depois que o objeto é destruído, a cadeia de caracteres previamente disponível a partir de seu membro de dados `nome` se torna inacessível (se não foi atribuída a nome de algum outro objeto ou a uma variável de cadeia de caracteres) e a memória ocupada por essa cadeia não pode mais ser liberada. Esse é um problema com objetos que têm membros de dados que apontam para locais dinamicamente alocados. Para evitar o problema, a definição de classe deve incluir a definição de um destrutor. Um *destrutor* é uma função automaticamente invocada quando um objeto é destruído, o que tem lugar na saída do bloco no qual o objeto é definido ou na chamada de `delete`. Os destrutores não tomam argumentos e não retornam valores, de modo que pode haver somente um destrutor por classe. Para a classe `Node`, um destrutor pode ser definido como

```
-Node(){
    if(nome != 0)
        delete [] nome;
}
```

1.4.4 Ponteiros e Variáveis de Referência

Considere as seguintes declarações:

```
int n = 5, *p = &n, &r = n;
```

A variável `p` é declarada como sendo do tipo `int*`, um ponteiro para um inteiro, e `r` é do tipo `int&`, uma variável de referência de inteiro. Uma variável de referência precisa ser inicializada em sua declaração como uma referência a uma variável particular, e essa referência não pode ser mudada. Isso significa que uma variável de referência não pode ser nula. Uma variável de referência `r` pode ser considerada como um nome diferente para uma variável `n`, de modo que, se `n` muda, `r` muda também. Isso porque uma variável de referência é implementada como um ponteiro constante para a variável.

Depois das três declarações a declaração de impressão

```
cout << n << ' ' << *p << ' ' << r << endl;
```

produz 5 5. Depois da atribuição

```
n = 7;
```

a mesma declaração de impressão produz 7 7 7. Também, uma atribuição

```
*p = 9;
```

fornecerá o resultado 9 9 9 e a atribuição

```
r = 10;
```

leva ao resultado 10 10 10. Essas declarações indicam que, em termos de notação, o que podemos realizar com o desreferenciamento de variáveis ponteiros é realizado sem o desreferenciamento de variáveis de referência. Isso não é um acidente porque, como mencionado, as variáveis de referência são implementadas como ponteiros constantes. Em vez da declaração

```
int& r = n;
```

podemos usar uma declaração

```
int *const r = &n;
```

onde `r` é um ponteiro constante para um inteiro, o que significa que a atribuição

```
r = q;
```

onde *q* é outro ponteiro, é um erro, porque o valor de *r* não pode mudar. No entanto, a atribuição

```
*r = 1;
```

é aceitável se *n* não for um inteiro constante.

É importante notar a diferença entre o tipo `int *const` e o tipo `const int *`. O segundo é um tipo de ponteiro para um inteiro constante:

```
const int *s = &m;
```

depois do qual a atribuição

```
s = &m;
```

onde *m* em um inteiro (seja constante ou não) é admissível, mas a atribuição

```
*s = 2;
```

é errónea, mesmo se *m* não é uma constante.

As variáveis de referência são usadas na passagem de argumentos por referência para chamadas de funções. A passagem por referência é requerida se um parâmetro atual deve ser mudado permanentemente, durante a execução de uma função. Isso pode ser realizado com ponteiros (e em C, é o único mecanismo disponível para se passar por referência) ou com variáveis de referência. Por exemplo, depois de se declarar uma função

```
void f1(int i, int* j, int& k){
    i = 1;
    *j = 2;
    k = 3;
}
```

os valores das variáveis

```
int n1 = 4, n2 = 5, n3 = 6;
```

depois de se executar a chamada

```
f1(n1, &n2, n3);
```

são *n1* = 4, *n2* = 2, *n3* = 3.

O tipo referência é também utilizado para indicar o tipo de retorno de funções. Por exemplo, tendo-se definido a função

```
int& f2(int a[], int i ){
    return a[i];
}
```

e declarado a matriz

```
int a[] = {1,2,3,4,5};
```

poderemos usar *f2()* em qualquer lado do operador de atribuição. Por exemplo, no lado direito,

```
n = f2(a, 3);
```

ou no lado esquerdo,

```
f2(a, 3) = 6;
```

que atribui 6 a *a[3]*, de modo que *a* = [1 2 3 6 5]. Note que podemos realizar o mesmo com ponteiros, mas o desreferenciamento tem que ser usado explicitamente:

```
int* f3(int a[], int i){
    return &a[i];
}
```

e então

```
*f3(a, 3) = 6;
```

1.4.5 Ponteiros para Funções

Como indicado na Seção 1.4.1, um dos atributos de uma variável é seu endereço que indica sua posição na memória do computador. O mesmo é verdade para as funções: um dos atributos de uma função é o endereço que indica o local do corpo da função na memória. Na chamada de função o sistema transfere o controle a esse local para executar a função. Por essa razão é possível usar ponteiros para funções. Esses ponteiros são muito úteis na implementação de funcionais (isto é, funções que tomam funções como argumentos), tal como a integral.

Considere a função simples

```
double f(double x){
    return 2*x;
}
```

Com essa definição *f* é um ponteiro para a função *f()*, *f** é a própria função e *(*f)(7)* é uma chamada para a função.

Considere agora a escrita de uma função C++ que calcula a seguinte soma:

$$\sum_{i=n}^m f(i)$$

Para calcular a soma temos que fornecer não somente os limites *n* e *m*, mas também uma função *f*. Em consequência, a implementação desejada deve permitir não somente a passagem de números como argumentos, mas também de funções. Isso é feito em C++ do seguinte modo:

```
double soma(double (*f)(double), int n, int m){
    double resultado = 0;
    for(int i = n; i <= m; i++)
        resultado += f(i);
    return resultado;
}
```

Nessa definição da *soma()* a declaração do primeiro argumento formal

```
double (*f*)(double)
```

significa que *f* é um ponteiro para uma função com um argumento do tipo `double`, e um valor de retorno do tipo `double`. Note a necessidade dos parênteses ao redor de **f*. Pelo fato de os parênteses terem precedência sobre o operador de dereferência ***, a expressão

```
double *f(double)
```

declara a função que retorna um valor `double`.

disso, ainda que `p` aponte para o `objeto3`, a instrução `p->h()` resulta em erro de compilação, porque o compilador não encontra `h()` na `Classe1`, onde `Classe1*` é ainda o tipo do ponteiro `p`. Para o compilador não importa que `h()` seja definida na `Classe3` (seja ela virtual ou não).

O polimorfismo é uma ferramenta poderosa em POO. Basta enviar uma mensagem padrão para vários objetos diferentes sem especificar como a mensagem será compreendida. Não há necessidade de se conhecer de que tipo o objeto é. O recebedor é responsável por interpretar a mensagem e compreendê-la. O expedidor não tem que modificar a mensagem, dependendo do tipo do recebedor. Não há necessidade de declarações `switch` ou `if-else`. Novas unidades também podem ser adicionadas a um programa complexo, sem necessidade de recompilar o programa inteiro.

■ 1.6 C++ E A PROGRAMAÇÃO ORIENTADA AO OBJETO

Até aqui assumiu-se que C++ é uma LOO, e todas as características das LOOs que nós discutimos foram ilustradas com o código em C++. No entanto, C++ não é uma LOO pura. Ele é mais orientado a objetos do que C ou Pascal, que não têm características orientadas a objetos, ou Ada, que suporta classes (pacotes) e instâncias, mas é menos orientado a objetos do que LOOs puras, tais como Smalltalk ou Eiffel.

Em C++ a abordagem orientada a objetos não é obrigatória. Podemos programar em C++ sem saber que talas características são parte da linguagem. A razão para isso é a popularidade de C. C++ é um superconjunto de C, de modo que um programador C pode facilmente mudar para C++, adaptando-se somente as suas características mais amigáveis, tais como E/S, mecanismo de chamada por referência, valores default para parâmetros de função, sobrecarga de operadores, funções em linha e similares. Usando-se uma LOO, tal como C++, não garante que se pratique a POO. Por outro lado, invocando-se o mecanismo completo de classes e funções-membro pode nem sempre ser necessário, especialmente em pequenos programas, por isso, não forçar a POO não é necessariamente uma desvantagem. Ademais, C++ é mais fácil de se integrar com o código existente em C do que outras LOOs.

C++ tem excelentes recursos de encapsulamento que permitem um ocultamento de informação bem controlado. Há, no entanto, um afrouxamento dessa regra no uso das assim chamadas funções amigas. O problema é que a informação privada de uma certa classe não pode ser acessada por qualquer um, e a informação pública é acessível por qualquer usuário. Mas algumas vezes gostaríamos de permitir que somente alguns usuários tivessem acesso à área privada de informação. Isso pode ser realizado se a classe listar as funções do usuário como suas amigas. Por exemplo, se a definição é

```
classe C {
    int n;
    friend int f();
} ob;
```

a função `f()` tem acesso à variável `n` que pertence à classe C, como em

```
int f()
{
    return 10 * ob.n; }
```

Isso poderia ser considerado uma violação do princípio de ocultamento de informação; no entanto, a própria classe C concede o direito de tornar público a alguns usuários o que é privado e inacessível a outros. Assim, desde que a classe tenha controle sobre o que considerar uma função amiga, o mecanismo de função amiga pode ser considerado uma extensão do princípio de ocultamento de informação. Esse mecanismo é reconhecidamente usado para facilitar a programação e agilizar a execução, pois reescrever o código sem usar as funções amigas pode ser um grande problema. Tal afrouxamento de algumas regras, a propósito, não é incomum na ciência de computação e podemos mencionar a existência de laços em linguagens funcionais,

tal como a LISP, ou o armazenamento de alguma informação no início de arquivos de dados em violação do modelo de banco de dados relacionais, como no dBaseIII+.

■ 1.7 A BIBLIOTECA DE FORMATOS PADRÃO

C++ é uma linguagem orientada a objetos, mas extensões recentes à linguagem trouxeram para um nível mais alto. A adição mais significativa à linguagem é a Biblioteca de Formatos Padrão (em inglês, Standard Template Library – STL), desenvolvida principalmente por Alexander Stepanov e Meng Lee. A biblioteca inclui três tipos de entidades genéricas: contêineres, iteradores e algoritmos. Os algoritmos são funções usadas freqüentemente que podem ser aplicadas a diferentes estruturas de dados. A aplicação é intermediada por iteradores que determinam quais algoritmos podem ser aplicados a que tipos de objetos. A STL libera os programadores de escreverem suas próprias implementações de várias classes e funções. Em vez disso, eles podem usar implementações genéricas pré-empacotadas, adaptadas para o problema em questão.

1.7.1 Contêineres

Um contêiner é uma estrutura de dados que contém alguns objetos que usualmente são do mesmo tipo. Tipos diferentes de contêineres organizam os objetos dentro deles diferentemente. Embora o número de organizações diferentes seja teoricamente ilimitado, somente um pequeno número deles tem significado prático, e as organizações mais freqüentemente usadas estão incorporadas na STL. A STL inclui os seguintes contêineres: `deque`, `list`, `map`, `multimap`, `set`, `multiset`, `stack`, `queue`, `priority_queue` e `vector`.

Os contêineres STL estão implementados como classes de formato (`template`) que incluem funções-membro, e estas especificam que operações podem ser realizadas sobre os elementos armazenados na estrutura de dados especificada pelo contêiner ou sobre a própria estrutura de dados. Algumas operações podem ser encontradas em todos os contêineres, embora possam estar implementadas diferentemente. As funções-membro comuns a todos os contêineres incluem o construtor `default`, o construtor de cópia, o destrutor, `empty()`, `max_size()`, `size()`, `swap()`, `operator=`, exceto em `priority_queue`, seis funções de operador relacional sobrecarregado (`operador<` etc.). Ademais, as funções-membro comuns a todos os contêineres, exceto `stack`, `queue` e `priority_queue`, incluem as funções `begin()`, `end()`, `rbegin()`, `rend()`, `erase()` e `clear()`.

Os elementos armazenados nos contêineres podem ser de quaisquer tipos e têm que fornecer pelo menos um construtor `default`, um destrutor e um operador de atribuição. Isso é particularmente importante para os tipos definidos pelo usuário. Alguns compiladores podem exigir também que alguns operadores relacionais sejam sobrecarregados (pelo menos os operadores `==` e `<`, mas podem ser `!=` e `>` também), ainda que o programa não os use. Também, um construtor de cópia e a função `operador=` devem ser fornecidos se os membros de dados são ponteiros, porque as operações de inserção usam uma cópia do elemento que está sendo inserido e não o próprio elemento.

1.7.2 Iteradores

Um iterador é um objeto usado para referenciar um elemento armazenado em um contêiner. Assim, ele é uma generalização do ponteiro. Um iterador permite o acesso à informação incluída em um contêiner, de modo que as operações desejadas possam ser realizadas sobre esses elementos.

Como uma generalização dos ponteiros, os iteradores retêm a mesma notação de desreferenciamento. Por exemplo, `*i` é um elemento referenciado pelo iterador `i`. Além disso, a aritmética do iterador é similar à aritmética do ponteiro, embora nem todas as operações sobre os iteradores sejam permitidas em todos os contêineres.

Nenhum iterador é suportado nos contêineres `stack`, `queue` e `priority_queue`. As operações de iterador para as classes `list`, `map`, `multimap`, `set` e `multiset` são como a seguir (`i1` e `i2` são iteradores, `n` é um número):

```
i1++, ++i1, i1--, -i1
i1 = i2
i1 == i2, i1 != i2
*i1
```

Em adição a essas operações, as operações de iterador para as classes `deque` e `vector` são como estas:

```
i1 < i2, i1 <= i2, i1 > i2, i1 >= i2
i1 + n, i1 - n
i1 += n, i1 -= n
i1[n]
```

1.7.3 Algoritmos

A STL fornece cerca de 70 funções genéricas, chamadas algoritmos, que podem ser aplicadas aos contêineres STL e às matrizes. Uma lista de todos os algoritmos está no Apêndice B. Esses algoritmos estão implementando operações que são muito freqüentemente usadas na maioria dos programas, tais como localizar um elemento em um contêiner, inserir um elemento em uma seqüência de elementos, remover um elemento de uma seqüência, modificar elementos, comparar elementos, encontrar um valor baseado em uma seqüência de elementos, ordenar a seqüência de elementos e assim por diante. Quase todos os algoritmos STL usam iteradores para indicar o intervalo dos elementos no qual operam. O primeiro iterador referencia o primeiro elemento do intervalo, e o segundo referencia um elemento *depois* do último elemento do intervalo. Em consequência, assume-se que sempre é possível atingir a posição indicada pelo segundo iterador incrementando-se o primeiro. Eis alguns exemplos.

A chamada

```
random_shuffle(c.begin(), c.end());
```

aleatoriamente reordena todos os elementos do contêiner `c`. A chamada

```
i3 = find(i1, i2, c1);
```

retorna um iterador que indica a posição do elemento `c1` no intervalo `i1` até (mas não incluindo) `i2`. A chamada

```
n = count_if(i1, i2, NumImpar);
```

conta, por meio do algoritmo `count_if()`, os elementos no intervalo indicado pelos iteradores `i1` e `i2` para os quais uma função booleana definida pelo usuário de um argumento `NumImpar()` retorna `true`.

Os algoritmos são funções acrescentadas às funções-membro fornecidas pelos contêineres. No entanto, alguns algoritmos estão também definidos como funções-membro para proporcionar melhor desempenho.

1.7.4 Objetos de Função

Em C++ o operador de chamada de função () pode ser tratado como qualquer outro operador; em particular, ele pode ser sobre carregado. Pode também retornar qualquer tipo e tomar qualquer número de argumentos, mas, como o operador de atribuição, ele pode ser sobre carregado somente como uma função-membro. Qualquer objeto que inclua uma definição do operador de chamada de função é chamado de *objeto de função*.

Um objeto de função é um objeto, mas comporta-se como se fosse uma função. Quando o objeto de função é chamado, seus argumentos tornam-se os argumentos do operador de chamada de função.

Considere o exemplo de se obter a soma dos números que resulta de se aplicar uma função `f` aos números inteiros no intervalo $[n,m]$. Uma implementação da `soma()` apresentada na Seção 1.4.5 dependia do uso de um ponteiro de função como um argumento da função `soma()`. O mesmo pode ser realizado definindo-se primeiro uma classe que sobre carrega o operador de chamada de função:

```
class classf {
public:
    classf() {}
    double operator() (double x) {
        return 2*x;
    }
};
```

e definindo

```
double soma2(classf f, int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f(i);
    return result;
}
```

que difere de `soma()` somente no primeiro parâmetro, que é um objeto de função, não uma função; senão, ela seria a mesma. A nova função pode agora ser chamada, como em

```
classf cf;
cout << soma2(cf, 2, 5) << endl;
```

ou simplesmente

```
cout << soma2(classf(), 2, 5) << endl;
```

O último modo de chamada exige uma definição do construtor `classf()` (mesmo que ele não tenha corpo) para se criar um objeto do tipo `classf()` quando `soma2()` é chamada.

O mesmo pode ser realizado sem sobre carregar o operador de chamada de função, como exemplificado nas duas seguintes definições:

```
class classf2 {
public:
    classf2 () {}
    double marcha (double x) {
        return 2*x;
    }
};
double soma3 (classf2 f, int n, int m) {
    double result = 0;
    for (int i = n; i <= m; i++)
        result += f.run(i);
    return result;
}
```

e uma chamada

```
cout << soma3(classif2(), 2, 5) << endl;
```

A STL depende muito fortemente dos objetos de função. O mecanismo dos ponteiros de função é insuficiente para os operadores predefinidos. Como poderemos passar um menos unário para `soma()`? A sintaxe `soma(~, 2, 5)` é ilegal. Para contornar o problema a STL define em `<functional>` objetos de função para os operadores C++ comuns. Por exemplo, o menos unário é definido como

```
template<class T>
struct negate : public unary_function<T, T> {
    T operator()(const T& x) const {
        return -x;
    }
};
```

Agora, depois de se redefinir a função `soma()` para que ela se torne uma função genérica:

```
template<class F>
double soma(F f, int n, int m) {
    double resultado = 0;
    for (int i = n; i <= m; i++)
        resultado += f(i);
    return resultado;
}
```

a função pode também ser chamada com o objeto de função `negate`,

`soma(negate<double>(), 2, 5)`.

■ 1.8 VETORES NA BIBLIOTECA DE FORMATOS PADRÃO

O container mais simples da STL é o vetor, que é uma estrutura de dados com blocos de memória contíguos, tal como uma matriz. Devido aos locais de memória serem contíguos, eles podem ser acessados aleatoriamente, de modo que o tempo de acesso de qualquer elemento do vetor é constante. A armazenagem é gerenciada automaticamente, por isso, na tentativa de inserir um elemento dentro de um vetor cheio, um bloco de memória maior é alocado para o vetor, os elementos do vetor são copiados para o novo bloco e o bloco antigo é liberado. Um vetor é, assim, uma matriz flexível, isto é, uma matriz cujo tamanho pode ser dinamicamente modificado.

A Figura 1.3 lista alfabeticamente todas as funções-membro do vetor. Uma aplicação dessas funções está ilustrada na Figura 1.4. Os conteúdos dos vetores afetados estão mostrados como comentários nas linhas nas quais as funções-membro são chamadas. Os conteúdos de um vetor são impressos com a função genérica `printVector()`, mas no programa da Figura 1.4 somente uma chamada aparece.

Para usar a classe vetor o programa tem que conter a instrução `include`

```
#include <vector>
```

A classe vetor tem quatro construtores. A declaração

```
vector<int> v5(5);
```

usa o mesmo construtor que a declaração

```
vector<int> v2(3, 7);
```

FIGURA 1.3 Uma lista alfabética das funções-membro na classe `vector`.

Função-Membro	Operação
<code>void assign(primeiro, ultimo)</code>	remove todos os nós no vetor e nele insere os elementos do intervalo indicado pelos iteradores <code>primeiro</code> e <code>ultimo</code>
<code>void assign(n, el = T())</code>	remove todos os nós no vetor e insere nele <code>n</code> cópias de <code>el</code>
<code>T& at(n)</code>	retorna o elemento na posição <code>n</code> do vetor
<code>const T& at(n) const</code>	retorna o elemento na posição <code>n</code> do vetor
<code>T& back()</code>	retorna o último elemento do vetor
<code>const T& back() const</code>	retorna o último elemento do vetor
<code>iterator begin()</code>	retorna um iterador que referencia o primeiro elemento do vetor
<code>const_iterator begin() const</code>	retorna um iterador <code>const</code> que referencia o primeiro elemento do vetor
<code>size_type capacity()</code>	retorna o número de elementos que podem ser estocados no vetor
<code>void clear()</code>	remove todos os elementos no vetor
<code>bool empty() const</code>	retorna <code>true</code> se o vetor não inclui elemento e, caso contrário, <code>false</code>
<code>iterator end()</code>	retorna um iterador que está adiante do último elemento do vetor
<code>const_iterator end() const</code>	retorna um iterador <code>const</code> que está adiante do último elemento do vetor
<code>iterator erase(i)</code>	remove o elemento referenciado pelo iterador <code>i</code> e devolve um iterador que referencia o elemento posterior ao removido
<code>iterator erase(primeiro, ultimo)</code>	remove os elementos no intervalo indicado pelos iteradores <code>primeiro</code> e <code>ultimo</code> e devolve um iterador que referencia um elemento posterior ao removido
<code>T& front()</code>	retorna o primeiro elemento do vetor
<code>const T& front() const</code>	retorna o primeiro elemento do vetor
<code>iterator insert(i, el = T())</code>	insere <code>el</code> antes do elemento referenciado pelo iterador <code>i</code> e devolve o iterador que referencia o elemento mais novo inserido

Continua

mas para o vetor `v5` o elemento com o qual ele está preenchido é determinado pelo construtor inteiro default, que é zero.

O vetor `v1` é declarado vazio e então novos elementos são inseridos com a função `push_back()`. A adição de um elemento ao vetor é usualmente rápida, a menos que o vetor esteja cheio e tenha que ser copiado para um novo bloco. Essa situação ocorre se o tamanho do vetor é igual a sua capacidade. Mas se o vetor tem algumas células não utilizadas, ele pode acomodar um novo elemento imediatamente em tempo constante. Os valores correntes dos parâmetros podem ser testados com a função `size()`, que retorna o número de elementos correntemente no vetor, e a função `capacity()`, que retorna o número de células disponíveis no vetor. Se necessário, a capacidade pode ser modificada com a função `reserve()`. Por exemplo, depois de executar

FIGURA 1.3 (continuação)

Função-Membro	Operação
void insert(i, n, el)	insere n cópias de el antes do elemento referenciado pelo iterador i
void insert(i, primeiro, ultimo)	insere elementos a partir do intervalo indicado pelos iteradores primeiro e ultimo antes do nó referenciado pelo iterador i
size_type max_size() const	retorna o número máximo de elementos para o vetor
T& operator[]	operador subscrito
const T& operator[] const	operador subscrito
void pop_back()	remove o último elemento do vetor
void push_back(el)	insere el no final do vetor
reverse_iterator rbegin()	retorna um iterador que referencia o último elemento do vetor
const_reverse_iterator rbegin() const	retorna uma iterador const que referencia o último elemento do vetor
reverse_iterator rend()	retorna um iterador que está antes do primeiro elemento do vetor
const_reverse_iterator rend() const	retorna um iterador const que está antes do primeiro elemento do vetor
void reserve(n)	reserva espaço suficiente para o vetor para conter n itens se sua capacidade é menor que n
void resize(n, el = T())	torna o vetor apto para conter n elementos adicionando n - size() mais posições com o elemento el, ou descartando size() - n posições em excesso a partir do final do vetor
size_type size() const	retorna o número de elementos no vetor
void swap(v)	troca o conteúdo do vetor com o conteúdo de outro vetor v
vector()	constrói um vetor vazio
vector(n, el = T())	constrói um vetor com n cópias de el do tipo T (se el não é fornecido, um construtor default T() é usado)
vector(primeiro, ultimo)	constrói um vetor com os elementos de intervalo indicado pelos iteradores primeiro e ultimo
vector(v)	construtor de cópia

```
v3.reserve(6);
```

o vetor v3 = (2 3) retém os mesmos elementos e o mesmo tamanho = 2, mas sua capacidade mudou de 2 para 6. A função `reserve()` afeta somente a capacidade do vetor, não o seu conteúdo. A função `resize()` afeta os conteúdos e possivelmente a capacidade. Por exemplo, o vetor v4 = (1 2 3 4 5) de tamanho = capacidade = 5 modifica-se depois da execução de

```
v4.resize(7);
```

para v4 = (1 2 3 4 5 0 0), tamanho = 7, capacidade = 10 e depois de outra chamada para `resize()`,

FIGURA 1.4 Um programa que demonstra a operação das funções-membro do vetor.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional> // greater<T>

using namespace std;

template<class T>
void printVector(char *s, const vector<T>& v) {
    cout << s << " = ";
    if (v.size() == 0) {
        cout << "\n";
        return;
    }
    (vector<T>::const_iterator i = v.begin());
    for (; i < v.begin() + v.size() - 1; i++)
        cout << *i << ' ';
    cout << *i << "\n";
}

bool f1(int n) {
    return n < 4;
}

void main() {
    int a[] = {1,2,3,4,5};
    vector<int> v1; // v1 esta vazio, tamanho = 0, capacidade = 0
    printVector(v1);
    for (int j = 1; j <= 5; j++)
        v1.push_back(j); // v1 = (1 2 3 4 5), tamanho = 5, capacidade = 8
    vector<int> v2(3,7); // v2 = (7 7 7)
}
```

Continua

```
v4.resize(3);
```

para v4 = (1 2 3), tamanho = 3, capacidade = 10. Isso indica que o novo espaço está alocado para vetores, mas não é retornado.

Note que não há a função-membro `push_front()`. Isso reflete o fato de que adicionar um novo elemento à frente do vetor é uma operação complexa, porque exige que todos os elementos sejam deslocados de uma posição para se criar espaço para o novo elemento. Essa é uma operação que consome tempo e pode ser realizada com a função `insert()`, é também uma função que automaticamente aloca mais memória para o vetor, se necessário. Outras funções que realizam essa tarefa são os construtores, a função `reserve()` e `operator=`.

Os elementos do vetor podem ser acessados com a notação de subscrito usada para as matrizes, como em

```
v4[0] = n;
```

FIGURA 1.4 (continuação)

```

vector<int> ::iterator i1 = v1.begin() + 1;
vector<int> v3(i1,i1+2); // v3 = (2 3), tamanho = 2, capacidade = 2
vector<int> v4(v1); // v4 = (1 2 3 4 5), tamanho = 5, capacidade = 5
vector<int> v5(5); // v5 = (0 0 0 0 0)
v5[1] = v5.at(3) = 9; // v5 = (0 9 0 9 0)
v3.reserve(6); // v3 = (2 3), tamanho = 2, capacidade = 6
v4.resize(7); // v4 = (1 2 3 4 5 0 0), tamanho = 7, capacidade = 10
v4.resize(3); // v4 = (1 2 3), tamanho = 3, capacidade = 10
v4.clear(); // v4 está vazio, tamanho = 0, capacidade = 10 (!)
v4.insert(v4.end(),v3[1]); // v4 = (3)
v4.insert(v4.end(),v3.at(1)); // v4 = (3 3)
v4.insert(v4.end(),2,4); // v4 = (3 3 4 4)
v4.insert(v4.end(),v1.begin() + 1,v1.end() - 1); // v4 = (3 3 4 4 2 3 4)
v4.erase(v4.end() - 2); // v4 = (3 3 4 4 2 4)
v4.erase(v4.begin(), v4.begin() + 4); // v4 = (2 4)
v4.assign(3,8); // v4 = (8 8 8)
v4.assign(a,a+3); // v4 = (1 2 3)
vector<int> ::reverse_iterator i3 = v4.rbegin();
for ( ; i3 != v4.rend(); i3++)
    cout << *i3 << ' ';
cout << endl; // print: 3 2 1

// algorithms

v5[0] = 3; // v5 = (3 9 0 9 0)
replace_if(v5.begin(),v5.end(),f1,7); // v5 = (7 9 7 9 7)
v5[0] = 3; v5[2] = v5[4] = 0; // v5 = (3 9 0 9 0)
replace(v5.begin(),v5.end(),0,7); // v5 = (3 9 7 9 7)
sort(v5.begin(),v5.end()); // v5 = (3 7 7 9 9)
sort(v5.begin(),v5.end(),greater<int>()); // v5 = (9 9 7 7 3)
v5.front() = 2; // v5 = (2 9 7 7 3)
}

```

ou com iteradores com a notação de desreferenciamento usada para ponteiros, como em

```

vector<int> ::iterator i4 = v4.begin();
*i1 = n;

```

Note que algumas funções-membro têm o tipo de retorno `T&` (isto é, um tipo de referência). Por exemplo, para um vetor inteiro, a marca da função-membro `front()` é

```
int& front();
```

Isso significa que, por exemplo, `front()` pode ser usada tanto para o lado esquerdo como para o lado direito do operador de atribuição:

```
v5.front() = 2;
```

```
v4[1] = v5.front();
```

Note que `front()` é um exemplo de uma função-membro sobreescrita, porque pode retornar uma referência a um valor ou uma referência constante a um valor. Para ver a diferença considere estas duas atribuições:

```
int& n1 = v5.front(); // use: T& front()
const int& n2 = v5.front(); // use: const T& front() const
```

Muito importante: `front()` retorna um valor, não uma referência a um valor, não obstante, a declaração do valor de retorno estar como `T&`. Para ver a diferença considere mais uma atribuição:

```
int n3 = v5.front();
```

Neste ponto os valores das três variáveis são:

```
n1 = 2, n2 = 2, n3 = 2.
```

Mas depois da atribuição

```
v5.front() = 3;
```

os valores das três variáveis são

```
n1 = 3, n2 = 3, n3 = 2.
```

Todos os algoritmos STL podem ser aplicados a vetores. Por exemplo, a chamada

```
replace(v5.begin(),v5.end(),0,7);
```

substitui todos os 0s com 7s no vetor `v5`, de modo que `v5 = (3 9 0 9 0)` se torna `v5 = (3 9 7 9 7)`, e a chamada

```
sort(v5.begin(),v5.end());
```

ordena na ordem ascendente o vetor `v5`. Alguns algoritmos permitem usar parâmetros funcionais. Por exemplo, se o programa inclui a definição de função

```
bool f1(int n) {
    return n < 4;
}
```

então a chamada

```
replace_if(v5.begin(),v5.end(),f1,7);
```

aplica `f1()` a todos os elementos de `v5` e substitui todos os elementos menores que 4 por 7. Neste caso `v5 = (3 9 0 9 0)` se torna `v5 = (7 9 7 9 7)`. Incidentalmente, um modo mais crítico de realizar o mesmo resultado sem a necessidade de definir explicitamente `f1` é dado por

```
replace_if(v5.begin(),v5.end(),bind2nd(less<int>(),4),7);
```

Nessa expressão, `bind2nd(op, a)` é uma função genérica que se comporta como se convertesse um objeto de função de dois argumentos em um objeto de função de um argumento, fornecendo (vinculando) o segundo parâmetro. Ela faz isso criando objetos de função de dois argumentos, nos quais uma operação de dois argumentos `op` toma a como o segundo argumento.

Os algoritmos de ordenação permitem a mesma flexibilidade. No exemplo de ordenação do vetor `v5`, a ordenação é ascendente. Como podemos ordená-lo de forma descendente? Um modo é fazer a ordenação ascendente e então invertê-lo com o algoritmo `reverse()`. Outro modo é forçar o `sort()` a aplicar o operador `>` na tomada de suas decisões. Isso é feito diretamente usando-se um objeto de função como parâmetro, como em

```
sort(v5.begin(), v5.end(), greater<int>());
ou indiretamente, como em
```

```
sort(v5.begin(), v5.end(), f2);
onde f2 é definido como
```

```
bool f2(int m, int n) {
    return m > n;
}
```

O primeiro método é preferível, mas isso é possível somente porque o objeto de função `greater` já está definido na STL. Esse objeto de função está definido como uma estrutura de formato que, em essência, sobrecarrega genericamente o operador `>`. Em consequência, `greater<int>()` significa que o operador deve ser aplicado a inteiros.

Essa versão do algoritmo `sort()`, que toma um argumento funcional, é particularmente útil quando necessitamos ordenar objetos mais complexos do que inteiros e usar diferentes critérios. Considere a seguinte definição de classe:

```
class Pessoa {
public:
    Pessoa (char *n = "", int a = 0) {
        nome = new char[strlen(n)+1];
        strcpy(nome, n);
        idade = a;
    }
    bool operator==(const Pessoa& p) const {
        return strcmp(nome, p.nome) == 0 && idade == p.idade;
    }
    bool operator<(const Pessoa& p) const {
        return strcmp(nome, p.nome) < 0;
    }
    bool operator>(const Pessoa& p) const {
        return !(*this == p) && !(*this < p);
    }
private:
    char *nome;
    int idade;
    friend bool idadeMenor(const Pessoa&, const Pessoa&);
};
```

Agora, com a declaração

```
vector<Pessoa> v6(1, Pessoa("Gregg", 25));
```

adicionando-se a `v6` mais dois objetos

```
v6.push_back(Pessoa("Ann", 30));
v6.push_back(Pessoa("Bill", 20));
```

e executando

```
sort(v6.begin(), v6.end());
```

`v6` muda de `v6 = ({"Gregg", 25} {"Ann", 30} {"Bill", 20})` para `v6 = ({"Ann", 30} {"Bill", 20} {"Gregg", 25})`, organizado na ordem ascendente, porque a versão de `sort()` com somente dois argumentos de iterador usa o operador `<` sobrecarregado na classe `Pessoa`. À chamada

```
sort(v6.begin(), v6.end(), greater<Pessoa>());
```

muda `v6 = ({"Ann", 30} {"Bill", 20} {"Gregg", 25})` para `v6 = ({"Gregg", 25} {"Bill", 20} {"Ann", 30})`, organizado na ordem descendente, porque essa versão de `sort()` depende do operador `>` sobrecarregado para essa classe. O que devemos fazer para ordenar os objetos pela idade? Nesse caso uma função precisa ser definida, como em

```
bool idadeMenor(const Pessoa& p1, const Pessoa& p2) {
    return p1.idade < p2.idade;
}
```

e então usá-la como argumento na chamada para `sort()`,

```
sort(v6.begin(), v6.end(), menorIdade);
```

que faz `v6 = ({"Gregg", 25} {"Bill", 20} {"Ann", 30})` mudar para `v6 = ({"Bill", 20} {"Gregg", 25} {"Ann", 30})`.

■ 1.9 ESTRUTURAS DE DADOS E PROGRAMAÇÃO ORIENTADA A OBJETOS

Embora os computadores operem com bits, usualmente não pensamos nestes termos; de fato, não gostamos disso. Embora um número inteiro seja uma seqüência de, digamos, 16 bits, preferimos ver um inteiro como uma entidade com sua própria individualidade, a qual é refletida em operações que podem ser realizadas em inteiros, mas não em variáveis de outros tipos. É como um inteiro usa bits como seus blocos de constituição, outros objetos podem usar inteiros como seus elementos atômicos. Alguns tipos de dados já estão desenvolvidos em uma linguagem particular, mas alguns tipos de dados podem e necessitam ser definidos pelo usuário. Novos tipos de dados têm estruturas distintas, novas configurações de seus elementos, e essas estruturas determinam o comportamento dos objetos desses novos tipos. A tarefa dada ao domínio das estruturas de dados é explorar tais novas estruturas e investigar seus comportamentos em termos de exigências de tempo e de espaço. Diferente da abordagem orientada a objetos, onde começamos com o comportamento e tentamos então encontrar os tipos de dados mais adequados que permitem levar em conta um desempenho eficiente das operações desejáveis, agora começamos com uma especificação do tipo de dados de alguma estrutura de dados e, então, observamos o que ela pode fazer, como ela o faz e quão eficientemente. O campo de estruturas de dados é concebido para construir ferramentas para serem incorporadas e usadas pelos programas de aplicação, e para encontrar estruturas de dados que possam realizar certas operações rapidamente e sem impor muita carga à memória do computador. Esse campo está interessado em construir classes concentrando-se na mecânica dessas classes, em suas engrenagens e rodas dentadas, que na maioria dos casos não estão visíveis ao usuário das classes. O campo das estruturas de dados investiga a operabilidade dessas classes e suas melhorias modificando as estruturas de dados encontradas dentro das classes, já que se tem acesso direto a elas. Ele afia as ferramentas e aconselha os usuários sobre a quais propósitos elas podem ser aplicadas. Devido à herança, o usuário pode adicionar mais algumas operações a essas classes e tentar espremer delas mais do que o projetista da classe já fez, mas, devido às estruturas de dados estarem escondidas do usuário, essas novas operações podem ser testadas ao serem rodadas e não pelo acesso ao interior da classe, a menos que o usuário tenha acesso ao código fonte.

O campo de estruturas de dados funciona melhor se realizado no modo orientado a objetos. Dessa forma, pode construir ferramentas sem o perigo de essas ferramentas serem inadvertidamente mal-empregadas. Encapsulando-se as estruturas de dados dentro de uma classe e tornando público somente o que é

necessário para o uso apropriado da classe, o campo das estruturas de dados pode desenvolver ferramentas cujas funções não são comprometidas por interferências desnecessárias.

1.10 ESTUDO DE CASO: ARQUIVOS DE ACESSO NÃO-SEQUENCIAL

Esse estudo de caso é primariamente projetado para ilustrar o uso das classes genéricas e da herança. A STL será aplicada a estudos de casos de capítulos posteriores.

Do ponto de vista dos sistemas operacionais, os arquivos são coleções de bytes, independentemente de seu conteúdo. Do ponto de vista do usuário, os arquivos são coleções de palavras, números, seqüências de dados, registros e assim por diante. Se o usuário quer acessar a quinta palavra em um arquivo de texto, um procedimento de busca varre seqüencialmente o arquivo começando na posição 0 e verifica todos os bytes no caminho. Ele conta o número de seqüências de caracteres em branco e, depois de pular quatro seqüências (ou cinco, se uma seqüência de brancos inicia o arquivo), pára, porque encontra o começo da quinta seqüência de não brancos ou da quinta palavra. Esta palavra pode começar em qualquer posição do arquivo. É impossível ir para uma posição particular de qualquer arquivo de texto e estar certo de que essa é a posição de início da quinta palavra do arquivo.

Idealmente, queremos ir diretamente a uma posição do arquivo e estarmos certos de que a quinta palavra começa ali. O problema é causado pelo comprimento das palavras precedentes e pelas seqüências de brancos. Se sabemos que cada palavra ocupa a mesma quantidade de espaço, é possível ir diretamente à quinta palavra, indo-se à posição 4 * comprimento(palavra). Mas, devido às palavras serem de comprimentos variados, isso pode ser realizado atribuindo-se o mesmo número de bytes a cada palavra; se uma palavra for menor, alguns caracteres de enchimento são adicionados para preencher o espaço remanescente; se for maior, a palavra é truncada. Desse modo, uma nova organização é imposta ao arquivo. O arquivo é agora tratado não meramente como uma coleção de bytes, mas como uma coleção de registros. Em nosso exemplo, cada registro consiste de uma palavra. Se uma solicitação chega para acessar a quinta palavra, a palavra pode ser diretamente acessada sem ver as palavras precedentes. Com a nova organização, criamos um arquivo de acesso não-sequencial.

Um arquivo de acesso não-sequencial permite o ingresso direto a cada registro. Os registros usualmente incluem mais itens do que uma palavra. O exemplo precedente sugere um modo de se criar um arquivo de acesso aleatório, usando registros de comprimentos fixos. Nossa tarefa neste estudo de caso é escrever um programa genérico que gere um arquivo de acesso aleatório para qualquer tipo de registro. As operações do programa estão ilustradas para um arquivo que contém registros de pessoal, cada registro consistindo de cinco membros de dados (número do seguro social, nome, cidade, ano do nascimento e salário), e para um arquivo de estudante que armazena registros de estudantes. Os últimos registros têm os mesmos membros de dados que os registros de pessoal, mas a informação sobre a especialização acadêmica. Isso nos permite ilustrar a herança.

Neste estudo de caso um programa genérico de arquivo de acesso não-sequencial insere novo registro em um arquivo, encontra um registro no arquivo e modifica um registro. O nome do arquivo tem que ser fornecido pelo usuário e, se o arquivo não for encontrado, ele é criado; caso contrário, é aberto para a leitura e para a escrita. O programa é mostrado na Figura 1.5.

A função `find()` determina se um registro está no arquivo. Ela realiza a pesquisa seqüencialmente, comparando cada registro recuperado `tmp` ao registro buscado `d`, usando um operador de igualdade `==` sobrecarregado. A função usa o fato de que o arquivo é não-sequencial, pesquisando-o registro a registro, não byte a byte. Para estar seguro os registros são formados de bytes e todos os bytes pertencendo a um registro particular têm que ser lidos, mas somente os bytes exigidos pelo operador de igualdade participam da comparação.

FIGURA 1.5 Listagem de um programa para gerenciar arquivos de acesso não-sequencial.

```
***** personal.h *****
#ifndef PERSONAL
#define PERSONAL

#include <fstream.h>
#include <string.h>

class Personal {
public:
    Personal();
    Personal(char*, char*, char*, int, long);
    void writeToFile(fstream&) const;
    void readFromFile(fstream&);
    void readKey();
    int size() const {
        return 9 + nameLen + cityLen + sizeof(year) + sizeof(salary);
    }
    bool operator==(const Personal& pr) const {
        return strcmp(pr.SSN, SSN) == 0;
    }
protected:
    const int nameLen, cityLen;
    char SSN[10], *name, *city;
    int year;
    long salary;
    ostream& writeLegibly(ostream&);
    friend ostream& operator<<(ostream& out, Personal& pr) {
        return pr.writeLegibly(out);
    }
    istream& readFromConsole(istream&);
    friend istream& operator>>(istream& in, Personal& pr) {
        return pr.readFromConsole(in);
    }
};

#endif

***** pessoal.cpp *****
#include "personal.h"

Personal::Personal() : nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
```

FIGURA 1.5 (continuação)

```

}

Personal::Personal(char *ssn, char *n, char *c, int y, long s) :
    nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
    strcpy(SSN,ssn);
    strcpy(name,n);
    strcpy(city,c);
    year = y;
    salary = s;
}

void Personal::writeToFile(fstream& out) const {
    out.write(SSN,9);
    out.write(name,nameLen);
    out.write(city,cityLen);
    out.write(reinterpret_cast<const char*>(&year),sizeof(int));
    out.write(reinterpret_cast<const char*>(&salary),sizeof(int));
}

void Personal::readFromFile(fstream& in) {
    in.read(SSN,9);
    in.read(name,nameLen);
    in.read(city,cityLen);
    in.read(reinterpret_cast<char*>(&year),sizeof(int));
    in.read(reinterpret_cast<char*>(&salary),sizeof(int));
}

void Personal::readKey() {
    char s[80];
    cout << "Enter SSN: ";
    cin.getline(s,80);
    strncpy(SSN,s,9);
}

ostream& Personal::writeLegibly(ostream& out) {
    SSN[9] = name[nameLen] = city[cityLen] = '\0';
    out << "SSN = " << SSN << ", nome = " << nome
        << ", cidade = " << city << ", ano = " << year
        << ", salario = " << salary;
    return out;
}

istream& Personal::readFromConsole(istream& in) {
    char s[80];
}

```

FIGURA 1.5 (continuação)

```

cout << "SSN: ";
entrada.getline(s,80);
strncpy(SSN,s,9);
cout << "Name: ";
entrada.getline(s,80);
strncpy(name,s,nameLen);
cout << "City: ";
entrada.getline(s,80);
strncpy(city,s,cityLen);
cout << "Birthyear: ";
entrada >> year;
cout << "Salary: ";
entrada >> salary;
entrada.getline(s,80); // get '\n'
return in;
}

//***** student.h *****
#include "personal.h"

class Student : public Personal {
public:
    Student();
    Student(char*,char*,char*,int,long,char*);
    void writeToFile(fstream&) const;
    void readFromFile(fstream&);

    int size() const {
        return Personal::size() + majorLen; //major = especialidade
    }
protected:
    char *major;
    const int majorLen;
    ostream& writeLegibly(ostream&);

    friend ostream& operator<<(ostream& out, Student& sr) {
        return sr.writeLegibly(out);
    }
    istream& readFromConsole(istream&);

    friend istream& operator>>(istream& in, Student& sr) {
        return sr.readFromConsole(in);
    }
};

//***** student.cpp *****

```

FIGURA 1.5 (continuação)

```

#include "student.h"

Student::Student() : majorLen(10) {
    Personal();
    major = new char[majorLen+1];
}

Student::Student(char *ssn, char *n, char *c, int y, long s, char *m) :
    majorLen(11) {
    Personal(ssn,n,c,y,s);
    major = new char[majorLen+1];
    strcpy(major,m);
}

void Student::writeToFile(fstream& out) const {
    Personal::writeToFile(out);
    out.write(major,majorLen);
}

void Student::readFromFile(fstream& in) {
    Personal::readFromFile(in);
    in.read(major,majorLen);
}

ostream& Student::writeLegibly(ostream& out) {
    Personal::writeLegibly(out);
    major[majorLen] = '\0';
    out << ", especialidade = " << major;
    return out;
}

istream& Student::readFromConsole(istream& in) {
    Personal::readFromConsole(in);
    char s[80];
    cout << "Major: ";
    in.getline(s,80);
    strncpy(major,s,9);
    return in;
}

//***** database.h *****
#ifndef DATABASE
#define DATABASE

```

FIGURA 1.5 (continuação)

```

template<class T>
class Database {
public:
    Database();
    void run();
private:
    fstream database;
    char fName[20];
    ostream& print(ostream&);
    void add(T& d);
    bool find(const T& d);
    void modify(const T& d);
    friend ostream& operator<<(ostream& out, Database& db) {
        return db.print(out);
    }
};

template<class T>
Database<T>::Database() {
    cout << "File name: ";
    cin >> fName;
}

template<class T>
void Database<T>::add(T& d) {
    database.open(fName,ios::in|ios::out|ios::binary);
    database.seekp(0,ios::end);
    d.writeToFile(database);
    database.close();
}

template<class T>
void Database<T>::modify(const T& d) {
    T tmp;
    database.open(fName,ios::in|ios::out|ios::binary);
    while (!database.eof()) {
        tmp.readFile(database);
        if (tmp == d) { // sobrecarregado ==
            cin >> tmp; // sobrecarregado >>
            database.seekp(-d.size(),ios::cur);
            tmp.writeToFile(database);
            database.close();
            return;
        }
    }
}

```

FIGURA 1.5 (continuação)

```

database.close();
cout << "O registro a ser modificado não está no banco de dados\n";
}

template<class T>
bool Database<T>::find(const T& d) {
    T tmp;
    database.open(fName,ios::in|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == d) { // sobrecarregado ==
            database.close();
            return true;
        }
    }
    database.close();
    return false;
}

template<class T>
ostream& Database<T>::print(ostream& out) {
    T tmp;
    database.open(fName,ios::in|ios::binary);
    while (1) {
        tmp.readFromFile(database);
        if (database.eof())
            break;
        out << tmp << endl; //sobrecarregado <<
    }
    database.close();
    return out;
}

template<class T>
void Database<T>::run() {
    char option[5];
    T rec;
    cout << "1. Adiciona 2. Procura 3. Modifica um registro; 4. Sair\n";
    cout << "Entre com uma opção: ";
    cin.getline(option,4); //obtem a opção junto com '\n';
    while (cin.getline(option,4)) {
        if (*option == '1') {
            cin >> rec; // sobrecarregado >>
            add(rec);
        }
        else if (*option == '2') {

```

FIGURA 1.5 (continuação)

```

        rec.readKey();
        cout << "O registro é ";
        if (find(rec) == false)
            cout << "não ";
        cout << "esta no banco de dados\n";
    }
    else if (*option == '3') {
        rec.readKey();
        modify(rec);
    }
    else if (*option != '4')
        cout << "opção errada\n";
    else return;
    cout << *this; // sobrecarregado <<
    cout << "Entre com uma opção: ";
}
#endif

//***** database.cpp *****

#include "database.h"

template<class T>
Database<T>::Database() {
    cout << "Nome do arquivo: ";
    cin >> fName;
}

template<class T>
void Database<T>::add(T& d) {
    database.open(fName,ios::in|ios::out|ios::binary);
    database.seekp(0,ios::end);
    d.writeToFile(database);
    database.close();
}

template<class T>
void Database<T>::modify(const T& d) {
    T tmp;
    database.open(fName,ios::in|ios::out|ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);

```

FIGURA 1.5 (continuação)

```

        if (tmp == d) {
            cin >> tmp;
            database.seekp(-d.size(), ios::cur);
            tmp.writeToFile(database);
            database.close();
            return;
        }
    database.close();
    cout << "O registro a ser modificado não está no banco de dados\n";
}

template<class T>
bool Database<T>::find(const T& d) {
    T tmp;
    database.open(fName, ios::in | ios::binary);
    while (!database.eof()) {
        tmp.readFromFile(database);
        if (tmp == d) {
            database.close();
            return true;
        }
    }
    database.close();
    return false;
}

template<class T>
ostream& Database<T>::print(ostream& out) {
    T tmp;
    database.open(fName, ios::in | ios::binary);
    while (1) {
        tmp.readFromFile(database);
        if (database.eof())
            break;
        out << tmp << endl;
    }
    database.close();
    return out;
}

//***** useDatabase.cpp ****
#include <iostream>
#include "student.h"
#include "personal.h"

```

FIGURA 1.5 (continuação)

```

#include "database.h"

void main() {
    Database<Personal> db;
    // Database<Student> db;
    db.run();
}

```

A função `modify()` atualiza a informação armazenada em um registro particular. O registro é primeiro recuperado do arquivo, também usando a pesquisa seqüencial, e a nova informação é lida do usuário usando o operador sobreescarregado de entrada `>>`. Para armazenar o registro atualizado `tmp` no arquivo, `modify()` força o ponteiro de arquivo `database` a ir de volta para o início do registro `tmp` que acabou de ser lido; caso contrário, o registro que está a seguir de `tmp` no arquivo seria reescrito. A posição de início de `tmp` pode ser determinada de imediato, porque cada registro ocupa o mesmo número de bytes; em consequência, é suficiente pular de volta o número de bytes ocupados por um registro.

Isto é realizado chamando-se `database.seekp(-d.size(), ios::cur)`, onde `size()` precisa ser definido na classe `T`, que é o tipo de classe para o objeto `d`.

A classe genérica `Database` inclui mais duas funções. A função `add()` coloca um registro no final do arquivo. A função `print()` imprime o conteúdo do arquivo.

Para ver a classe `Database` em ação, temos que definir uma classe específica que determina o formato de um registro em um arquivo de acesso não-seqüencial. Como exemplo, definimos a classe `Personal` com cinco membros de dados, `SSN`, `name`, `city`, `year` e `salary`. Os três primeiros membros de dados são cadeias de caracteres (strings), mas somente `SSN` é sempre do mesmo tamanho; em consequência, o tamanho está incluído na sua declaração, `char SSN[10]`. Para se ter levemente mais flexibilidade com as outras duas cadeias de caracteres, duas constantes são usadas, `nameLen` e `cityLen`, cujos valores estão nos construtores. Por exemplo,

```

Personal::Personal() : nameLen(10), cityLen(10) {
    name = new char[nameLen+1];
    city = new char[cityLen+1];
}

```

Note que não podemos inicializar constantes com atribuições, como em

```

Personal::Personal () {
    nameLen = cityLen = 10;
    char name[nameLen+1];
    char city[cityLen+1];
}

```

Mas essa sintaxe peculiar usada em C++ para a inicialização de constantes em classes pode ser usada para inicializar variáveis.

A estocagem de dados a partir de um objeto exige um cuidado particular, que é a tarefa da função `writeToFile()`. O membro de dados `SSN` é o mais simples de se manusear. O número de Segurança Social inclui sempre nove dígitos; em consequência, o operador de saída `<<` pode ser usado. No entanto, os comprimentos dos nomes e das cidades e as seções de um registro no arquivo de dados designados para esses dois membros de dados precisam sempre ter o mesmo comprimento. Para garantir isso a função

`write()`, como em `out.write(name, nameLen)`, é usada para enviar as duas cadeias de caracteres para o arquivo, porque a função escreve um número especificado de caracteres da cadeia de caracteres – incluindo os caracteres nulos '\0', – que não é enviado com o operador `<<`.

Outro problema é colocado pelos membros de dados numéricos, `year` e `salary`, particularmente esse segundo membro de dados. Se `salary` é escrito para o arquivo com o operador `<<`, então `salary` de 50000 é escrito como uma cadeia de caracteres de cinco bytes de comprimento '50000', e `salary` de 100000, como uma cadeia de caracteres de seis bytes de comprimento '100000', o que viola a condição de que cada registro no arquivo de acesso aleatório deve ser do mesmo comprimento. Para evitar o problema, os números são estocados na forma binária. Por exemplo, 50000 é representado no membro de dados `salary` como a cadeia de caracteres de 32 bits 00000000000000001100001101010000 (assumindo que as variáveis `long` sejam estocadas em quatro bytes). Podemos agora tratar essa sequência de bits como representando não um número do tipo `long`, mas uma string de quatro caracteres, 00000000, 00000000, 11000011, 01010000, isto é, os caracteres cujos códigos ASCII são, em decimais, os números 0, 0, 195 e 80. Nesse modo, independentemente do valor de `salary`, o valor é sempre estocado em quatro bytes. Isto é realizado com a instrução

```
out.write(reinterpret_cast<const char*>(&salary), sizeof(long));
```

que faz a função `write()` tratar `salary` como uma cadeia de caracteres de quatro bytes de comprimento, convertendo o endereço `&salary` em `const char*` e especificando o comprimento do tipo `long`.

Uma abordagem similar é usada para se ler dados de um arquivo de dados, que é a tarefa de `readFromFile()`. Em particular, as cadeias de caracteres que devem ser armazenadas em membros de dados numéricos têm que ser convertidas de cadeias de caracteres para números. Para o membro `salary` isso é feito com a instrução

```
in.read(reinterpret_cast<char*>(&salary), sizeof(long));
```

que converte `&salary` para `char*`, com o operador `reinterpret_cast`, e especifica que os quatro bytes (`sizeof(long)`) devem ser lidos no membro de dados `salary` do tipo `long`.

Esse método de estocar registros em um arquivo de dados cria um problema de legibilidade, particularmente no caso de números. Por exemplo, 50000 está estocado como quatro bytes: dois caracteres nulos, um caractere especial e um P maiúsculo. Para um leitor humano, não é óbvio que esses caracteres representam 50000. Em consequência, uma rotina especial é necessária para produzir registros em uma forma legível. Isso é realizado sobrecarregando o operador `<<`, que usa uma função auxiliar `writeLegibly()`. A classe `database` sobrecarrega também o operador `<<`, que usa sua própria auxiliar `print()`. A função lê repetitivamente os registros do arquivo de dados no objeto `tmp` com `readFromFile()` e produz `tmp` em uma forma legível com o operador `<<`. Isso explica por que esse programa usa duas funções para ler e duas para escrever: uma é para manter os dados em um arquivo de acesso não-sequencial, e a outra é para ler e escrever os dados em uma forma legível.

Para testar a flexibilidade da classe `Database`, outra classe de usuário é definida, a classe `Student`. Esta classe é usada também para mostrar mais um exemplo de herança.

A classe `Student` usa os mesmos membros de dados que a classe `Personal`, sendo definida como uma classe derivada de `Personal`, mais um membro, o membro `major` do tipo de cadeia de caracteres. O processamento da entrada e da saída nos objetos da classe `Student` é muito similar ao da classe `Personal`, mas o membro adicional precisa ser considerado. Isso é feito redefinindo-se as funções da classe base e, ao mesmo tempo, reutilizando-as. Considere a função `writeToFile()` para escrever os registros dos estudantes em um arquivo de dados no formato de comprimento fixo:

```
void Student::writeToFile(fstream& out) const {
    Personal::writeToFile(out);
    out.write(major, majorLen);
}
```

A função usa `writeToFile()`, da classe base, para inicializar os cinco membros de dados, `SSN`, `name`, `city`, `year` e `salary`, e inicializa o membro `major`. Note que o operador de resolução de escopo `::` precisa ser usado para indicar claramente que `writeToFile()`, que está sendo definido para a classe `Student`, chama `writeToFile()` já definida na classe base `Personal`. No entanto, a classe `Student` herda, sem qualquer modificação, a função `readKey()` e o operador sobrecarregado `==`, porque em ambos os objetos `Personal` e `Student` a mesma tecla é usada para unicamente identificar qualquer registro, a saber, `SSN`.

■ 1.11 EXERCÍCIOS

- Se `i` é um valor inteiro e `p` e `q` são ponteiros para inteiros, quais das seguintes atribuições causam erro de compilação?
 - `p = &i;`
 - `p = *&i;`
 - `p = &i;`
 - `i = *&p;`
 - `i = **p;`
 - `i = &*p;`
 - `i = &p;`
 - `i = &*&p;`
 - `i = &*&p;`
- Identifique os erros:
 - ```
char* f(char *s) {
 char ch = 'A';
 return &ch;
}
```
  - ```
char *s1;
strcpy(s1,s2);
```
 - ```
char *s1;
s1 = new char[strlen(s2)];
strcpy(s1,s2);
```
- Uma vez que as declarações
 

```
int intArray[] = {1, 2, 3}, *p = intArray;
```

 tenham sido feitas, qual será o conteúdo de `intArray` e `p` depois de se executar
  - `*p++;`
  - `(*p)++;`
  - `*p++; (*p)++;`
- Usando somente ponteiros (sem indexação de matriz), escreva:
  - Uma função para somar todos os números em uma matriz de inteiros.
  - Uma função para remover todos os números ímpares de uma matriz ordenada; a matriz deve permanecer ordenada. Seria mais fácil escrever essa função se a matriz não estivesse ordenada?
- Usando somente ponteiros, implemente as seguintes funções de cadeias de caracteres:
  - `strlen()`
  - `strcmp()`
  - `strcat()`
  - `strchr()`

6. Qual é a diferença entre `if (p == q) {...} e if(*p == *q){...}?`
7. As primeiras versões do C++ não suportavam formatos (templates), mas as classes genéricas podiam ser introduzidas usando-se macros parametrizadas. Em que aspecto o uso de formatos é melhor do que o uso de macros?
8. Qual é o significado das partes `private`, `protected` e `public` das classes?
9. Qual deve ser o tipo de construtores e destrutores definidos nas classes?
10. Assuma a seguinte declaração de classe:

```
template<class genTipo>
class genClasse {
 ...
 char aFuncao(...);
 ...
};
```

O que está errado com essa definição de classe?

```
char genClasse::aFuncao(...){ ... };
```

11. A sobrecarga é uma ferramenta poderosa em C++, mas existem algumas exceções. Que operadores não podem ser sobreescritos?
12. Se a classeA inclui uma variável n private, uma variável m protected e uma variável k public, e a classeB é derivada da classeA, quais dessas variáveis podem ser usadas na classeB? Pode n, na classeB, se tornar private? protected? public? E quanto às variáveis m e k? Faz diferença se a derivação da classeB for private, protected ou public?

13. Transforme a declaração

```
template<class genTipo,> int size = 50>
class genClasse {
 genTipo estoCagem[tamanho];

 void funMembro() {

 if (algumaVariavel < tamanho) { }

 }
};
```

que usa a variável de tipo inteiro tamanho como um parâmetro para o template, em uma declaração de genClasse, que não inclua tamanho como parâmetro para o template e que ainda leve em consideração o valor de tamanho. Considere uma declaração do construtor de genClasse. Há alguma vantagem de uma versão sobre a outra?

14. Qual é a diferença entre funções-membro que são do tipo `virtual` e aqueles que não o são?
15. O que acontece se a declaração de genClasse

```
class genClasse {

 virtual void process01(char);
 virtual void process02(char);
```

```
};
```

for seguida por esta declaração de classeDerivada?

```
class classeDerivada : public genClasse {

 void process01(int);
 int process02(char);
};
```

Que funções-membro são invocadas se a declaração de dois ponteiros

```
genClasse *objetoPtr1 = &classeDerivada,
*objetoPtr2 = &classeDerivada;
```

é seguida por estas instruções?

```
objetoPtr1->process01(1000);
objetoPtr2->process02('A');
```

## ■ 1.12 TAREFAS DE PROGRAMAÇÃO

1. Escreva uma classe Fração que defina a soma, a subtração, a multiplicação e a divisão de frações, sobreescrargendo-se os operadores padrões para essas operações. Escreva um membro de função para fatores de redução e sobreescrita dos operadores de E/S, para entrar e sair com frações.
2. Escreva a classe Quaternion que define as quatro operações dos quaternions e as duas operações de E/S. Os quaternions, como definidos em 1843 por William Hamilton e publicado em sua *Lectures on Quaternions*, em 1853, são uma extensão dos números complexos. Os quaternions são quádruplas de números reais,  $(a, b, c, d) = a + bi + cj + dk$ , onde  $i = (1, 0, 0, 0)$ ,  $j = (0, 1, 0, 0)$  e  $k = (0, 0, 1, 0)$ , e as seguintes equações valem:

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ ij &= k, \quad jk = i, \quad ki = -j, \quad ji = -k, \quad kj = -i, \quad ik = -j \\ (a + bi + cj + dk) &+ (p + qi + rj + sk) \\ &= (a + p) + (b + q)i + (c + r)j + (d + s)k \\ (a + bi + cj + dk) &\cdot (p + qi + rj + sk) \\ &= (ap - bq - cr - ds) + (aq + bp + cs - dr)i \\ &\quad + (ar + cp + dq - bs)j + (as + dp + br - cq)k. \end{aligned}$$

Use essas equações na implementação de uma classe quaternion.

3. Escreva um programa para reconstruir um texto a partir da concordância de palavras. Esse era um problema real de se reconstruir alguns textos não publicados dos Papiros do Mar Morto usando concordâncias. Por exemplo, aqui está o poema de William Wordsworth, *Nature and the Poet*, e uma concordância de palavras que correspondem ao poema.

So pure the sky, so quiet was the air!  
 So like, so very like, was day to day!  
 Whene'er I look'd, thy image still was there;  
 It trembled, but it never pass'd away.

A concordância de 33 palavras é esta:

```

1:1 so quiet was the *air!
1:4 but it never pass'd *away.
1:4 It trembled, *but it never
1:2 was *day to day!
1:2 was day to *day!
1:3 thy *image still was there;
1:2 so very like, *was day
1:3 thy image still *was there;
1:3 *Whene'er I look'd,

```

Nessa concordância cada palavra está mostrada no contexto de até cinco palavras, e a palavra referida em cada linha está precedida de um asterisco. Para concordâncias maiores, dois números têm que ser incluídos, um número correspondendo a um poema e um número da linha onde as palavras podem ser encontradas. Por exemplo, assumindo que 1 é o número de *Nature and the Poet*, a linha "1:4 but it never pass'd \*away" significa que a palavra "away" neste poema é encontrada na linha 4. Note que as marcas de pontuação estão incluídas no contexto.

Escreva um programa que carregue uma concordância a partir de um arquivo e crie um vetor onde cada célula esteja associada a uma linha da concordância. Então, usando a busca binária, reconstrua o texto.

4. Modifique o programa do estudo de caso para manter uma ordem durante a inserção de novos registros no arquivo de dados. Isso exige a sobre carga do operador < em `Personal` e em `Student`, para serem usados em uma função `add()` modificada em `Database`. A função encontra uma posição apropriada para um registro d, move todos os registros no arquivo para criar espaço e escreve d no arquivo. Com a nova organização do arquivo de dados, `find()` e `modify()` podem também ser modificados. Por exemplo, `find()` para a busca seqüencial quando ela encontra um registro maior do que o registro procurado (ou atinge o fim de arquivo). Uma estratégia mais eficiente pode usar a busca binária, discutida na Seção 2.7.
5. Escreva um programa que mantenha uma ordem no arquivo de dados indiretamente. Use um vetor de ponteiros de posição de arquivo (obtido por meio de `tellg()` e `tellp()`) e mantenha o vetor na ordem classificada, sem mudar a ordem dos registros no arquivo.
6. Modifique o programa do estudo de caso para remover os registros do arquivo de dados. Defina a função `isNull()` nas classes `Personal` e `Student` para determinar se um registro é nulo. Defina a função `writeNullToFile()` nas duas classes para sobre escrever um registro a ser removido por um registro nulo. Um registro nulo pode ser definido como tendo um caractere não-numérico (uma lápide) na primeira posição do membro `SSN`. Então defina a função `remove()` em `Database` (muito similar a `modify()`), que localize a posição de um registro a ser removido e o sobre escreva com o registro nulo. Depois que uma sessão for terminada, um destrutor `Database` deve ser invocado, que copie os registros não nulos para um novo arquivo de dados, remova o antigo arquivo de dados e renomeie o novo arquivo de dados com o nome do antigo arquivo de dados.

## Bibliografia

- Breymann, Ulrich, *Designing Components with the C++ STL*, Harlow: Addison-Wesley, 2000.
- Budd, Timothy, *Data Structures in C++ Using the Standard Template Library*, Reading, MA: Addison-Wesley, 1998.
- Cardelli, Luca and Wegner, Peter, "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys* 17 (1985), 471–522.

Deitel, Harvey M. and Deitel, P.J., *C++: How to Program*, Upper Saddle River, NJ: Prentice Hall, 1998.

Egg, Raimund K., *Programming in an Object-Oriented Environment*, San Diego: Academic Press, 1992.

Flaming, Bryan, *Practical Data Structures in C++*, New York: Wiley, 1993.

Johnsonbaugh, Richard and Kalin, Martin, *Object-Oriented Programming in C++*, Upper Saddle River, NJ: Prentice Hall, 1999.

Khoshafian, Setrag and Razmik, Abnous, *Object Orientation: Concepts, Languages, Databases, User Interfaces*, New York: Wiley, 1995.

Lippman, Stanley B. and Lajoie, Josée, *C++ Primer*, Reading, MA: Addison-Wesley, 1998.

Meyer, Bertrand, *Object-Oriented Software Construction*, Upper Saddle River, NJ: Prentice Hall, 1997.

Stroustrup, Bjarne, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1997.

Wang, Paul S., *C++ with Object-Oriented Programming*, Boston: PWS, 1994.