DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Estruturas de dados

Ivan Luiz Marques Ricarte

http://www.dca.fee.unicamp.br/~ricarte/

2008

Sumário

Estruturas de dados

Em diversos contextos, disciplinas associadas à programação recebem a denominação de "processamento de dados". Esta denominação não é gratuita — de fato, embora seja possível criar procedimentos que não manipulem nenhum dado, tais procedimentos seriam de pouco valor prático. Uma vez que procedimentos são, efetivamente, processadores de dados, a eficiência de um procedimento está muito associada à forma como seus dados são organizados. Estrutura de dados é o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento.

Nesta apostila são apresentadas algumas estruturas de dados, com ênfase naquelas que são utilizadas posteriormente no decorrer do texto. Assim, algumas estruturas de importância para outros tipos de aplicações — como a representação de matrizes esparsas, fundamental para a área de computação científica — não estão descritas aqui.

As estruturas de dados definem a organização, métodos de acesso e opções de processamento para coleções de itens de informação manipulados pelo programa. Dois tipos de coleções são apresentados. Estruturas lineares são aquelas que mantém os seus itens de forma independente de seus conteúdos, ou seja, na qual qualquer tipo de interpretação dos dados que são armazenados é irrelevante para a manutenção da estrutura. Estruturas associativas são aquelas que levam em consideração a interpretação do valor (ou de parte dele) para a manutenção dos itens na estrutura.

Apresenta-se inicialmente uma visão conceitual de cada tipo de estrutura, com ilustrações que utilizam estruturas pré-definidas na biblioteca padrão de gabaritos (STL) da linguagem C++. São apresentados também aspectos da organização interna de uma estrutura de dados. Tais aspectos são relevantes para o projeto e implementação de uma nova estrutura de dados e normalmente não são manipulados por um programador que simplesmente utiliza estruturas já existentes na linguagem. Entretanto, é importante que o usuário detenha tal conhecimento por dois motivos. O primeiro é simplesmente ter parâmetros para poder selecionar qual a implementação mais adequada, se houver mais de uma disponível, para a sua aplicação. O segundo motivo é ter conhecimento para, se for necessário por não haver nenhuma implementação disponível, desenvolver a sua própria implementação de uma estrutura adequada às suas necessidades.

Capítulo 1

Tipos de dados

Internamente, todos os dados são representados no computador como seqüências de bits, ou seja, uma seqüência de dígitos binários (o nome *bit* é derivado da expressão *binary digit*), onde cada bit é usualmente representado pelos símbolos 0 ou 1. Esta é a forma mais conveniente para manipular os dados através de circuitos digitais, que podem diferenciar apenas entre dois estados (*on* ou *off*, verdadeiro ou falso, 0 ou 1). Em *assembly*, tipicamente todos os valores escalares são representados na forma de *bytes* (grupos de 8 bits), *words* (dois bytes ou 16 bits) ou *long words* (quatro bytes ou 32 bits) — uma seqüência de n bits pode representar uma faixa com 2^n valores distintos. Nestes casos, a interpretação desses valores é tipicamente delegada ao programador da aplicação.

Em linguagens de programação de alto nível, é desejável ter flexibilidade para lidar com diferentes interpretações para essas seqüências de bits de acordo com o que elas representam. Esse grau de abstração é oferecido por meio dos tipos primitivos da linguagem, que estabelecem estruturas de armazenamento e conjuntos de operações para esses valores. Além disso, as linguagens de programação oferecem usualmente mecanismos para trabalhar com conjuntos de valores e, em alguns casos, oferecem a possibilidade de criar novos tipos que podem ser usados pelos programadores. Tais princípios de representação são apresentados a seguir.

1.1 Tipos primitivos

O formato de representação interna (ou seja, como uma seqüência de bits é traduzida para um valor) pode variar de computador para computador, embora haja um esforço crescente para uniformizar a representação de tipos básicos. Assim, um caráter usualmente ocupa um byte com conteúdo definido por algum padrão de codificação (EBCDIC, ASCII, Unicode, ISO); um número inteiro tem uma representação binária inteira (sinal e magnitude, complemento de um ou complemento de dois); e um valor real é usualmente representado tipicamente no formato sinal, mantissa e expoente. Alguns desses formatos são apresentados no Apêndice 4.

A linguagem C++, apresentada no Apêndice 5, suporta a definição de valores escalares através da declaração de variáveis em um dos seus tipos de dados básicos, que são: `bool`, para representar um valor booleano; `char`, para representar um caráter; `int`, para um valor numérico inteiro; `float`, para um valor numérico real com precisão simples; e `double`,

para representar um valor numérico real com precisão dupla.

1.1.1 Valores booleanos

Uma variável do tipo `bool` de C++ pode receber um valor `true` ou `false`, apenas:

```
bool flag = false;
...
if (...) flag = true;
```

Quando há uma expressão lógica, associada por exemplo a uma condição em um comando `if` ou `while`, o seu resultado é um valor do tipo `bool`. Os operadores relacionais em C++, tipicamente usados nesse tipo de expressões, são:

>	maior que
>=	maior que ou igual a
<	menor que
<=	menor que ou igual a
==	igual a
!=	diferente de

Observe que o operador de igualdade é ==, e não =. Esta é uma causa comum de erros para programadores que estão acostumados com outras linguagens onde = é um operador relacional.

Valores booleanos podem ser combinados com os operadores booleanos:

`&&`	AND
`||`	OR
!	NOT

O operador `&&` *(and)* resulta verdadeiro quando os dois valores de seus operandos são verdadeiros. O operador `||` *(or)* resulta verdadeiro quando pelo menos um de seus dois operandos é verdadeiro. Além destes dois conectores binários, há também o operador unário de negação, `!`, que resulta falso quando o operando é verdadeiro ou resulta verdadeiro quando o operando é falso.

Expressões lógicas complexas, envolvendo diversos conectores, são avaliadas da esquerda para a direita. Além disto, `&&` tem precedência maior que `||`, e ambos têm precedência menor que os operadores lógicos relacionais e de igualdade. Entretanto, recomenda-se sempre a utilização de parênteses em expressões para tornar claro quais operações são desejadas. A exceção a esta regra ocorre quando um número excessivo de parênteses pode dificultar ainda mais a compreensão da expressão; em tais casos, o uso das regras de precedência da linguagem pode facilitar o entendimento da expressão.

É bom ressaltar que na linguagem C não existe o tipo `bool`. Nesse caso, valores lógicos são representados como valores inteiros e a interpretação desses valores é que determina o valor lógico — um valor 0 equivale a `false` e qualquer valor diferente de 0 é interpretado como `true`.

1.1.2 Caracteres

Uma variável do tipo `char` ocupa um byte com o valor binário da representação de um caráter:

```
char letra = 'A';
```

O padrão de representação de caracteres básico, ASCII, permite representar 128 caracteres distintos (valores entre 0 e 127), entre os quais estão diversos caracteres de controle (tais com ESC, associado à tecla de *escape*, e CR, associado ao *carriage return*) e de pontuação. Uma tabela com a representação interna desses valores está no Apêndice 4.1. Há outros formatos além de ASCII, tais como o padrão ISO8859 para a representação associada à faixa de valores entre 128 e 255 organizando-a em diversos subconjuntos, dos quais o ISO8859-1 (Latin-1) é o mais utilizado. O padrão de representação Unicode integra várias famílias de caracteres em uma representação unificada (em um ou dois bytes), sendo que a base dessa proposta engloba as codificações ASCII e ISO8859-1.

Em C++, os valores de variáveis do tipo caráter podem ser manipulados diretamente pelo valor inteiro correspondente ou, de forma mais conveniente, pela representação do caráter entre aspas simples. Assim, o caráter `'A'` equivale a uma seqüência de bits que corresponde ao valor hexadecimal 41 (representada em C++ pela seqüência `0x41`), pelo valor decimal 65 ou ainda pelo valor octal 101 (representada em C++ por 0101 — são octais os valores iniciados com o dígito 0).

Além dos caracteres alfanuméricos e de pontuação, que podem ser representados em uma função diretamente pelo símbolo correspondente entre aspas simples, C++ também define representações para caracteres especiais de controle do código ASCII através de seqüências de escape iniciados pelo símbolo \ (barra invertida ou contrabarra). As principais seqüências são apresentadas na Tabela 1.1.

Tabela 1.1 Caracteres representados por seqüências de escape.

`\n`	nova linha	`\t`	tabulação
`\b`	retrocesso	`\r`	retorno de carro
`\f`	alimentação de formulário	`\\`	contrabarra
`\'`	apóstrofo	`\"`	aspas
`\0`	o caráter NUL	`\xxx`	qualquer padrão de bits `xxx` em octal

C++ não oferece, além das operações que podem ser aplicadas sobre a representação inteira de um caráter, operações primitivas específicas para a manipulação de caracteres. Ao invés disso, funções para esse fim estão disponíveis na biblioteca de rotinas padronizadas de C. A maior parte dessas rotinas são de verificação da categoria à qual o caráter pertence, como `isalpha` (é um caráter alfabético), `isdigit` (é a representação de um dígito) ou `isspace` (é espaço em branco). Como tais rotinas fazem herança da herança de C, o tratamento de valores booleanos associado a elas é a mesma daquela linguagem. Há ainda duas rotinas de conversão, `toupper` (converter para maiúscula) e `tolower` (converter para minúscula). Para usar uma dessas rotinas, o programador deve incluir o arquivo de cabeçalho `cctype`:

```
#include <cctype>
```

```
...
char ch;
...
if (isalpha(ch)) ...
```

1.1.3 Valores numéricos inteiros

O tipo `int` representa um valor inteiro que pode ser positivo ou negativo:

```
int total = 0;
```

As quatro operações aritméticas podem ser aplicadas a valores inteiros, por meio dos operadores `+`, `-`, `*` e `/`. É importante destacar que quando os dois operandos da divisão são valores inteiros, a operação de **divisão inteira** é realizada. Assim, o resultado da expressão `7/2` é 3, não 3,5. Há também o operador `%` (módulo) que resulta no resto da divisão inteira — por exemplo, `7%2` é 1.

A linguagem C++ oferece também operadores que trabalham sobre a representação binária de valores inteiros e caracteres. Estes operadores são:

`&`	*AND* bit-a-bit
`|`	*OR* bit-a-bit
`^`	*XOR* bit-a-bit
`<<`	deslocamento de bits à esquerda
`>>`	deslocamento de bits à direita
`~`	complemento de um (inverte cada bit)

Expressões envolvendo esses operadores tomam dois argumentos — exceto pelo operador `~`, que é unário. Por exemplo,

```
a = x & 0177;
b &= ~0xFF;
c >>= 4;
```

Na primeira expressão, a variável `a` recebe os sete bits menos significativos da variável `x`. A segunda expressão, que utiliza a forma abreviada de operação com atribuição, reseta os oito bits menos significativos da variável `b`. Na terceira expressão, a variável `c` tem seus bits deslocados de quatro posições à direita.

O número de bytes ocupado por este tipo (e conseqüentemente a faixa de valores que podem ser representados) refletem o tamanho "natural" do inteiro na máquina onde o programa será executado. Usualmente, quatro bytes (32 bits) são reservados para o tipo `int` nos computadores atuais, permitindo representar valores na faixa entre -2^{31} a $+2^{31}-1$, ou $-2\,147\,483\,648$ a $2\,147\,483\,647$ em complemento de dois (Apêndice 4.2).

Uma forma de descobrir a quantidade de bytes alocado a uma variável do tipo `int` é aplicar o operador `sizeof`, que indica o número de bytes associado a uma variável ou a um tipo. Por exemplo, a expressão

```
cout << sizeof(int) << endl;
```

apresenta na saída o valor 4.

Para expressar o valor máximo (maior positivo) ou o mínimo (mais negativo) associado a uma variável do tipo `int`, C++ define constantes simbólicas no arquivo de cabeçalho `limits`, `INT_MAX` e `INT_MIN`, respectivamente:

```
#include <limits>
...
cout << "Maximo: " << INT_MAX << endl;
cout << "Minimo: " << INT_MIN << endl;
```

Variáveis inteiras podem ser qualificadas na sua declaração como `short` ou `long` e `unsigned`. Um tipo `unsigned int` indica que a variável apenas armazenará valores não-negativos. Os modificadores `short` e `long` modificam o espaço reservado para o armazenamento da variável. Um tipo `short int` indica que (caso seja possível) o compilador deverá usar um número menor de bytes para representar o valor numérico — usualmente, dois bytes são alocados para este tipo. Uma variável do tipo `long int` indica que a representação mais longa de um inteiro deve ser utilizada, sendo que usualmente quatro bytes são reservados para variáveis deste tipo.

Estas dimensões de variáveis denotam apenas uma situação usual definida por boa parte dos compiladores, sendo que não há nenhuma garantia quanto a isto. A única afirmação que pode ser feita sobre a dimensão de inteiros em C++ é que uma variável do tipo `short int` não terá um número maior de bits em sua representação do que uma variável do tipo `long int`.

Em C++, a representação de constantes associadas a valores inteiros pode ser expressa em decimal, octal ou hexadecimal. Números com representação decimal são denotados por qualquer seqüência de algarismos entre `0` e `9` que inicie com um algarismo diferente de `0` — 10, 127, 512 etc. Números em octal são seqüências de algarismos entre `0` e `7` iniciadas por `0` — 012 (decimal 10), 077 (decimal 63). Números em hexadecimal são seqüências de algarismos entre `0` e `F` iniciadas com o prefixo `0x` — `0xF` (decimal 15), `0x1A` (decimal 26). As representações octal e hexadecimal são atrativas como formas compactas de representação de valores binários — cada algarismo da representação octal pode ser diretamente expandido para uma seqüência de três bits, e da representação hexadecimal para seqüências de quatro bits. Assim, tanto `127` quanto `0177` quanto `0x7F` correspondem a uma mesma seqüência de bits, 01111111.

1.1.4 Valores numéricos reais

Os tipos `float` e `double` representam valores em **ponto flutuante**, limitados apenas pela precisão da máquina que executa o programa. O tipo `float` oferece seis dígitos de precisão enquanto que `double` suporta o dobro da precisão de um `float`.

Valores com representação em ponto flutuante (reais) são representados em C++ através do uso do ponto decimal, como em `1.5` para representar o valor um e meio. A notação exponencial também pode ser usada, como em `1.2345e-6` ou em `0.12E3`.

A linguagem de programação C++ define para estes tipos as quatro operações aritméticas elementares, por meio dos operadores `+`, `-`, `*` e `/`. Outras operações são executadas por meio de rotinas da biblioteca matemática de C. Por exemplo,

```
#include <cmath>
...
double pi = 4*atan(1.0);
```

As rotinas definidas nessa biblioteca incluem funções trigonométricas, tais como `cos`, `sin`, `tan`, `acos`, `asin` e `atan`; hiperbólicas (`cosh`, `sinh`, `tanh`); exponenciais e logarítmicas (`exp`, `log`, `log10`); potências (`pow`, `sqrt`); e arredondamento (`floor`, `ceil`).

1.1.5 Declaração de variáveis

Variáveis representam uma forma de identificar por um nome simbólico uma região da memória que armazena um valor sendo utilizado por uma função. Em C++, uma variável deve estar associada a um dos tipos de dados primitivos ou a uma classe definida pelo programador, sendo neste caso denominada um objeto.

Toda variável que for utilizada em uma função C++ deve ser previamente declarada. A forma geral de uma declaração de variável é:

```
tipo  nome_variavel;
```

ou

```
tipo  nome_var1, nome_var2, ... ;
```

onde `nome_var1, nome_var2, ...` são variáveis de um mesmo tipo de dado. Para declarar uma variável escalar de nome `i` do tipo inteiro em um programa C++, a seguinte expressão seria utilizada:

```
int i;
```

Essa declaração reserva na memória um espaço para a variável `i`, suficiente para armazenar a representação binária em complemento de dois do valor associado à variável, que inicialmente é indefinido. Outros exemplos válidos de declaração de variáveis em C++ são:

```
int um_inteiro;
unsigned int outro_inteiro;
char c1, c2;
float salarioMedio;
double x,
       y;
```

Como pode ser observado no exemplo acima, diversas variáveis de um mesmo tipo podem ser declaradas em um mesmo comando, sendo que o nome de cada variável neste caso estaria separado por vírgulas.

Identificadores, ou nomes de variáveis, podem ser de qualquer tamanho, sendo que usualmente nomes significativos devem ser utilizados. C++ faz distinção entre caracteres maiúsculos e caracteres minúsculos, de forma que a variável de nome `salariomedio` é diferente de outra variável com nome `salarioMedio`.

Há restrições aos nomes de variáveis. Palavras associadas a comandos e definições da linguagem (tais como `if`, `for` e `int`) são **reservadas**, não podendo ser utilizadas para o

nome de variáveis. A lista de palavras reservadas em C são apresentadas no Apêndice 5.2. O nome de uma variável pode conter letras e números, mas deve começar com uma letra.

Caso se deseje ter um valor definido desde o princípio para uma variável, a declaração deve ser acompanhada da correspondente inicialização, como em

```
int a = 0,
    b = 20;
char c = 'X';
long int d = 12345678L;
```

Na inicialização da variável `d`, o sufixo `L` indica que essa constante é do tipo `long`.

Uma variável cujo valor não será alterado pelo programa pode ser qualificada como `const`, como em

```
const int notaMaxima = 10;
```

Neste caso, a tentativa de alterar o valor da variável `notaMaxima` seria sinalizada como um erro pelo compilador. Variáveis deste tipo são obrigatoriamente inicializadas no momento de sua declaração.

1.1.6 Ponteiros e referências

Em C++, é possível ter acesso não apenas ao valor de uma variável mas também ao seu endereço. Há duas formas de indicar nesta linguagem que uma determinada variável será manipulada através de seu endereço. Na forma mais simples, através de variáveis referências, o uso do endereço é transparente para o programador. Na outra forma, ponteiros, o programador é responsável por operar explicitamente com os endereços das variáveis.

O operador unário `&`, aplicado a uma variável existente, retorna o endereço de memória associado à variável. A manipulação explícita de endereços dá-se através da utilização de **ponteiros**. Ponteiros constituem um dos recursos mais utilizados na programação C e C++. Eles fornecem um mecanismo poderoso, flexível e eficiente de acesso a variáveis. Há computações que só podem ser expressas através das referências a variáveis, mecanismo suportado com o uso de ponteiros.

Para definir que uma variável vai guardar um endereço, o operador unário `*` é utilizado na declaração, como em

```
int *ap;
```

Neste exemplo, `ap` é uma variável do tipo **ponteiro para inteiro**, ou seja, ela irá receber um endereço de uma variável inteira. Para se obter o endereço de uma variável, o operador unário `&` pode ser utilizado. No exemplo acima,

```
int x;
ap = &x;
```

Após esta instrução, diz-se que `ap` **aponta para** `x`, ou seja, `ap` tem como conteúdo o endereço da variável `x`. É possível acessar o valor da variável `x` através do ponteiro usando o operador unário `*`, como em

```
int y;
y = *ap;
```

Neste caso, a variável `y` recebe o conteúdo da variável apontada por `ap`, ou seja, o conteúdo de `x` nesta seqüência de execução.

Observe que a combinação `*ap` é um inteiro, que pode assim ser atribuído a outro inteiro. Esta associação pode facilitar a compreensão da declaração de ponteiros. No exemplo, `int *ap` pode ser lido como "`*ap` é um inteiro."

Também seria possível definir o valor de `x` através do ponteiro, como em

```
*ap = y;
```

Neste caso, o conteúdo da variável apontada por `ap` (que é `x`) recebe o valor de `y`.

Ponteiros podem tomar parte em expressões aritméticas. Assim, a seguinte expressão é perfeitamente válida:

```
y = *ap + 1;
```

ou seja, `y` receberia o valor de `x` (variável apontada por `ap`) incrementado de um.

Observe que a expressão `ap + 1` expressa um incremento do ponteiro e não do conteúdo armazenado nesse endereço. Como `ap` é do tipo inteiro, essa expressão deve ser interpretada como "o inteiro que está armazenado na posição de memória seguinte ao inteiro armazenado em `ap`".

A aritmética de ponteiros, ou seja, a realização de operações aritméticas envolvendo ponteiros, é bastante restrita. Afinal, não faz muito sentido multiplicar ou dividir o valor de um ponteiro por outro. Operações válidas envolvem um deslocamento por um certo número de posições, tanto para a frente (soma de um valor inteiro a um ponteiro) como para trás (subtração de um valor inteiro de um ponteiro), e a distância, expressa pela diferença entre dois ponteiros. No caso de deslocamento, o resultado da operação é um ponteiro; para a distância, o resultado é um valor inteiro.

Nas operações da aritmética de ponteiros, a unidade do valor inteiro corresponde ao tamanho em bytes do tipo manipulado pelo ponteiro. Assim, se o valor de `ap` é 1000 e a execução ocorre em uma máquina na qual um inteiro é representado em dois bytes, então `ap+1` resulta em 1002. Porém, a mesma expressão executada em uma outra máquina com inteiros representados em quatro bytes retornaria 1004. Com a aritmética de ponteiros, no entanto , o programador não precisa se preocupar com esses detalhes do armazenamento, trabalhando simplesmente com o conceito de "o próximo inteiro."

O **ponteiro nulo** é um valor que representa "nenhuma posição de memória", sendo útil para indicar condições de erro e evitar o acesso a áreas inválidas de memória. Em C++, um ponteiro nulo é qualquer variável do tipo ponteiro cujo valor é 0.

Uma variável referência torna transparente para o programador o fato de que um ponteiro está sendo utilizado internamente para a manipulação da variável. Para declarar uma variável deste tipo, o operador `&` é novamente utilizado:

```
int& a;
```

No momento do uso, a variável referência é utilizada da mesma forma que uma variável normal. Por exemplo, para adicionar 10 ao conteúdo da posição associada à variável `a`, a seguinte expressão pode ser utilizada:

```
a += 10;
```

Observe que ponteiros podem ser declarados sem ter um valor inicial associado, mas isto não pode ocorrer com referências.

1.2 Tipos definidos pelo programador

Linguagens de programação orientadas a objetos, como C++, oferecem aos programadores a possibilidade de estender o conjunto de tipos que podem ser usados em suas implementações. O mecanismo básico para tal fim é a definição de **classes**.

A definição de uma classe nada mais contém do que a definição de uma estrutura interna e de um conjunto de operações que podem ser aplicados a variáveis desse tipo, que são chamadas de **objetos**. As operações podem ser implementadas como métodos, sendo neste caso definidas na forma de rotinas e invocadas pelo nome. Alternativamente, C++ permite que o comportamento de um operador já existente seja alterado por meio do mecanismo de sobrecarga de operadores.

Não está no escopo deste texto apresentar como classes e métodos são definidos ou como operadores são sobrecarregados, mas é importante compreender como esses elementos são utilizados. A classe `string` ilustra bem esses conceitos.

1.2.1 Strings em C++

Em C++, a definição da classe *string* torna transparente a forma de armazenamento da seqüência de caracteres e sua manipulação. Assim, o programador pode abstrair-se de detalhes como terminadores de seqüência ou dos limites de armazenamento associados ao tamanho do arranjo, preocupações usuais na manipulação de strings em C.

Para utilizar *strings* em C++, o programador deve incluir em seu programa o arquivo de cabeçalho `string`. Com essa inclusão, ele pode criar objetos desse tipo de forma similar à declaração de variáveis de tipos primitivos. Por exemplo, no segmento de código

```
#include <string>
 ...
string estouVazia;
```

um objeto *string* `estouVazia` é criado, neste caso sem nenhum conteúdo inicial. Outras formas possíveis de criação de objetos desse tipo são:

```
string socorro("Alguem me ajude!");
string sos(socorro);
string toAqui = "Ta me ouvindo?";
```

No primeiro caso, é utilizada a forma de inicialização de objetos, passando um argumento entre parênteses — no caso, uma *string* constante. O segundo exemplo é similar, mas usa para inicialização um objeto já existente ao invés da constante. No terceiro caso, é usada uma forma que combina a declaração do objeto e uma atribuição.

Métodos são semelhantes a funções, com a diferença que devem ser aplicados a um objeto. A forma de aplicação de um método dá-se através do operador **.** (ponto) após o objeto. Por exemplo, para obter o tamanho da seqüência armazenado no objeto `sos`, utiliza-se o método `size` através de uma expressão como a seguinte:

```
int tamanho = sos.size();
```

Há duas formas de obter os elementos individuais de uma seqüência. Uma utiliza o método `at`, que recebe um argumento inteiro que indica a posição desejada, sendo que 0 é a primeira posição. Por exemplo,

```
char c = sos.at(2);
```

teria como resultado o caráter `'g'`. A outra forma utiliza o conceito de **sobrecarga de operadores** da linguagem C++, que permite associar um novo comportamento aos operadores da linguagem quando aplicados a objetos de uma classe. No caso da classe `string`, um novo comportamento foi definido para o operador de indexação, `[]`. Assim, a expressão acima poderia ter sido representada como

```
char c = sos[2];
```

Outros operadores sobrecarregados para objetos da classe `string` incluem o operador de soma, +, para a concatenação de seqüências; e os operadores de comparação (==, !=, >, <, <= e <=), modificados para avaliar o conteúdo das seqüências e não seus endereços de memória, o que é causa de um erro comum na manipulação de arranjos de caracteres em C

1.2.2 Bibliotecas de classes

Embora o conjunto de tipos básicos de C++ seja relativamente restrito, a possibilidade de ampliar esses tipos com a definição de novas classes abre muitas possibilidades para os programadores. Em algumas situações, é necessário que o programador crie suas novas classes, com estruturas e comportamentos específicos para a sua aplicação. Há, entretanto, um grande conjunto de classes que são de uso comum e para as quais não faria sentido haver uma nova definição cada vez que um programador tivesse que fazer sua aplicação. Tais classes pré-definidas podem ser utilizadas por meio das bibliotecas de classes.

A classe `string` é um exemplo de uma classe que faz parte da biblioteca padrão de C++, ou seja, é um recurso que todo compilador da linguagem deve oferecer. Outros recursos oferecidos pela biblioteca padrão de C++ incluem as classes para manipulação de entrada e saída de dados *(streams)* e as classes e funções parametrizadas da *Standard Template Library* (STL).

As classes da STL utilizam o recurso de definição parametrizada suportada por C++. Tal recurso permite que uma definição genérica seja especificada para a manipulação de elementos de diferentes tipos de modo que, quando necessário, o compilador saiba instanciar corretamente uma definição para o tipo desejado. Considere por exemplo a definição de uma função `swap` para trocar os conteúdos de duas posições de memória. Para duas posições do tipo `int`, a definição seria

```
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
 }
```

Para utilizar o mesmo comportamento com outro tipo de dados — `double` ou `long`, por exemplo — seria necessário, sem o mecanismo de definição parametrizada, que novas funções fossem definidas com a substituição de `int` pelo tipo desejado. Com a definição parametrizada, isso não é necessário. A especificação da função genérica indica a posição onde um tipo (primitivo ou classe) irá ser substituído, quando necessário:

```
template<typename T> void swap(T& a, T& b) {
   T temp = a;
   a = b;
   b = temp;
 }
```

Quando o compilador encontrar uma invocação à função `swap`, como em

```
int a = 10, b = 39;
...
swap (a, b);
```

ele reconhece que é preciso construir uma instância da função para o tipo inteiro, automaticamente. Não é preciso que o programador dê nenhuma indicação adicional para isso.

Do mesmo modo, classes parametrizadas podem ser definidas com referências a elementos de tipos genéricos, que podem ser substituídos por tipos efetivos no momento da declaração dos objetos daquela classe. Para o programador, o uso de uma classe parametrizada é simples; requer que ele especifique, no momento da criação do objeto, qual é o tipo que deve ser utilizado internamente pela classe. Essa especificação dá-se através da indicação do tipo entre os símbolos de menor e maior. Por exemplo, seja `cp` uma classe parametrizada que requer a especificação de um tipo. Se for criado um objeto `c1` onde esse tipo seja `int`, então a expressão correspondente seria

```
cp<int> c1;
```

O conjunto de classes e algoritmos da biblioteca STL definem recursos de uso genérico para permitir que o programador trabalhe com estruturas de dados usuais. Esses recursos estão agrupados em três grandes categorias (*containers*, iteradores, algoritmos) e utilizam o mecanismo de definições parametrizadas para poder trabalhar com qualquer tipo de conteúdo. Para cada classe genérica há um conjunto básico de métodos para a sua manipulação (a API — *Application Programming Interface*). No fundo o conhecimento desta API é a única informação relevante para um programador que utiliza um elemento da biblioteca.

1.3 Tipos agregados

As linguagens de programação normalmente oferecem facilidades para construir agregados contíguos e uniformes, ou seja, nos quais todos os elementos são de um mesmo tipo e estão armazenados em uma área contígua de memória. São os **arranjos**; seus elementos podem ser acessados através de um índice representando a posição desejada.

Em C++, arranjos são definidos e acessados através do operador de indexação `[]`, como em:

```
int elem[5];
...
elem[0] = 1;
```

Neste exemplo, um arranjo de nome `elem` é definido. Este arranjo tem espaço para armazenar cinco valores inteiros, que serão referenciados no programa como `elem[0]`, `elem[1]`, `elem[2]`, `elem[3]` e `elem[4]`. Observe através desse exemplo que o primeiro elemento de um arranjo em C++ é sempre o elemento de índice 0 (`elem[0]`). Conseqüentemente, para um arranjo com `N` elementos o último elemento é o de índice `N-1` (`elem[4]`, no exemplo).

A implementação de arranjos está intimamente ligada ao conceito de ponteiros. Quando se cria, como acima, um arranjo com cinco inteiros, reserva-se o espaço na memória para armazená-los e atribui-se um nome para o endereço inicial dessa área — em outras palavras, um ponteiro. O acesso ao valor `i`-ésimo elemento do arranjo `elem`, `elem[i]`, equivale à expressão

```
*(elem + i)
```

As duas formas de indicar o elemento acessado podem ser usadas indistintamente. Entretanto, enquanto o valor de uma variável ponteiro que não seja `const` pode ser alterado, o valor base de um arranjo não pode.

É fundamental entender em que região da memória os dados de um arranjo serão armazenados. Todo processo mantém uma área de dados na memória e também uma área de pilha, que é utilizada na passagem de parâmetros entre funções e para o armazenamento de valores temporários, como as variáveis locais de uma função. Se um arranjo é declarado como uma variável local de alguma função, então os valores de seus elementos só serão preservados enquanto a função estiver "ativa", ou seja, enquanto ela ainda não tiver concluído seu escopo. Quando a função retorna, ela libera o espaço que ocupava na pilha e suas variáveis locais são descartadas.

Caso seja necessário que a informação seja mantida além do tempo de execução de uma função, ela deve estar alocada à área de dados. Em C e C++, há duas maneiras básicas de fazer isto. A primeira é ter a variável declarada fora do escopo de uma função. Deste modo, a variável é alocada, por padrão, à área de dados estática do programa. A outra forma é declarar a variável no escopo da função mas acrescentar na declaração a palavra-chave `static`, uma indicação de que a variável não deve ser armazenada na pilha como o padrão para variáveis locais.

Há uma terceira maneira de manter esses valores além do escopo da função na qual eles foram gerados, por meio da alocação dinâmica de memória. Neste caso, a aplicação solicita ao sistema operacional a expansão da sua área de dados obtendo espaço de uma área suplementar, o *heap*. Em C++, o operador `new` é utilizado para esse fim — ele recebe uma indicação de qual tipo serão os elementos dessa área e, no caso de agregados, para quantos elementos deve ser reservado espaço; o retorno é um ponteiro para o início da área alocada. Por exemplo,

```
string *ustr = new string;
int *iptr = new int[100];
```

Na primeira expressão, o operador `new` é utilizado para reservar espaço para um único elemento — neste caso, um objeto da classe `string`. Na outra expressão, uma área para cem valores inteiros é alocada.

Áreas que foram alocadas com o operador `new` devem ser liberadas após o uso. Um programa que aloca muitas áreas de memória sem liberá-las sofre do problema de **vazamento de memória**, que pode causar até mesmo a interrupção de sua execução por falta de recursos. A liberação de área alocada em C++ é feita com o operador `delete`. Um cuidado que deve ser tomado é que este operador deve usar a mesma forma que foi usada na alocação — se uma área para vários elementos foi alocada, o operador de liberação da área deve indicar o fato. Assim, para os exemplos acima, as formas corretas para os correspondentes usos de `delete` são

```
delete ustr;
delete [] iptr;
```

Se a forma `delete iptr` fosse utilizada, o compilador não indicaria um erro, mas apenas a primeira posição da área seria liberada e o vazamento de memória ainda existiria.

Capítulo 2

Estruturas lineares

Estruturas lineares são aquelas que mantêm os seus itens de informação de forma independente de seus valores. A única informação utilizada pela estrutura é a posição do item; qualquer manipulação relativa ao conteúdo ou valor desse item é atribuição da aplicação.

Estruturas lineares são, ao menos conceitualmente, naturais para programadores. Os arranjos oferecem uma forma básica de definir um agregado de dados linear e com acesso indexado. No entanto, são estruturas estáticas, ou seja, não há como modificar a dimensão de um arranjo após sua criação. Ademais, mesmo para uma estratégia de organização simples como a estrutura linear, há operações que não ocorrem eficientemente em um arranjo. Um exemplo evidente é a inserção de novo elemento em uma posição intermediária de um arranjo parcialmente preenchido — seria necessário mover todos os elementos já inseridos em posições posteriores uma posição adiante. Se o arranjo tiver vários elementos, tal operação pode ser extremamente lenta.

A STL de C++ oferece um elenco de classes que definem estruturas lineares, que podem ser utilizados em diferentes situações, de acordo com as necessidades do programador. As estruturas lineares básicas da STL de C++ são `vector`, `deque` e `list`.

2.1 vector

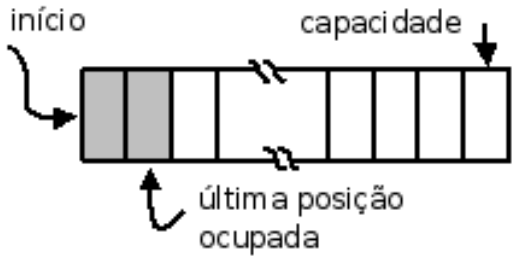
Uma coleção do tipo vetor pode ser vista como um arranjo cuja capacidade pode variar dinamicamente. Se o espaço reservado for totalmente ocupado e espaço adicional for necessário, este será alocado automaticamente — o programador não precisa se preocupar com a capacidade de armazenamento ou com a ocupação até o momento.

Sempre que uma coleção dinâmica for necessária, o programador deve considerar o vetor como uma opção, pois é o tipo de estrutura mais simples e com menor sobrecarga de memória para o seu armazenamento. Se as características de manipulação de elementos definidas pelo vetor forem no entanto inadequadas para a aplicação, então as outras opções de estruturas devem ser avaliadas.

2.1.1 Estrutura interna

Para que o programador possa utilizar a coleção sem se preocupar com tais aspectos, o implementador da classe providenciou os cuidados para que, internamente, as informações necessárias fossem mantidas — por exemplo, qual é a quantidade total de elementos e qual é a última posição ocupada na coleção (Figura 2.1).

Figura 2.1 Estrutura do tipo vetor



2.1.2 Criação

Para criar uma coleção desse tipo, é preciso incluir no programa o arquivo de cabeçalho `vector`. Por exemplo, para criar um objeto `vint` que é um vetor de elementos do tipo `int`,

```
#include <vector>
...
vector<int> vint;
```

Para criar um vetor de outros tipos definidos pelo programador, é preciso incluir as declarações associadas a esse outro tipo, usualmente em um arquivo de cabeçalho. Por exemplo, para criar um vetor de *strings*:

```
#include <vector>
#include <string>
...
vector<string> umaLista;
```

Quando criado dessa forma, o vetor está vazio e tem tamanho 0. É possível criar um vetor com algum conteúdo inicial usando a forma alternativa:

```
vector<int> ovi(10,0);
```

Neste caso, o vetor é criado com dez elementos com o mesmo valor (0).

À medida que o vetor vai recebendo novos elementos, o espaço interno é realocado, caso necessário. O programador pode obter quanto espaço há internamente para os elementos do vetor com a aplicação do método `capacity`:

```
int espaco = vint.capacity();
```

Se o programador tem uma boa noção de quantos elementos serão armazenados no vetor, ele pode evitar realocações intermediárias de espaço ao solicitar uma alteração nessa capacidade com o método `reserve`.

2.1.3 Operações

A forma básica de inserção de um elemento no vetor sua colocação ao final do vetor, por meio do método `push_back`. Por exemplo, para o vetor `vint` definido acima, o trecho de código

```
unsigned int pos = 0;
for (pos=0; pos<10; ++pos)
    vint.push_back(pos+10);
```

preenche as dez posições iniciais do vetor com os valores de 10 a 19.

O acesso ao conteúdo do vetor pode se dar de duas formas. O método `at` recebe por argumento o índice ou posição do elemento desejado — como para arranjos, o primeiro elemento tem índice 0 — e retorna a referência para o elemento indicado. A outra forma é por meio do operador de indexação `[]` sobrecarregado na classe, que é utilizado como se o vetor fosse um arranjo:

```
int valor = vint[pos];
```

A quantidade de elementos em um vetor é obtida com a aplicação do método `size`:

```
int tamanho = vint.size();
```

Para verificar se a coleção está vazia, o método `empty` (que retorna um valor booleano) é mais eficiente do que comparar `size` com o valor 0.

A remoção do elemento que está no final do vetor ocorre eficientemente com a aplicação do método `pop_back`, pois neste caso o tempo da operação independe da quantidade de elementos armazenados no vetor, ou seja, têm complexidade temporal constante. Deve-se observar que, como internamente o vetor utiliza para o armazenamento dos dados um arranjo, inserções e remoções de elementos em uma posição intermediária não são eficientes, pois têm complexidade linear.

2.1.4 Iteradores

A varredura, ou seja, a operação de percorrer os elementos de um agregado, pode realizado por meio das funções providas para o acesso a seus elementos. Por exemplo, para listar o conteúdo do vetor `vint`, o seguinte fragmento de código poderia ser utilizado:

```
cout << "vint: ";
for (pos=0; pos < vint.size(); ++pos)
   cout << vint[pos] << "   ";
cout << endl;
```

A forma mais eficiente e aplicável também a outras estruturas, entretanto, faz uso do conceito de **iteradores**. Um iterador é um objeto interno a um agregado que, por conhecer a forma como seus elementos são armazenados, implementa eficientemente a varredura. Por exemplo, um iterador para um vetor de inteiros é declarado

```
vector<int>::iterator itv;
```

O programador, por sua vez, não precisa se preocupar como os elementos são armazenados — apenas utiliza as funções associadas ao iterador para percorrer os elementos. Para obter a referência a um objeto iterador no início da coleção, o método `begin` é utilizado:

```
itv = vint.begin();
```

Já o método `end` retorna um iterador posicionado após o último elemento do agregado. Os operadores sobrecarregados `*`, `++` e `!=` são usados para, respectivamente, obter a referência para o elemento na posição corrente do iterador, avançar o iterador para o próximo elemento e comparar iteradores. Dependendo da estrutura interna da coleção, outros tipos de iteradores podem ser oferecidos, com mais operações de acesso. Iteradores bidirecionais sobrecarregam o operador de decremento (`--`). Iteradores de acesso direto permitem o uso da aritmética de ponteiros, ou seja, permitem saltos maiores do que uma posição durante a varredura.

Deste modo, o fragmento de código para listar os elementos do vetor `vint` usando o iterador `itv` pode ser concluído

```
cout << "vint: ";
while (itv != vint.end()) {
   cout << *itv << "   ";
   ++itv;
 }
cout << endl;
```

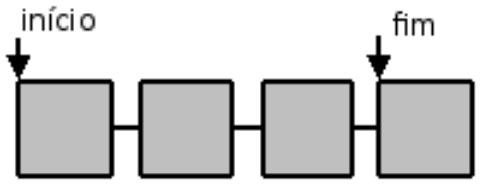
Há algumas operações das coleções que requerem como argumento um iterador, ou seja, trabalham durante a varredura da coleção. É o caso dos métodos `insert` e `erase`, por exemplo, que são utilizados para inserir ou para remover, respectivamente, um elemento de uma posição arbitrária da coleção.

2.2 deque

O deque é uma estrutura linear, similar a um vetor, mas com informação mantida internamente sobre a posição das suas duas extremidades, inicial e final (Figura 2.2).

Como no vetor, manipulações em posições intermediárias da estrutura, com os métodos `insert` e `erase`, não são eficientes. No entanto, o deque permite operar eficientemente com ambas as extremidades da coleção. Além dos métodos presentes em `vector`, `push_back` e `pop_back`, que respectivamente inserem ou removem o elemento na extremidade final, o deque tem implementações eficientes para os métodos `push_front` e `pop_front`, que manipulam a extremidade inicial da estrutura.

Figura 2.2 Estrutura do tipo deque



Como para qualquer estrutura seqüencial de C++, os métodos `size` e `empty` retornam informação sobre a ocupação da coleção. Iteradores são utilizados como em `vector`, como ilustrado no exemplo abaixo:

```
#include <deque>
...
  deque<int> dqi;

  int pos=0;
  while (pos < 10) {
      dqi.push_front(pos++);
      dqi.push_back(pos++);
  }

  deque<int>::iterator itd;
  for (itd = dqi.begin(); itd != dqi.end(); ++itd)
      cout << *itd << " ";
```

A execução do código desse exemplo apresenta na tela:

```
8 6 4 2 0 1 3 5 7 9
```

2.2.1 Stack

Além das coleções, a biblioteca STL também oferece alguns **adaptadores de coleções**, que usam alguma coleção internamente para oferecer um elemento com maior grau de abstração. Este é o caso de `stack` ou pilha, uma estrutura linear com restrição na política de acesso aos seus elementos — a ordem de saída é inversa á ordem de entrada. Esta política é usualmente denominada LIFO *(last in, first out)*, ou seja, último que entra é o primeiro que sai.

Um `stack` de STL oferece as operações `push` para inserir um elemento no topo da pilha e `pop` para remover o elemento no topo da pilha. Além dessas operações básicas de pilha, a classe tem também os métodos `top`, para inspecionar o elemento que está no topo, `empty` para testar se a pilha está vazia e `size` para obter a quantidade de elementos na pilha. O exemplo a seguir ilustra a utilização desses métodos:

```
#include <stack>
...
  stack<int> pilha;

  for (int pos=0; pos<5; ++pos)
      pilha.push(pos);

  while (! pilha.empty()) {
      cout << pilha.top() << " ";
      pilha.pop();
  }
  cout << endl;
```

Este exemplo, quando executado, apresenta na tela o resultado

```
4  3  2  1  0
```

Embora possa usar qualquer estrutura linear como elemento interno, a implementação padrão de `stack` utiliza o `deque`. Caso deseje-se utilizar outra estrutura, como um vetor, essa pode ser especificada no momento da declaração:

```
stack<int, vector<int> > pilha;
```

2.2.2 Queue

Assim como `stack`, a coleção do tipo `queue` é uma adaptação sobre uma estrutura linear, que implementa uma política restrita de acesso — neste caso, a política FIFO *(first in, first out)*, ou seja, o primeiro elemento que entra é o primeiro elemento que sai. Essa estrutura é a implementação de uma fila, que é suportada eficientemente por uma estrutura do tipo `deque`.

Assim como `stack`, as operações suportadas por um `queue` são `push`, `pop`, `front`, `empty` e `size`. A diferença está no comportamento do método `pop`, que remove o elemento que está no início da fila. Uma diferença está no método de inspeção de conteúdo — ao invés de `top`, na fila há os métodos `front` e `back` para inspecionar o elemento que está no início ou no fim da fila, respectivamente.

Compare o código anterior, que usa pilha, com este exemplo que usa fila:

```
#include <queue>
     ...
    queue<int> fila;
    for (int pos=0; pos<5; ++pos)
        fila.push(pos);

    while (! fila.empty()) {
        cout << fila.front() << " ";
        fila.pop();
    }
    cout << endl;
}
```

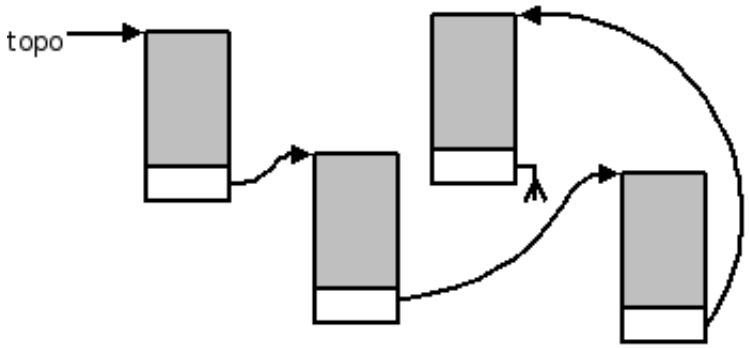
A execução desse código de exemplo resulta em

```
0 1 2 3 4
```

2.3 list

Uma estrutura linear do tipo `list` é uma implementação de uma lista ligada. Em uma estrutura deste tipo, os elementos na coleção não estão armazenados necessariamente em posições contíguas. Assim, junto a cada elemento é preciso manter a informação sobre a localização do próximo elemento da lista (Figura 2.3). Uma referência para o primeiro elemento da lista deve ser mantida para permitir o acesso a qualquer outro elemento.

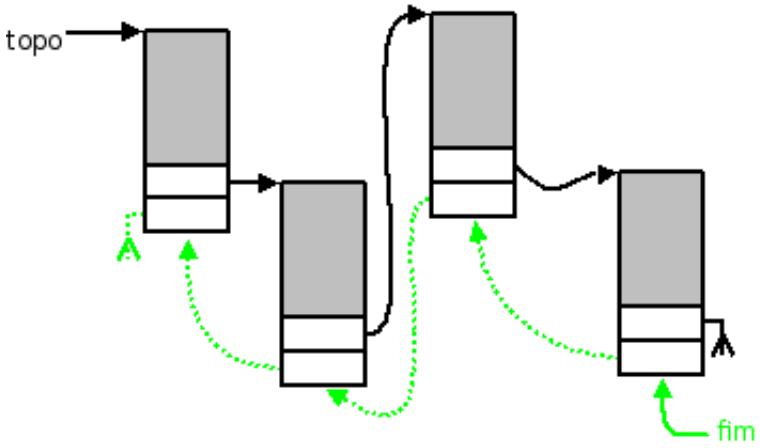
Figura 2.3 Estrutura do tipo lista — simplesmente ligada



Durante uma varredura na lista, a informação sobre a posição atual deve ser mantida para buscar o próximo elemento. Para cada nova varredura, entretanto, o acesso deve ser reiniciado a partir do primeiro elemento da coleção.

A classe `list` da biblioteca STL de C++ implementa, na verdade, uma lista duplamente ligada. Neste caso, além da informação sobre o próximo elemento da lista, associado a cada elemento deve haver também informação sobre a localização do elemento anterior da lista (Figura 2.4). Deste modo, varreduras nos dois sentidos — do início para o fim ou do fim para o início — são eficientemente implementadas.

Figura 2.4 Estrutura do tipo lista — duplamente ligada



A classe suporta os métodos de acesso presentes nas outras estruturas lineares. Os métodos `push_front` e `pop_front` manipulam o elemento no topo da lista, ao passo que os métodos `push_back` e `pop_back` manipulam o último elemento da lista. Pela sua característica, a estrutura do tipo lista executa eficientemente inserções e remoções de elementos em posições intermediárias. As implementações dos métodos `erase` e `insert` na posição corrente têm complexidade temporal constante, ou seja, dada uma posição para efetuar a operação, não há dependência em quantos elementos há antes ou depois dele na lista.

Em contrapartida, o acesso direto a elementos armazenados no interior da lista não é eficiente. Por exemplo, para acessar o sétimo elemento da lista é necessário percorrer os seis primeiros elementos. Por este motivo, a operação `at` não é aplicável e o operador de acesso indexado `[]` não é sobrecarregado para este tipo de estrutura.

Para percorrer uma lista, um iterador pode ser obtido como para as outras estruturas lineares. O exemplo a seguir ilustra o uso de um iterador para realizar a varredura em ordem reversa:

```
#include <list>
...
  list<int> lista;

  for (int pos=0; pos<5; ++pos)
      lista.push_back(pos);

  list<int>::reverse_iterator ril;
  ril = lista.rbegin();
  while ( ril != lista.rend() ) {
      cout << *ril << " ";
      ril++;
  }
  cout << endl;
```

O resultado da execução desse código é a apresentação dos elementos da lista do fim para o início:

```
4 3 2 1 0
```

O iterador para `list` é bidirecional, ou seja, sobrecarrega tanto o operador de incremento como o de decremento, mas não de acesso direto.

2.3.1 Aspectos de implementação

Uma **lista ligada** é uma estrutura que corresponde a uma seqüência lógica de entradas ou **nós**. Tipicamente, em uma lista ligada há um ou dois pontos conhecidos de acesso — normalmente o **topo** da lista (seu primeiro elemento) e eventualmente o fim da lista (seu último elemento). Cada nó armazena também a localização do próximo elemento na seqüência, ou seja, de seu **nó sucessor**.

Em uma lista simplesmente ligada (Figura 2.3), a informação em um nó pode ser abstraída para dois campos de interesse: *info*, o conteúdo do nó, e *next*, uma referência para o próximo nó da lista. A entrada que determina o topo da lista deve ser registrada à parte da lista ou em um nó especial, o **nó descritor**, com conteúdo vazio. A entrada que marca o fim da lista não precisa de indicação especial — tipicamente, a referência vazia como valor de *next* marca o final da lista.

Para apresentar como uma lista simplesmente ligada pode ser implementada, considera-se aqui que o nó de uma lista é um tipo definido pelo programador com a seguinte estrutura:

$$\begin{array}{lllll} \text{NODE} \models & \text{info} & : & \text{OBJECT} \\ & \text{next} & : & \text{NODE} \end{array}$$

O tipo OBJECT é usado aqui para indicar que o conteúdo poderia ser de qualquer tipo, de acordo com a aplicação.

Como listas são estruturas dinâmicas, normalmente são definidos procedimentos que permitem criar e remover nós na memória. Neste texto, a criação e remoção de um nó estarão associadas respectivamente aos procedimentos:

CREATENODE(OBJECT e). Aloca recursos (área de memória) para guardar a informação especificada nos argumentos. Retorna uma referência para o nó criado, do tipo NODE; e

DELETENODE(NODE n). Libera os recursos usados pelo nó.

A informação sobre a estrutura de uma lista ligada está distribuída ao longo de seus nós. Assim, a única informação adicional requerida para manter a lista é a especificação de seu nó descritor:

$$\text{LIST} \models \quad \text{top} \quad : \quad \text{NODE}$$

Na criação de uma lista, o nó descritor está inicialmente vazio, de forma que seu campo *next* tem o valor nulo. Assim, um procedimento para verificar se a lista está vazia deve verificar o valor desse campo. Esse procedimento está descrito no Algoritmo 2.1.

Algoritmo 2.1 Verificação se a lista ligada está vazia.

```
ISEMPTY(LIST l)
1  if l.top.next = NIL
2     then return true
3     else  return false
```

Estabelecer a conexão entre dois nós é uma operação simples e freqüente na manipulação de listas. Para estabelecer a ligação entre um nó já pertencente a uma lista e um novo nó, basta fazer com que o novo nó referencie no campo *next* o nó que anteriormente era referenciado pelo nó original — mesmo que esse campo tenha o valor nulo. Para concluir a conexão, o nó original deve ter atualizado o campo *next* para referenciar o novo nó.

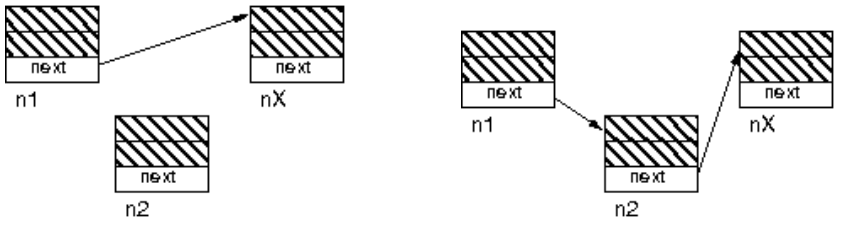
O efeito dessa conexão é ilustrado na Figura 2.5. Neste caso, o nó $n2$ deve ser inserido entre o nó $n1$ e o nó nX. A situação original, antes da ligação, mostra que $n1.next$ é nX — este é o valor que $n2.next$ deve assumir após a ligação, ao passo que $n1.next$ deverá receber o valor $n2$.

O procedimento LINKNODE, apresentado no Algoritmo 2.2, descreve como estabelecer essa ligação entre os dois nós que são passados como argumento.

Algoritmo 2.2 Ligação de dois nós.

```
LINKNODE(NODE n1, NODE n2)
1  n2.next ← n1.next
2  n1.next ← n2
```

Figura 2.5 Efeito da aplicação do procedimento LINKNODE.



(a) Antes da ligação (b) Após a ligação

Há duas possibilidades que podem ser consideradas para a inserção de um novo nó em uma das extremidades da lista, dependendo da opção de se inserir o novo nó no início (antes do primeiro elemento) ou no final (após o último elemento) da lista.

A primeira dessas possibilidades está representada através do procedimento INSERT, que recebe como argumentos as referências para a lista e para o nó a ser inserido. O Algoritmo 2.3 apresenta esse procedimento, que simplesmente estabelece a ligação do nó descritor com o novo nó.

Algoritmo 2.3 Inserção de nó no topo da lista ligada.

```
INSERT(LIST l, NODE n)
1  LINKNODE(l.top, n)
```

O procedimento que acrescenta um nó ao final da lista necessita varrer a lista completamente em busca do último nó, aquele cujo campo *next* tem o valor nulo. Para tanto, requer a utilização de uma variável interna que indica qual o nó está atualmente sendo analisado. No momento em que o campo *next* desse nó tiver o valor nulo, então sabe-se que o último nó da lista foi localizado. Esse procedimento, APPEND, está descrito no Algoritmo 2.4.

Algoritmo 2.4 Inserção de nó no final da lista ligada.

```
APPEND(LIST l, NODE n)
1  declare curr : NODE
2  curr ← l.top
3  while curr.next ≠ NIL
4  do curr ← curr.next
5  LINKNODE(curr, n)
```

De forma similar, o procedimento para retirar um nó do início da lista é mais simples que aquele para retirar um nó do fim da lista, pois este requer a varredura completa da lista. Nos dois casos, o valor de retorno é uma referência ao nó retirado; a partir dessa referência, a aplicação pode determinar o que deseja fazer com o nó, se manipular a informação nele contida ou simplesmente liberar os recursos com o procedimento DELETENODE. Um valor de retorno nulo indica que a operação foi especificada sobre uma lista vazia.

O procedimento que retira o primeiro nó da lista é apresentado no Algoritmo 2.5. Embora a linha 5 desse algoritmo não seja absolutamente necessária, ela garante que não há meios de acesso aos nós da lista exceto pelos procedimentos definidos. Se ela não estivesse presente, seria possível que a aplicação, ao obter o nó com a informação de endereço do seu antigo sucessor, tivesse acesso a um nó da lista de forma direta.

Algoritmo 2.5 Retirada do primeiro nó da lista ligada.

```
REMOVEFIRST(LIST l)
1  declare first : NODE
2  first ← l.top.next
3  if first ≠ NIL
4    then LINKNODE(l.top, first.next)
5         first.next ← NIL
6  return first
```

O procedimento para a retirada de um nó do fim da lista é descrito no Algoritmo 2.6. Da mesma forma que no procedimento de remoção do primeiro elemento da lista, a situação de manipulação de uma lista vazia deve receber tratamento especial. Como no procedimento de inserção, uma varredura de toda a lista é feita mantendo-se uma referência para o nó sob análise; adicionalmente, mantém-se uma referência para o nó anterior a este de forma a permitir a atualização da indicação do fim da lista.

Algoritmo 2.6 Retirada do último nó da lista ligada.

```
REMOVELAST(LIST l)
1  declare pred, curr : NODE
2  pred ← l.top
3  curr ← l.top.next
4  if curr ≠ NIL
5    then while curr.next ≠ NIL
6         do pred ← curr
7            curr ← curr.next
8         pred.next ← NIL
9  return curr
```

Dependendo da aplicação, outros procedimentos podem ser associados à manipulação de uma lista ligada, tais como obter o número de nós na lista, SIZE(); concatenar ou combinar duas listas, CONCAT(); inserir elemento em posição específica da lista, INSERTAT(); ou remover elemento em posição específica, REMOVEAT().

2.4 Busca em estruturas lineares

Para muitas aplicações, o conjunto de operações apresentado até aqui para estruturas lineares e seus adaptadores (como pilha e fila) suporta todas as necessidades. Entretanto, há

situações nas quais é necessário descobrir se um determinado valor está presente ou não na coleção. Em tais casos, operações de busca são utilizadas.

A busca em uma estrutura linear usualmente percorre a coleção a partir do primeiro elemento. Se este é o elemento procurado, então a busca está concluída. Caso contrário, a pesquisa deve prosseguir com o próximo elemento e assim sucessivamente, até que o elemento procurado seja encontrado ou até que a pesquisa conclua no último elemento da lista sem que o elemento desejado tenha sido encontrado.

Esse procedimento é ilustrado para a estrutura de lista simples apresentada acima. Neste caso, considera-se que o campo de conteúdo tem uma informação c que é utilizada como a chave para a busca — deve ser igual ao valor k passado como argumento. O procedimento, apresentado no Algoritmo 2.7, retorna uma referência para o campo de informação do nó encontrado ou o valor nulo se não for encontrado nenhum nó que satisfaça a condição de busca.

Algoritmo 2.7 Busca de nó com chave especificada em lista ligada.

FIND(LIST l, KEY k)
1 **declare** $curr : Node$
2 $curr \leftarrow l.top.next$
3 **while** $curr \neq$ NIL
4 **do if** $curr.info.c = k$
5 **then return** $curr.info$
6 **else** $curr \leftarrow curr.next$
7 **return** NIL

A biblioteca STL de C++ implementa algoritmos de busca que podem ser usados com as estruturas lineares aqui apresentadas. O algoritmo de busca mais simples é `find` que, como os demais procedimentos genéricos de STL, está declarado no arquivo de cabeçalho `algorithm`. Este procedimento recebe três argumentos, dois iteradores para especificar o início e o final da região a ser procurada e, como terceiro argumento, o valor que está sendo procurado.

Por exemplo, para procurar pelo valor 12 no vetor `vint` inteiro, o seguinte fragmento de código seria utiizado:

```
#include <algorithm>
...
vector<int>::iterator resultado;
resultado = find(vint.begin(), vint.end(), 12);
```

O resultado da busca também é um iterador sobre a coleção, que indicará a posição na qual o elemento foi encontrado ou, caso o elemento não esteja na coleção, retorna o mesmo iterador que `end` retorna.

Os algoritmos da biblioteca STL que são aplicados aos containers também podem ser aplicados a arranjos. Neste caso, o "iterador" de início é o ponteiro para o início do arranjo e, para o final, o ponteiro para a posição imediatamente após a última. Por exemplo, para realizar a busca de um valor `x` em um arranjo de inteiros `a` com `t` posições, a seguinte invocação é utilizada:

```
int* resultado = find(a, a+t, x);
```

O atrativo desse procedimento de busca linear é a simplicidade. Porém, seu uso está restrito a estruturas pequenas, pois caso contrário ele é muito ineficiente. O tempo de pesquisa cresce linearmente com o número de entradas na estrutura, ou seja, o algoritmo apresenta complexidade temporal $O(n)$. Se for possível manter os dados da estrutura segundo algum critério de ordenação, é possível melhorar o desempenho da busca numa estrutura linear. Neste caso, é possível utilizar no momento da busca uma estratégia análoga àquela utilizada ao procurar uma palavra no dicionário:

1. Faz-se uma estimativa da posição aproximada da palavra e abre-se na página estimada.
2. Se a palavra não foi encontrada nessa página, pelo menos sabe-se em que direção buscar, se mais adiante ou mais atrás no dicionário. O processo de estimar a nova página de busca repete-se na parte do dicionário que pode conter a palavra.

Esse mecanismo de busca só é aplicável porque existe uma ordenação possível das palavras com base na precedência das letras no alfabeto (a chamada ordem lexicográfica) e esta ordenação é utilizada no dicionário. Se o dicionário mantivesse as palavras sem nenhuma ordem, esse tipo de busca não seria possível. Com base nesse mesmo princípio, a busca em um vetor pode ser melhorada se seu conteúdo puder ser ordenado. Entretanto, deve-se observar que nem todos os domínios de dados são ordenáveis.

O algoritmo de **busca binária** utiliza esse princípio de busca numa coleção ordenada. No caso, a estimativa que é feita para a posição a ser buscada é a posição mediana do restante do vetor que ainda precisa ser analisado. No início, este "restante" é o vetor completo; assim, a sua posição central é a primeira a ser analisada. Se seu conteúdo não for a entrada buscada, analisa-se a metade que resta, ou a inferior (se a chave encontrada tem valor maior que a procurada) ou a superior (em caso contrário). O procedimento assim se repete, até que se encontre a entrada desejada ou que a busca se esgote sem que esta seja encontrada.

O Algoritmo 2.8 descreve essa forma de busca. Os dois argumentos são o vetor T e o elemento que será utilizado como chave de busca c, cujo tipo é genericamente aqui denotado como ELEMENT. O retorno é uma indicação se o elemento está presente ou não nesta coleção. As variáveis bot, mid e top referem-se a posições no vetor — respectivamente o início, o meio e o fim da área ainda por ser pesquisada. A notação $\lfloor x \rfloor$ denota o maior inteiro cujo valor é menor ou igual a x.

Deve-se observar que a busca binária assume que os dados da coleção estão mantidos em posições contíguas de memória, como forma de estimar qual deve ser a próxima posição a ser analisada a cada iteração do procedimento. Embora conceitualmente nada impeça que o algoritmo possa ser aplicado a uma lista ordenada, sua implementação não seria eficiente neste caso.

A biblioteca STL de C++ também traz uma implementação genérica para o algoritmo de busca binária. Como `find`, os argumentos para `binary_search` são os dois iteradores que definem a faixa de busca e o valor do elemento a ser buscado. O retorno, entretanto, é apenas um valor booleano: `true` se o valor foi localizado, `false` caso contrário:

```
#include <algorithm>
  ...
bool result;
```

Algoritmo 2.8 Busca binária em um vetor.

```
BINARYSEARCH(VECTOR T, ELEMENT c)
 1  declare bot, mid, top : INTEGER
 2  bot ← 0
 3  top ← T.size() − 1
 4  while bot ≤ top
 5  do mid ← ⌊(bot + top)/2⌋
 6     if c > T[mid]
 7       then bot ← mid + 1
 8       else if c < T[mid]
 9               then top ← mid − 1
10               else return true
11  return false
```

```
result = binary_search(vint.begin(), vint.end(), 12);
```

A manutenção da ordem em um vetor dá-se através de procedimentos auxiliares de ordenação, uma das áreas relevantes de estudos em estruturas de dados que é descrita na seqüência.

2.5 Ordenação

Procedimentos de ordenação trabalham com valores de um domínio no qual uma ordem parcial pode ser estabelecida entre seus elementos. Em outras palavras, dados dois elementos e_1 e e_2, é possível afirmar se $e_1 < e_2$ ou não. Quando é possível estabelecer esse tipo de relação entre dois elementos quaisquer de uma coleção, então é possível realizar a ordenação de seus elementos.

Por simplicidade, é assumido nessa apresentação que os conteúdos que serão ordenados estão sempre contidos em memória. Os algoritmos de ordenação que trabalham com essa restrição são denominados algoritmos de **ordenação interna**. Algoritmos de ordenação externa manipulam conjuntos de valores que podem estar contidos em arquivos maiores, armazenados em discos ou outros dispositivos de armazenamento externos à memória principal. Os algoritmos de ordenação interna (em memória) são convencionalmente baseados em estratégias de comparação *(quicksort, heapsort)* ou em estratégias de contagem *(radixsort)*.

2.5.1 Algoritmos básicos

Um algoritmo básico de ordenação é o algoritmo de ordenação pela **seleção do menor valor** *(selection sort)*, que pode ser sucintamente descrito como a seguir. Inicialmente, procure a entrada que apresenta o menor valor de todo o vetor. Uma vez que seja definido que posição contém esse valor, troque seu conteúdo com aquele da primeira posição do vetor; desta forma, o menor valor estará na posição correta. Repita então o procedimento para o restante do vetor, excluindo os elementos que já estão na posição correta.

Esse procedimento é descrito no Algoritmo 2.9, que recebe como argumento o vetor a ser ordenado. No procedimento, $pos1$ é a posição sob análise, a qual irá receber o menor valor do restante da coleção; ou seja, as posições anteriores a $pos1$ já estão ordenadas. A variável $pos2$ varre a parte da coleção que ainda não foi ordenada, enquanto a variável min mantém a posição na qual foi encontrado o menor valor até o momento.

Algoritmo 2.9 Ordenação de vetor pela seleção do menor valor.

```
SELECTIONSORT(VECTOR T)
1  declare min, pos1, pos2 : INTEGER
2  for pos1 ← 0 to T.size() − 2
3  do min ← pos1
4     for pos2 ← pos1 + 1 to T.size() − 1
5     do if T[pos2] < T[min]
6           then min ← pos2
7     T.swap(pos1, min)
```

Neste algoritmo, o laço de iteração mais externo indica o primeiro elemento no vetor não ordenado a ser analisado — no início, esse é o seu primeiro elemento. As linhas de 4 a 6 procuram, no restante do vetor, o elemento com menor valor. Na linha 7, o método $swap$ troca o conteúdo das duas entradas nas posições especificadas — nada é feito se as duas posições indicadas são iguais.

O exemplo a seguir ilustra como um pequeno vetor não-ordenado é manipulado por este algoritmo até ser completamente ordenado. Cada linha nesse quadro corresponde ao estado da coleção após a invocação do procedimento de troca de posições.

(pos)	0	1	2	3	4	
Inicial:	15	8	12	7	10	pos1=0, min=3
1:	7	8	12	15	10	pos1=1, min=1
2:	7	8	12	15	10	pos1=2, min=4
3:	7	8	10	15	12	pos1=3, min=4
Final:	7	8	10	12	15	

Este tipo de algoritmo de ordenação é razoável para manipular coleções com um pequeno número de elementos, mas à medida que esse tamanho cresce o desempenho torna seu uso inviável — sua complexidade temporal é $O(n^2)$, conseqüência do duplo laço de iteração que varre o vetor até o final.

Há outros algoritmos com mesma complexidade temporal mas com pequenas variações de estratégia para ordenação. Por exemplo, o algoritmo de ordenação da bolha *(bubble sort)* percorre a coleção de elementos trocando a posição de dois elementos consecutivos cada vez que estiverem fora de ordem. Ao final de cada iteração, o elemento maior estará no fim da coleção e pode ficar fora da próxima rodada; ao mesmo tempo, elementos menores mover-se-ão em direção ao início da lista.

O seguinte quadro mostra o comportamento do algoritmo da bolha para ordenar a mesma coleção do exemplo acima. A linha horizontal marca o ponto no qual cada uma das varreduras internas é concluída.

(pos)	0	1	2	3	4	
Inicial:	15	8	12	7	10	
1:	8	15	12	7	10	troca 0, 1
2:	8	12	15	7	10	troca 1, 2
3:	8	12	7	15	10	troca 2, 3
4:	8	12	7	10	15	troca 3, 4
5:	8	12	7	10	15	0, 1 ok
6:	8	7	12	10	15	troca 1, 2
7:	8	7	10	12	15	troca 2, 3
8:	7	8	10	12	15	troca 0, 1
9:	7	8	10	12	15	1, 2 ok
Final:	7	8	10	12	15	0,1 ok

2.5.2 Quicksort

Há, felizmente, outros algoritmos de ordenação com melhor comportamento de desempenho em situações onde a quantidade de elementos cresce. Ainda por meio de comparações entre os elementos, há algoritmos com complexidade temporal $O(n \log n)$.

O algoritmo de ordenação **quicksort** é baseado no princípio de "dividir para conquistar:" o conjunto de elementos a ser ordenado é dividido em dois subconjuntos (partições), que sendo menores irão requerer menor tempo total de processamento que o conjunto total, uma vez que o tempo de processamento para a ordenação não é linear com a quantidade de elementos. Em cada partição criada, o procedimento pode ser aplicado recursivamente, até um ponto onde o tamanho da partição seja pequeno o suficiente para que a ordenação seja realizada de forma direta por outro algoritmo. O ponto crítico é definir tais partições de forma que todos os elementos em uma delas não sejam maiores que os elementos na outra partição. Deste modo, ao concatenar as partições ordenadas a coleção inteira estará ordenada.

O Algoritmo 2.10 apresenta uma versão básica do procedimento QUICKSORT. Neste exemplo, os argumentos da função incluem a indicação das posições inicial e final, respectivamente $init$ e end, para determinar qual é o segmento do vetor a ser ordenado a cada invocação. O ponto crítico deste algoritmo está na forma de realizar a partição — um elemento é escolhido como **pivô**, ou separador de partições. Neste algoritmo, o pivô é selecionado sempre como o primeiro elemento do segmento que deve ser ordenado.

O procedimento é recursivo. Após a determinação do elemento pivô e da realização das trocas de posições para garantir que todos os elementos no segmento à esquerda do pivô são menores que os elementos à direita do pivô, o próprio procedimento QUICKSORT é invocado para ordenar cada um desses segmentos — o pivô é excluído pois já foi colocado na posição correta. O particionamento repete-se até que a condição de parada seja alcançada, quando o segmento tem no máximo uma posição, ou seja, $init$ não é menor que end.

Neste Algoritmo 2.10, a linha 2 estabelece a condição de parada da recursão. Se há mais de um elemento para ordenar, então a varredura deve ser realizada para encontrar a posição correta do pivô, que é mantida na variável *part*. O laço externo, com início na linha 5, é executado até que essa posição seja encontrada, condição estabelecida na linha 13. Os dois laços internos (linhas 6–7 e 8–9) procuram elementos que devem ser trocados para garantir que todos os elementos no segmento à esquerda não são maiores que os elementos do segmento à direita do pivô — essa troca é realizada na linha 11. Quando não há mais

Algoritmo 2.10 Ordenação de vetor por *quicksort*.

```
QUICKSORT(VECTOR T, INTEGER init, INTEGER end)
 1  declare pos1, pos2, part : INTEGER
 2  if init < end
 3    then pos1 ← init + 1
 4         pos2 ← end
 5         while true
 6         do while T[pos1] < T[init] ∧ pos1 ≤ end
 7            do pos1 ← pos1 + 1
 8            while T[pos2] > T[init] ∧ pos2 > init
 9            do pos2 ← pos2 − 1
10            if pos1 < pos2
11              then T.swap(pos1, pos2)
12              else part ← pos2
13                   break
14         T.swap(init, part)
15         QUICKSORT(T, init, part − 1)
16         QUICKSORT(T, part + 1, end)
```

elementos nessa condição, a posição correta para o pivô foi encontrada. Neste caso, o pivô é colocado nessa posição (linha 14) e QUICKSORT invocado para cada um dos segmentos.

Considere a aplicação do quicksort ao exemplo com os valores (15, 8, 12, 7, 10). As linhas horizontais delimitam novas invocações recursivas do procedimento quicksort:

(pos)	0	1	2	3	4	
1	15	8	12	7	10	init:0, end:4 part:4 – swap(0,4)
2	10	8	12	7	**15**	init:0, end:3
2a	10	8	7	12	**15**	swap(2,3) part: 2 – swap(0,2)
3	7	8	**10**	12	**15**	init:0, end:1 part:0 – swap(0,0)
4	**7**	8	**10**	12	**15**	init:0, end:-1
5	**7**	8	**10**	12	**15**	init:1, end:1
6	**7**	**8**	**10**	12	**15**	init:3, end:3
7	**7**	**8**	**10**	**12**	**15**	init:5, end:4

As posições destacadas em negrito são aquelas que o quicksort descobriu estarem no lugar correto, seja porque foi escolhida como pivô (correspondente à posição *part* anterior), seja porque os valores de *init* e *end* são idênticos na invocação.

Este algoritmo básico tem uma deficiência evidente se os valores de pivô são sempre menores ou sempre maiores que os demais elementos do vetor — por exemplo, se o vetor já está ordenado. Um melhor desempenho poderia ser obtido obtendo-se o valor médio de três amostras como ponto de partida para o pivô — por exemplo, entre os valores no início, meio e fim da partição sob análise. Dessa forma, haveria melhores chances de obter como

resultado partições de tamanhos mais balanceados, característica essencial para atingir um bom desempenho para esse algoritmo. Outra otimização usual é interromper a recursividade quando o segmento tem poucos elementos, de modo que algum algoritmo mais simples possa ser utilizado sem prejuízo do desempenho.

Quicksort é um algoritmo rápido em boa parte dos casos onde aplicado, com complexidade temporal média $O(n \log n)$. Entretanto, no pior caso essa complexidade pode degradar para $O(n^2)$. Mesmo assim, implementações genéricas desse algoritmo são usualmente suportadas em muitos sistemas — por exemplo, pela rotina `qsort` da biblioteca padrão da linguagem C e em versões antigas da rotina `sort` da biblioteca de algoritmos de C++. Entre os principais atrativos de *quicksort* destacam-se o fato de que na maior parte dos casos sua execução é rápida e de que é possível implementar a rotina sem necessidade de espaço de memória adicional.

Outros algoritmos de ordenação que apresentam a mesma complexidade temporal que a média de quicksort, $O(n \log n)$, são *merge sort*, *heap sort* e *introsort*.

2.5.3 Radix sort

Existe ainda uma terceira classe de algoritmos de ordenação, para os quais a definição da posição ordenada de um elemento se dá pela contagem do número de elementos com cada valor e não pela sua comparação com os demais elementos. O princípio básico é simples. Considere por exemplo uma coleção de elementos a ordenar onde as chaves podem assumir N valores diferentes. Cria-se então uma tabela com N contadores e varre-se a coleção do início ao fim, incrementando-se o contador correspondente à chave i cada vez que esse valor for encontrado. Ao final dessa varredura conhece-se exatamente quantas posições serão necessárias para cada valor; os elementos são transferidos para as posições corretas na nova coleção, agora ordenada.

Claramente, a aplicação desse princípio básico de contagem a domínios com muitos valores torna-se inviável. Por exemplo, se os elementos são inteiros de 32 bits, o algoritmo de contagem básico precisaria de uma tabela com cerca de quatro bilhões (2^{32}) de contadores.

Radix sort é um algoritmo baseado neste conceito de ordenação por contagem que contorna este problema ao aplicar o princípio da ordenação por contagem a uma parte da representação do elemento, a **raiz**. O procedimento é repetido para a raiz seguinte até que toda a representação dos elementos tenha sido analisada. Por exemplo, a ordenação de chaves inteiras com 32 bits pode ser realizada em quatro passos usando uma raiz de oito bits, sendo que a tabela de contadores requer apenas 256 (2^8) entradas.

O procedimento para execução de *radix sort* é descrito no Algoritmo 2.11. Para essa descrição, assumiu-se que elementos do vetor são inteiros positivos, que serão analisados em blocos de R bits a cada passo. As variáveis internas ao procedimento são $pass$, que controla o número de passos executados e também qual parte do elemento está sob análise, iniciando pelos R bits menos significativos; pos, que indica qual posição do vetor está sob análise; $radixValue$, o valor da parte do elemento (entre 0 e $2^R - 1$) no passo atual; $count$, a tabela de contadores; e T_{aux}, uma cópia do vetor ordenado segundo a raiz sob análise ao final de cada passo. A notação $\lceil x \rceil$ denota o menor inteiro cujo valor é maior ou igual a x.

O laço mais externo do algoritmo RADIXSORT (linhas 4 a 16) é repetido tantas vezes quantas forem necessárias para que a chave toda seja analisada em blocos de tamanho R bits. Utiliza-se na linha 4 um operador SIZEOFBITS para determinar o tamanho do elemento

Algoritmo 2.11 Ordenação de vetor por *radixsort*.

```
RADIXSORT(VECTOR T, INTEGER R)
 1  declare pass, pos, radixValue : INTEGER
 2  declare count : array[2^R] of INTEGER
 3  declare T_aux : VECTOR
 4  for pass ← 1 to ⌈SIZEOFBITS(INTEGER)/R⌉
 5  do for radixValue ← 0 to 2^R − 1
 6     do count[radixValue] ← 0
 7     for pos ← 0 to T.size() − 1
 8     do radixValue ← (T[pos] >> (R × (pass − 1))) & (2^R − 1)
 9        count[radixValue] ← count[radixValue] + 1
10     for radixValue ← 1 to 2^R − 1
11     do count[radixValue] ← count[radixValue] + count[radixValue − 1]
12     for pos ← T.size() − 1 to 0
13     do radixValue ← (T[pos] >> (R × (pass − 1))) & (2^R − 1)
14        T_aux[count[radixValue] − 1] ← T[pos]
15        count[radixValue] ← count[radixValue] − 1
16     T ← T_aux
```

em bits. Em C ou C++, este seria implementado com o operador `sizeof`, que retorna o tamanho do tipo em bytes, e a informação sobre quantos bits há em um byte.

O primeiro laço interno do algoritmo (linhas 5 e 6) simplesmente inicializa o arranjo de contadores, pois este será reutilizado em todos os demais passos do laço. No laço seguinte (linhas 7 a 9), o vetor é percorrido para avaliar o valor da raiz em cada posição (linha 8, usando os operadores binários SHIFT, `>>` e AND, `&`) e assim atualizar a contagem de valores (linha 9).

Na seqüência (linhas 10 e 11), gera-se a soma acumulada de contadores, o que permite avaliar quantas chaves com raiz de valor menor que o indicado existem. Essa informação permitirá que, no próximo laço (linhas 12 a 15), o vetor auxiliar T_{aux} seja preenchido colocando cada entrada do vetor T na nova posição que lhe corresponde segundo esse valor de raiz; cada vez que uma entrada é colocada na tabela, o valor do contador associado deve ser decrementado (linha 15) para que o elemento com a próxima raiz de mesmo valor seja colocada na posição correta, anterior à última ocupada.

Finalmente, o vetor T recebe a tabela ordenada segundo a raiz e o procedimento é repetido para o bloco de bits seguinte da chave. Após a varredura do último bloco (o mais significativo), o vetor estará completamente ordenada.

Considere novamente o exemplo do vetor com os elementos (15, 8, 12, 7, 10). Para o exemplo da ordenação com radix sort, será utilizada a representação binária sem sinal com quatro bits e uma raiz de dois bits:

pos	valor	binário
0	15	11.11
1	8	10.00
2	12	11.00
3	7	01.11
4	10	10.10

Neste caso, a tabela de contadores tem apenas quatro valores e o algoritmo estará completo em dois passos. Ao final da execução da linha 11, no primeiro passo, essa tabela terá os seguintes valores:

raiz	linhas 7–9	linhas 10–11
0	2	2
1	0	2
2	1	3
3	2	5

Ainda no passo 1, a execução das linhas 12–15 cria o vetor T_{aux}, que será o vetor T para o passo 2. A primeira posição a ser analisada é a 4, pois a varredura do vetor nesse laço é do final para o início. O valor dessa posição é 10 mas, na raiz (dois bits menos significativos), o valor 2 é analisado. Na tabela de contadores a contagem acumulada associada ao valor 2 é 3; portanto, o valor 10 deve ocupar a posição 2 do novo vetor e o contador é decrementado para 2. O processo se repete para a próxima posição (em ordem reversa), ou seja, o valor na posição 3, que é 7. Na raiz desse passo, o contador para o valor 3 é analisado e 7 ocupará a posição 4 do novo vetor. O contador então é decrementado para 4, de forma que o próximo valor que tiver raiz 3 irá ocupar a posição 3.

Ao final desse processo de transferência dos valores para o novo vetor, ao final do passo o vetor T terá os elementos na ordem (8, 12, 10, 15, 7) — que estaria ordenado se apenas os dois bits menos significativos fossem considerados. O passo 2 irá verificar os dois bits mais significativos nesse novo vetor. Como no passo 1, a tabela de contadores é calculada e, ao final da linha 11, terá o conteúdo:

raiz	linhas 7–9	linhas 10–11
0	0	0
1	1	1
2	2	3
3	2	5

A execução das linhas 12–15 no segundo passo leva todos os elementos do vetor para a posição correta no novo T_{aux}.

Este algoritmo requer, para seu correto funcionamento, que a ordenação utilizada em cada etapa intermediária seja estável, ou seja, que dois elementos com mesmo valor mantenham suas posições relativas na nova seqüência ordenada. *Radix sort* é um algoritmo rápido, mas apresenta como principal desvantagem a necessidade de espaço adicional de memória — uma área do mesmo tamanho ocupado pelo conjunto de elementos sendo ordenado é necessária para receber os dados re-ordenados após cada contagem. Quando o espaço de memória não é um recurso limitante, *radix sort* é um algoritmo atrativo, com complexidade temporal linear $O(n)$.

2.5.4 Ordenação em STL

A biblioteca STL oferece uma implementação de um algoritmo genérico para ordenação de valores de uma coleção. Para usar a função `sort`, é preciso especificar a faixa da coleção a ser ordenada por meio de dois iteradores, um para o início e outro para o final. Por exemplo, para ordenar o vetor `vint`:

```
#include <algorithm>
...
sort (vint.begin(), vint.end());
```

A mesma função pode ser aplicada para a ordenação de arranjos. Por exemplo:

```
int arr[10];
...
sort (arr, arr+10);
```

A função `sort` não tem valor de retorno. A implementação usada por STL, baseada no algoritmo *introsort*, garante que a complexidade temporal é $O(n \log n)$ tanto no caso médio como no pior caso.

Para estruturas do tipo `list`, a ordenação é realizada por meio de um método da classe e não pelo algoritmo genérico da biblioteca STL. Por exemplo, a execução do fragmento de código

```
    list<int> lista;

    for (int pos=0; pos<10; ++pos)
        lista.push_front(pos);

    list<int>::iterator itl;
    cout << "Inicial: ";
    for (itl=lista.begin(); itl != lista.end(); ++itl)
        cout << *itl << " ";

    lista.sort();

    cout << endl << "Final: ";
    for (itl=lista.begin(); itl != lista.end(); ++itl)
        cout << *itl << " ";
    cout << endl;
```

resulta em

```
Inicial: 9 8 7 6 5 4 3 2 1 0
Final: 0 1 2 3 4 5 6 7 8 9
```

STL tem também um adaptador de estrutura linear, `priority_queue`, que usa internamente o conceito de ordenação. As operações oferecidas são as mesmas de `queue`, mas com os elementos de maior valor sendo mantidos no topo da fila.

Capítulo 3

Estruturas associativas

Estruturas associativas são aquelas que permitem o acesso a seus elementos de forma independente de sua posição, com base apenas no seu valor. Em alguns casos, não é o valor do elemento completo que é utilizado, mas apenas uma parte dele; neste caso, essa parte é conhecida como **chave**.

Este tipo de estrutura é a base conceitual para a construção de tabelas, peça de fundamental importância para o desenvolvimento de *software* de sistema. Um de seus usos principais é na construção de tabelas de símbolos, amplamente utilizadas em compiladores e montadores. Tabelas são também amplamente utilizadas em sistemas operacionais para a manutenção de informação de apoio à execução de programas, como as tabelas de processos e as tabelas de arquivos.

3.1 set

Um conjunto é uma estrutura associativa que mantém elementos sem que haja repetição de valores. Desse modo, esta estrutura busca representar o conceito matemático de conjuntos.

Na biblioteca STL, conjunto é implementado pela classe `set`:

```
#include <set>
...
set<int> conj;
```

A implementação de `set` mantém os valores ordenados. Se o tipo de conteúdo do conjunto for uma classe definida pelo usuário, então é preciso informar um segundo parâmetro na declaração do conjunto, que é a função de comparação. Entretanto, se a classe já tiver uma definição para o operador menor que (<), este será utilizado quando o segundo parâmetro é omitido.

Os métodos básicos que podem ser utilizados com os elementos de um conjunto são essencialmente os mesmos que podem ser usados por uma estrutura linear: `size` e `empty` para avaliar a quantidade de elementos na coleção, `begin` e `end` para obter iteradores para varrer o conteúdo da estrutura.

A inserção de elementos no `set` é feita com o método `insert`. Como não há elementos repetidos, a inserção ocorre apenas se o elemento não está presente na coleção ainda. Na

forma básica, o método recebe apenas um argumento com o valor a ser armazenado:

```
conj.insert(99);
conj.insert(11);
conj.insert(55);
conj.insert(11);

set<int>::iterator its;
its = conj.begin();
while (its != conj.end()) {
    cout << *its << " ";
    its++;
}
```

Para este exemplo, o resultado apresentado é

```
11  55  99
```

O retorno desta forma do método `insert`, ignorado neste exemplo, é um par de elementos: um iterador, que indica a posição na qual o elemento está armazenado, e um valor booleano que sinaliza se o valor foi armazenado nesta invocação. Pares de elementos em C++ são definidos pela classe parametrizada `pair`, com elementos `first` e `second`:

```
int x;
pair<set<int>::iterator, bool> res;
...
res = conj.insert(x);
if (! res.second)
    cout << *res.first << " repetido!" << endl;
```

Outra opção para este método recebe dois argumentos, um iterador com a sugestão para a posição da inserção e outro com o valor a ser armazenado. O retorno, neste caso, é apenas o iterador para a posição efetiva na qual o elemento foi (ou estava) armazenado.

Para descobrir se um objeto faz ou não parte da coleção, o método `find` é utilizado. Diferentemente do que foi apresentado para as estruturas lineares, neste caso `find` é uma função membro da classe e não o algoritmo genérico de STL. Recebe apenas um argumento, o valor a ser buscado, e retorna o iterador posicionado no elemento encontrado:

```
set<int>::iterator result;
result = conj.find(25);
```

O método retorna o iterador para o final da coleção (o mesmo que `end` retorna) se o valor não está na coleção. A busca é executada com complexidade temporal $O(\log n)$.

Além desses métodos da classe `set`, a biblioteca STL de C++ oferece funções genéricas que implementam operações para conjuntos, tais como união (`set_union`) e interseção (`set_intersection`). Tais funções recebem como argumentos iteradores e trabalham sobre qualquer tipo de coleção que esteja ordenada, não apenas com objetos da classe `set`. Os dois primeiros iteradores indicam o início e o fim da primeira coleção ordenada, os dois

iteradores seguintes, o início e o fim da segunda coleção, e o quinto argumento é o iterador para o início da coleção na qual o resultado será armazenado.

Além de conjuntos implementados com `set`, STL oferece uma estrutura alternativa, `multiset`, que tem as mesmas operações mas permite a duplicação de elementos com mesmo valor:

```
#include <set>
...
multiset<int> bag;
```

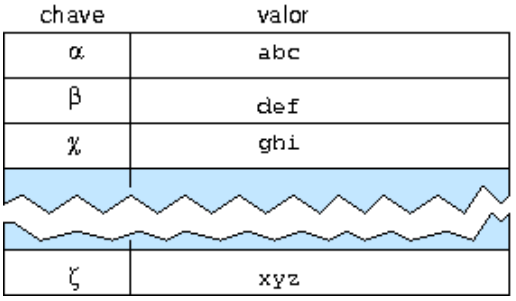
O método `count`, também presente na interface de `set`, retorna a quantidade de vezes que um determinado elemento aparece na coleção. O método `equal_range` retorna um par de iteradores que delimitam o segmento da coleção que contém elementos do mesmo valor.

Uma revisão do padrão da linguagem C++ pretende incorporar à biblioteca STL duas outras classes para manipular conjuntos cujas chaves não sejam ordenadas. Essas classes são `unordered_set`, para coleções sem elementos repetidos, e `unordered_multiset`, para coleções que podem conter elementos repetidos. Embora não sejam parte do padrão, algumas implementações da STL feitas por diferentes distribuidores de compiladores C++ já oferecem classes com funcionalidade equivalente, como as classes `hash_set` e `hash_multiset` disponibilizadas inicialmente na STL do compilador da empresa SGI e depois incorporada também aos compiladores g++ e Visual C++.

3.2 map

Um mapa é uma estrutura associativa na qual os elementos armazenados estão organizados na forma de pares *(chave, valor)*, de tal forma que é possível ter acesso a *valor* (que pode eventualmente ser um objeto com uma estrutura complexa) a partir da *chave*. Conceitualmente, um mapa pode ser visualizado como uma tabela (Figura 3.1). Assim, é possível obter o valor `def` a partir da especificação da chave β, ou o valor `xyz` a partir de ζ.

Figura 3.1 Visão conceitual de uma tabela.



Uma tabela de símbolos, utilizada em compiladores para manter a informação sobre cada variável de um programa, é um exemplo de uma possível aplicação de uma estrutura do tipo mapa. Neste caso, a chave é usualmente o nome de uma variável e o valor é o conjunto de informações sobre a variável, tais como o seu tipo, endereço de memória e local

de definição. Já em uma tabela de arquivos, uma estrutura utilizada pelo sistema operacional para manter a informação sobre cada um dos arquivos associados a um processo, a chave pode ser um identificador inteiro (por exemplo, o terceiro arquivo aberto pelo processo) e o valor o conjunto de informações sobre o arquivo, tais como a posição corrente no arquivo.

Em C++, a biblioteca STL implementa mapas com a classe `map`. A declaração de uma coleção deste tipo especifica dois parâmetros, o primeiro para o tipo da chave e o segundo para o tipo do valor:

```
#include <map>
#include <string>
...
map<string,int> compras;
```

Neste exemplo, a chave é uma string e o valor associado é um inteiro.

Para operar com os elementos de um mapa, o operador `[]` é sobrecarregado. Por exemplo, para associar o valor 6 à chave *vinho*, basta utilizar a expressão:

```
compras["vinho"] = 6;
```

Do mesmo modo, para obter o valor associado a uma chave, o mesmo operador pode ser utilizado:

```
int quantidade = compras["vinho"];
```

Além da sobrecarga desse operador, os métodos básicos presentes nos outros tipos de estrutura (como `empty` e `size`) também estão disponíveis para mapas.

Iteradores para coleções do tipo `map` percorrem elementos que são pares, com o primeiro elemento correspondente à chave e o segundo, ao valor:

```
map<string,int>::iterator itm;
itm = compras.begin();
while (itm != compras.end()) {
    cout << (*itm).first << ": " ;
    cout << (*itm).second << endl;
    itm++;
}
```

A estrutura `map` não admite duplicação de chaves; STL oferece também uma implementação `multimap`, na qual as chaves podem ser repetidas:

```
#include <map>
#include <string>
...
multimap<string,int> mm;
```

Como para `multiset`, `count` retorna a quantidade de elementos com a chave especificada e `equal_range` retorna um par de iteradores para o segmento da coleção que contém os elementos com a mesma chave, especificada como argumento.

Da mesma forma que para `set` e `multiset`, a manutenção de coleções do tipo mapa cujas chaves possam ser mantidas independentemente de ordenação é suportada em algumas implementações de STL por meio das classes não padronizadas `hash_map` e `hash_multimap`. Na revisão da especificação de STL, as correspondentes classes `unordered_map` e `unordered_multimap` deverão ser incorporadas ao padrão.

3.3 Aspectos de implementação

As estratégias de implementação apresentadas nesta seção são amplamente utilizadas em estruturas de dados; a STL as aplica na construção das suas classes que implementam as coleções associativas. Árvores são utilizadas na implementação das classes que implementam estruturas associativas, `set` e `map`, e em suas versões que permitem repetições, `multiset` e `multimap`.

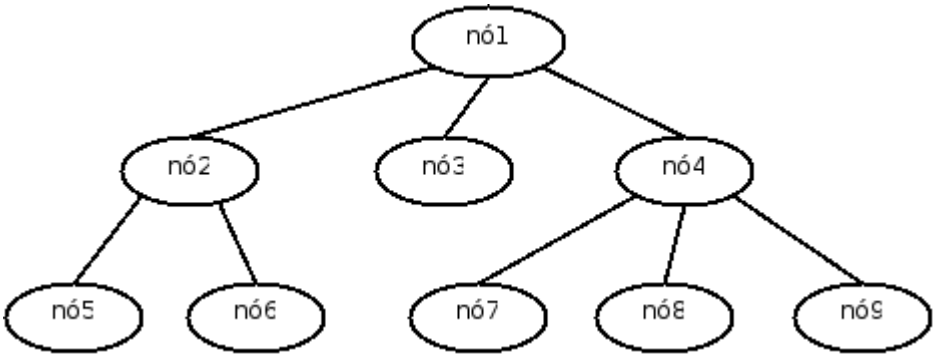
No entanto, para cada uma dessas classes existe uma versão que oferece exatamente o mesmo conjunto de operações mas que utiliza hashing para a implementação: `hash_set`, `hash_map`, `hash_multiset` e `hash_multimap`. Essas classes, presentes em diversas implementações de STL apesar de não serem parte do padrão atual da linguagem, permitem a manutenção de valores ou de chaves independentemente de critérios de ordenação.

3.3.1 Árvores

Uma árvore é uma estrutura que contém um conjunto finito de um ou mais elementos, usualmente denominados **nós**, sendo que um desses nós é especialmente designado como o **nó raiz**. O nó raiz é o ponto de entrada para a estrutura. Associado a cada nó podem estar associados 0 ou mais subconjuntos disjuntos de nós, tal que cada um desses conjuntos é em si uma árvore, denominada sub-árvore.

A representação esquemática de árvores usualmente coloca a raiz no topo, com a árvore crescendo para baixo, como apresentado na Figura 3.2. Neste exemplo, *nó1* é a raiz da árvore e este nó tem três sub-árvores. A primeira sub-árvore tem três nós, sendo o *nó2* a raiz dessa sub-árvore. A segunda sub-árvore tem apenas um nó, *nó3*, que é a raiz da sub-árvore e não tem nenhuma sub-árvore. A terceira sub-árvore tem quatro nós, com *nó4* como raiz.

Figura 3.2 Representação gráfica de uma árvore.

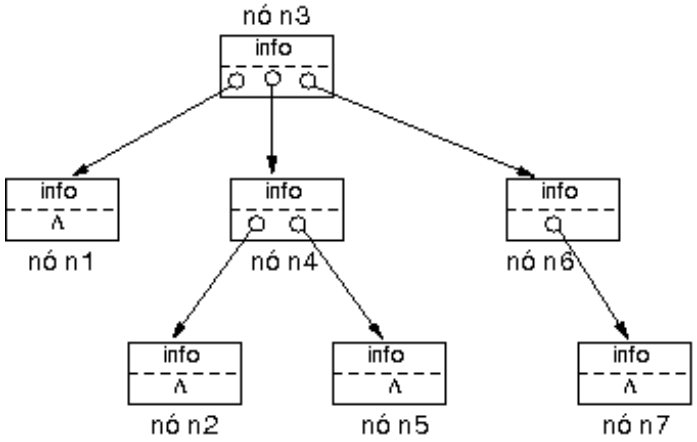


O número de sub-árvores de um nó é o **grau do nó**. No exemplo, o nó *nó1* tem grau 3;

nó2, 2; e *nó3* tem grau 0. O **grau da árvore** é o maior valor de grau de nó entre todos os nós da árvore; no exemplo, a árvore tem grau 3. Um nó que não tem sub-árvores, ou seja, cujo grau é 0, é normalmente denominado **nó folha** da árvore. No exemplo da Figura 3.2, a árvore tem seis folhas: *nó3, nó5, nó6, nó7, nó8, nó9*. Os nós raízes das sub-árvores de um nó são usualmente chamados de **nós filhos** desse nó, que recebe também o nome de **nó pai** daqueles nós. No exemplo, *nó5* e *nó6* são filhos do *nó2*; o *nó4* é pai dos nós *nó7, nó8, nó9*. A estrutura de uma árvore é hierárquica, ou seja, cada nó tem apenas um nó pai.

Sob o ponto de vista de implementação, a estrutura de árvore pode ser visualizada como uma extensão de uma lista ligada na qual um nó pode ter mais de um sucessor (Figura 3.3).

Figura 3.3 Representação esquemática de uma implementação de árvore.



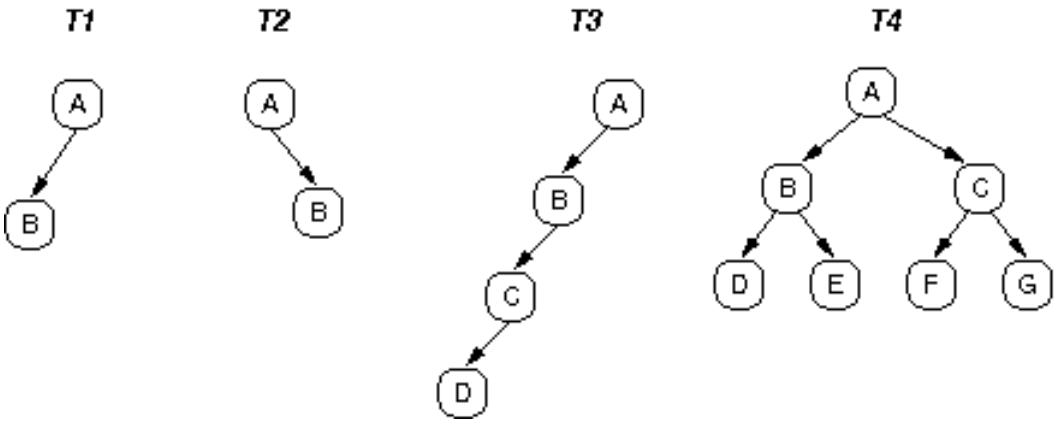
A Figura 3.3 ilustra um exemplo de uma árvore T, que tem o nó raiz $n3$ e as sub-árvores T_1, T_2 e T_3. A sub-árvore T_1 tem o nó raiz $n1$ e não contém sub-árvores; sub-árvore T_2 tem o nó raiz $n4$ e sub-árvores T_4 e T_5; e a sub-árvore T_3 tem o nó raiz $n6$ e sub-árvore T_6. No próximo nível, as sub-árvores T_4, T_5 e T_6 têm respectivamente os nós raízes $n2$, $n5$ e $n7$ e não têm sub-árvores.

Um tipo especial de árvore é a **árvore binária**. Uma árvore binária tem um nó raiz e no máximo duas sub-árvores, uma sub-árvore esquerda e uma sub-árvore direita. Em decorrência dessa definição, o grau de uma árvore binária é limitado a dois. A Figura 3.4 ilustra alguns exemplos de árvores binárias.

Observe na figura que $T1$ e $T2$ são árvores binárias distintas pois, ao contrário da definição genérica de árvores, há diferença de tratamento para a árvore binária entre a sub-árvore direita e a sub-árvore esquerda. Outra diferença de definição para árvores binárias é que elas podem eventualmente ser vazias, algo que a definição de árvore genérica não permite. $T3$ é uma árvore binária degradada (equivalente a uma lista linear), enquanto $T4$ é uma árvore binária completa e **balanceada**, ou seja, na qual as sub-árvores tem igual tamanho. A inserção de mais um nó a $T4$ forçosamente implicaria na adição de mais um nível à árvore, o que a transformaria numa árvore binária incompleta — ou seja, haveria a possibilidade de incluir mais nós sem alterar a altura da árvore. No entanto, a árvore ainda seria balanceada, pois uma diferença máxima de um nível entre as alturas das sub-árvores é aceitável.

Uma das principais aplicações de árvores é a manutenção de estruturas nas quais a ordem é importante. Para manter a ordem dos nós de uma árvore binária, três estratégias podem ser

Figura 3.4 Árvores binárias



utilizadas:

Pré-ordem é a estratégia de varredura de uma árvore binária na qual o primeiro nó é o nó raiz, seguido pela sub-árvore esquerda em pré-ordem e finalmente pela sub-árvore direita em pré-ordem;

Intra-ordem é a estratégia de varredura de árvore binária na qual lê-se primeiro a sub-árvore esquerda em intra-ordem, seguido pelo nó raiz e finalmente pela sub-árvore direita em intra-ordem;

Pós-ordem é a estratégia de varredura na qual primeiro lê-se os nós da sub-árvore esquerda em pós-ordem, depois os nós da sub-árvore direita em pós-ordem e finalmente o nó raiz.

Aplicando essas estratégias à árvore $T4$ (Figura 3.4), com pré-ordem a seqüência de nós da árvore seria A, B, D, E, C, F, G; com intra-ordem, D, B, E, A, F, C, G; e com a pós-ordem, D, E, B, F, G, C, A.

A estratégia intra-ordem é usualmente utilizada para a manutenção de coleções ordenadas por valores. Todos os valores na sub-árvore esquerda de um nó precedem o valor armazenado nesta raiz; similarmente, todos os valores na sub-árvore direita são maiores que esse valor. Deste modo, a busca por um valor na árvore inicia-se pelo nó raiz. Se o valor armazenado for igual ao buscado, encerra-se a busca. Caso o valor buscado seja menor que a raiz, a busca é repetida na sub-árvore esquerda; caso seja maior, na sub-árvore direita. As outras estratégias de varredura são utilizadas no processamento da representação interna de comandos em compiladores, por meio das árvores sintáticas.

Para inserir um novo valor na árvore binária, estratégia similar é utilizada. Se a árvore estiver vazia, o elemento é inserido na raiz. Caso a raiz exista, o elemento é inserido na sub-árvore esquerda, se seu valor for menor ou igual àquela armazenado na raiz, ou na sub-árvore direita, caso contrário. Ao aplicar recursivamente essa estratégia de armazenamento, o local correto para o elemento é encontrado.

A busca por um elemento em uma árvore tem tempo proporcional à altura da árvore, ou seja, tem complexidade temporal $O(\log n)$. Para que essas buscas sejam eficientes, é

importante que a árvore esteja balanceada — caso contrário, uma busca na árvore poderia degradar para $O(n)$ no pior caso. Na prática, implementações de árvores como em STL utilizam estruturas de árvores binárias auto-balanceadas, que realizam transformações na árvore de forma a garantir que a estrutura esteja sempre balanceada. Implementações usuais de árvores auto-balanceadas incluem árvores AVL e árvores vermelho-preto.

3.3.2 Tabelas hash

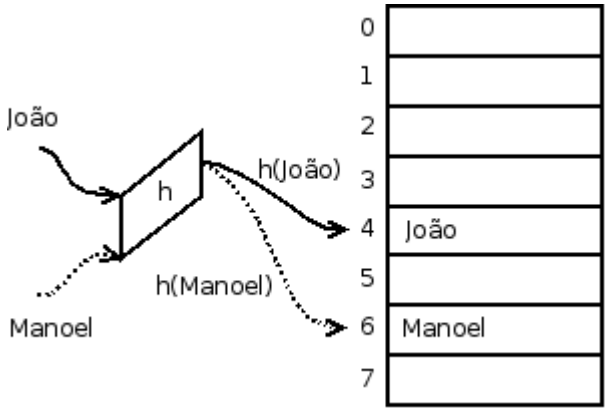
Em coleções associativas mantidas com estrutura de árvore, a busca por um elemento ou uma chave ocorre sempre através de comparações a partir do elemento raiz. Uma estrutura alternativa para coleções associativas é manter os elementos em um mapa ou tabela, no qual a posição de cada elemento possa ser determinada diretamente a partir de seu valor ou do valor de sua chave. Uma estrutura desse tipo é denominada **tabela *hash***. A função que transforma o valor do elemento ou da chave para um inteiro, que é a posição do elemento na tabela hash, é chamada **função de *hashing***.

Tabelas hash são estruturas de acesso direto. Numa tabela hash ideal, o tempo de acesso a qualquer elemento independente da quantidade de elementos — é constante, pois depende apenas do tempo necessário para o cômputo da função de hashing. Portanto, a complexidade temporal de acesso a tabelas hash é $O(1)$.

A Figura 3.5 ilustra o conceito de operação da tabela hash. Neste caso, a tabela permite o armazenamento de até oito entradas, nas posições de 0 a 7. Os elementos são strings e a função de hashing adotada, a título de ilustração, é simplesmente o tamanho da string:

```
int h(string s) {
    return s.size();
}
```

Figura 3.5 Tabela hash



É evidente que essa função de hashing impõe um limite ao domínio de elementos que podem ser armazenados na coleção — por exemplo, *h(Gumercindo)*=10, que indicaria uma posição inexistente na tabela. Há também uma posição, a primeira, que nunca seria ocupada por nenhum elemento.

Funções de hashing devem ter como contra-domínio posições válidas na tabela. Um dos métodos utilizados para alcançar esse objetivo é o chamado **método da divisão**. Nesse tipo de função, o elemento tem seu valor inicialmente transformado para um valor inteiro positivo qualquer, sem limitação. Esse valor é então dividido por M, o número de posições na tabela. O resto dessa divisão é um valor entre 0 e $M - 1$, que é uma posição válida na tabela. No exemplo acima, a função de hashing calculada usando esse método seria:

```
int h(string s) {
    return s.size() % 8;
}
```

Com essa função, o elemento *Gumercindo* seria mapeado para a posição 2 da tabela.

Como é possível perceber desse pequeno exemplo, a tabela hash combina aspectos da alocação contígua com a alocação não-contígua. Por um lado, o espaço total da tabela hash é alocado previamente e ocupa uma área contígua. Por outro lado, a ocupação da tabela não é contígua, pois a posição de cada nó depende da aplicação da função hash — assim, pode haver espaços desocupados entre dois elementos da coleção.

Idealmente, cada chave processada por uma função de hashing gera uma posição diferente na tabela. No entanto, na prática existem **sinônimos** — chaves distintas que resultam em um mesmo valor da função de hashing. Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma **colisão**. Obviamente, colisões degradam o desempenho de busca e de armazenamento em uma tabela hash. Por esse motivo, a escolha de boas funções de hashing é essencial — certamente, o tamanho da string não é uma boa escolha.

Uma boa função hash deve apresentar duas propriedades básicas: seu cálculo deve ser rápido e deve gerar poucas colisões. Além disso, é desejável que ela leve a uma ocupação uniforme da tabela para conjuntos de chaves quaisquer. No caso do método da divisão, uma recomendação é que o tamanho da tabela seja um número primo.

Outro método aplicado para restringir um valor numérico arbitrário para uma posição em uma tabela cuja dimensão é uma potência de 2 é o **método do meio do quadrado**. Nesse tipo de função, o valor inteiro obtido a partir do valor do elemento ou chave é elevado ao quadrado. Os r bits no meio da representação binária do valor resultante são utilizados como o endereço em uma tabela com 2^r posições.

A título de ilustração, considere ainda o exemplo acima, no qual o valor inteiro inicial associado ao elemento é simplesmente a dimensão da string. Considere adicionalmente que o domínio de entrada não tem strings maiores que 11 caracteres, de modo que 7 bits são suficientes para a representação binária do quadrado desse valor. O elemento *João* seria coincidentemente mapeado para a posição 4, pois $4^2 = 16$, cujos três bits do meio da representação binária (`0010000`) representam o valor 4. Já os elementos *Manoel* e *Gumercindo* seriam ambos mapeados para a posição 1, pois $6^2 = 36$ tem representação binária `0100100` (três bits do meio: `001`) e $10^2 = 100$ tem representação binária `1100100`, com os mesmos três bits `001` no meio.

Uma técnica usual para produzir valores inteiros a partir de elementos de qualquer tipo e que sejam menos previsíveis do que o exemplo acima, do comprimento da string, é a técnica de *folding*. Nessa técnica, a representação do elemento é dividida em segmentos de igual tamanho (exceto por um segmento, eventualmente) e cada um deles é interpretado como um

valor inteiro. A soma ou outra operação de combinação de todos os valores assim obtidos será a entrada para a função de hashing.

Considere a aplicação da técnica de folding para gerar um valor inteiro associado à string *Manoel* do exemplo acima. Uma estratégia usual para tratamento de strings nessa técnica é utilizar o valor ASCII de cada caráter como unidade de segmentação. Para essa string:

Caráter	Hexadecimal	Decimal
M	4D	77
a	61	97
n	6E	110
o	6F	111
e	65	101
l	6C	108

Se a soma é utilizada como função para combinar os valores dos segmentos, então o valor inteiro que seria gerado como entrada para o método da divisão ou do meio do quadrado seria $77 + 97 + 110 + 111 + 101 + 108 = 604$. Outra função usual para a combinação é a função binária ou exclusivo (XOR), que pode ser aplicado a segmentos de um ou mais bytes. Por exemplo, aplicada a segmentos de um byte para a mesma string, o valor resultante seria 36.

Como deve ter ficado evidente a partir dos exemplos apresentados, o processamento de tabelas hash deve prever algum mecanismo para o tratamento de colisões. As formas mais usuais de tratamento de colisão são por endereçamento aberto ou por encadeamento.

Na técnica de tratamento de colisão por **endereçamento aberto**, a estratégia é utilizar o próprio espaço da tabela que ainda não foi ocupado para armazenar a chave que gerou a colisão. Quando a função hash gera para uma chave uma posição que já está ocupada, o procedimento de armazenamento verifica se a posição seguinte também está ocupada; se estiver ocupada, verifica a posição seguinte e assim por diante, até encontrar uma posição livre. A entrada é então armazenada nessa posição.

Considere o exemplo da aplicação do método do meio do quadrado com a dimensão da string, no qual os elementos *Manoel* e *Gumercindo* foram mapeados para a mesma posição (1). Se *Manoel* foi inserido na tabela antes de qualquer outro elemento mapeado para a mesma posição, então irá ocupar a posição 1. Quando o elemento *Gumercindo* for inserido na tabela, vai encontrar a posição 1 ocupada pelo elemento *Manoel*. Então irá ocupar a posição 2, se estiver livre.

Nesse tipo de tratamento, considera-se a tabela como uma estrutura circular, onde a primeira posição sucede a última posição. Se a busca por uma posição livre retorna à posição inicialmente determinada pela função de hashing, então a capacidade da tabela está esgotada e uma mensagem de erro é gerada.

No momento da busca, essa varredura da tabela pode ser novamente necessária. Se a chave buscada não está na posição indicada pela função de hashing e aquela posição está ocupada, a chave pode eventualmente estar em outra posição na tabela. Assim, é necessário verificar se a chave não está na posição seguinte. Se, por sua vez, essa posição estiver ocupada com outra chave, a busca continua na posição seguinte e assim por diante, até que se encontre a chave buscada ou uma posição livre.

Na técnica de tratamento de colisão por **encadeamento**, para cada posição onde ocorre colisão cria-se uma área de armazenamento auxiliar, externa à área inicial da tabela hash.

Normalmente essa área é organizada como uma lista ligada que contém todas os elementos que foram mapeados para a mesma posição da tabela. No momento da busca, se a posição correspondente ao elemento na tabela estiver ocupada por outro elemento, é preciso percorrer apenas a lista ligada correspondente àquela posição até encontrar o elemento buscado ou alcançar o final da lista.

Hashing é uma técnica simples e amplamente utilizada na programação de sistemas. Quando a tabela hash tem tamanho adequado ao número de chaves que irá armazenar e a função de hashing utilizada é de boa qualidade, a estratégia de manipulação por hashing é bastante eficiente.

A grande vantagem na utilização da tabela hash está no desempenho — enquanto a busca linear tem complexidade temporal $O(N)$ e a busca binária ou em árvores tem complexidade $O(\log N)$, o tempo de busca na tabela hash é praticamente independente do número de chaves armazenadas na tabela, ou seja, tem complexidade temporal $O(1)$. Outro aspecto importante é que as estratégias mais eficientes baseadas em comparação, com coleções ordenadas ou árvores binárias, demandam que o conjunto de valores assumido pelos elementos ou chaves seja ordenável, algo que não é necessário em hashing. No entanto, o desempenho de hashing pode degradar sensivelmente em situações nas quais um grande número de colisões pode ocorrer — em situações extremas, chegar até $O(n)$.

Capítulo 4

Representação interna de valores

Os valores simbólicos, caracteres ou constantes numéricas, apresentados em expressões nos arquivos dos programas, precisam ser convertidos para representações binárias adequadas ao processador que irá usar esses argumentos em suas operações. Aqui são apresentadas as representações padrões utilizadas para esses símbolos.

4.1 Representação de caracteres

A seguinte tabela apresenta a representação dos 128 caracteres iniciais associados à representação ASCII e ISO8859-1.

Cód	0x-0	0x-1	0x-2	0x-3	0x-4	0x-5	0x-6	0x-7	0x-8	0x-9	0x-A	0x-B	0x-C	0x-D	0x-E	0x-F
0x0-	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0x1-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0x2-	[esp]	!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
0x3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6-	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7-	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

4.2 Representação numérica binária

Inteiros sem sinal têm uma representação computacional (em números binários) equivalente à representação usual para números decimais, ou seja, através da atribuição de pesos associados à posição de cada bit. Grande parte dos computadores atuais utilizam 32 bits para representar números inteiros, o que permite representar $4.924.967.296$ valores distintos. (A geração mais recente de computadores suporta também inteiros com 64 bits.) Uma seqüência binária

$$s_{n-1}s_{n-2}s_{n-3}\ldots s_2s_1s_0$$

está associada ao valor inteiro

$$\sum_{i=0}^{n-1} s_i \cdot 2^i$$

onde $s_i \in \{0, 1\}$. O bit s_{n-1} é chamado *bit mais significativo* (MSB), enquanto que s_0 é o *bit menos significativo* (LSB).

A representação de inteiros com sinal pode usar outros formatos. A forma mais básica é a representação em sinal e magnitude, onde o bit mais significativo denota o sinal associado ao restante da seqüência ($s_{n-1} = 1$ indicaria que o número é negativo). Este formato tem a desvantagem de ter duas representações diferentes para o valor zero, além de ter circuitos complicados para suportar operações básicas, diferenciando adição de subtração, por exemplo.

Outra formato suportado para representar inteiros com sinal é a representação em complemento de um. A representação para um número negativo neste formato pode ser obtida facilmente a partir da representação do número positivo correspondente simplesmente complementando cada bit da seqüência, ou seja, trocando 0's por 1's e 1's por 0's. Apesar de simplificar circuitos para operações básicas, este formato ainda mantém duas representações distintas para o valor zero.

O formato mais aceito para inteiros com sinal é a representação em complemento de dois. Para obter a representação de um número negativo neste formato, computa-se inicialmente a representação em complemento de um e adiciona-se 1 ao bit menos significativo. Neste caso, o valor inteiro associado à seqüência $s_{n-1} \ldots s_0$ é

$$\sum_{i=0}^{n-2} s_i \cdot 2^i - s_{n-1} \cdot 2^n.$$

Este formato mantém a simplicidade dos circuitos aritméticos e tem apenas uma representação para o valor zero. Uma característica que lhe é peculiar é o fato de que a faixa de valores representáveis não é simétrica em torno de 0, havendo um valor negativo a mais que a quantidade de valores positivos distintos. Por exemplo, seqüências de cinco bits podem representar valores entre -16 (`10000`) e +15 (`01111`).

No formato de representação para números reais, associado ao conceito de notação científica, cada valor (pertencente ao domínio dos reais) é representado por um sinal, uma mantissa e um expoente. Entre as inúmeras combinações possíveis de formatos de representação que seguem esta filosofia básica, o padrão IEEE-754 tem sido o mais aceito e usualmente suportado em hardware (através das unidades de ponto flutuante em co-processadores ou incorporados a CPUs). Este formato suporta representações de números reais em *precisão simples* (32 bits, dos quais 8 para a representação do expoente e 23 para a representação da mantissa), em *precisão dupla* (64 bits, sendo 11 para o expoente e 53 para a mantissa) e em precisão estendida (80 bits). Há também representações especiais para os valores $-\infty$, $+\infty$ e `NaN` (*Not a Number*, associado ao resultado de operações sem significado matemático, tal como a divisão de zero por zero).

Parece evidente que a representação binária, apesar de ideal para o processador, é de difícil manipulação por humanos. Por este motivo, adota-se usualmente a representação hexadecimal para denotar seqüências binárias.

A vantagem da representação hexadecimal sobre a decimal, que usamos no dia a dia, é a fácil associação com seqüências binárias. A tradução é direta: cada seqüência de quatro bits corresponde a um símbolo hexadecimal. A tabela a seguir define este mapeamento:

binário	**hexa**	**binário**	**hexa**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

A representação octal também permite um mapeamento similar, de três bits para um dígito entre 0 e 7. Entretanto, a representação hexadecimal também apresenta a vantagem de alinhamento com um byte (8 bits, dois dígitos hexadecimais) e palavras de 16 bits (quatro dígitos).

Capítulo 5

A linguagem de programação C++

O fato de uma linguagem ter sido desenvolvida com uma aplicação em mente não significa que ela não seja adequada para outras aplicações. A linguagem C, juntamente com sua "sucessora" C++, é utilizada para um universo muito amplo de aplicações. Um dos atrativos dessas linguagens é sua flexibilidade: o programador tem à sua disposição comandos que permitem desenvolver programas com características com alto nível de abstração e ao mesmo tempo trabalhar em um nível muito próximo da arquitetura da máquina, de forma a explorar os recursos disponíveis de forma mais eficiente. Por este motivo, o número de aplicações desenvolvidas em C e C++ é grande e continua a crescer.

5.1 Fundamentos de C++

Nesta seção são apresentados alguns princípios que permitirão dar início às atividades de programação em C++. Outros aspectos da linguagem serão apresentados conforme a necessidade de ilustrar os conceitos apresentados ao longo do texto.

5.1.1 Organização de programas

Um programa em C++ deve ser organizado como um conjunto de objetos que interagem para realizar alguma tarefa. Esta estratégia de organização de programas é usualmente conhecida como orientação a objetos e é um paradigma de programação suportado por C++.

Objetos nada mais são do que variáveis, cuja estrutura interna pode ser complexa e que têm operações específicas para sua manipulação. A definição da estrutura e do comportamento de objetos dá-se através da especificação de classes. Algumas classes e objetos estão especificados e definidos pela própria linguagem (fazem parte de sua biblioteca padrão), outros podem ser especificados e definidos pelo programador.

Além de objetos, o programador C++ pode definir funções. Cada função tem um nome, uma lista de argumentos (que pode ser vazia) e um tipo de retorno (que também pode ser vazio, indicado pela palavra-chave `void`). Cada argumento e o valor de retorno pode ser de um dos tipos primitivos da linguagem ou do tipo de um objeto. A ordem e os tipos dos argumentos, juntamente com o nome da função, definem sua **assinatura**. Sobrecarga *(overloading)* é o mecanismo da linguagem que permite a existência de mais de uma função

com o mesmo nome, desde que o restante de suas assinaturas (listas de argumentos) sejam diferentes.

Uma dessas funções tem um papel especial e deve ser definida pelo programador. A função `main` estabelece o ponto de início de execução de qualquer aplicação desenvolvida em C ou em C++. Ela estabelece o mecanismo básico de comunicação entre a aplicação e o ambiente (sistema operacional) no qual esta está executando. A assinatura da função `main` determina como se dá esta comunicação. Do ambiente de execução para a aplicação, a forma básica é através da passagem de argumentos no momento da invocação do programa; são os argumentos da linha de comando. Desta forma, `main` recebe dois parâmetros, como indicado abaixo na sua assinatura:

```
int main(int argc, char *argv[]) {
    ...
}
```

O primeiro parâmetro, que tipicamente recebe o nome `argc` *(argument count)*, indica o número de palavras (*strings* separadas por espaços) presentes na linha de comando, incluindo o próprio nome do programa. Por exemplo, uma chamada a um programa de nome `eco` com dois argumentos, como

```
eco um dois
```

faria com que o valor de `argc` passado para a função `main` fosse igual a três.

O segundo parâmetro, cujo nome típico é `argv` *(argument value)*, é um arranjo de ponteiros para caracteres, onde cada elemento do arranjo representa uma das palavras da linha de comando. Essa é a forma que a linguagem C utiliza para representar strings de caracteres. Assim, no exemplo acima a função `main` receberia as seguintes *strings* nesta variável:

- Em `argv[0]`, a seqüência de caracteres `"eco"`;
- em `argv[1]`, a seqüência de caracteres `"um"`; e
- em `argv[2]`, a seqüência de caracteres `"dois"`.

Observe que `argv[0]` sempre armazenará o nome do programa sendo executado, enquanto que `argv[i]` armazenará o i-ésimo argumento passado para o programa, para i variando de `1` até `argc-1`.

Quando o programa não faz uso dos argumentos da linha de comando, é usual omitir da definição da função a declaração dos parâmetros:

```
int main() { ... }
```

O valor de retorno da função `main` é repassado para o ambiente de execução quando do final da execução da aplicação. Para tanto, utiliza-se o comando `return` da linguagem, que pode ocorrer sem argumentos quando a função não tiver retorno (for do tipo `void`) ou deve ter um argumento que é uma expressão cujo resultado é de um tipo compatível com o tipo de retorno da função.

Por convenção, um valor de retorno para a função `main` diferente de 0 serve para indicar ao sistema operacional (ou a um outro processo que tenha ativado este programa) que alguma

condição de erro ocorreu que impediu o programa de completar com sucesso sua execução; o valor de retorno 0 indica a terminação sem problemas. Alternativamente, a função `main` pode encerrar com a invocação da função `exit`, que recebe esse valor de retorno para o ambiente como argumento.

Strings em C

Um dos tipos de agregados que mais ocorre na programação de sistemas é a seqüência de caracteres, ou *string*. Apesar de C++ ter uma classe `string` definida em sua biblioteca padrão, a linguagem C não suporta esse tipo básico; ao invés, utiliza uma convenção para tratamento de arranjos de caracteres que permite o uso de diversas funções de manipulação de *strings* na linguagem. Como algumas funções em C++ preservam compatibilidade com esse tipo de representação, como a própria função `main` acima, ele é apresentado a seguir.

Por convenção, C considera como uma *string* uma seqüência de caracteres armazenada sob a forma de um arranjo de tipo `char` cujo último elemento é o caráter NUL, tipicamente representado na forma de caráter, `'\0'`, ou simplesmente pelo seu valor, 0. Por exemplo, uma *string* C poderia ser declarada e inicializada como em

```
char exemplo[4] = {'a','b','c','\0'};
```

Observe que o espaço para o caráter `'\0'` deve ser previsto quando dimensionando o tamanho do arranjo de caracteres que será manipulado como *string*. No exemplo, o arranjo de quatro caracteres pode receber apenas três letras, já que o último caráter está reservado para o NUL.

C e C++ suportam uma forma de representação de uma *string* constante através do uso de aspas:

```
char exemplo[4] = "abc";
```

Este exemplo é equivalente ao anterior — a *string* `"abc"` contém quatro caracteres, sendo que o caráter `'\0'` é automaticamente anexado à *string* pelo compilador.

Funções que manipulam *strings* trabalham usualmente com a referência para o início da seqüência de caracteres, ou seja, com um ponteiro para a *string*. A manipulação de ponteiros é fonte usual de confusão em qualquer linguagem.

Considere, por exemplo, um trecho de código com duas variáveis do tipo ponteiro para caracteres `s1` e `s2`. Supondo que as duas variáveis tivessem sido declaradas e seus conteúdos devidamente inicializado com *strings*, não seria possível copiar o conteúdo de `s2` para `s1` simplesmente por atribuição, como em

```
s1 = s2;
```

Da mesma forma, não seria possível comparar seus conteúdos diretamente, como em

```
if (s1 != s2)
    ...
```

Nessas duas situações, o que estaria envolvido na expressão seriam os endereços armazenados por essas variáveis. No exemplo da atribuição, o endereço armazenado em `s2` estaria

sendo atribuído à variável `s1`. Na comparação, os dois endereços em `s1` e `s2` estariam sendo comparados e não os respectivos conteúdos.

Em C, particularmente para *strings* de caracteres, um conjunto de rotinas foi definido como parte da biblioteca de funcionalidades básicas da linguagem, tais como `strcpy` (cópia) e `strcmp` (comparação). Para utilizar essas funções em um programa C++, o arquivo de cabeçalho `cstring` deve ser incluído no programa fonte.

5.1.2 Expressões

O corpo de uma função, como `main`, é definido através dos comandos que serão por ela executados. Esses comandos são indicados como uma seqüência de expressões válidas da linguagem.

Antes de mais nada, é interessante que se apresente a forma de se expressar comentários em um programa C++. A forma preferencial de incluir comentários no código é através da seqüência `//`, que faz com que o restante da linha seja interpretado como comentário. Comentários no padrão da linguagem C, indicados pelos terminadores `/*` (início de comentário) e `*/` (fim de comentário), também são aceitos; quaisquer caracteres entre estes dois pares de símbolos são ignorados pelo compilador. Comentários em C não podem ser aninhados, mas podem se estender por diversas linhas e podem começar em qualquer coluna. Por exemplo,

```
1    /*  Exemplo de
2     *  comentario
3     */
4    void func( ) {
5       // esta funcao nao faz coisa alguma
6    }
```

A posição das chaves não é relevante.

As expressões na linguagem C++ são sempre terminadas pelo símbolo `;` (ponto e vírgula). Uma *expressão nula* é constituída simplesmente pelo símbolo terminador. Assim, a função do exemplo acima é equivalente a

```
1    void func( )
2    {
3     ;
4    }
```

O comando de atribuição em C++ é indicado pelo símbolo =, como em

```
1    void func() {
2       int a, b, c;
3       a = 10;       // a recebe valor 10
4       b = c = a;    // b e c recebem o valor de a (10)
5    }
```

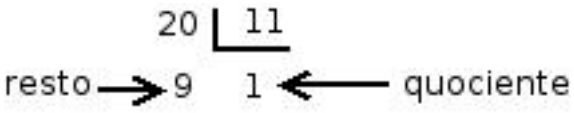
Observe neste exemplo que a atribuição pode ser encadeada — na última linha da função acima, `c` recebe inicialmente o valor da variável `a`, e então o valor de `c` será atribuído à variável `b`.

Expressões aritméticas em C++ podem envolver os operadores binários (isto é, operadores que tomam dois argumentos) de *soma* (+), *subtração* (−), *multiplicação* (⋆), *divisão* (/). Valores negativos são indicados pelo operador unário −. Adicionalmente, para operações envolvendo valores inteiros são definidos os operadores de resto da divisão inteira ou *módulo* (%), incremento (++) e decremento (−−). Por exemplo,

```
1       void func() {
2         int a=10, b, c, d;
3
4         b = 2*a;        // b recebe 20
5         a++;            // a recebe 11
6         c = b/a;        // c recebe 1
7         d = b%a;        // d recebe 9
8       }
```

A Figura 5.1 ilustra o resultado associado às duas últimas linhas.

Figura 5.1 Resultados da divisão inteira.



Cada um dos operadores de incremento e decremento tem duas formas de uso, dependendo se eles ocorrem antes do nome da variável (pré-incremento ou pré-decremento) ou depois do nome da variável (pós-incremento ou pós-decremento). No caso do exemplo acima, onde o operador de incremento ocorre de forma isolada em uma expressão (sozinho na linha), as duas formas possíveis são equivalentes. A diferença entre eles ocorre quando estes operadores são combinados com outras operações. No exemplo acima, as linhas de atribuição à `b` e incremento de `a` poderiam ser combinados em uma única expressão,

```
b = 2*a++;
```

Neste caso, o valor de `a` é inicialmente utilizado na expressão (e portanto `b` recebe 2*10) e apenas depois será incrementado (a forma pós-incremento). Observe como essa expressão é diferente de

```
b = 2*(++a);
```

pois neste caso o valor de `a` seria inicialmente incrementado (forma pré-incremento) e apenas depois utilizado no restante da expressão.

Na prática, os parênteses na expressão acima poderiam ser omitidos uma vez que a precedência do operador de incremento é maior que da multiplicação — ou seja, o incremento será avaliado primeiro. O Apêndice 5.3 apresenta a ordem de avaliação para todos os operadores da linguagem.

C++ tem também uma forma compacta de representar expressões na forma

```
var = var op (expr);
```

onde uma mesma variável `var` aparece nos dois lados de um comando de atribuição. A forma compacta é

```
var op= expr;
```

Por exemplo, as expressões

```
a += b;
c *= 2;
```

são respectivamente equivalentes a

```
a = a + b;
c = c * 2;
```

5.1.3 Expressões condicionais

Um tipo especial de expressão é a **expressão condicional**, cujo resultado é um valor que será interpretado como falso ou verdadeiro. Em C++, uma expressão desse tipo tem como resultado um valor do tipo `bool`. Como a linguagem C não suporta diretamente um tipo de dado booleano, ela trabalha com representações inteiras para denotar estes valores — o resultado de uma expressão condicional é um valor inteiro que será interpretado como falso quando o valor resultante da expressão é igual a 0, e como verdadeiro quando o valor resultante é diferente de 0.

Uma expressão condicional usualmente envolve a comparação de valores através dos operadores relacionais. Os operadores relacionais em C++ são:

>	maior que	>=	maior que ou igual a
<	menor que	<=	menor que ou igual a
==	igual a	!=	diferente de

Aqueles que conhecem outras linguagens de programação, como Pascal, devem observar que o operador de igualdade é ==, e não =. Esta é uma causa comum de erros para programadores que estão acostumados a utilizar = como um operador relacional.

Expressões condicionais elementares (comparando duas variáveis ou uma variável e uma constante) podem ser combinadas para formar expressões complexas através do uso de operadores booleanos. Estes operadores são

&&	AND
||	OR
!	NOT

O operador `&&` *(and)* resulta verdadeiro quando as duas expressões envolvidas são verdadeiras. O operador `||` *(or)* resulta verdadeiro quando pelo menos uma das duas expressões envolvidas é verdadeira. Além destes dois conectores binários, há também o operador unário de negação, `!`, que resulta falso quando a expressão envolvida é verdadeira ou resulta verdadeiro quando a expressão envolvida é falsa.

Expressões lógicas complexas, envolvendo diversos conectores, são avaliadas da esquerda para a direita. Além disto, `&&` tem precedência maior que `||` e ambos têm precedência menor que os operadores lógicos relacionais e de igualdade. Entretanto, recomenda-se

sempre a utilização de parênteses para tornar claro qual é a ordem desejada de avaliação das expressões. A exceção a esta regra ocorre quando um número excessivo de parênteses pode dificultar ainda mais a compreensão da expressão; em tais casos, o uso das regras de precedência da linguagem pode facilitar o entedimento da expressão.

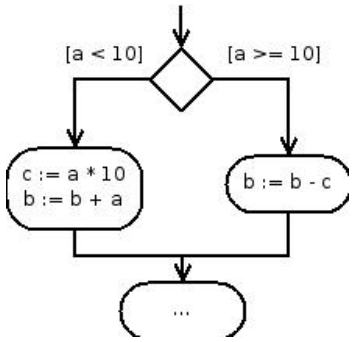
5.1.4 Controle do fluxo de execução

O corpo de funções e métodos em C++ é expresso usando construções da programação estruturada, a qual permite agrupar comandos na forma de seqüências intercaladas com comandos de seleção e repetição.

A seqüência de comandos em uma função C++ é denotada simplesmente pela ordem da ocorrência das expressões no código, como já ilustrado em exemplos anteriores.

A construção de seleção, um comando do tipo `IF-THEN-ELSE`, é expressa em C++ com as palavras-chaves `if...else`. A Figura 5.2 apresenta um exemplo com a representação gráfica deste comando, usando a notação de diagrama de atividades de UML (a *Unified Modeling Language* definida pelo consórcio OMG, *Object Management Group*), e a correspondente codificação em C++. Após a palavra-chave `if` deve haver uma expressão condicional entre parênteses. Se a expressão for avaliada como verdadeira, então a expressão sob `if` será realizada; se for falsa, a expressão sob `else` será executada.

Figura 5.2 Seleção com if...else



(a) Representação gráfica

```
if (a < 10 ) {
   c = a * 10;
   b = b + a;
} else
   b = b - c;
```

(b) Expressão em C++

Este exemplo também introduz o conceito de **expressão composta**, ou seja, a primeira das expressões no *if-else* deste exemplo é na verdade um bloco contendo diversas expressões. Neste caso, o bloco de comandos que deve ser executado nessa condição deve ser delimitado por chaves { e }. Algumas observações adicionais relevantes com relação a este comando são:

1. Em C++, há diferenças entre letras minúsculas e maiúsculas. Como todos os comandos em C++, as palavras chaves deste comando estão em letras minúsculas. Assim, as formas *IF* (ou *If* ou *iF*) não são formas válidas em C++ para denotar o comando `if`.

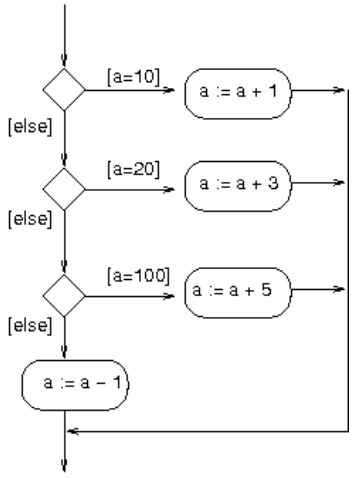
2. Ao contrário do que ocorre em Pascal ou FORTRAN, a palavra *then* não faz parte da sintaxe deste comando em C++.

3. A cláusula *else* pode ser omitida quando a expressão a executar na condição falsa for nula.

4. No caso de haver mais de um `if` que possa ser associado a uma cláusula `else`, esta será associada ao comando `if` precedente mais próximo.

Outra construção estruturada de seleção suportada por C++ é o comando *switch case* (Figura 5.3). Neste caso, após a palavra-chave `switch` deve haver uma *variável* entre parênteses, que deve ser do tipo inteiro ou caráter. Após a especificação dessa variável, segue-se uma lista de valores possíveis para a variável que devem ser considerados na seleção. Cada elemento da lista (ou cada caso) é iniciado com a palavra-chave `case` seguida por um valor ou uma expressão inteira e o caráter `':'`.

Figura 5.3 Seleção em C++ usando a forma switch... case. Observe que o conjunto de ações associado a cada caso encerra-se com a palavra-chave *break*.



(a) Representação gráfica

```
switch (a) {
case 10:
   ++a;
   break;
case 20:
   a += 3;
   break;
case 100:
   a += 5;
   break;
default:
   a += 10;
}
```

(b) Expressão em C++

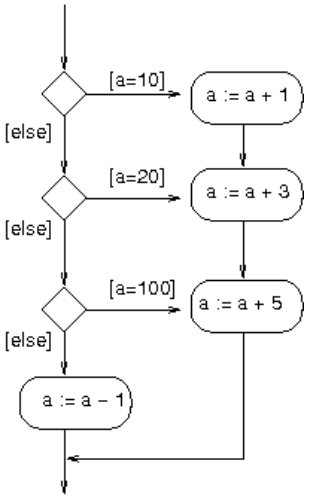
Neste exemplo, a variável `a` pode ser do tipo `int` ou `char`. A palavra-chave especial `default` indica que ação deve ser tomada quando a variável assume um valor que não foi previsto em nenhum dos casos anteriormente listados. Assim como a condição `else` no comando `if` é opcional, a condição `default` também é opcional para o `switch-case`. Observe também a importância da palavra-chave `break` para delimitar o escopo de ação de cada caso — fossem omitidas as ocorrências de `break` no exemplo, a semântica associada ao comando seria essencialmente diferente (Figura 5.4).

Comandos de repetição ou iteração em C++ são suportados em três formas distintas. A primeira forma é `while`, na qual uma condição é verificada antes da primeira execução do bloco de comandos. Se a condição for avaliada como falsa, o bloco não é executado nenhuma vez (Figura 5.5).

Outra forma de expressar um comando de repetição em C++ é através do comando `do while`, que garante a execução do bloco de comandos pelo menos uma vez antes de avaliar a condição de terminação (Figura 5.6).

A terceira forma associada ao comando de repetição em C++, `for`, facilita a expressão de iterações associadas a contadores, com uma variável que é inicializada e incrementada para

Figura 5.4 Seleção em C++ usando a forma switch... case com a omissão da palavra-chave *break* em cada bloco de comandos.

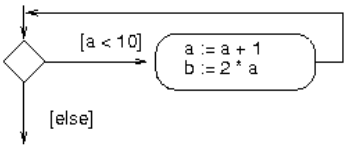


(a) Representação gráfica

```
switch (a) {
case 10:
   ++a;

case 20:
   a += 3;

case 100:
   a += 5;

default:
   a += 10;
}
```

(b) Expressão em C++

Figura 5.5 Comando de repetição while.

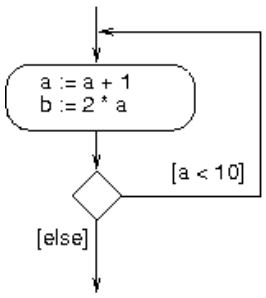


(a) Representação gráfica

```
while (a < 10)

   b = 2 * ++a;
```

(b) Expressão em C++

Figura 5.6 Comando de repetição do while.



(a) Representação gráfica

```
do
   b = 2 * ++a;
while (a < 10);
```

(b) Expressão em C++

controlar a execução repetida. Considere que, na situação apresentada na Figura 5.5, a variável `a` tivesse sido anteriormente inicializada com o valor 0. O comando `for` correspondente seria então:

```
for (a=0; a<10; ++a)
    b = 2*(a+1);
```

Neste exemplo, `a` é uma variável que tem a função de contador, assumindo valores 0, 1, ..., 9. Enquanto o valor de `a` for menor que 10 (a condição de término da iteração), a expressão (simples ou composta) no corpo da iteração será repetidamente avaliada. Se a variável não tiver sido declarada anteriormente, a declaração pode ser combinada com a inicialização do laço, como em

```
for (int a=0; a<10; ++a)
```

Qualquer que seja forma usada para indicar o comando de repetição — `while`, `do while` ou `for` — há duas formas de se interromper a seqüência de execução do bloco de comandos. A primeira forma, usando a palavra-chave `continue`, serve para indicar o fim prematuro de uma iteração. A outra forma de interrupção de um comando de repetição é o comando `break`, que indica o fim prematuro de todo o comando de iteração. Por exemplo, em

```
for (a=0; a<10; ++a) {
  if (b == 0) {
    b = a;
    continue;
  }
  c = c/b;
  b = b-1;
}
```

se a linha com o comando `continue` for executada, o valor de `a` será incrementado e então o teste da iteração será reavaliado para definir a continuidade ou não do laço de repetição. Já no exemplo abaixo,

```
for (a=0; a<10; ++a) {
  if (b == 0)
    break;
  c = c/b;
  b = b-1;
}
```

quando (se) `b` assumir o valor 0, o laço será simplesmente interrompido e a primeira instrução após o bloco `for` será executada.

5.1.5 Arquivos em C

Na linguagem de programação C, a informação sobre um arquivo é acessada através de um descritor cuja estrutura é definida no arquivo de cabeçalho `stdio.h`. Um programa C que vá manipular arquivos deve então incorporar ao início de seu programa fonte a linha de inclusão desse arquivo de cabeçalho:

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

Esse arquivo de cabeçalho define o nome de tipo `FILE` associado a essa estrutura. Não é necessário conhecer o formato interno dessa estrutura para manipular arquivos. O programador C, para acessar arquivos, define variáveis ponteiros para este tipo, `FILE *`, que são manipuladas diretamente pelas funções da biblioteca padrão de entrada e saída. Tais variáveis são usualmente chamadas de **manipuladores de arquivo**.

Assim, a função que vai manipular o arquivo deve incluir a declaração de uma variável manipulador de arquivo, como em:

```
FILE *arqFonte;
```

O objetivo de manipular um arquivo é realizar operações de leitura e escrita sobre seu conteúdo. Para que essas operações de transferência de dados tenham sucesso, é preciso que haja a permissão adequada para a operação. Por exemplo, um teclado seria um dispositivo que não aceita saída de dados (escrita), mas apenas entrada (leitura).

Para abrir um arquivo em C, a rotina `fopen` é invocada recebendo dois parâmetros. O primeiro é uma *string* com o nome do arquivo que será aberto. O segundo parâmetro é outra *string* que especifica o modo de acesso, que pode conter os caracteres `r` (leitura), `w` (escrita), `a` (escrita ao final — *append*), e `b` (acesso em modo binário). O valor de retorno é o manipulador alocado para o arquivo aberto.

Por exemplo, para realizar a leitura do conteúdo de um arquivo `teste.asm`, a seguinte expressão poderia ser usada no programa:

```
arqFonte = fopen("teste.asm", "r");
```

Caso o arquivo não possa ser aberto, a função `fopen` retorna o ponteiro nulo. Assim, para verificar de o arquivo foi aberto sem problemas, é necessário testar o valor de retorno:

```
if (arqFonte != 0) {
    /*  tudo bem */
}
else {
    /* erro */
}
```

Encerradas as operações sobre um arquivo, ele deve ser fechado. Isso permite que o sistema operacional libere o espaço ocupado pelas informações sobre o arquivo para que esse mesmo espaço possa ser reocupado para a manipulação de outros arquivos. Esta liberação é importante, uma vez que sistemas operacionais tipicamente limitam a quantidade de arquivos que podem ser abertos simultaneamente devido a restrições de espaço alocado para essas estruturas auxiliares.

Para fechar um arquivo previamente aberto, a rotina `fclose` pode ser usada. Ela recebe como argumento o manipulador do arquivo e não retorna nenhum valor. Assim, após encerrada a operação com o arquivo a expressão `fclose(arqFonte);` fecha-o.

Quando o arquivo é aberto, a posição corrente (mantida internamente pelo sistema) é o início do arquivo. A cada operação executada sobre o arquivo, essa posição é atualizada. O valor da posição corrente pode ser obtido pela função `ftell`. A função `feof` retorna um

valor verdadeiro (inteiro diferente de 0) se a posição corrente para o arquivo indicado é o final do arquivo, ou falso (inteiro igual a 0) em caso contrário.

Na maior parte dos exemplos analisados neste texto, os arquivos estarão sendo manipulados de forma seqüencial. Assim, na leitura de um arquivo contendo texto, após a leitura de um caráter a posição corrente do arquivo estará indicando o próximo caráter; após a leitura de uma linha, a posição indicada será o início da próxima linha. A rotina C para obter um caráter de um arquivo é `fgetc`:

```
int fgetc(FILE *stream);
```

O valor de retorno de `fgetc` é um inteiro, que pode conter o código ASCII do caráter ou o valor `EOF` (definido em `stdio.h`), que indica o final do arquivo ou a ocorrência de alguma condição de erro.

Uma linha de um arquivo texto nada mais é do que uma seqüência de caracteres seguido por um caráter terminador de linha *(newline)*. Tipicamente, o caráter terminador de linha adotado é o CR (ASCII 0x0D), embora alguns sistemas operacionais adotem o par CR/LF (o par de valores 0x0D e 0x0A) como terminador de linha. A linguagem C traduz o terminador de linha para o caráter `'\n'`.

Para ler uma linha de um arquivo texto, a função da biblioteca padrão C `fgets` pode ser utilizada:

```
char *fgets(char *s, int size, FILE *stream);
```

Essa função recebe três argumentos. O primeiro é o endereço de um arranjo de caracteres que irá receber a linha lida; esse arranjo deve ter capacidade para pelo menos `size` caracteres. O segundo é o número máximo de caracteres que deve ser lido da linha, caso a linha tenha mais caracteres que essa quantidade. O terceiro parâmetro é o manipulador do arquivo de onde a linha será lida. O retorno é um ponteiro para o início do arranjo com a linha, ou o ponteiro nulo caso o final do arquivo tenha sido atingido. Se houver espaço para o terminador de linha no arranjo, ele será incluído. Após o último caráter lido, a rotina inclui o terminador de *string* `'\0'`.

Operações correspondentes para a escrita em arquivos são oferecidas pela biblioteca padrão de C. Para escrever um caráter na posição corrente de um arquivo, a rotina `fputc` é usada:

```
int fputc(int c, FILE *stream);
```

Para escrever uma *string*, a rotina `fputs` pode ser utilizada:

```
int fputs(const char *s, FILE *stream);
```

Neste caso, a *string* apontada por `s` (sem o terminador de *string* `'\0'`) é escrita no arquivo.

A função `fseek` permite modificar a posição corrente para um ponto arbitrário do arquivo, se tal operação for permitida. O primeiro argumento dessa função é o manipulador do arquivo; o segundo, um valor `long` indicando o deslocamento desejado; e o terceiro, um valor inteiro inidcando a referência para o deslocamento, que pode ser o início do arquivo (`SEEK_SET`), a posição corrente (`SEEK_CUR`) ou o final do arquivo (`SEEK_END`). Um valor de retorno 0 indica que a operação foi completada com sucesso. A função `rewind` retorna a posição corrente para o início do arquivo, sem valor de retorno.

5.2 Palavras reservadas em C e C++

asm	auto	break	case
catch	char	class	const
continue	default	delete	do
double	else	enum	extern
float	for	friend	goto
if	inline	int	long
new	operator	private	protected
public	register	return	short
signed	sizeof	static	struct
switch	template	this	throw
try	typedef	union	unsigned
virtual	void	volatile	while

5.3 Precedência de operadores

Na tabela a seguir resume-se a precedência dos operadores da linguagem C, assim como sua associatividade. Operadores em uma mesma linha têm a mesma precedência, e as linhas estão ordenadas em ordem decrescente de precedência.

Operador	**Associatividade**
`( ) [ ] -> .`	esq-dir
`! ~ ++ -- -` (type) `* & sizeof`	dir-esq
`* / %`	esq-dir
`+ -`	esq-dir
`<< >>`	esq-dir
`< <= > >=`	esq-dir
`== !=`	esq-dir
`&`	esq-dir
`^`	esq-dir
`|`	esq-dir
`&&`	esq-dir
`||`	esq-dir
`?:`	dir-esq
`= += -=` etc.	dir-esq
`,`	esq-dir

Capítulo 6

Exercícios

1. Crie funções em C++ para realizar as seguintes conversões entre tipos:

   (a) um caráter que representa um dígito decimal para o valor inteiro correspondente (retorna -1 se o caráter não é um dígito);

   (b) um caráter que representa um dígito hexadecimal para o valor inteiro correspondente (retorna -1 se o caráter não é um dígito hexadecimal);

   (c) uma string com uma seqüência de dígitos decimais, opcionalmente precedida com o caráter `'-'`, para o valor inteiro correspondente;

   (d) um valor inteiro para uma string contendo a representação decimal desse valor, precedida pelo caráter `'-'` se o valor for negativo;

   (e) um valor inteiro para uma string com 32 caracteres contendo a representação binária em complemento de dois desse valor.

2. Crie uma função em C++ que recebe duas string e retorna verdadeiro se elas forem iguais independentemente se seus caracteres estiverem em minúsculas ou em maiúsculas. Assim, a invocação com duas strings cujo conteúdos são `"Abc"` e `"aBc"` deve retornar `true`.

3. ROT13 é um antigo e simples procedimento para criptografar textos, que substitui cada caráter por outro que está distante 13 posições no alfabeto — A por N (e N por A), B por O e assim sucessivamente.

   (a) Crie uma função em C++ que receba uma string e retorne outra string criptografada por esse esquema. Caracteres não-alfabéticos devem permanecer inalterados e a caixa do caráter (maiúscula ou minúscula) deve ser preservada.

   (b) Considere que um container de strings contém palavras, todas em letras minúsculas. Apresente uma função C++ que aplique o procedimento de criptografia ROT13 a cada palavra e retorne uma lista de pares de strings que são equivalentes em ROT13. Por exemplo, se `terra` e `green` são elementos do container, a lista de saída deve conter uma string com o par `terra:green`.

   (c) Mostre como estender ROT13 para contemplar caracteres de pontuação e outros elementos da tabela ASCII. Ao invés de 13, qual é o passo usado neste caso?

4. Algumas implementações da biblioteca STL de C++ oferecem *containers* complementares, como é o caso de `slist` para a implementação de uma lista simplesmente ligada. No entanto, neste caso, as operações `insert` e `erase` têm complexidade linear ao invés de constante, como é o caso de `list`. Por que isto acontece? Pesquise essas implementações e mostre qual é a alternativa que elas propõem para aliviar esse problema.

5. Uma das formas possíveis de implementação de uma lista simplesmente ligada é através da utilização de dois arranjos, um para os elementos armazenados e outro contendo, na mesma posição correspondente ao elemento, a indicação da posição no arranjo para o próximo elemento. Um índice inválido (por exemplo, -1) é utilizado para indicar que não há próximo elemento. Usando esta estratégia, mostre como seriam realizadas as seguintes operações na lista:

   (a) inserir novo elemento no início;

   (b) inserir novo elemento no final;

   (c) retirar o elemento do início;

   (d) retirar o elemento do final;

   (e) inserir um novo elemento em posição intermediária indicada;

   (f) retirar o elemento da posição intermediária indicada.

6. Uma possível implementação de uma lista simplesmente ligada utiliza um arranjo onde cada posição armazena dois valores, sendo o primeiro uma chave de busca e o segundo uma posição para o próximo elemento da lista. Um valor inválido nesta posição (aqui denotado Λ) indica que não há próximo elemento. Uma variável auxiliar *(cabeça)* indica qual a posição do arranjo que tem o primeiro elemento da lista. Outra variável, *livre*, aponta para o início da lista de elementos livres. Utilizando essa abordagem de implementação, considere a estrutura com os seguintes elementos:

Cabeça: 3		
Livre: 5		
Posição	Chave	Próximo
0	3	6
1	9	2
2	2	Λ
3	5	7
4	7	0
5	1	4
6	4	Λ
7	3	1

   (a) Apresente uma representação gráfica para a lista ligada ocupada e para a lista de elementos livres.

   (b) Apresente o estado final dessa estrutura após cada passo da execução da seguinte seqüência de operações:

- Inserir no início da lista um elemento com chave 8;
- Retirar da lista um elemento com chave 3; e
- Acrescentar no final da lista um elemento com chave 6.

(c) Como essas operações poderiam ser implementadas em cada uma das estruturas lineares da STL de C++? Em qual delas a implementação seria mais eficiente?

(d) Considere uma instalação de C++ que não suporte as estruturas de STL. Neste caso, usando apenas essa estrutura de arranjos, proponha uma implementação de um conjunto de funções que suporte as operações de manipulação de lista.

7. Para a seguinte lista de valores inteiros,

```
123, 091, 325, 129, 245, 003
```

apresente todos os passos intermediários da ordenação usando

(a) Ordenação pela seleção do menor valor;

(b) Quicksort;

(c) Radix sort tendo como raiz um dígito na representação decimal;

(d) Radix sort tendo como raiz três bits da representação binária.

8. Para o algoritmo de ordenação da bolha *(bubble sort)*:

(a) apresente uma descrição do algoritmo em pseudo-código, com notação similar à utilizada para a apresentação dos demais algoritmos de ordenação no texto;

(b) aplique o algoritmo de ordenação ao conjunto de valores do exercício anterior;

(c) compare este algoritmo com o de ordenação pela seleção do menor valor em termos dos números de comparações e números de trocas. Como esse resultado pode ser generalizado, em termos de projeção, para um número n qualquer de elementos?

9. O algoritmo quicksort apresentado no texto não faz um bom particionamento quando o valor na posição *init*, escolhida como pivô, é menor ou maior que todos os demais valores considerados na invocação. Modifique o algoritmo de modo que o pivô seja selecionado pela estratégia **mediana de três**, ou seja, ao invés de tomar o pivô diretamente do valor inicial tome a valor intermediário entre as três primeiras posições. Modifique também de forma que, se a faixa especificada tiver apenas três valores ou menos, a ordenação não use quicksort mas uma estratégia de comparação direta de valores.

10. Uma estrutura de árvore binária foi utilizada para armazenar nós de informação com chaves de valor 2, 11, 23, 7, 5, 41, 13 e 19, inseridos nesta ordem.

(a) Apresente graficamente a organização desta árvore após a inserção de cada um dos elementos acima.

(b) Mostre uma representação esquemática de como seria a estrutura da árvore caso ela fosse balanceada, considerando que a árvore adote a estratégia de varredura intra-ordem.

(c) Repita o item *(b)* considerando que a árvore adote a estratégia pré-ordem.

(d) Repita o mesmo item considerando que a árvore adote a estratégia pós-ordem.

11. Uma árvore binária contendo como chaves valores do tipo inteiro foi construída segundo a seguinte regra de formação:

    - Cada nó da árvore ocupa três posições no arranjo;
    - A primeira das três posições de um nó contém o valor da chave;
    - A segunda das três posições de um nó contém a posição no arranjo coma raiz da sub-árvore esquerda; um valor -1 indica que não há sub-árvore esquerda para este nó;
    - A terceira das três posições de um nó contém a posição no arranjo coma raiz da sub-árvore direita; um valor -1 indica que não há sub-árvore direita para este nó.

    O arranjo `bt1` a seguir representa a informação de uma árvore construída segundo esta regra:

    ```
    int bt1[] = { 10,  3, 12,  7,  6,  9,  5, -1,
                  -1,  8, -1, -1, 15, 15, 18, 12,
                  -1, -1, 19, -1, -1};
    ```

    (a) Ilustre graficamente a estrutura da árvore binária representada por `bt1`.

    (b) Uma tentativa de implementação para realizar a varredura intra-ordem numa estrutura desse tipo foi proposta como a seguir:

    ```
    void btscan(int *t) {
      if (*(t+1) != -1) btscan(t+*(t+1));
      cout << *t;
      if (*(t+2) != -1) btscan(t+*(t+2));
    }
    ```

    No entanto, a invocação `btscan(bt1)` gera um erro fatal de execução, o que indica que a lógica desta função apresenta um problema. Qual é o problema e qual a sua solução?

12. O conjunto de números inteiros sem sinal

    ```
    62512, 61544, 63489, 64118, 60382, 64550
    ```

    representa identificadores que serão usados como chaves em uma pesquisa em uma tabela hash de 16 posições. Mostre como seria a distribuição dessas chaves usando como função hash:

    (a) O método do meio do quadrado;

(b) O método da divisão.

13. Uma aplicação usual da técnica de folding para elementos do tipo string utiliza como segmento um byte, cujo valor é a representação ASCII de cada caráter. Considerando que os elementos que serão armazenados numa tabela hash com 13 posições, usando o método da divisão, sejam

```
UNICAMP
Universidade
Campinas
Engenharia
Software
```

indique qual a posição associada a cada elemento quando o valor inteiro é gerado a partir da

(a) Soma dos valores ASCII de cada caráter (folding de um caráter combinado com soma aritmética).

(b) Aplicação da função binária XOR entre todos os caracteres individualmente (folding de um caráter combinado com soma ou-exclusivo).

(c) Aplicação da função binária XOR entre pares de caracteres a partir da primeira posição (folding de dois caracteres combinado com soma ou-exclusivo).