Introdução à Programação

Curso em C++

Dr. Alan R. R. de Freitas

ALANDEFREITAS.COM

Edição: 19 de Março de 2015

Conteúdo

Programação Estruturada

1. Introdução ao C++

1.1 Porquê Programar

Sabemos que a utilização de computadores é claramente uma vantagem em nossas vidas. O computador é um facilitador da nossa comunicação, do nosso planejamento e essa praticidade nos permite trabalhar menos e nos fornece diversão. Mas por que fazer nossos próprios programas?

Uma resposta simples para isso é que tudo é uma questão de automatização. Programadores têm muitas vezes a reputação de serem preguiçosos, mas muitas vezes pessoas preguiçosas são aquelas boas em encontrar soluções fáceis e eficientes para problemas difíceis.

Sendo assim, você já parou para pensar em quantas tarefas repetitivas temos em nossas vidas? Em nossa profissão? Na família? Nós programadores costumamos ter ferramentas próprias para o dia-a-dia: calendários, lembretes, processadores de dados, e várias outras.

A programação é necessária em tarefas avançadas de todas as áreas do conhecimento. Cálculo é uma delas. Muitos problemas matemáticos só puderam ser resolvidos por causa da capacidade dos computadores atuais. Todo ano, milhares de problemas novos e mais difíceis são solucionados.

Outra vantagem de se saber programar é saber por que um computador pode falhar. Nós programadores, usualmente, reconhecemos o motivo de erros em programas. Isso só acontece porque já passamos pelos mesmos erros em nossos próprios programas.

Além disso, a programação te ajuda a pensar e a quebrar problemas grandes em partes menores. O único limite no seu programa vai ser a sua imaginação. Uma ideia simples pode mudar a vida de milhões de pessoas.

1.2 Linguagem C++

A linguagem C++ foi desenvolvida por Bjarne Stroustrup quando trabalhava para a Bell Labs, durante a década de 1980. Nesta época, o Unix, sistema operacional também desenvolvido pela Bell Labs, era desenvolvido na linguagem C. Era importante que fosse mantida compatibilidade entre a nova linguagem C++ e a antiga linguagem C.

Então em 1983 o nome da linguagem foi alterado de “C com Classes”, *C with Classes* em

inglês, para C++. Uma novidade importante da linguagem C++ é que ela contém uma biblioteca padrão de recursos que também foi sendo melhorada ao longo do tempo.

Uma característica importante do C++ é ser considerada de médio nível. Isso que dizer que ela não é tão distante da linguagem da máquina, a língua dos computadores. Ao mesmo tempo ela também não é tão distante de uma linguagem com um alto nível de abstração, a tornando acessível a nós humanos.

Estas características tornam o C++ uma linguagem altamente eficiente e a fazem uma das linguagens comerciais mais populares atualmente.

Como referência para leitores que já são programadores porém não estão acostumados com a linguagem C++, recomendamos a leitura direta da Seção 13, onde são apresentados os principais comandos da linguagem.

1.3 Compilador

Como faremos para conversar com computadores se falamos línguas tão diferentes? Esta é a tarefa de um programa de computador chamado **compilador**.

O compilador é uma ferramenta que nos serve como uma ponte para fazer esta comunicação. De modo breve, a comunicação funciona da seguinte maneira:

1. Escreveremos nossa mensagem para o computador em uma linguagem de programação preferêncial. Neste curso, utilizaremos a linguagem **C++**.
2. O compilador, por sua vez, recebe esse código que é, então, transformado em linguagem de máquina.
3. O novo código em linguagem de máquina pode ser utilizado diretamente pelo computador, finalizando assim nossa comunicação.

1.4 Tipos Fundamentais de Dados

Uma linguagem de máquina, que é entendida pelo computador, é cheia de 0's e 1's mas não bem assim que nós nos comunicamos com nossos programas. Se enviarmos 0100110101 para um computador, como ele conseguirá identificar se o que queremos é um letra minúscula ou uma letra maiúscula? Ou como diferenciar um número inteiro de um número real?

Para isto precisamos identificar os tipos de dados. Há diferentes tipos de dados que podemos usar e eles têm diferentes tamanhos. Na Tabela 1.1 temos os quatro tipos fundamentais de dados e suas respectivas palavras-chave que as representam em C++.

Tipos de dados	Palavra-Chave	Exemplos
Tipo lógico	`bool`	V ou F
Números inteiros	`short` < `int` < `long`	... -3, -2, -1, 0, 1, 2, 3 ...
Números reais	`float` < `double`	3.32, 4.78, 7.24, -3.14, 0.01
Caracteres	`char`	a, b, c, d, e, f ...#, $, [, ., \ ..

Tabela 1.1: Tipos Fundamentais de Dados em C++

O tipo lógico, que possui `bool` como palavra chave, representa valores que são **verdadeiros** ou **falsos**. O **verdadeiro** é representado pela palavra-chave `true` e o **falso** é representado pela palavra-chave `false`.

Os **números inteiros**, possuem `short`, `int` e `long` como palavras-chave. `short` é utilizado para números inteiros menores, `int`, para números inteiros que podem ser maiores e `long`, para

números que podem ser ainda maiores. Estes tipos de dados podem representar valores como `7`, `-5`, `-1`, `0`, `+3`, `-2`, ou `10`.

Já os **números reais** possuem as palavras-chave `float` e `double`. Da mesma maneira, ambas as palavras indicam a capacidade do número. Exemplos desse tipo de dados são números como `3.32`, `4.78`, `7.24`, `-3.14` ou `0.01`.

Por último, temos os **caracteres**, que podem representar letras ou símbolos. Eles possuem `char` como palavra-chave, e servem para representar dados como: `h`, `@`, `C`, `r`, `e`, `f`, `#`, `$` ou `}`.

2. Estruturas sequenciais

2.1 Primeiros programas

Para criarmos nosso primeiro programa em C++ começaremos com este código.

```
1 #include <iostream>
2
3 int main(){
4     std::cout << "Olá␣mundo!" << std::endl;
5     return 0;
6 }
```

O comando apresentado na primeira linha, `#include <iostream>`, inclui a biblioteca com recursos para fluxo de entrada e saída, chamados também de ***in and out stream***. Quando dizemos entrada, essa será feita usualmente via teclado. Quando dizemos saída, essa será feita pelo monitor.

É na terceira linha que começa a função principal do nosso programa, chamada de `main()` em inglês. As chaves indicam onde começa e termina essa função.

Logo em seguida temos o comando `cout`. Dizemos que `cout` é um objeto que faz saída de dados para o monitor. Esses dados são enviados para ele em cadeia com os operadores de inserção (<<), que estão a sua direita.

Para evitar um possível conflito entre os nomes, os recursos do C++ são divididos em **espaços de nomes**. O `cout`, por exemplo, é precedido de `std` e dois pontos seguidos (`::`) para indicar que ele pertence ao espaço de nomes padrão da linguagem, conhecido também como ***standard***.

Temos agora a linha 4, onde o operador de inserção é então utilizado para enviar a frase `‘‘Olá, Mundo!’’` para `cout`. Cada letra desta frase é um dado do tipo `char`, que representa um caractere. Lembre-se dos tipos fundamentais de dados apresentados na Tabela 1.1. No código acima, o símbolo ␣ indica uma barra de espaços dentro de uma sequência de caracteres.

As aspas ao redor da expressão `‘‘Olá, Mundo!’’` indicam que este é um conjunto de `char`s, e não uma palavra especial que é reservada da linguagem.

Além desta frase, uma quebra de linha é também enviada para `cout`. Essa quebra de linha é representada pelo comando `endl`, que também pertence ao espaço de nomes `std`.

Todos os comandos em C++ **devem** ser acompanhados de um ponto e vírgula indicando a onde termina aquele comando.

O comando `return` (*retornar*, em inglês), indica que a função `main` retorna daquele ponto, finalizando assim o programa. Mas esse comando não só encerra o programa como também retorna um valor do tipo `int`, que é no caso um `0`. Este `0` é utilizado usualmente por programadores para indicar que o programa encerrou sem erros.

O comando `int`, antes da definição `main()` na linha 3, indica que a própria função `main` irá retornar um `int`. Discutiremos o significado desse comando em lições futuras sobre funções.

Lembre-se que a função `main` é obrigatória em todos os programas.

Este é o resultado do programa acima:

```
Olá, Mundo!
```

2.2 Segundo programa em C++ e espaço de nomes padrão

Veja o código abaixo:

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

Na linha 3 deste segundo programa em C++, indicamos com o comando `namespace` que o **espaço de nomes** `std` deve ser utilizado se nenhum outro **espaço de nomes** for mencionado. Eliminamos assim os vários `stss` e seguindos de dois pontos (`::`) em nosso código.

Como podem ver, as variáveis em C++ são organizadas nestes **espaços de nomes** para evitar conflito entre duas variáveis que por acaso tenham o mesmo nome identificador.

Assim como nosso primeiro programa, o programa acima irá mostrar na tela a seguinte mensagem:

```
Olá, Mundo!
```

2.3 Comentários

Em alguns casos, um código pode se tornar tão complexo que nem o próprio programador consegue entendê-lo. Em casos como este, é bom deixar comentários para outros programadores ou mesmo para si mesmo.

Não se preocupe, pois os comentários são partes do código que são ignoradas pelo compilador e servem apenas para deixar lembretes ou informações sobre o código para outros programadores.

O comando usado para marcar o início de um comentário de uma linha são duas barras para a direita (`//`). Já os comandos barra-asterisco (`/*` e asterisco-barra (`*/`) são utilizados para iniciar e encerrar um bloco de comentários, respectivamente. Veja o exemplo abaixo:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     cout << "Olá, Mundo!" << endl; // Tudo isto é ignorado
7     // cout << "Olá, Mundo!" << endl;
8     cout << "Olá, Mais uma vez" << endl;
9     /* Aqui se inicia um bloco de comentários.
10    Tudo dentro do bloco é ignorado. */
11    //cout << "Este"Olá, Mundo!" não sera impresso!" << endl;
12    return 0;
13 }
```

Temos acima mais um exemplo de programa em C++. Neste programa, temos comandos que imprimem a frase "Olá, Mundo!", mas temos também vários comentários. Estes comentários são trechos no código que serão ignorados.

Logo na linha 6, a mensagem "Olá, Mundo!" é enviada para o monitor. Após o comando, porém, há uma mensagem que é ignorada pelo computador por ser um comentário.

Ao fim da linha 6, a seguinte mensagem terá sido mostrada no monitor:

```
Olá, Mundo!
```

Como a linha 7 é toda um comentário, ela é totalmente ignorada. Passamos então direto para a linha 8 do código, que imprime novamente uma mensagem:

```
Olá, Mundo! Olá!  Mais uma vez!
```

Temos em seguida nas linhas 9 e 10, todo um bloco de comentários entre os comandos `/*` e `*/` que é também ignorado. A linha 11 também é um comentário de uma linha inteira feito com o comando `//`. O programa retorna da linha 12 com o comando `return 0`.

2.4 Criando variáveis

Os dados do nosso programa serão guardados em **variáveis**, que são espaços alocados na memória do computador para nossos dados, como representado na Figura 2.1

Para cada variável é necessário informar qual o seu **tipo de dado** e dar a ela um nome **identificador** que a referencia.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     // criando uma variável do tipo inteira com nome x
7     int numero;
8     // criando uma variável do tipo char com o nome letra
9     char letra;
10    // criando uma variável do tipo double com nome num_real
```

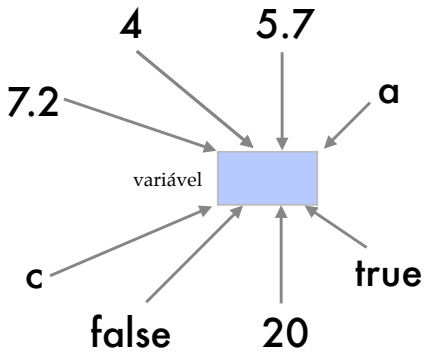


Figura 2.1: Uma variável é um espaço alocado na memória do computador para guardarmos dados

```
11      double num_real;
12      // criando uma variável do tipo bool com nome teste
13      bool teste;
14      cout << "Este programa criou 4 variáveis!" << endl;
15      return 0;
16  }
```

Neste exemplo, temos um programa onde 4 variáveis são criadas. A primeira delas, criada na linha 5, irá guardar um número inteiro (`int`) e seu nome identificador é `numero`. Note que o nome que identifica a variável `numero` não contém acentos.

Na linha 9 criamos uma variável chamada `letra`, para dados do tipo `char`. Na linha 7, uma variável do tipo `double`, que pode armazenar números reais. Por fim, na linha 8, uma variável do tipo `bool`, que pode guardar dados lógicos. Lembre-se dos tipos de dados apresentados na Seção 1.4.

Por enquanto, nosso programa não utilizou as variáveis criadas para armazenar dados. Por isto elas são representadas por caixas que ainda estão fazias, já que nenhum valor foi definido para as variáveis. Após a linha 13 do código isto seria representado como na Figura 2.2.

Figura 2.2: Quatro variáveis criadas que ainda não têm valores especificados.

Finalmente na próxima linha, este programa imprime uma mensagem simples, como já temos feito até agora.

```
Este programa criou 4 variáveis!
```

Os identificadores são usados para dar nomes às variáveis ou funções que criamos. Estudaremos as funções mais adiante. Neste programa criamos variáveis com os identificadores `numero`, `letra`, `num_real` e `teste`.

Exitem algumas regras para identificadores:

- O primeiro caractere de um identificador deve ser uma letra ou sinal de sublinha (_). Isto quer dizer que nomes como `8nome` ou `@nome` não são válidos.
- As letras minúsculas e maiúsculas também são identificadas de formas diferentes. Ou seja, para o C++, uma variável **x** minúsculo é diferente de uma variável **X** maiúsculo.

3. Operadores básicos

Os operadores são muito importantes em programação, pois eles alteram o valor das variáveis ou usam variáveis existentes para criarem novas, como apresentado na Figura 3.2.

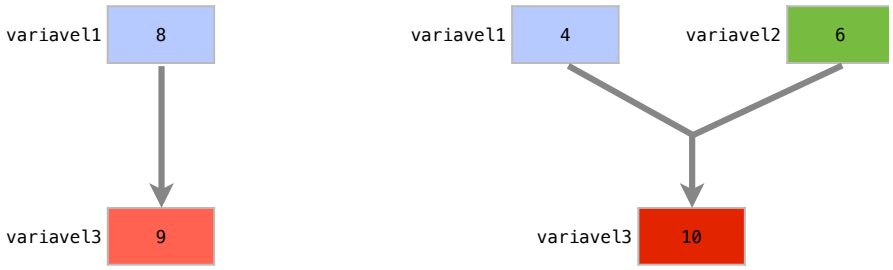


Figura 3.1: Exemplos de operadores. Operadores podem alterar o valor de uma variável ou criar novas variáveis a partir de variáveis existentes.

3.1 Operador de atribuição

O operador representado pelo sinal de igualdade (=) em C++ não diz que dois valores são iguais. Ele na verdade, atribui o valor da direta à variável. Por isto, chamamos ele de operador de atribuição. Sendo assim, não confunda este operador de atribuição com um operador de igualdade. Ele transfere um valor a uma variável e esse valor pode ser utilizado posteriormente. Quando fazemos isto, o valor antigo da variável é perdido.

Neste código utilizaremos um operador de atribuição para dar um valor a uma variável:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
```

```
6     int numero;
7     numero = 4;
8     cout << "A␣variável␣numero␣do␣tipo␣int␣tem␣valor";
9     cout << numero << endl;
10    return 0;
11 }
```

Logo no início do código, na linha 6, criamos uma variável chamada `numero`, para dados do tipo `int`. Na linha 7, esta variável recebe o valor 4 através do operador de atribuição. Assim temos o seguinte estado da variável:

O operador = utilizado na linha 7 disse que **atribuiremos** à variável `numero` o valor 4, mas não diz que o numero **é igual** a 4. Se atribuírmos outro valor à variável, o valor 4 será substituído pelo novo valor.

Ao imprimir a mensagem, o identificador `numero` é enviado para cout e é substituído pelo valor da variável `numero`.

```
A variável numero do tipo int tem valor 4
```

3.2 Fluxo de entrada

De modo análogo a `cout`, o `cin` é um objeto global definido em `iostream` e é utilizado para que o usuário possa fazer entradas no programa. Assim, dados podem ser atribuídos diretamente às variáveis, como apresentado na Figura 3.2.

Figura 3.2: O fluxo de entrada é utilizado para que o usuário possa dar valores a variáveis.

Esse mecanismo é importante para a interação com a pessoa que utiliza o programa. Dessa forma ele é complementar ao operador de atribuição para dar valor às variáveis.

Neste exemplo, temos um código onde utilizamos o fluxo de entrada para dar valor às variáveis:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     double numero1;
7     double numero2;
8     double soma;
9     cout << "digite␣o␣primeiro␣numero:␣";
10    cin >> numero1;
11    cout << "digite␣o␣segundo␣numero:␣";
12    cin >> numero2;
13    soma = numero1 + numero2;
14    cout << "A␣soma␣dos␣numeros␣é␣" << soma << endl;
15    return 0;
16 }
```

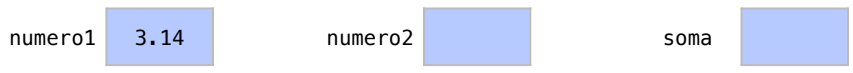
Logo no início do código, nas linhas 6 a 8, criamos 3 variáveis que receberão seus valores através do fluxo de entrada:

O comando que aparece na linha 9 imprime uma mensagem pedindo ao usuário que execute uma ação. Repare que não há o comando `endl` para passar para a próxima linha. Isso quer dizer que qualquer outro texto aparecerá na mesma linha pois não houve uma quebra de linha.

```
digite o primeiro numero:
```

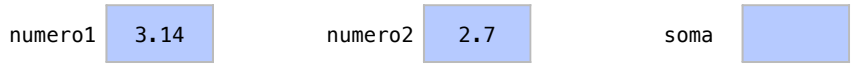
O `cin`, após a linha 8, é usado para obter um valor que virá do usuário pelo teclado. Por isto, o programa fica parado até que o usuário digite um valor e aperte a tecla *Enter*. Vamos supor que o valor `3.14` tenha sido digitado. Como não houve quebra de linha no comando anterior, o valor `3.14` aparece em frente ao último texto apresentado e o valor é, então, guardado na variável `numero1`, como indicado para o `cin`.

```
digite o primeiro numero:  3.14
```

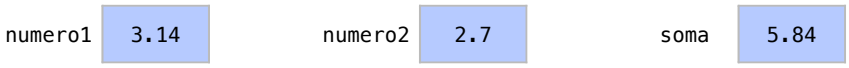


Vamos agora para o próximo passo na linha 10. Da mesma maneira, o programa espera uma entrada de dados do usuário novamente e recebe desta vez o valor `2.7`, por exemplo.

```
digite o primeiro numero:  3.14
digite o segundo numero:  2.7
```



Logo após, na linha 12, é atribuido à variável `soma` o valor `numero1` somado com o valor `numero2`. De modo que a cada variável há um valor atribuído agora.



O programa finaliza com uma última mensagem.

```
digite o primeiro numero:  3.14
digite o segundo numero:  2.7
A soma dos numeros é 5.84
```

3.2.1 Alterações possíveis

Vejamos agora algumas alterações possíveis no programa apresentado.

É possível termos uma lista separada por vírgulas para declaração das variáveis do mesmo tipo, como a acontece na linha 6 do programa a seguir.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      double numero1, numero2, soma;
7      cout << "digite␣o␣primeiro␣numero:␣";
8      cin >> numero1;
9      cout << "digite␣o␣segundo␣numero:␣";
10     cin >> numero2;
11     soma = numero1 + numero2;
12     cout << "A␣soma␣dos␣numeros␣é␣" << soma << endl;
13     return 0;
14 }
```

Apesar de possível, é mais interessante fazer a declaração de variáveis separadamente, pois isso nos permite inserir comentários mais facilmente e deixa o código mais legível.

Há ainda uma segunda alteração possível. Neste código a seguir, fizemos a adição dos dois números diretamente nos parâmetros do `cout`, na linha 12:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      double numero1, numero2;
7      cout << "digite␣o␣primeiro␣numero:␣";
8      cin >> numero1;
9      cout << "digite␣o␣segundo␣numero:␣";
10     cin >> numero2;
11     cout << "A␣soma␣dos␣numeros␣é␣";
12     cout << numero1 + numero2 << endl;
13     return 0;
14 }
```

Assim, utilizamos uma variável a menos. O resultado da soma fica armazenado temporariamente para que o `cout` possa utilizá-lo. Ao fazer isto, após o comando com o `cout`, o resultado da soma é descartado.

3.3 Operadores aritméticos

Além do operador de atribuição, representado por um sinal de igualdade (=), existem os operadores aritméticos. Esses são utilizados principalmente para tipos de dados que representam números.

Já utilizamos um operador aritmético de soma, representado pelo próprio sinal de mais (+), no exemplo anterior. A Figura 3.3 representa como um operador aritmético utiliza os valores de duas variáveis para criar uma nova variável.

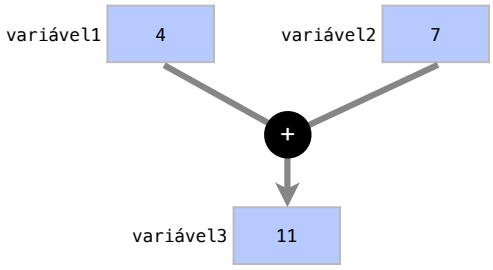


Figura 3.3: Exemplo de operador aritmético. Os valores de duas variáveis são utilizados para a criação de uma nova variável.

A Tabela 3.1 apresenta os operadores aritméticos.

Operador	Comando
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Resto da divisão inteira	%

Tabela 3.1: Tipos Fundamentais de Dados em C++

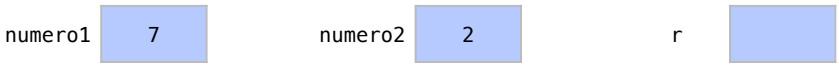
Neste exemplo, utilizaremos os operadores apresentados na Tabela:

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int numero1;
6      int numero2;
7      int r;
8      cout << "digite␣o␣primeiro␣numero:␣";
9      cin >> numero1;
10     cout << "digite␣o␣segundo␣numero:␣";
11     cin >> numero2;
12     r = numero1 + numero2;
13     cout << "Adição␣dos␣numeros␣=␣" << r << endl;
```

```
14      r = numero1 - numero2;
15      cout << "Subtração␣dos␣numeros␣=␣" << r << endl;
16      r = numero1 * numero2;
17      cout << "Multiplicação␣dos␣numeros␣=␣" << r << endl;
18      r = numero1 / numero2;
19      cout << "Divisão␣dos␣numeros␣=␣" << r << endl;
20      r = numero1 % numero2;
21      cout << "Resto␣da␣divisão␣=␣" << r << endl;
22      return 0;
23  }
```

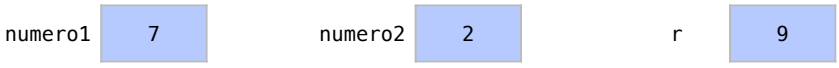
Como usual, criamos as variáveis que serão utilizadas no programa, nas linhas 5 a 7. Em seguida, nas linhas 8 a 11, o próprio usuário atribui valores às variáveis `numero1` e `numero2`, como já aprendemos na Seção 3.2. Neste exemplo, vamos supor que o usuário escolheu os valores 7 e 2:

```
Digite o primeiro numero 7
Digite o segundo numero 2
```



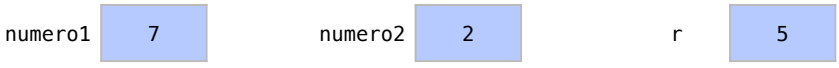
Veja na linha 12 o operador de adição, utilizado para fazer a soma que é atribuída a `r`. O comando com um `cout` imprime o resultado `r`, na linha 13, em seguida.

```
Digite o primeiro numero 7
Digite o segundo numero 2
Adição dos numeros = 9
```



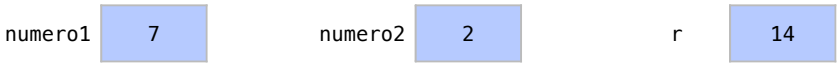
Fazemos o mesmo com o operador de subtração, nas linhas 13 e 14:

```
Digite o primeiro numero 7
Digite o segundo numero 2
Adição dos numeros = 9
Subtração dos numeros = 5
```



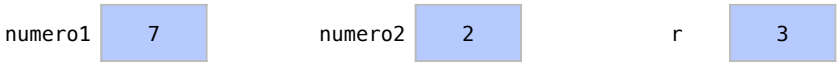
Fazemos o mesmo com o operador de multiplicação, nas linhas 15 e 16:

```
Digite o primeiro numero 7
Digite o segundo numero 2
Adição dos numeros = 9
Subtração dos numeros = 5
Multiplicação dos numeros = 14
```



Fazemos o mesmo com o operador de divisão, nas linhas 17 e 18:

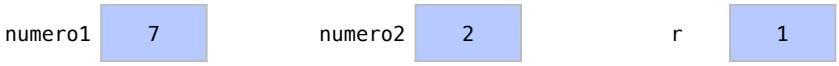
```
Digite o primeiro numero 7
Digite o segundo numero 2
Adição dos numeros = 9
Subtração dos numeros = 5
Multiplicação dos numeros = 14
Divisão dos numeros = 3
```



Repare que o resultado da divisão de 7 por 2 é 3 e não 3.5. Isto acontece porque `r` é uma variável do tipo `int`, que representa apenas **números inteiros**.

Veja enfim, na linha 20, o operador de resto da divisão inteira, que tem o seu resultado apresentado na linha 21.

```
Digite o primeiro numero 7
Digite o segundo numero 2
Adição dos numeros = 9
Subtração dos numeros = 5
Multiplicação dos numeros = 14
Divisão dos numeros = 3
Resto da divisão = 1
```



O programa apresenta o resto da divisão inteira de 7 por 2, que é igual a 1. Este operador só está disponível para dados de tipo inteiro, como `int`. Isso ocorre pois não existe resto da divisão inteira para dados que representam números reais, como `double`.

Os operadores aritméticos são usualmente utilizados para dados aritméticos, como `int` e `double`. Porém, os operadores aritméticos podem também ser utilizados para outros tipos de dados, como `bool` e `char`. Nestes casos, os operadores têm outros significados.

3.4 Operadores de atribuição aritméticos

Existem outros operadores de atribuição além do operador representado pelo sinal de igualdade =. Esses outros operadores de atribuição são utilizados para abreviar expressões aritméticas e devem ser utilizados sempre que possível. A alteração dos valores se dá como apresentado na Figura 3.4.

Imagine qualquer instrução onde somamos o valor de uma variável `x` a 2 e atribuímos o resultado à própria variável `x`. Isto pode ser convertido na forma `x = x + 2`. De forma mais geral, considere qualquer comando onde atribuímos à própria variável o valor de uma variável `x` somado a uma constante `c`. Isto pode ser convertido em um comando da forma `x = x + c`.

Um operador de atribuição aritmético permite que um comando `x = x + c` seja substituído por um comando na forma `x += c`. O comando `x = x + c` cria uma nova variável temporária com o valor de `x + c`, que será atribuído a `x`. Por outro lado, o comando `x += c` apenas incrementa o valor dex.

A Tabela 3.4 apresenta todos os operadores de atribuição artméticos e a mesma operação poderia ser feita com um comando que utilizasse operadores comuns aritméticos e de atribuição.

Operador	Exemplo	Comando Equivalente
+=	x += 6	x = x + 6
-=	x -= 3	x = x -3
*=	x *= 2	x = x * 2
/=	x /= 4	x = x / 2
%=	x %= 7	x = x % 7

Tabela 3.2: Operadores de Atribuição Aritméticos

Neste exemplo, utilizamos os operadores de atribuição aritméticos:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int numero1;
7      int numero2;
8      cout << "digite o primeiro numero: ";
9      cin >> numero1;
10     cout << "digite o segundo numero: ";
11     cin >> numero2;
12     numero1 += numero2;
13     cout << "Somando";
14     cout << numero2 << "->" << numero1 << endl;
15     numero1 -= numero2;
16     cout << "Subtraindo";
17     cout << numero2 << "->" << numero1 << endl;
18     numero1 *= numero2;
```

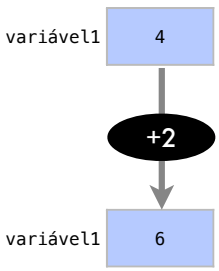


Figura 3.4: Exemplo de operador de atribuição aritmético. Os valor de uma variável é transformado de acordo com uma expressão matemática.

```
19      cout << "Multiplicando por";
20      cout << numero2 << "->" << numero1 << endl;
21      numero1 /= numero2;
22      cout << "Dividindo por";
23      cout << numero2 << "->" << numero1 << endl;
24      numero1 %= numero2;
25      cout << "Resto da divisão por";
26      cout << numero2 << "->" << numero1 << endl;
27      return 0;
28  }
```

São criadas nossas variáveis nas linhas 6 e 7. Em seguida, nas linhas 8 a 11, valores são atribuídos a elas:

```
digite o primeiro numero 8
digite o segundo numero 3
```

Na linha 12, o comando (+=) equivale a dizer que `numero1` recebe o próprio valor de `numero1` mais o valor de `numero2`. As linha 13 e 14 imprimem o resultado:

```
digite o primeiro numero 8
digite o segundo numero 3
Somando 3 -> 11
```

Após a impressão do resultado, vamos para a próxima operação de atribuição aritmética. No próximo comando na linha 15, o `numero1` recebe seu próprio valor menos o `numero2`. Imprimimos novamente o resultado nas linhas 17 e 18, onde o valor de `numero1` volta a ser 8:

```
digite o primeiro numero 8
digite o segundo numero 3
Somando 3 -> 11
Subtraindo 3 -> 8
```

Em seguida, na linha 18, o `numero1` recebe seu próprio valor vezes o `numero2`, e o resultado é impresso pelo código das linhas 19 e 20:

```
digite o primeiro numero 8
digite o segundo numero 3
Somando 3 -> 11
Subtraindo 3 -> 8
Multiplicando por 3 -> 24
```

Na linha 21, `numero1` recebe seu próprio valor dividido por `numero2` com a operação `/=`, ficando com 8 como resultado. Os valores das variáveis são impressos pelas linhas 22 e 23:

```
digite o primeiro numero 8
digite o segundo numero 3
Somando 3 -> 11
Subtraindo 3 -> 8
Multiplicando por 3 -> 24
Dividindo por 3 -> 8
```

Por fim, na linha 24, `numero1`, que tem valor 8, recebe o resto da divisão de seu próprio valor por `numero2`, que tem valor 3. O nosso valor de `numero1` passa a ser 2. Isto é impresso pelo código das linhas 25 e 26:

```
digite o primeiro numero 8
digite o segundo numero 3
Somando 3 -> 11
Subtraindo 3 -> 8
Multiplicando por 3 -> 24
Dividindo por 3 -> 8
Resto da divisão por 3 -> 2
```

3.5 Operadores de incremento e decremento

Assim como os operadores de atribuição aritméticos, os operadores de incremento e decremento são utilizados também para alterar o valor de variáveis. Estes operadores são usados para adicionar ou subtrair somente uma unidade do valor de uma variável, como apresentado na Figura 3.6.

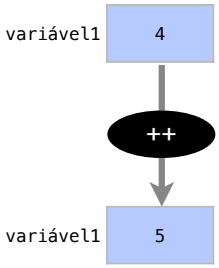


Figura 3.5: Exemplo de operador de atribuição aritmético. Os valor de uma variável é transformado de acordo com uma expressão matemática.

O operador de incremento é representado por dois sinais de adição seguidos (++) e ele serve para incrementar em uma unidade o valor de uma variável qualquer (x++). Como esta é uma operação comum, isto tira o trabalho de dizermos que `x = x + 1` ou que `x += 1`. Podemos apenas dizer x++.

A posição do operador também altera seu modo de funcionamento. Para utilizar e depois incrementar valores usamos x++. Chamamos este de um operador de **pós-incremento**. Para incrementar e depois utilizar valores usamos ++x. Chamamos este de um operador **prefixado**. De modo análogo, no caso do decremento, podemos utilizar x- ou -x. Isto ficará mais claro neste exemplo, onde mostraremos a diferença entre os vários operadores de incremento e decremento:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int x;
7      x = 0;
8      cout << "x␣=␣" << x++ << endl;
9      cout << "x␣=␣" << x << endl;
10     cout << "x␣=␣" << ++x << endl;
11     cout << "x␣=␣" << x << endl;
12     cout << "x␣=␣" << x-- << endl;
13     cout << "x␣=␣" << x << endl;
14     cout << "x␣=␣" << --x << endl;
15     cout << "x␣=␣" << x << endl;
16     return 0;
17 }
```

Iniciamos nossa função principal `main` com a criação de uma variável `x` na linha 6. Esta variável inicia com valor `0` que lhe é atribuído na linha 7 do código. Assim temos a seguinte condição das variáveis:

Na linha 8, utilizamos um operador de incremento em uma variável que está sendo enviada para um `cout`. Como utilizamos o operador de **pós-incremento** (x++), a variável será primeiramente utilizada no comando `cout`. Isto imprime seu valor:

```
x = 0
```

Apenas após sua utilização, o operador da linha 8 incrementa o valor de `x`, que passa a ser 1.

No comando seguinte da linha 9, simplesmente imprimimos o valor da variável `x`, que agora vale 1.

```
x = 0
x = 1
```

Agora na linha 10, utilizamos um operador de **incremento prefixado** (`++x`), onde `x` é incrementado e depois utilizado. Primeiramente, `x` é incrementado passando a valer 2 e em seguida seu valor é impresso:

```
x = 0
x = 1
x = 2
```

Na linha 11 imprimimos novamente o valor de `x`, sem alterá-lo:

```
x = 0
x = 1
x = 2
x = 2
```

De modo análogo, na linha 12 utilizamos o Operador de **pós-decremento** `x--`, onde `x` é utilizado e apenas depois decrementado. Isso imprime o valor 2 e depois altera o valor de `x` para 1:

```
x = 0
x = 1
x = 2
x = 2
x = 2
```

A linha 13 do código apenas imprime o valor de `x`, que agora é `1`:

```
x = 0
x = 1
x = 2
x = 2
x = 2
x = 1
```

Por fim, na linha 14 usamos o operador de **decremento prefixado** `-x`, para que `x` seja decrementado e depois utilizado. Assim, `x` passa primeiro a valer `0` e este valor é impresso em seguida:

```
x = 0
x = 1
x = 2
x = 2
x = 2
x = 1
x = 0
```

Após isto, na linha 15, o valor final de `x` é impresso mais uma vez e encerramos o programa:

```
x = 0
x = 1
x = 2
x = 2
x = 2
x = 1
x = 0
x = 0
```

Como vimos, o operador prefixado incrementa (`++i`) ou decrementa (`-i`) o valor da variável antes de usa-la. Apenas após a alteração, ela é utilizada com o valor modificado na expressão em que aparece. O operador de pós-incremento (`i++`) ou pós-decremento (`i-`) altera o valor da variável apenas depois que a mesma é utilizada na expressão em que aparece. Um quadro geral com os operadores de incremento e decremento é apresentado na Tabela 3.5.

Operador	**Significado**	**Representação**
Pós-incremento	Utiliza e depois incrementa	`i++`
Incremento prefixado	Incrementa e depois utiliza	`++i`
Pós-decremento	Utiliza e depois decrementa	`i--`
Decremento prefixado	Decrementa e depois utiliza	`--i`

Tabela 3.3: Operadores de Incremento de Decremento

3.6 Dados lógicos e comparações

É muito comum em nossas vidas termos dilemas que precisamos resolver. Dilemas são situações onde temos apenas duas opções. Por exemplo: Esquerda ou direita? Para cima ou para baixo? Dentro ou fora? Melhor ou pior? Sim ou não? Verdadeiro ou Falso? Zero ou um? Oito ou oitenta?

No caso da programação, variáveis lógicas são utilizadas para guardar o resultado de tais dilemas. Estas variáveis são criadas com a palavra-chave `bool` e podem assumir apenas dois valores: Verdadeiro, que é representado por `true`, e falso, que é representado por `false`.

Neste exemplo abaixo, criaremos variáveis lógicas e mostraremos seus valores:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      bool var1;
7      var1 = true
8      bool var2;
9      var2 = false;
10     cout << "var1 = " << var1 << endl;
11     cout << "var2 = " << var2 << endl;
12     return 0;
13 }
```

É criada a variável `var1` na linha 6. Logo em seguida, na linha 7, `var1` recebe `true`, ou seja, verdadeiro. Na linha 8 é criada a variável `var2`. Em seguida, na linha 9, `var2` recebe `false`, ou seja, falso. Assim, temos o seguinte estado das variáveis:

Nas linhas 10 e 11, imprimimos os valores destas variáveis. Veja como, em vez das palavras `true` e `false`, o `cout` imprime `true` como 1 e `false` como 0:

```
var1 = 1
var2 = 0
```

3.7 Operadores relacionais

Os operadores relacionais são utilizados para comparar variáveis de qualquer tipo. O resultado dessa comparação é um dado do tipo `bool`, que diz se a comparação é verdadeira ou falsa. Usualmente, este operadores são mais utilizados para gerar dados do tipo `bool`, do que os valores `true` e `false` diretamente, como fizemos no último exemplo. A Figura 3.6 mostra um exemplo de operador relacional onde perguntamos se 3 é maior que 7 e temos `false` (falso) como resposta.

A Tabela 3.7 apresenta um quadro geral com os operadores relacionais que temos em C++. Tanto o “maior que”quando o “menor que", tanto em C++ quanto em álgebra são representados

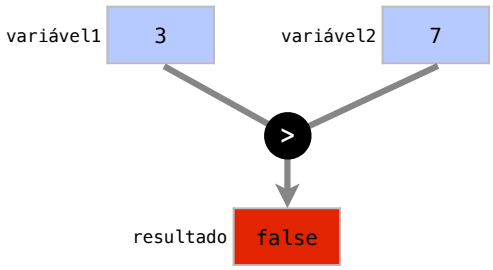


Figura 3.6: Exemplo de operador relacional. O valor de duas variáveis são comparados e temos um resultado lógico, que é verdadeiro ou falso.

Significado	C++	Em álgebra
Maior que	>	$>$
Menor que	<	$<$
Maior ou igual	>=	$\geq$
Menor ou igual	<=	$\leq$
Igual	==	$=$
Diferente	!=	$\neq$

Tabela 3.4: Operadores de Incremento de Decremento

pelos símbolos > e <. O "Maior ou igual" e o "menor ou igual"que são representados em matemática por $\geq$ ou $\leq$, em C++ são representados por >= e <=.

O "igual", em C++, é representado por dois sinais de igualdade juntos (==). Note como isto é diferente da igualdade em álgebra, que, naturalmente, usa apenas um sinal de igualdade ($=$). Note que não podemos utilizar apenas um sinal de igualdade para representar igualdade em C++ pois este seria o operador de atribuição em C++, que aprendemos na Seção 3.1. O "diferente", em C++, é representado por `!=`. Isto é o que, em álgebra, seria representado por um símbolo da desigualdade, que é um sinal de igualdade cortado ao meio ($\neq$).

É importante não confundir o operador relacional de **igualdade** (==) com o operador de **atribuição** (=). O primeiro é representado por dois sinais de igualdade e compara duas variáveis. O segundo é representado por apenas um sinal de igualdade e serve para atribuir valores a uma variável. Só se faz comparações com o operador de igualdade ==.

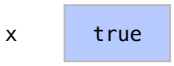
Neste exemplo, mostraremos como a comparação de duas variáveis nos retorna um valor do tipo `bool`:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6     bool x;
7     x = 2 < 9;
8     cout << x << endl;
9     cout << (2 < 3) << endl;
```

```
10      cout << (4 > 5) << endl;
11      cout << (1 <= 7) << endl;
12      cout << (2 >= 2) << endl;
13      cout << (4 == 4) << endl;
14      cout << (4 != 4) << endl;
15      return 0;
16  }
```

No exemplo, criamos uma variável `x` na linha 6 para guardar dados do tipo `bool`. Na linha 7, `x` recebe um `bool` que diz se dois é menor que nove (`2 < 9`). Como dois é de fato menor que nove, esta variável recebe `true`, ou verdadeiro. Em seguida, na linha 8, imprimimos `x`, onde seu valor `true` é impresso como 1:

```
1
```



O valor de `x` fica então como `true`. Como não precisamos armazenar o resultado de uma expressão neste exemplo, podemos utilizar e imprimir diretamente seu resultado como fazemos nesta próxima impressão, na linha 9. Neste comando, perguntamos se 2 é menor que 3. Como o resultado da comparação é verdadeiro, `true` é retornado e o valor `1` é impresso:

```
1
1
```

Em seguida, na linha 10, testamos se 4 é maior que 5, retornando falso.

```
1
1
0
```

Testamos se 1 é menor ou igual a 7, na linha 11, com resultado verdadeiro:

```
1
1
0
1
```

Logo na linha 12, se 2 é maior ou igual a 2, com resultado verdadeiro.

```
1
1
0
1
1
```

Na linha 13, se 4 é igual a 4, com resultado verdadeiro.

```
1
1
0
1
1
1
```

E finalmente na linha 14, se 4 é diferente de 4, com resultado falso:

```
1
1
0
1
1
1
0
```

3.8 Operadores lógicos

Como vimos, os operadores de relação são úteis para expressar condições simples como $x < 10$ (`x < y`), $y \geq 1000$ (`y >= 1000`), ou $x \neq y$ (`x != y`). Estes operadores geram novos dados lógicos a partir da comparação de duas variáveis quaisquer.

Já os operadores lógicos são os que geram novos `bool`s a partir de outros `bool`s, como apresentado na Figura 3.7. Eles são utilizados para testar múltiplas condições, que são representadas pelos `bool`s.

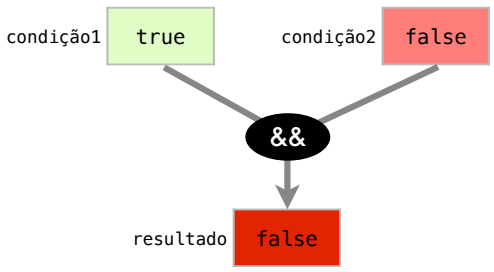


Figura 3.7: Exemplo de operador lógico. Estes operadores criam novos dados lógicos a partir de outros dados lógicos.

A Tabela 3.8 apresenta os operadores lógicos e o que os mesmos retornam. O "E" lógico, em C++ é representado por `&&`, retorna se dois `bool`s são verdadeiros. O "OU" lógico, em C++ é representado por duas barras verticais seguidas (`||`), retorna se pelo menos um de dois `bool`s é `true`. Já o "NÃO" lógico ou negação, em C++ é representado por um ponto de exclamação (`!`), retorna o inverso de um `bool`.

A Tabela 3.8 apresenta alguns exemplos de utilização de operadores lógicos, com seus significados. Na expressão entre parênteses temos duas expressões distintas, também entre parênteses. Na primeira expressão, perguntamos se $2 > 7$, o que é falso. Em seguida, perguntamos se $6 > 3$, o que é verdadeiro. O operador de "E" lógico `&&` pergunta se as duas expressões são verdadeiras, o que é falso em nossa primeira expressão. No segundo exemplo, perguntamos se $2 < 7$ e $6 > 3$, o que é verdadeiro.

Nome	Em C++	Retorna
E lógico	`&&`	se dois `bools` são `true`
OU lógico	`||`	se um dos dois `bools` é `true`
Não lógico ou Negação	`!`	o inverso de um `bool`

Tabela 3.5: Operadores Lógicos.

No terceiro exemplo, temos uma expressão que pergunta se $2 < 6$, o que é verdadeiro. A segunda expressão pergunta se $6 < 3$, o que é falso. O operador de "OU" lógico pergunta se pelo menos uma das duas expressões é verdadeira, o que torna toda a expressão verdadeira. Em seguida, no quarto exemplo, perguntamos se $7 < 2$ ou ao menos $6 < 3$. O resultado da expressão completa é falso.

No último exemplo, na expressão que utiliza o "NÃO" lógico, temos uma expressão que pergunta se $2 < 3$, o que é verdadeiro. Porém o operador de "NÃO" lógico recebe este verdadeiro e o transforma em falso.

Expressão	Resultado	Significado
`((2 > 7) && (6 > 3))`	`false`	$2 > 7$ e $6 > 3$?
`((2 < 7) && (6 > 3))`	`true`	$2 < 7$ e $6 > 3$?
`((2 < 7) || (6 < 3))`	`true`	$2 < 7$ ou $6 < 3$?
`((7 < 2) || (6 < 3))`	`false`	$7 < 2$ e $6 < 3$?
`!(2 < 3)`	`false`	$2 < 3$? Inverta a resposta.

Tabela 3.6: Exemplos de utilização dos Operadores Lógicos.

3.8.1 Resumo dos operadores lógicos

É comum resumir o funcionamento de operadores lógicos através de tabelas verdade. A Tabela 3.7 apresenta a Tabela Verdade do operador "E"lógico. Neste operador, sempre que uma das expressões utilizadas pelo operador for falsa, temos `false` como resultado. Mas se temos as expressões 1 e 2 como `true`, temos então `true` como resultado da aplicação do operador.

Expressão 1	Expressão 2	Expressão 1 && Expressão 2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Tabela 3.7: Operadores Lógicos.

Em álgebra, podemos dizer que $3 < x < 7$. Esta é uma notação matematicamente correta. Porém, em C++, devemos utilizar `(3 < x) && (3 < 7)`. Precisamos disto pois estas são duas operações distintas de comparação.

A Tabela 3.8 apresenta a Tabela Verdade do operador "OU" lógico `||`. Na tabela verdade do operador "OU" lógico, temos uma situação onde temos `false` como resposta sempre que as duas expressões sendo comparadas também sejam falsas. Para todos os outros casos, temos `true` como resultado, já que uma das expressões é sempre verdadeira.

Expressão 1	Expressão 2	Expressão 1 || Expressão 2
false	false	false
false	true	true
true	false	true
true	true	true

Tabela 3.8: Operadores Lógicos.

A Tabela 3.9 apresenta a Tabela Verdade do operador "NÃO" lógico `!`. A tabela verdade do operador de "NÃO" lógico é bem mais simples. Se temos um expressão verdadeira, logo "NÃO" lógico retornará `false` e vice-versa.

Expressão	!Expressão
true	false
false	true

Tabela 3.9: Operadores Lógicos.

3.8.2 Exemplo

Neste exemplo, usaremos operadores lógicos para unir expressões condicionais.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     bool x;
7     x = ((2 > 7) && (6 > 3));
8     cout << x << endl;
9     x = ((true) || (false));
10    cout << x << endl;
11    cout << ((2 < 7) && (6 > 3)) << endl;
12    cout << ((2 < 7) || (6 < 3)) << endl;
13    cout << ((7 < 2) || (6 < 3)) << endl;
14    x = !x;
15    cout << x << endl;
```

```
16      cout << !(2 < 3) << endl;
17      return 0;
18  }
```

Uma variável para dados lógicos é criada logo na linha 6, primeira linha dentro da funçao `main`. Unimos então duas expressões na linha 7 abaixo. Perguntamos se $2 > 7$, o que é falso. Depois perguntamos se $6 > 3$, o que é verdadeiro. O "E" lógico `&&` pergunta se as duas condições são verdadeiras, o que é falso. O resultado é atribuído à variável `x`, que é impressa como 0 na linha 8.

```
0
```

x false

Em seguida, na linha 9, temos um valor booleano que representa verdadeiro e outro valor que representa falso. O operador de "OU" lógico pergunta se ao menos uma das duas expressões é verdadeira, e o resultado é verdadeiro. Este resultado é guardado em `x` e é impresso como 1.

```
0
1
```

x true

No próximo comando, na linha 11, imprimimos se $2 < 7$ e $6 > 3$, o que é verdadeiro, e a impressão será 1.

```
0
1
1
```

Em seguida, na linha 12, se $2 < 7$ ou $6 < 3$, o que também é verdadeiro, imprimimos então 1

```
0
1
1
1
```

A próxima comparação, linha 13, pergunta se $7 < 2$ ou $6 < 3$. Como as duas expressões são falsas, o resultado é falso.

```
0
1
1
1
0
```

O comando que aparece na linha 14 utiliza o operação de Não-lógico para inverter o valor de `x`. O valor invertido de `x` é atribuído à própria variável `x`, que passa a ser falso. A linha 15 imprime o valor de `x`.

```
0
1
1
1
0
0
```

```
x    false
```

O último exemplo, na linha 16, é uma impressão de um resultado da expressão invertida diretamente no comando de impressão. Testamos se $2 < 4$, que é `true`, e invertemos o resultado para `false`. O resultado é impresso como `0` e encerramos o programa:

```
0
1
1
1
0
0
0
```

3.9 Exercícios

Exercício 3.1 Faça um programa que leia uma altura e uma peso de uma pessoa e imprima seu *Índice de Massa Corporal* (IMC), que é calculado com a fórmula:

$$IMC = \frac{peso}{altura^2}$$

Exemplo de saída:

```
Digite a altura da pessoa: 1.8
Digite o peso da pessoa: 68
O Índice de Massa Corporal desta pessoa é 20.9876
```

Note que, por padrão, não existe em C++ um operador "∧"pré-definido para exponenciação. Assim, $altura^2 = altura \times altura$. ■

Exercício 3.2 Faça um programa que leia uma temperatura em Celsius e imprime esta temperatura em Fahrenheit. Considere a fórmula:

$$F = \frac{9C}{5} + 32$$

Exemplo de saída:

```
Digite a temperatura em Celsius: 20.5
20.5 graus Celsius equivalem a 68.9 graus Fahrenheit
```

■

Exercício 3.3 Faça um programa que leia o raio r de um círculo e imprima sua área A e seu perímetro p.

$$A = \pi \times r^2 \qquad\qquad p = 2 \times \pi \times r$$

Exemplo de saída:

```
Digite o raio do círculo: 4.12
Um círculo de raio 4.12 possui área 53.29 e perímetro 25.87
```

Considere que $\pi = 3.14$. ■

Exercício 3.4 Faça um programa que leia dois valores a e b e imprima o resultado de $(b^3 + ab) - 2b + a \bmod b$. Em notação matemática, mod representa o resto da divisão de dois inteiros. Note que não se calcula resto da divisão de números reais.

Exemplo de saída:

```
Digite o valor de a: 5
Digite o valor de b: 7
f(x) = 369
```

■

4. Estruturas condicionais

A todo momento estamos tomando decisões que dependem de condições. Se o carro está sujo, então precisamos lavar. Se não choveu, vamos regar as plantas. Se estiver escuro acenderemos as luzes.

Códigos com estruturas condicionais também funcionam assim. Estas estruturas nos dão capacidade para representar estas situações.

Veja que nos códigos que criamos até o momento, apenas estruturas sequenciais foram utilizadas. Nestas estruturas, todos os comandos são executados de cima para baixo, como representado na Figura 4.1.

Com as estruturas condicionais, podemos dizer que alguns trechos do código nem sempre serão executados. Estes trechos estarão condicionados a alguma condição, como representado na Figura 4.2.

4.1 Condicional *se*

A instrução `if` (*se*, em inglês), é uma instrução de uma única seleção. Ela seleciona ou ignora um grupo de ações de acordo com uma condição. Esta condição pode fazer com que o programa ignore ou não certo bloco de comandos. Toda instrução de controle espera um dado do tipo `bool` indicando se a condição é `true`, verdadeira, ou `false`, falsa.

4.1.1 Sintaxe do *se*

Assim é a organização geral de uma instrução `if`:

```
1 if (condição){
2     comandos;
3     comandos;
4 }
```

A `condição` entre parênteses deve ser um dado do tipo `bool`. Este `bool` pode ser simplesmente `true` ou `false`, neste caso a condição seria sempre verdadeira ou falsa. Por isso, normalmente, vamos usar uma expressão que retorna um `bool`, como `x < 3`.

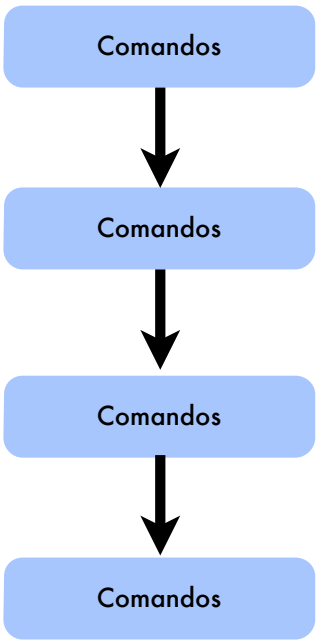


Figura 4.1: Representação de um programa com estrutura sequencial.

As chaves { e } indicam onde se inicia e termina o bloco de comandos que serão executados apenas se a condição for verdadeira. Entre as chaves, podem ser inseridos quantos comandos forem necessários.

Perceba que os comandos pertencentes ao `if` estão alinhados mais à direita do restante do código. Isto é fundamental para que o código seja legível quando temos muitos comandos. Esta organização se chama **indentação**.

4.1.2 Exemplo

Neste exemplo, vamos criar um bloco do código que dependerá de uma condição.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     double largura;
7     cout << "Digite␣a␣largura␣do␣quadrado:␣";
8     cin >> largura;
9     if (largura >= 0){
10      cout << "A␣área␣do␣quadrado␣é␣";
11      cout << largura * largura << endl;
12    }
13    return 0;
14 }
```

No início da função principal, criamos uma variável na linha 6 e, na linha 7, pedimos ao usuário que digite a largura de um quadrado. Este valor será utilizado para imprimir a área do quadrado. Porém, apenas valores não negativos são válidos como largura do quadrado. Caso o número digitado seja menor que 0, não queremos que o valor seja impresso, pois ele é inválido.

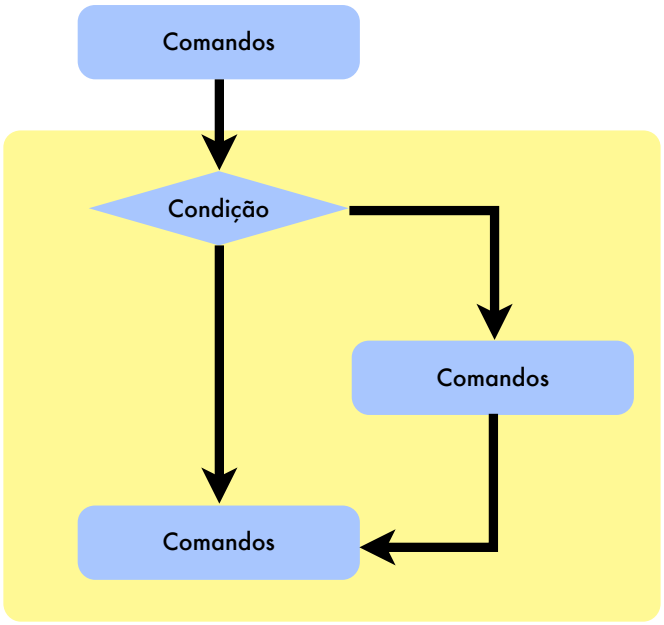


Figura 4.2: Representação de um programa com estrutura condicional.

Neste exemplo, supomos que o usuário digitou 5 como largura do quadrado.

```
Digite a largura do quadrado:  5
```

largura	5

Testamos, na linha 9, a condição `largura > 0`, que neste caso equivale a `5>0` e é verdadeira. Isto faz com que os comandos dentro do bloco sejam executados. Veja que como os comandos das linhas 10 e 11 pertencem ao `if`, eles estão indentados mais à direita. Como a condição era verdadeira, os comandos das linhas 10 e 11 são executados e a área do quadrado, 25, é impressa:

```
Digite a largura do quadrado:  5
A área do quadrado é 25
```

Vamos supor agora que o programa seja executado novamente e o usuário digitou `-3` como largura do quadrado.

```
Digite a largura do quadrado:  -3
```

largura	-3

A condição `-3 >= 0` é falsa. Assim, o bloco do `if`, composto pelas linhas 10 e 10, será ignorado e o programa será encerrado sem a mensagem:

```
Digite a largura do quadrado:  -3
```

4.2 *Se-senão*

Além de fazermos tarefas que dependem de uma condição, também fazemos usualmente muitas decisões entre duas opções. Suponha que temos uma festa hoje. Podemos nos perguntar se a festa será a noite. Se sim, podemos usar um terno. Senão, podemos usar uma bermuda.

A instrução `if-else` (*se-senão* em inglês), é uma instrução de seleção dupla. Ela seleciona entre dois blocos de comandos, como representado na Figura 4.3. Diferentes ações são tomadas em vez de apenas se ignorar um bloco. Uma condição decide qual bloco de comandos vamos utilizar.

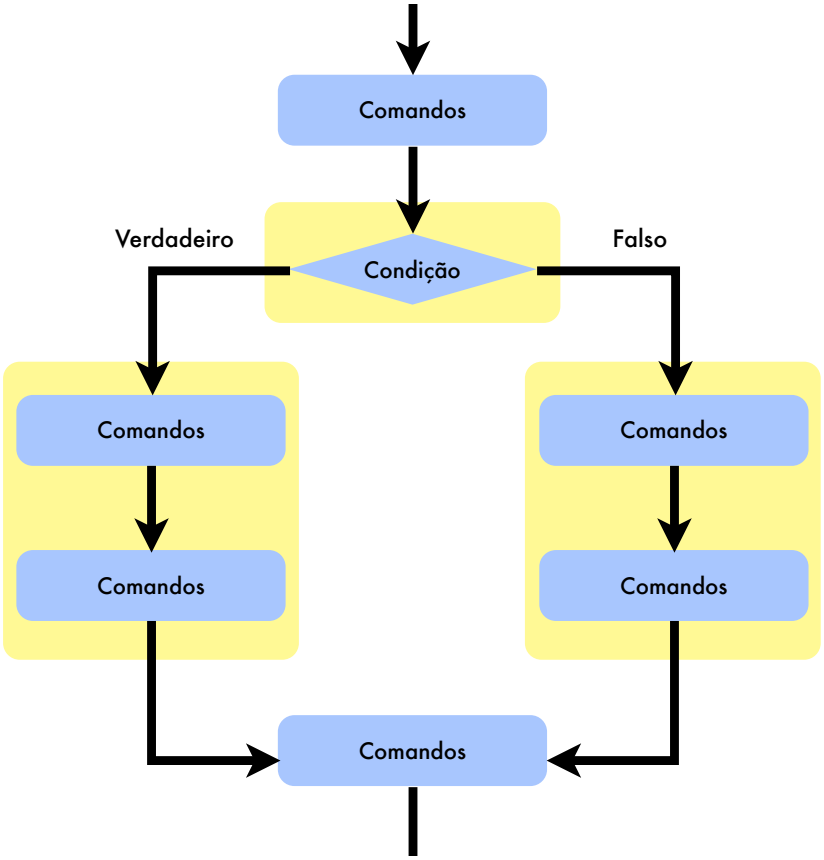


Figura 4.3: Representação de uma instrução de seleção dupla.

4.2.1 Se-senão sintaxe

Esta é a estrutura geral das instruções de *se-senão*.

```
1 if (condição){
2     comandos;
3     comandos;
4 } else {
5     comandos;
6     comandos;
7 }
```

A `condição` entre parênteses deve ser representada ainda por um valor do tipo `bool`. As chaves `{` e `}` agora indicam onde se iniciam e onde terminam dois blocos distintos de comandos. Um bloco, composto pelos comandos das linhas 2 e 3, é executado se a condição for `true` e o outro, composto pelos comandos das linhas 5 e 6, se a condição for `false`. Perceba também a **indentação**, fundamental para organização do código. Os blocos são todos indentados à direita dentro dos blocos.

4.2.2 Exemplo

Neste exemplo, vamos pedir um número ao usuário e imprimir uma mensagem informando se este número é par ou ímpar:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int numero;
7      cout << "Digite um número: ";
8      cin >> numero;
9      if (numero % 2 == 0){
10         cout << "O número é par" << endl;
11     } else {
12         cout << "O número é ímpar" << endl;
13     }
14     return 0;
15 }
```

Na linha 6, criamos uma variável `numero` e, na linha 7 do código, pedimos ao usuário que digite um número:

```
Digite um número:  7
```

A expressão condicional `numero % 2 == 0`, que utilizamos na linha 9, testa se a divisão de `numero` por 2 tem resto `0`. Repare que dividir um número por 2 e ter resto zero significa que este número é par. Como o resultado é `false`, o bloco de comandos do `else`, formado pelo comando da linha 12, é então executado. Temos neste bloco um comando que imprime na tela que o número digitado é ímpar.

```
Digite um número:  7
O número é ímpar
```

4.3 *Se-senão* aninhados

Existem ocasiões onde há várias condições possíveis que levam a diferentes escolhas. Imagine que você está de frente a um caixa eletrônico, se a sua opção for o “extrato", uma tela 1 será

exibida. Se escolher “saldo", uma tela 2 será exibida. Se a sua opção for “depósito", uma tela 3 será exibida, ou se a sua opção for “saque", então uma tela 4 será exibida.

A instrução `if-else` aninhada é utilizada como instrução de seleção múltipla para testar vários casos, como representado na Figura 4.4. Ela seleciona entre vários grupos diferentes de comandos. Existe agora não apenas uma condição mas um grupo de condições que escolhem entre grupos de blocos de comandos.

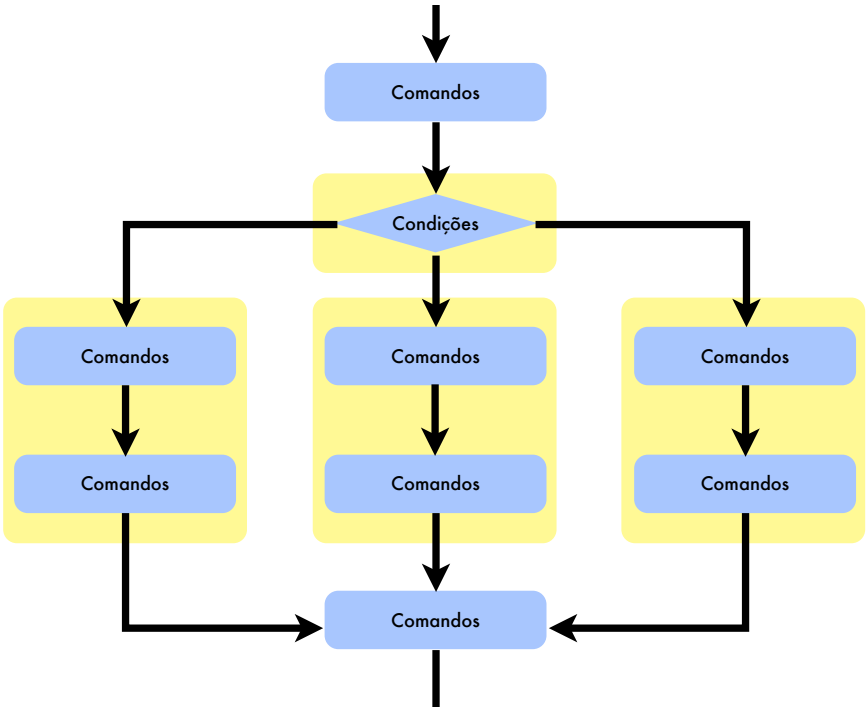


Figura 4.4: Representação de uma instrução de seleção múltipla.

4.3.1 Se-senão alinhados- sintaxe

Esta é a sintaxe deste tipo de estrutura condicional:

```
1  if (condição1){
2     comandos;
3     comandos;
4  } else if (condição2){
5     comandos;
6     comandos;
7  } else if (condição3){
8     comandos;
9     comandos;
10 } else {
11    comandos;
12    comandos;
13 }
```

Há agora várias condições possíveis que podem ser escolhidas. Sendo assim, podem ser criados quantos blocos forem necessários. Para cada condição a mais, temos um bloco a mais. A primeira `condição` que for `true` determina qual bloco será executado. Se nenhuma for `true`, então o bloco do `else` é executado. Perceba que a **indentação** se mantém.

4.3.2 Exemplo

Neste exemplo, vamos determinar qual o conceito de um aluno de acordo com sua nota.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     int nota;
7     cout << "Digite␣a␣nota␣do␣aluno:␣";
8     cin >> nota;
9     if (nota >= 90){
10       cout << "O␣conceito␣do␣aluno␣é␣A" << endl;
11    } else if (nota >= 80){
12       cout << "O␣conceito␣do␣aluno␣é␣B" << endl;
13    } else if (nota >= 70){
14       cout << "O␣conceito␣do␣aluno␣é␣C" << endl;
15    } else if (nota >= 60){
16       cout << "O␣conceito␣do␣aluno␣é␣D" << endl;
17    } else{
18       cout << "O␣conceito␣do␣aluno␣é␣E" << endl;
19    }
20    return 0;
21 }
```

Nas linhas 6 a 8 o usuário determina que a nota de um aluno. Neste exemplo, a nota do aluno é 74, o que quer dizer que o aluno deve ter conceito C.

```
Digite a nota do aluno:  74
```

Como a condição da linha 9, `nota >= 90`, é falsa, o primeiro bloco de comandos, na linha 10, é ignorado. Como `nota >= 80`, condição da linha 11, também é falso, o segundo bloco de comandos, da linha 12, é também ignorado.

Na linha 13 temos a condição `nota > 70`, que é verdadeira. O bloco de comandos correspondente, na linha 14, será executado. Este bloco de comandos imprime na tela que o conceito do aluno é C.

```
Digte a nota do aluno:  74
O conceito do aluno é C
```

Saímos então do bloco alinhado e vamos direto para a linha 20, onde a função é encerrada. Repare que a nota também era maior que 60, mas apenas o primeiro bloco `true` é considerado.

4.4 Outros operadores condicionais

Apesar das instruções apresentadas serem suficientes para representar qualquer estrutura condicional, existem outros operadores disponíveis que podem ser convenientes de se conhecer.

4.4.1 Operador ternário

Esta é a sintaxe do operador ternário:

```
condicao?expressao1:expressao2;
```

Temos uma condição seguida de duas expressões, separadas por dois pontos `:`. Se a condição for verdadeira, a primeira expressão será retornada. Se a condição for falsa, a segunda expressão será retornada.

O operador ternário pode ser substituído por um `if-else`, onde a primeira expressão fica no primeiro bloco e a segunda expressão no segundo bloco.

```
if (condicao){
    expressao1;
} else{
    expressao2;
}
```

No exemplo abaixo, utilizamos o operador ternário para damos à variável `max` o valor `1` ou `0`, de acordo com a condição `x>y`:

```
max=(x>y)?1:0;
```

O mesmo código pode ser representado com um `if-else` onde a condição é `x > y`. No primeiro bloco, atribuímos 1 a `max`. No segundo, atribuímos 0. Veja o exemplo:

```
if (x>y){
    max = 1;
} else{
    max = 0;
}
```

Assim, o operador ternário faz o mesmo que pode ser feito com um `if-else`. O operator ternário pode diminuir o número de linhas para se representar uma seleção dupla. Porém, é comum evitar o uso desta instrução pois ela dificulta a leitura do código.

4.4.2 Switch

O `switch` é um recurso para estruturas múltiplas de seleção. Sua é sintaxe é definida da seguinte maneira:

```
switch(variável){
    case constante1:
        comandos;
        break;
    case constante2:
        comandos;
        break;
    default:
        comandos;
}
```

Temos uma `variável` que será comparada com várias `constantes`. Caso a variável seja igual a uma destas constantes, um conjunto de comandos é executado.

O `switch` pode ser facilmente substituído por um `if-else` aninhado da seguinte forma:

```
1 if (variável == contante1){
2     comandos;
3 } else if (variável == constante2){
4     comandos;
5 } else {
6     comandos;
7 }
```

No `if-else`, podemos comparar a variável a cada uma das constantes do `switch`.

Neste exemplo seguinte, temos um `switch` para definir o que fazer de acordo com uma opção do usuário.

```
1  switch(opcao){
2      case 1:
3          cout << "Opção␣1␣escolhida" << endl;
4          break;
5      case 2:
6          cout << "Opção␣2␣escolhida" << endl;
7          break;
8      default:
9          cout << "Opção␣inválida␣escolhida" << endl;
10 }
```

A variável opcao é enviada para o `switch` e comparada para cada caso. O `switch` é um recurso muito utilizado para apresentar um menu de opções para o usuário.

Este `switch` pode ser substituído por um `if-else` onde comparamos opcao com cada possibilidade de resposta:

```
1 if (opcao == 1){
2     cout << "Opção␣1␣escolhida" << endl;
3 } else if (opcao == 2){
4     cout << "Opção␣2␣escolhida" << endl;
5 } else {
6     cout << "Opção␣inválida␣escolhida" << endl;
7 }
```

4.5 Exercícios

Exercício 4.1 Faça um programa que leia um número de usuário e uma senha numérica. O programa deve dizer se os valores digitados são válidos ou não. As senhas válidas são:

Número de usuário	Senha
982753	83928
263572	49582
275493	72648

Exemplo de saída:

```
Digite o número do usuário: 876342
```

```
Digite a senha do usuário: 20495
Usuário Inválido
```

■

Exercício 4.2 Faça um programa que leia 5 números e diga no final quantos números eram pares e quantos números eram ímpares.

Neste exercício, **apenas 2 variáveis poderão ser utilizadas**. Uma para receber o número e outra para contar quantas vezes o número recebido foi par ou ímpar.

Exemplo de saída:

```
Digite o primeiro número: 7
Digite o segundo número: 15
Digite o terceiro número: 13
Digite o quarto número: 4
Digite o quinto número: 2
Você digitou 2 números pares e 3 números ímpares
```

■

Exercício 4.3 Faça um programa que leia um valor para uma variável x e então calcule $f(x)$, sendo que:

$f(x) = x + 2x^2$ se $g(x) > 10$
$f(x) = 10$ se $g(x) \leq 10$
$g(x) = 5$ se $h(x) \leq 5$
$g(x) = h(x)$ se $h(x) > 5$
$h(x) = x^2 + 3x - 20$

Exemplo de saída:

```
Digite o valor de x: 15
f(x) = 465
```

■

Exercício 4.4 Faça um programa que leia a idade de um atleta e imprima sua categoria, sendo que:

Idade do Atleta	Categoria
5 a 7 anos	Infantil A
8 a 10 anos	Infantil B
11 a 13 anos	Juvenil A
14 a 17 anos	Juvenil B
18 a 25 anos	Sênior

Exemplo de saída:

```
Digite a idade do atleta: 17
Este é um atleta Juvenil B
```

Exercício 4.5 Faça um programa que leia:

- O salário de um empregado por hora trabalhada
- O número de horas trabalhadas
- O números de horas extras trabalhadas
- O número de dependentes

O programa retornará o salário final do empregado, sendo que:

- Há um benefício de R$128,00 por dependente
- Deve-se pagar imposto de renda de acordo com o salário
- Há um benefício de acordo com o salário após o imposto de renda

O imposto de renda é:

Salário	Imposto
Até R$1.434,59	0,00%
De R$1.434,60 até R$2.150,00	7,50%
De R$2.150,01 até R$2.866,70	15,00%
De R$2.866,71 até R$3.582,00	22,50%
Acima de R$3.582,01	27,50%

Os benefícios são:

Salário Líquido	Benefício
Até R$500,00	R$180,00
De R$500,00 até R$1.000,00	R$120
Acima de R$1.000,00	R$100,00

Exemplo de saída:

```
Digite o salário do empregado por hora trabalhada: 9.5
Digite o número de horas trabalhadas: 160
Digite o número de horas extras trabalhadas: 7
Digite o número de dependentes: 2
O salário final do empregado é R$1804.31
```

5. Operador de endereço e arranjos

5.1 Operador de endereço

O endereço de variáveis é conceito fundamental em C++. Cada variável declarada por nós possui fisicamente um endereço na memória do computador. A memória dos computadores possui várias posições, sendo que cada posições possui um endereço. Quando criamos uma variável, ela é guardada na memória. Cada valor precisa ficar em uma posição da memória. E cada uma destas posições tem um endereço. Isto está representado na Tabela 5.1.

Nomes das variáveis	Valores das Variáveis	Endereços das Posições
		24
		25
		26
		27
		28
`largura`	5	29
		30
		31
		32
		33
		34
		35

Tabela 5.1: Organização dos dados na memória se dá através de posições e endereços.

Em C++, o nome de uma variável, como `largura`, retorna diretamente seu valor, que no exemplo seria `5`. O operador de endereço, representado por um “e comercial"`&`, retorna o endereço de uma variável. Assim, `&largura` retorna o endereço `29` desta variável que, por sua

vez, contém fisicamente o valor 5.

Assim, cada variável possui um nome *identificador*, um *valor* e um *endereço*. O nome de uma variável já nos retorna seu valor. Já o operador de endereço (&) é usado para nos retornar o endereço desta variável.

Neste exemplo, criaremos uma variável e vamos imprimir seu valor e endereço:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int num;
7      cout << "Digite␣um␣número:␣";
8      cin >> num;
9      cout << "O␣número␣é␣" << num << endl;
10     cout << "O␣endereço␣do␣número␣é␣" << &num << endl;
11     return 0;
12 }
```

Criamos então, na linha 6, uma variável chamada num. A princípio, sabemos que a variável existe, mas não sabemos seu endereço na memória. Temos uma variável que foi criada e que ainda não tem um valor, mas não sabemos onde está alocada na memória:

Suponha então, que nas linhas 7 e 8 de código, o usuário dê o valor 65 a num.

```
Digite um número:  65
```

Como estamos acostumados, o cout da linha 9 imprime o valor do número num.

```
Digite um número:  65
O número é 65
```

Agora, porém, utilizamos o operador de endereço na linha 10 em num (&num) em vez de num, e o endereço de num na memória será exibido.

```
Digite um número:  65
O número é 65
O endereço de número é 0xbf8d456c
```

Na linha 11 finalizamos o programa. Repare que o endereço impresso da variável não é um número tão simples como os números de nosso exemplo anterior. Isso se deve à grande capacidade de números de endereços das memórias atuais.

5.2 Arranjos simples

Pelo o que vimos até agora, cada variável identifica um dado da memória. Porém, sabemos também que programas complexos trabalham com muitos dados, como milhares de estudantes, milhões de produtos ou centenas de números. Precisamos então de estruturas que representem vários dados de uma única vez.

Um arranjo é um endereço, onde há um grupo consecutivo de posições alocadas na memória. Todas estas posições guardam variáveis do mesmo tipo. Arranjos são então estruturas de dados muito úteis para se trabalhar com vários itens do mesmo tipo. Por exemplo, a Figura 5.2 representa um arranjo de números inteiros que tem nome identificador `x`.

Figura 5.1: Um arranjo de números inteiros.

O arranjo `x` apresentado nada mais é que **o endereço do primeiro elemento** deste arranjo de números inteiros. Para acessar elementos específicos precisamos de utilizar `x[i]`, onde este `i` é a posição no arranjo, como apresentado na Figura **??**. Formalmente, este `i` é chamado de índice do arranjo.

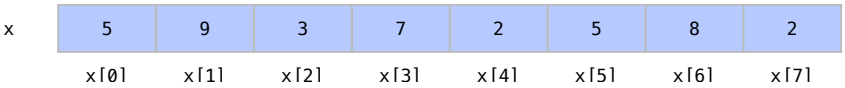


Figura 5.2: Os elementos de um arranjo podem ser acessados através de índices.

Repare que o primeiro índice `i` é `0` e o último índice `i` é `7`, ou seja, um a menos que o número de elementos no arranjo. Isso ocorre porque os índices representam quantas posições além do inicío do arranjo está o elemento.

Os índices são simplesmente números inteiros ou uma expressão do tipo inteiro. Suponha variáveis `a` e `b` de tipo inteiro:

Podemos acessar `x` na posição `a+b` com o comando `x[a+b]` para retornar o elemento que está em `x` na posição `5`.

Os endereços dos elementos do arranjo estão em sequência na memória. Assim, arranjo da Figura 5.2 ficaria guardado na memória como representado na Tabela 5.2.

Para criar um arranjo inicial em branco para dez elementos, utilizamos `int x[10]`.

```
1 int x[10];
```

Para criar um arranjo já com seus elementos utilize, então, `int x[]` e uma lista de elementos entre chaves, como no exemplo:

```
1 int x[] = {5, 9, 3, 7, 2, 5, 8, 2};
```

É fundamental lembrar que um arranjo ocupa posições contínuas na memória. Os índices *i* indicam quantas posições somamos ao endereço da variável para encontrar um elemento. Assim, o endereço de `x[3]`, ou `&x[3]`, nada mais é que o endereço de `x[0]` + 3, ou `&x[0]` + 3.

Nomes das variáveis	Valores das Variáveis	Endereços das Posições
		24
x[0]	5	25
x[1]	9	26
x[2]	3	27
x[3]	7	28
x[4]	2	29
x[5]	5	30
x[6]	8	31
x[7]	2	32
		33
		34
		35

Tabela 5.2: Um arranjo armazenado na memória. Os elementos do arranjo se encontram em posições sequenciais na memória.

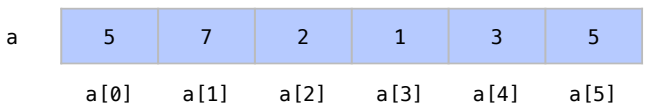
É importante entender que tentar acessar uma posição inexistente no arranjo é um erro grave. Não sabemos o que há na posição e quais as consequências de tal acesso.

Suponha que temos o arranjo apresentado na Figura 5.2 e vamos tentar acessar `x[9]`. Neste caso, teríamos um erro, pois de acordo com a Tabela 5.2, tentaríamos acessar a posição de memória no endereço 35. Não sabemos o que existe na posição 35 e nem mesmo que tipo de dado se encontra nesta posição.

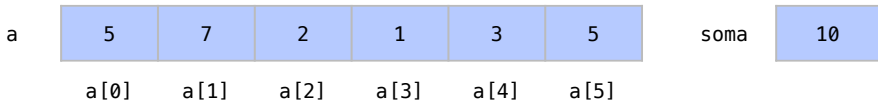
Neste próximo exemplo, criamos um arranjo com 6 elementos e somamos elementos de posições específicas:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int a[] = {5,7,2,1,3,5};
7      int soma;
8      soma = a[0] + a[2] + a[4];
9      cout << "a[0] + a[2] + a[4] -> " << soma << endl;
10     cout << "Endereço do arranjo -> " << a << endl;
11     return 0;
12 }
```

Na linha 6, criamos um arranjo `a` com 6 elementos, como o apresentado abaixo:



Em seguida, na linha 7, é criada uma variável `soma` e, na linha 8, a mesma recebe `a[0] + a[2] + a[4]`, que equivale a `5 + 2 + 3`, números das posições 0, 2 e 4 do arranjo.



Quando imprimimos `soma`, na linha 9, temos o resultado da operação.

```
a[0] + a[2] + a[4] -> 10
```

Na linha 10, quando imprimimos o valor de `a` sem especificar uma posição, temos o endereço na memória onde começa o arranjo `a`.

```
a[0] + a[2] + a[4] -> 10
Endereço do arranjo -> 0x7fff5f6bdb70
```

Lembre-se assim que um arranjo **é** um endereço onde começa uma sequência de elementos na memória.

5.3 Cadeias de caracteres

Como vimos, o tipo fundamental de dados do tipo `char` é utilizado para representar letras e caracteres. Sua relação com arranjos é fundamental para formar cadeias de caracteres. Estas cadeias representam palavras e frases, que não são um tipo fundamental de dado.

As cadeias de caracteres são arranjos simples de `char`s e que terminam com o caractere especial `\0`, que representa o fim da cadeia. Por exemplo, a Figura 5.3 apresenta um arranjo de `char`s chamado `b` e que carrega a palavra *abacate*.

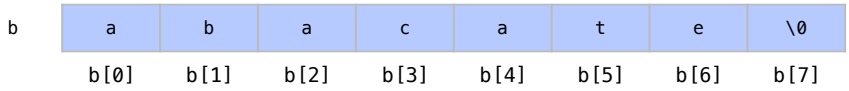


Figura 5.3: Cadeias de Caracteres são arranjos com dados do tipo `char`.

O caractere `\0` mostra que ali termina a palavra. São necessários assim 8 elementos para uma palavra de 7 letras. Como aprendemos na seção anterior, este arranjo poderia ter sido criado diretamente com o seguinte comando:

```
1 char b[] = {'a', 'b', 'a', 'c', 'a', 't', 'e', '\0'};
```

onde o arranjo `char b` recebe uma lista de `char`s, que são separados por vírgula (`,`) e aspas simples (`'`).

Porém, existe um atalho equivalente, que é atribuir a `b` todos os `char`s entre aspas duplas (`"`).

```
1 char b[] = "abacate";
```

Podemos atribuir dados às cadeias com o comando `cin`, como em `cin >> b`. Isto, porém, é muito perigoso pois a pessoa pode digitar mais que 7 letras. Neste caso, haverá uma tentativa de inserir elementos além do arranjo.

Por exemplo, em um comando `cin` para atribuir um valor a `b`, se a pessoa digita "laranjada", por exemplo, ocorre um erro por falta de espaço. Isto é apresentado na Figura 5.4, onde se

tenta atribuir a palavra `laranjada` ao arranjo já alocado da Figura 5.3. Como isto retornaria um erro, é preciso garantir que isto não ocorra, com arranjos que tenham espaço suficiente para as palavras possíveis de acordo com a aplicação.

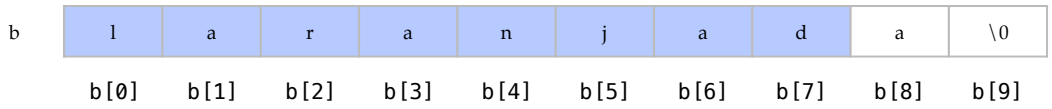


Figura 5.4: É necessário se precaver em relação ao número de posições disponíveis para palavras no arranjo.

Já o objeto `cout`, recebendo `b`, imprime a palavra `abacate` corretamente. Assim como `cin`, `cout` não se preocupa com o tamanho do arranjo. São impressos todos os caracteres até que se encontre o caractere especial `\0`, como no exemplo abaixo:

```
1 cout << b;
```

```
abacate
```

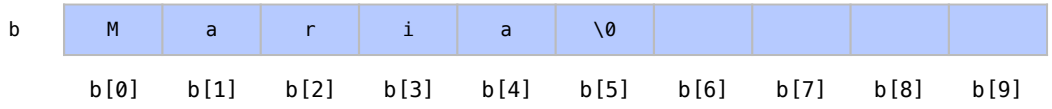
Como em qualquer arranjo `b[i]` retorna o `char` de `b` na posição `i` após o endereço inicial do arranjo.

Neste exemplo, criaremos uma cadeia de caracteres e imprimimos um de seus elementos:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      char nome[10];
7      cout << "Digite␣seu␣nome:␣";
8      cin >> nome;
9      cout << "Terceira␣letra␣de␣seu␣nome␣é␣";
10     cout << nome[2] << endl;
11     return 0;
12 }
```

Para isto, criamos um arranjo de `char`s com 10 elementos, na linha 6. Nas linhas 7 e 8, o usuário insere um nome nesse arranjo.

```
Digite seu nome:  Maria
```



Depois, nas linhas 9 e 10, se imprime o caractere que está na posição 2 do arranjo, ou seja, a terceira letra do nome:

```
Digite seu nome:  Maria
A terceira letra do seu nome é r
```

O programa então é encerrado. Mas repare que este código é propenso a erros. Na linha 8, o `nome` digitado poderia ter mais que 9 letras, e não haveria espaço no arranjo. Ou o `nome` poderia, na linha 10, ter menos que 3 letras e, no caso, estaríamos tentando acessar posições inexistentes, o que configura um erro grave novamente.

5.4 Classe string

Pode ser bastante trabalhoso lidar com todos os erros possíveis na representação de cadeias de caracteres com arranjos. Por isso o C++ nos oferece a classe `string`, para aliviar este problema. Esta é uma classe que define um tipo de dado *não fundamental* e por isto não está apresentado na Tabela 1.1 onde vimos os tipos dados. Os dados não fundamentais são compostos com os dados fundamentais da linguagem que já conhecemos. Neste caso, uma `string` organiza automaticamente as cadeias dentro de arranjos para o programador.

Vamos supor que `b` agora é uma `string` e não um arranjo simples de `char`. Esta string pode ser criada com o seguinte comando:

```
1 string b("abacate");
```

Se tentarmos executar um comando para que `b` receba a cadeia de caracteres ‘‘`laranjada`’’, mais espaço será alocado para atender à demanda e não teremos mais erros. Por não ser um tipo de dado fundamental, é necessário incluir em nosso código o cabeçalho `string` com o seguinte comando:

```
1 #include <string>
```

Neste próximo exemplo, o usuário irá inserir novamente um nome que será salvo em uma cadeia de caracteres:

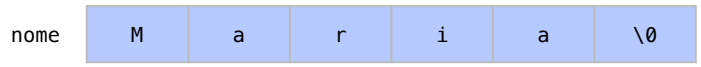
```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main(){
7      string nome;
8      cout << "Digite␣seu␣nome:␣";
9      cin >> nome;
10     cout << "Terceira␣letra␣de␣seu␣nome␣é␣";
11     cout << nome[2] << endl;
12     return 0;
13 }
```

Veja que desta vez, na linha 2, será necessário incluir o cabeçalho `string` para nos dar o recurso de utilizar `strings`, pois `string` não é um tipo fundamental de dados.

É criada, na linha 7, então, uma `string nome` que terá a capacidade que for demandada. Inicialmente, a `string` se encontra vazia.

Imprimimos umas mensagem na linha 8 e, quando o usuário faz sua entrada, na linha 9, é alocado um arranjo com o espaço para o nome digitado:

```
Digite seu nome:  Maria
```



Na linhas 10 e 11, retornamos a terceira letra do nome digitado, ou a segunda posição do arranjo:

```
Digite seu nome:  Maria
Terceira letra do seu nome é r
```

5.5 Arranjos multidimensionais

Os arranjos que vimos até agora são muito úteis para representar listas de elementos. Já os arranjos multidimensionais de duas dimensões costumam ser úteis para representar tabelas de valores organizados em linhas e colunas, como apresentado na Figura 5.5.

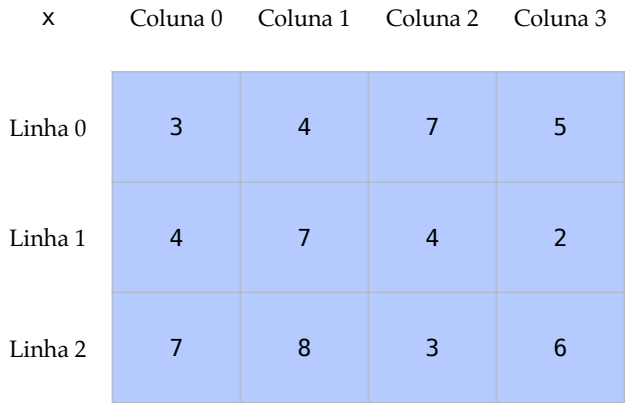


Figura 5.5: Exemplo de arranjo de duas dimensões. Estes arranjos são úteis para representar tabelas organizadas em linhas e colunas.

Sendo assim, precisamos de dois índices para encontrar um item dentro deste tipo de arranjo, como representado na Figura 5.6. Estes são arranjos bidimensionais, também conhecidos como 2D. Para acessar uma posição específica dentro desse arranjo `x` da Figura precisamos de dois índices entre colchetes `x[i][j]`. Assim como em arranjos simples, os índices começam em `0`. Por convenção, o primeiro índice indica a linha e o segundo a coluna. Se colocamos `0` no primeiro índice, acessamos elementos da primeira linha. Se colocamos `0` no segundo índice, acessamos elementos da primeira coluna.

Este exemplo de arranjo `x` pode ser criado com `x[3][4]`, onde 3 representa o número de linhas e 4 representa o número de colunas. Este arranjo pode receber diretamente seus elementos através de três conjuntos de números entre chaves, como apresentado abaixo:

```
1 int x[3][4] = {{3,4,7,5},{4,7,4,2},{7,8,3,6}};
```

x	Coluna 0	Coluna 1	Coluna 2	Coluna 3
Linha 0	`x[0][0]`	`x[0][1]`	`x[0][2]`	`x[0][3]`
Linha 1	`x[1][0]`	`x[1][1]`	`x[1][2]`	`x[1][3]`
Linha 2	`x[2][0]`	`x[2][1]`	`x[2][2]`	`x[2][3]`

Figura 5.6: Posições em um arranjo de duas dimensões. Assim como em arranjos simples, as posições de uma dimensão começam em 0.

Cada conjunto de números entre chaves representa uma linha do arranjo. Isto acontece porque um arranjo multidimensional é, fisicamente na verdade, um arranjo de arranjos.

Assim como um arranjo simples, um arranjo multidimensional também ocupa posições contínuas na memória. O arranjo da Figura 5.5, por exemplo, seria armazenado na memória como apresentado na Tabela 5.3.

Identificadores	**Valores das Variáveis**	**Endereços das Posições**
`x[0][0]`	3	24
`x[0][1]`	4	25
`x[0][2]`	7	26
`x[0][3]`	5	27
`x[1][0]`	4	28
`x[1][1]`	7	29
`x[1][2]`	4	30
`x[1][3]`	2	31
`x[2][0]`	7	32
`x[2][1]`	8	33
`x[2][2]`	3	34
`x[2][3]`	6	35

Tabela 5.3: Um arranjo multidimensional armazenado na memória. Assim como em arranjos simples, os elementos do arranjo multidimensional se encontram em posições sequenciais na memória.

Assim, cada valor do arranjo possui um **identificador** (representado aqui pelo identificador do arranjo e sua posição em colchetes), um **valor** e uma **posição**.

Todos os valores estão em sequência na memória justamente porque eles nada mais são do que arranjos de arranjos. No arranjo multidimensional `x`, `x[0]` é um arranjo de 4 elementos que se inicia na posição 24, `x[1]` inicia na posição 28 e `x[2]` na posição 32.

A mesma estrutura sintática de arranjos de arranjos, ou arranjos 2-D, pode ser utilizada para criar arranjos com qualquer número de dimensões, seja 3-D ou 4-D, por exemplo, como nos exemplos:

```
1 int x[3][2][5]; // Arranjo 3-D para números inteiros
```

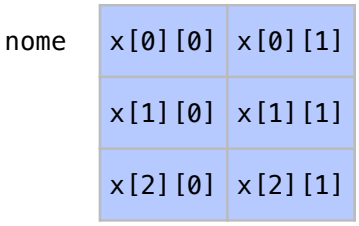
```
2 double y[4][8][15][26]; // Arranjo 4-D para números reais
```

Assim como em todos os arranjos, é importante lembrar que o primeiro elemento de uma dimensão sempre tem o índice 0 e todos os elementos estarão sempre contínuos na memória.

Neste exemplo, vamos trabalhar com arranjos de 2 dimensões:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     double x[3][2];
7     double resultado;
8     cout << "Digite a nota do aluno 0 na prova 0: ";
9     cin >> x[0][0];
10    cout << "Digite a nota do aluno 0 na prova 1: ";
11    cin >> x[0][1];
12    cout << "Digite a nota do aluno 1 na prova 0: ";
13    cin >> x[1][0];
14    cout << "Digite a nota do aluno 1 na prova 1: ";
15    cin >> x[1][1];
16    cout << "Digite a nota do aluno 2 na prova 0: ";
17    cin >> x[2][0];
18    cout << "Digite a nota do aluno 2 na prova 1: ";
19    cin >> x[2][1];
20    resultado = (x[0][0] + x[1][0] + x[2][0])/3;
21    cout << "Nota média na prova 0 = " << resultado << endl;
22    resultado = (x[1][0] + x[1][1])/2;
23    cout << "Nota final do aluno 1 = " << resultado << endl;
24    return 0;
25 }
```

Criamos no início do código, na linha 6, um arranjo de duas dimensões com três linhas e duas colunas. Assim os índices vão de `x[0][0]` até `x[2][1]` como apresentado:



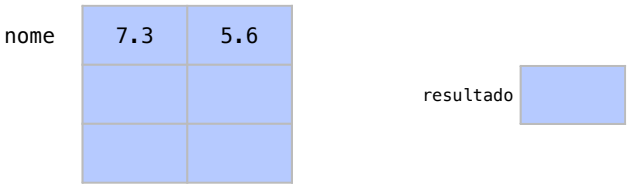
Neste exemplo, faremos com que cada linha da matriz represente um aluno e cada coluna uma prova feita por ele. Além do arranjo 2D, que encontra-se ainda vazio, criamos em seguida, na linha 7, uma variável que guardará os resultados dos cálculos.

Na linha 8, Pedimos ao usuário para que digite um nota. Na linha 9, guardamos o resultado em `x[0][0]`.

```
Digite a nota do aluno 0 na prova 0:  7.3
```

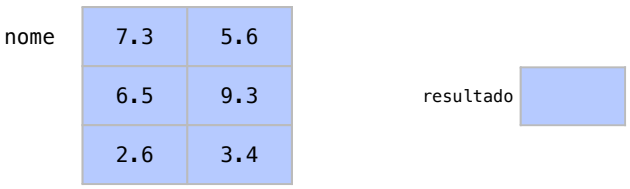
Esta posição `x[0][0]` se refere à primeira linha e a primeira coluna. Nos referimos então, ao aluno 0 na prova 0 para o usuário. Apenas para observação, poderíamos omitir esta informação do usuário e mostrar esta combinação como aluno 1 e prova 1. De modo análogo, nas linhas 10 e 11, damos valores ao elemento `x[0][1]`:

```
Digite a nota do aluno 0 na prova 0:  7.3
Digite a nota do aluno 0 na prova 1:  5.6
```



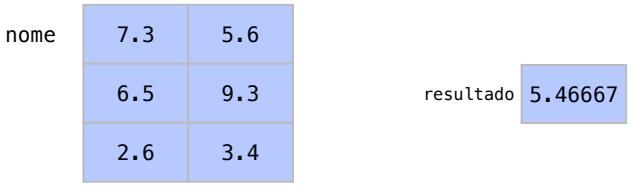
Nas linhas 12 a 19, fazemos o mesmo para todas as combinações de aluno e prova.

```
Digite a nota do aluno 0 na prova 0:  7.3
Digite a nota do aluno 0 na prova 1:  5.6
Digite a nota do aluno 1 na prova 0:  6.5
Digite a nota do aluno 1 na prova 1:  9.3
Digite a nota do aluno 2 na prova 0:  2.6
Digite a nota do aluno 2 na prova 1:  3.4
```



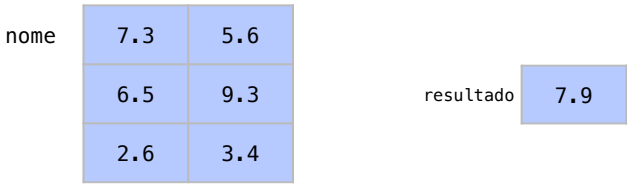
Somamos agora, na linha 20, todos os elementos da coluna 0 (`x[0][0] + x[1][0] + x[2][0]`) e dividimos por 3. O resultado disto, atribuído a `resultado` e impresso na linha 21, será a nota média dos alunos na primeira prova:

```
Digite a nota do aluno 0 na prova 0:  7.3
Digite a nota do aluno 0 na prova 1:  5.6
Digite a nota do aluno 1 na prova 0:  6.5
Digite a nota do aluno 1 na prova 1:  9.3
Digite a nota do aluno 2 na prova 0:  2.6
Digite a nota do aluno 2 na prova 1:  3.4
Nota media na prova 0 = 5.46667
```



De modo análogo, para saber qual foi a nota final do segundo aluno, somamos todos os elementos da linha 1 (`x[1][0] + x[1][1]`) e dividimos por 2, que é o número de colunas, ou de provas.

```
Digite a nota do aluno 0 na prova 0:  7.3
Digite a nota do aluno 0 na prova 1:  5.6
Digite a nota do aluno 1 na prova 0:  6.5
Digite a nota do aluno 1 na prova 1:  9.3
Digite a nota do aluno 2 na prova 0:  2.6
Digite a nota do aluno 2 na prova 1:  3.4
Nota media na prova 0 = 5.46667
Nota final do aluno 1 = 7.9
```



5.6 Exercícios

Exercício 5.1 Suponha um arranjo a com 5 elementos e outro arranjo b com 5 elementos. Faça um programa que calcule o produto escalar de a por b (Isto é, o primeiro elemento de a multiplicado pelo primeiro elemento de b mais o segundo elemento de a multiplicado pelo segundo de b mais ...).

```
Digite o primeiro elemento do arranjo a: 6
Digite o segundo elemento do arranjo a: 3
Digite o terceiro elemento do arranjo a: 4
Digite o quarto elemento do arranjo a: 8
Digite o quinto elemento do arranjo a: 3

Digite o primeiro elemento do arranjo b: 2
```

```
Digite o segundo elemento do arranjo b: 6
Digite o terceiro elemento do arranjo b: 7
Digite o quarto elemento do arranjo b: 2
Digite o quinto elemento do arranjo b: 5

O produto escalar de a[] por b[] é 89
```

■

Exercício 5.2 Fazer um algoritmo que:

1. Crie um arranjo de 5 elementos e o preencha de números
2. Procure a posição do menor elemento deste arranjo
3. Troque o menor elemento com elemento da primeira posição
4. Imprima os elementos do arranjo

```
Digite o primeiro elemento do arranjo: 6
Digite o segundo elemento do arranjo: 3
Digite o terceiro elemento do arranjo: 4
Digite o quarto elemento do arranjo: 2
Digite o quinto elemento do arranjo: 8

O menor elemento deste arranjo está na posição a[3]

Novo arranjo: 2 3 4 6 8
```

■

6. Estruturas de repetição

Um motivo pelo qual os computadores nos auxiliam muito em tarefas é porque são capazes de repetir tarefas milhões ou bilhões de vezes sem cometer erros. Este tipo de repetição poderia demorar séculos para nós, porém, são simples para essas máquinas.

Imagine quanto tempo levaria enviar uma carta para um milhão de pessoas. Compare agora com o tempo gasto para se enviar um e-mail para um milhão de pessoas.

As estruturas de repetição nos permitem realizar instruções repetidamente enquanto uma condição for verdadeira, como apresentado na Figura 6.1. Por retornar a um comando anterior no código, estruturas de repetição também são chamadas de instruções de **loop**, ou **laço**. Usualmente, os laços são repetidos um certo número de vezes ou até que algo aconteça.

6.1 Laços com contadores

O caso mais simples de repetição é aquele no qual queremos repetir exatamente a mesma tarefa um certo número de vezes.

Quando queremos que um bloco de comandos se repita um certo número de vezes, usamos uma **variável contadora** para guardar quantas vezes o bloco já foi executado. A variável contadora deve ser um número inteiro e pode ter qualquer nome identificador. Usualmente, ela leva tem o nome identificador de `i`, o que facilita o seu uso repetido. Cada repetição do laço é chamada de **iteração**.

6.1.1 Controle de fluxo-For

A instrução `for` é utilizada para repetições controladas por uma variável contadora. Ela tem a sintaxe apresentada.

```
1 for (inicialização; condição; incremento){
2     comandos;
3 }
```

A `inicialização` deste comando é utilizada para inicializar a variável contadora. A `condição` deste comando é utilizada para que a repetição continue acontecendo. Ela testa,

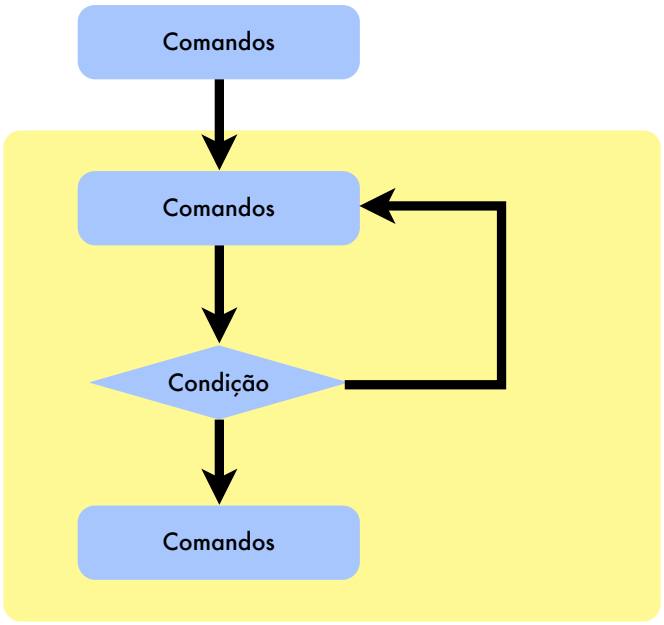


Figura 6.1: Representação de um programa com estrutura de repetição.

usualmente, se o contador atingiu um valor final. O `incremento` é a onde dizemos como será incrementada a variável contadora após cada repetição. Normalmente, somamos 1 à variável contadora.

Podemos colocar quantos `comandos` quisermos dentro do bloco de comandos de `for`. Inclusive, podemos colocar ali outras estruturas condicionais e até mesmo outras estruturas de repetição.

6.1.2 Repetição controlada por contador

Um **contador** utilizado em uma repetição precisa de: um nome **identificador**, um **valor inicial**, um **valor final**, e um **incremento** a cada passo.

Veja um exemplo para um bloco de comandos que será repetido 10 vezes.

```
1 for (i = 0; i < 10; i++){
2     comandos;
3 }
```

No exemplo, a `inicialização` é representada por uma variável `i` recebendo `0` (`i = 0`). A *condição* de repetição é que `i` seja menor que `10`. E a cada repetição feita devemos incrementar `i` em `1` (`i++`). Essa é a estrutura básica pra repetir um bloco de comandos 10 vezes.

A instrução diz que o contador `i` começa em `0` e repetimos os comandos enquanto ele for menor do que `10`. A cada repetição ele é incrementado em `1`.

Neste exemplo, temos um programa que imprime uma mesma mensagem 10 vezes:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6     int i; // contador
7     for (i = 0; i < 10; i++){
```

```
8           cout << "Olá mundo!" << endl;
9       }
10      return 0;
11  }
```

Para imprimir a mensagem seguidas vezes, criamos uma variável contadora `i`, na linha 6. Quando entramos no laço, na linha 7, a variável contadora é inicializada com `0`. A condição `i < 10` é testada e o bloco de comandos será executado porque `i < 10` é `true`. Todo o bloco da linha 8 é, então, executado.

```
Olá mundo!
```

Ao fim da execução do bloco, o comando de incremento `i++` é executado e `i` passa a ter valor `1`. A condição de parada `i < 10` é testada e o comando será executado pois `1`, valor de `i` após incremento, ainda é menor que `10`.

```
Olá mundo!
Olá mundo!
```

Veja que quando `i` é 2, `i < 10` é também verdadeiro. E assim em diante.

```
Olá mundo!
Olá mundo!
Olá mundo!
```

Após as 10 iterações deste laço, teremos o seguinte estado, com `i` igual a 9 e 10 mensagens impressas:

```
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
```

Ao fim da última repetição do laço, `i` chega a 10 através do incremento `i++` e então, a condição `i < 10` é falsa. Assim, o bloco não será mais executado e este é o fim deste programa.

6.2 Repetições determinadas pelo usuário

Compare agora estes dois programas. Apesar do segundo não usar estruturas de repetição, os dois imprimem o mesmo resultado na tela.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int i; // contador
7      for (i = 0; i < 10; i++){
8          cout << "Olá mundo!" << endl;
9      }
10     return 0;
11 }
```

```
1  include <iostream>
2
3  using namespace std;
4
5  int main(){
6      cout << "Olá mundo!" << endl;
7      cout << "Olá mundo!" << endl;
8      cout << "Olá mundo!" << endl;
9      cout << "Olá mundo!" << endl;
10     cout << "Olá mundo!" << endl;
11     cout << "Olá mundo!" << endl;
12     cout << "Olá mundo!" << endl;
13     cout << "Olá mundo!" << endl;
14     cout << "Olá mundo!" << endl;
15     cout << "Olá mundo!" << endl;
16     return 0;
17 }
```

No primeiro exemplo, podemos ver que o comando `for` nos permitiu não precisar reescrever o comando várias vezes. Um programa sequencial que simula o mesmo comportamento pode ser muito extenso. Imagine um caso onde fazemos milhões de repetições.

Na maior parte dos problemas, nem é mesmo possível fazer uma versão sequencial equivalente. Um exemplo disso é quando não sabemos quantas vezes o comando será repetido. Isso ocorre, por exemplo, em problemas onde o usuário decide quantas vezes um bloco de comandos será repetido.

Neste próximo exemplo, temos um código onde o usuário determina o número de repetições do bloco:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int i;
7      int n;
8      cout << "Quantas␣vezes␣você␣quer␣a␣mensagem?␣";
9      cin >> n;
10     for (i = 0; i < n; i++){
11         cout << "Olá␣mundo!" << endl;
12     }
13     return 0;
14 }
```

Neste exemplo, após criada a variável contadora, na linha 6, é criada uma variável para o número de repetições, na linha 7. Esta mesma variável, na linha 9, recebe do usuário a quantidade de vezes que ele deseja ver a mensagem. No caso, ele escolhe ver a mensagem 6 vezes:

```
Quantas vezes você quer a mensagem?  6
```

O laço que se inicia na linha 10 repete o bloco de comandos e incrementa o contador enquanto a condição for verdadeira. A condição se torna falsa quando a mensagem é impressa n vezes e então encerramos o programa.

```
Quantas vezes você quer a mensagem?  6
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
Olá mundo!
```

6.3 Repetição de processos similares

Nos últimos exemplos repetimos os mesmos comandos várias vezes. Mas nem sempre é exatamente isto o que acontece. É comum, também, repetir tarefas que sejam similares de acordo com a repetição do laço. Por exemplo, uma companhia pode enviar uma mensagem desejando um feliz ano novo para todos os seus clientes. Apesar de ser uma tarefa de repetição, cada mensagem possui algumas diferenças, como o nome do cliente.

Usualmente, uma mudança nas próprias condições das variáveis pode alterar como a repetição se comporta. Um exemplo disso, é quando usamos o contador dentro do próprio bloco de repetição.

Neste exemplo, criaremos uma estrutura de repetição onde cada iteração do bloco terá um resultado diferente:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      cout << "Digite␣o␣número␣final:␣";
9      cin >> n;
10     for (i = 0; i <= n; i++){
11         cout << i << "␣";
12     }
13     return 0;
14 }
```

No exemplo, a variável contadora será utilizada dentro do próprio bloco do `for` para imprimir os números de 0 até `n`.

Assim que é criada a variável contadora, na linha 6, cria-se também a variável com o número final da sequência impressa, na linha 7. O usuário, então, nas linhas 8 e 9, escolhe um número final para a sequência. No caso, ele escolhe 5:

```
Digite o número final:  5
```

Na linha 10, o contador `i` é inicializado com `0` e a condição `i <= n` é testada. Repare que neste exemplo testamos se `i <= n` e não `i < n`. Isso ocorre porque queremos também imprimir o número `n`. Na linha 11, o contador `i` é impresso e seguido de um espaço.

```
Digite o número final:  5
0
```

Logo após a impressão, `i` é incrementado (`i++`), a condição é testada novamente e imprimimos o valor de `i`.

```
Digite o número final:  5
0 1
```

Neste exemplo, a condição para continuar o laço é que `i <= 10`, pois queremos imprimir o número `n` também. O contador `i` é incrementado e ainda é menor ou igual a `n`.

```
Digite o número final:  5
0 1 2
```

Ao continuar executando o laço, todos os números menores ou iguais a 5 serão impressos até que `i` seja 6.

```
Digite o número final:  5
0 1 3 4 5
```

Vale lembrar que o programa pode aceitar qualquer número `n` como entrada do usuário. Caso o número digitado fosse 56, teríamos impresso todos os valores entre 0 e 56.

```
Digite o número final:  56
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56
```

6.4 Alterando variáveis externas

Algumas vezes, usamos estruturas de repetição também para alterar o valor de variáveis externas ao bloco de repetição. Um exemplo disso é quando fazemos o somatório de vários valores. A estrutura de repetição faz o cálculo do somatório com uma variável auxiliar, que será impressa após todo o processo.

Neste exemplo, uma variável `soma` será usada para calcular o somatório de todos os números de 1 até `n`:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      int soma;
9      cout << "Digite␣o␣número␣final:␣";
10     cin >> n;
11     soma = 0;
```

```
12      for (i = 1; i <= n; i++){
13          soma += i;
14      }
15      cout << "Somatório = " << soma << endl;
16      return 0;
17  }
```

Como fizemos nos últimos exemplos, após criada a variável contadora, na linha 6, é criada também a variável para o número final da série, na linha 7. Na linha 8, uma outra variável é criada para guardar o somatório dos números. Neste exemplo, nas linhas 8 e 9, a variável `n` para o número final da série recebe do usuário o número 4, enquanto `soma` é inicializada com `0` na linha 11:

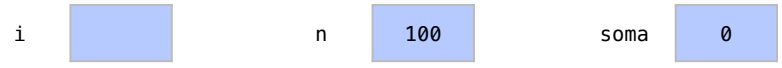
```
Digite o número final:  4
```

O nosso somatório começa em 0 e será incrementado durante o laço. O contador `i` se inicia agora, na linha 12, em `1`, pois não precisamos somar `0`. O laço se repetirá enquanto `i` for menor ou igual ao número digitado pelo usuário. Veja que, na linha 13, acumulamos cada valor de `i` à `soma`. Assim, o valor final de `soma` será igual a `1 + 2 + 3 + 4`. O resultado impresso é `10`.

```
Digite o número final:  4
Somatório = 10
```

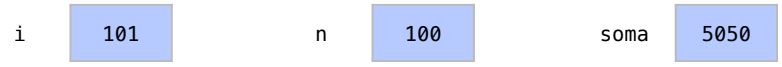
Outro exemplo de número escolhido pelo usuário seria 100:

```
Digite o número final:  100
```



Perceba como estruturas de repetição nos permitem realizar tarefas muito mais complexas.

```
Digite o número final:  100
Somatório = 5050
```



6.5 Aninhando estruturas de controles

É possível combinar estruturas de repetição com outras estruturas de controle. Isso pode ser feito, por exemplo, para garantir na repetição que existam determinadas condições.

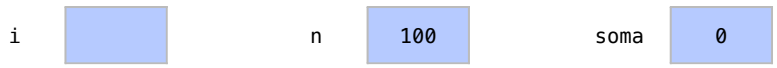
Veja este exemplo onde faremos o somatório de números múltiplos de 3.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      int soma;
9      cout << "Digite␣o␣número␣final:␣";
10     cin >> n;
11     soma = 0;
12     for (i = 1; i <= n; i++){
13         if (i % 3 == 0){
14             soma += i;
15         }
16     }
17     cout << "Somatório␣=␣" << soma << endl;
18     return 0;
19 }
```

Neste exemplo, só adicionaremos um número ao somatório se este for múltiplo de 3. A soma será então (`3 + 6 + 9 + 12 ...`).

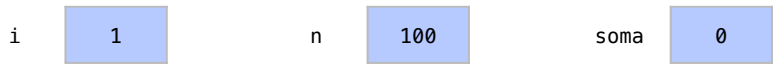
Nas linhas 6 a 11 são criadas e inicializadas as variáveis. O usuário define o valor`100` para a variável `n`:

```
Digite o número final:  100
```



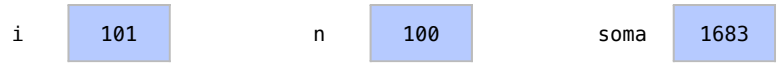
Na repetição, definida na linha 12, o contador `i` inicia em `1` novamente, pois não precisamos somar `0`. Logo na linha 13, se o resto da divisão de `i` por `3` for igual a `0`, o comando da linha 14 é executado. Na primeira iteração, como `i % 3 == 0` é `false`, nada acontece.

```
Digite o número final:  100
```



Continuamos nosso código e, quando a condição `i % 3 == 0` é verdadeira, este valor de `i` será adicionado à `soma`. Esse processo se repete até que `i` seja `101` e a condição de repetição seja `false`. Ao final temos então o somatório total de todos os número menores que `1000` e múltiplos de `3`, que é `1683`:

```
Digite o número final:  100
Somatório = 1683
```



6.6 Condições de inicialização

A condição de inicialização de um `for` pode ser qualquer expressão. Suponha que queremos calcular o resultado da seguinte série:

$$\frac{5}{3}+\frac{7}{4}+\frac{9}{5}+\frac{11}{6}+\cdots+\frac{99}{50}$$

Para isto, podemos somar elementos calculador em uma repetição onde o contador `i` vai de 3 a 50. Nesta série, a cada repetição somamos o elemento:

$$\frac{2i-1}{i}$$

Neste exemplo, vamos utilizar um contador `i` inicializado em 3 para calcularmos esta série.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      double soma;
9      double numerador;
10     double denominador;
11     cout << "Digite␣o␣contador␣final:␣";
12     cin >> n;
13     soma = 0;
14     for (i = 3; i <= n; i++){
15         numerador = 2*i-1;
16         denominador = i;
17         soma += numerador/denominador;
18     }
19     cout << "Somatório␣=␣" << soma << endl;
20     return 0;
21 }
```

Nas linhas 6 a 13 inicializamos os variáveis necessárias pelo programa. O usuário define um valor final para `n` de 50, que é o último número da série.

```
Digite o contador final:  50
```

i		n	50	soma	0	numera dor		denomi nador	

Entramos na estrutura de repetição na linha 14. Veja que a condição de inicialização é que `i = 3`. Fazemos isto pois 3 é o primeiro valor de `i` em nossa série. Nas linhas 15 a 17, a variável `soma` recebe o resultado de `5/3` na primeira iteração, quando `i` é 3.

i	3	n	50	soma	1.6666	numera dor	5	denomi nador	3

Após cada iteração da repetição, incrementamos o `i` e testamos a condição do laço. Na segunda iteração, `soma` é incrementado de `7/4`.

i	4	n	50	soma	3.4166	numera dor	7	denomi nador	4

Esse processo se repete até que `i` seja `n+1`, ou `51`. Neste ponto, o resultado do somatório é impresso na linha 19:

```
Digite o contador final:  50
Somatório = 93.0008
```

i	51	n	50	soma	93.0008	numera dor	99	denomi nador	50

6.7 Condições de incremento

As condições de incremento de um `for` também podem ser alteradas com qualquer outra expressão. Há vários motivos pelos quais faríamos isso:

- Para alterar a condição de incremento e dar saltos mais largos na variável contadora. Por exemplo, `i` pode aumentar em 7 a cada iteração.
- Podemos utilizar outra operação, por exemplo, `i` dobrar a cada iteração.
- Podemos até mesmo decrementar a variável contadora a cada passo, por exemplo, `i` diminuir em 1 a cada iteração.

Neste exemplo, calcularemos a soma de todos os múltiplos de 7 menores que um certo número limite. Temos então `7 + 14 + 21 + 38...`.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      double soma;
9      cout << "Digite o número final: ";
10     cin >> n;
11     soma = 0;
12     for (i = 7; i <= n; i += 7){
13         soma += i;
```

```
14      }
15      cout << "Somatório␣=␣" << soma << endl;
16      return 0;
17  }
```

As variáveis são criadas e inicializadas nas linhas 6 a 11, onde o usuário escolhe 1000 como um valor limite para o somatório:

```
Digite o número final:  1000
```

Na linha 12, o contador `i` já é inicializado com 7, o primeiro número da série. Como o contador `i` é menor ou igual ao o número final `n`, executamos os comandos do bloco:

Na estrutura de repetição, já incrementamos `i` em 7 para a próxima iteração com `i += 7`. O processo se repetirá enquanto `i <= 1000`.

```
Digite o número final:  1000
Somatório = 71071
```

Veja como neste exemplo utilizamos uma expressão diferente de incremento. Com a nova expressão não precisamos testar se `i` é multiplo de 7 pois já sabemos que apenas múltiplos de 7 serão atribuídos a `i`.

Neste segundo exemplo, vamos imprimir todas as potências de 2 entre 2 e um número `n` qualquer. Por exemplo, 2, 4, 8, 16, 32, 64, 128 e assim em diante.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main() {
6       int i;
7       int n;
8       cout << "Digite␣o␣número␣final:␣";
9       cin >> n;
10      for (i = 2; i < n; i*=2){
11          cout << i << "␣";
12      }
13      return 0;
14  }
```

Nas linhas 6 e 9 são criadas e inicializadas as variáveis, onde o usuário define 10000 como um valor limite:

```
Digte o número final:  10000
```

Na linha 10, o contador `i` é inicializado em no primeiro item da série, que é 2. O valor de `i` é impresso na linha 11:

```
Digite o número final:  10000
2
```

A operação de incremento multiplica `i` por 2, o transformando em 4.

```
Digite o número final:  10000
2 4
```

Esse processo se repete enquanto `i` for menor que 10000.

```
Digite o número final:  10000
2 4 8 16 32 64 128 256 512 1024 2048 4096 8192
```

No terceiro exemplo, vamos imprimir uma sequência de `n` até `1`. Por exemplo: `n`, `n-1`, `n-2`, `n-3` e assim em diante.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int n;
8      cout << "Digite␣o␣número␣inicial:␣";
9      cin >> n;
10     for (i = n; i > 0; i--){
11         cout << i << "␣";
12     }
13     return 0;
14 }
```

Nas linhas 6 a 9 temos as variáveis criadas e inicializadas, onde o usuário dá a `n` o primeiro numero da série, no caso `50`.

```
Digite o número inicial:  50
```

Na estrutura de repetição, na linha 10, `i` é inicializado nesse primeiro item da série, que é `n`. Seu valor é impresso na linha 11:

```
Digite o número inicial:  50
50
```

O bloco de comandos se repetirá enquanto `i` for maior que `0`. Repare agora que a expressão de incremento neste exemplo diminui o valor de `i`.

```
Digite o número inicial:  50
50 49
```

Assim, enquanto `i > 0`, todos os números entre `50` e `1` são impressos.

```
Digite o número inicial:  50
50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28
27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

6.8 Percorrer arranjos

Os arranjos são bastante relacionados à instrução `for`. Como arranjos têm um número grande de elementos, inicializá-los um por um é uma tarefa muito repetitiva. Nestes casos, a própria variável contadora do `for` pode ser utilizada dentro do bloco de repetição como índice para elementos do arranjo.

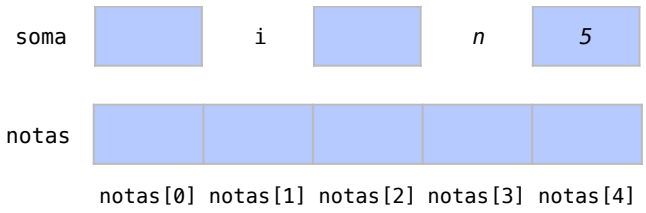
Neste exemplo, vamos dar valores às posições de um arranjo com uma estrutura de repetição:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
```

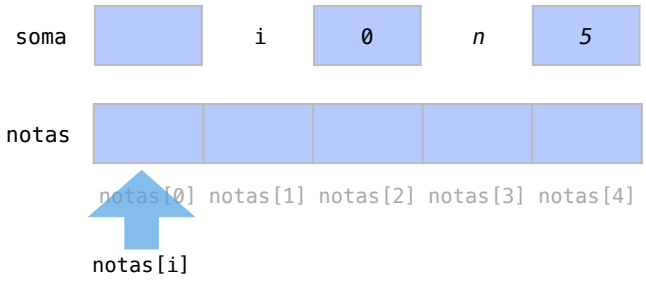
```
6     const int n = 5;
7     double notas[n];
8     double soma;
9     int i;
10    for(i=0; i<n; i++) {
11     cout << "Digite␣a␣nota␣do␣aluno␣" << i << ":␣";
12     cin >> notas[i];
13    }
14    soma = 0;
15    for(i = 0; i < n; i++) {
16     soma += notas[i];
17    }
18    cout << "Média␣das␣notas:␣" << soma/n << endl;
19    return 0;
20 }
```

Logo na linha 6, primeira linha da nossa função principal `main()`, criamos um `int` que tem valor 5. A instrução `const` que acompanha `int` indica que esta é uma **constante** em vez de uma variável. O valor de uma constante não pode ser alterada ao longo do programa.

Criamos um arranjo chamado `notas`, na linha 7. Apenas constantes podem definir o tamanho de arranjos em C++. A variável `soma` é criada na linha 8 para receber a soma de todas as notas do arranjo. A linha 9 cria a variável contadora `i`.

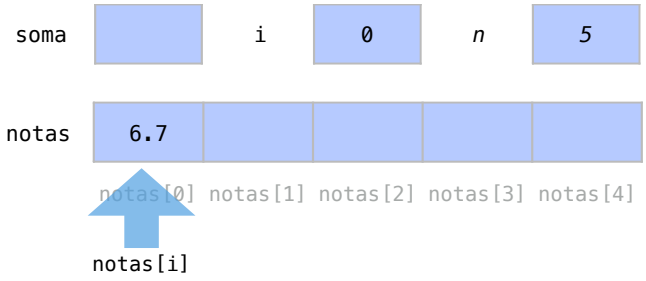


Na estrutura de repetição, definida na linha 10, a variável contadora `i` inicializa em `0` para trabalhar com `notas[0]`. Como `i` é `0`, sempre que nos referirmos a `notas[i]` estaremos nos referindo a `notas[0]`:



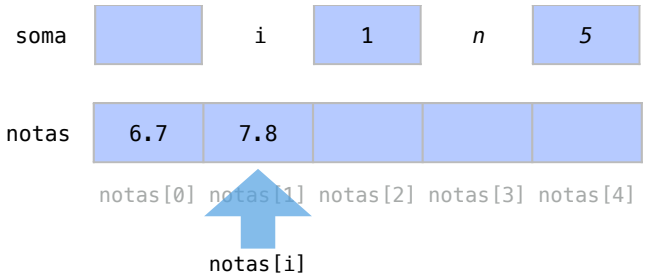
Na linha 11, pedimos agora para o usuário que insira a nota do aluno do primeiro aluno, o aluno que está na posição `i`, ou `0`, do arranjo. O resultado digitado pelo usuário é guardado em `notas[i]` (que no caso é `notas[0]`).

```
Digite a nota do aluno 0:  6.7
```



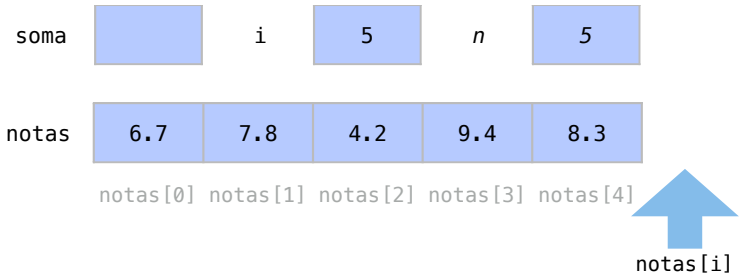
Na expressão de incremento `i++`, passamos então para o próximo aluno `i`. Note que com a alteração do valor de `i`, `notas[i]` passa a representar outra posição do arranjo. Assim, para cada posição `i` do arranjo, damos um valor de nota nas linhas 11 e 12.

```
Digite a nota do aluno 0:  6.7
Digite a nota do aluno 1:  7.8
```

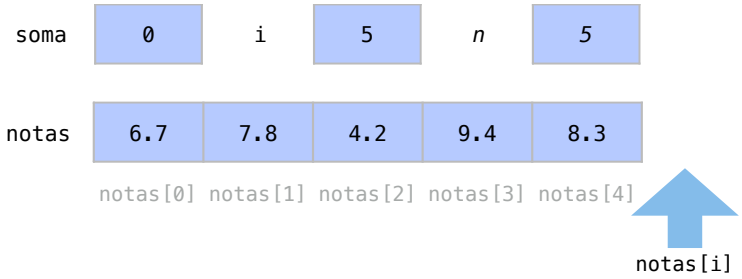


Após a última iteração da estrutura de repetição, `notas[i]` passa a representar `notas[5]`, que é uma posição fora do arranjo. O programa daria um erro de tentássemos acessar `notas[i]`. Porém, veja que a condição de repetição não é atendida e prosseguimos com o código.

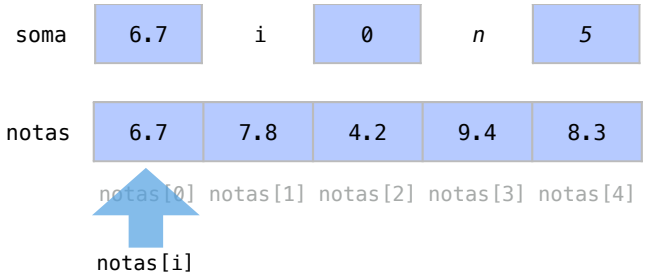
```
Digite a nota do aluno 0:  6.7
Digite a nota do aluno 1:  7.8
Digite a nota do aluno 2:  4.2
Digite a nota do aluno 3:  9.4
Digite a nota do aluno 4:  8.3
```



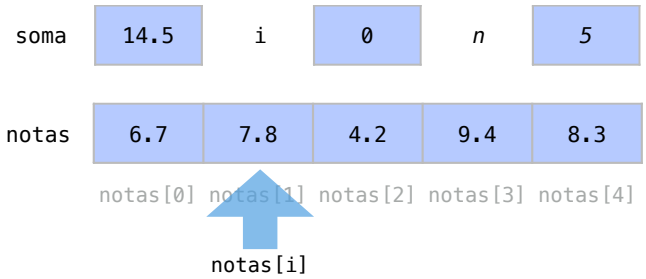
A variável `soma`, então, na linha 14, é inicializada em `0`. Utilizaremos esta variável para fazermos um somatório das notas no arranjo. Diferentemente dos exemplos anteriores somaremos notas na posição `i` em vez de, apenas, `i`, o índice.



Então, `i` volta a ser `0` na nova estrutura de repetição da na linha 15 e, `notas[i]` volta a indicar `notas[0]`. Veja que a `soma`, na linha 16, é incrementado de `notas[0]`.

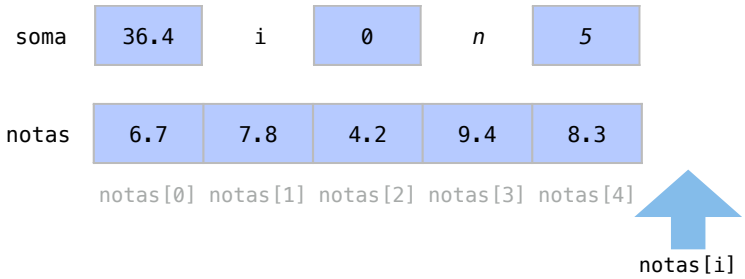


Agora, na segunda iteração da estrutura de repetição, `soma` é incrementado de `notas[1]`.



E ao fim de todo o processo processo de repetição, `soma` terá o somatório de todas as notas no arranjo, que é `36.4`, e `i` terá valor `5`, colocando `notas[i]` fora do arranjo novamente. Na linha 18, a soma das `notas` dividida por `n` nos permite imprimir a nota média nas provas.

```
Digite a nota do aluno 0:  6.7
Digite a nota do aluno 1:  7.8
Digite a nota do aluno 2:  4.2
Digite a nota do aluno 3:  9.4
Digite a nota do aluno 4:  8.3
Média das notas:  7.28
```



6.9 Laços aninhados

Vimos na seção 6.5 a junção de estruturas de repetição com estruturas de seleção. As estruturas de repetição podem ser combinadas também com outras estruturas de repetição para uma diversidade de tarefas que envolvem repetições dentro de repetições. Inclusive, o contador de uma estrutura de repetição pode servir de referência para o contador da outra estrutura mais interna.

Neste exemplo, usaremos dois contadores, `i` e `j`, que serão usados como contadores ao mesmo tempo para imprimir vários valores diversas vezes.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      int j;
8      int n;
9      cout << "Digite um número: ";
10     cin >> n;
11     for(i=1; i<=n; i++) {
12      for (j=0; j<i; j++){
13        cout << i ;
14      }
15      cout << endl;
16     }
17     return 0;
18 }
```

Neste código, para cada valor de `i` do `for` mais externo, um `for` interno vai imprimir `i` vezes o próprio valor `i`. O `for` mais interno utilize a variável `j` como contador. Depois do `for` interno, será dada uma quebra de linha com `endl`.

Nas linhas 6 a 10, inicializamos as variáveis e o usuário define que o valor de `n` será `5`.

```
Digite um número:  5
```

Veja que, no primeira laço de repetição, na linha 11, `i` começa em `1` e ele irá até `n`.

Na linha 12, `j` começa em 0 no segundo laço de repetição e ele irá até `i-1`. Ou seja, para cada valor de `i` em uma iteração no laço externo, este laço interno ocorre `i` vezes.

Perceba que após o laço mais interno, na linha 15, `endl` passa para a próxima linha encerrando, assim, uma iteração do laço externo.

```
Digite um número:  5
1
```

Enfim, `i` é incrementado para 2. Na primeira iteração do laço interno, `i`, ou 2, é impresso uma vez.

```
Digite um número:  5
1
2
```

Na segunda iteração, `i` é impresso uma segunda vez:

```
Digite um número:  5
1
22
```

A fim da segunda iteração, `j` passa a valer 2 e o laço interno se encerra.Veja que o laço interno imprime sempre `i` vezes o valor de `i`. E `endl` passa para a próxima linha.

```
Digite um número:  5
1
22
```

O contador `i` é mais uma vez incrementado para 3 e seguiremos então, para a próxima iteração. Nesta próxima iteração, o valor de `i` é impresso 3 vezes no laço mais interno.

```
Digite um número:  5
1
22
333
```

Esse processo se repete até que `i` seja igual a 6, interrompendo assim o laço de repetição na linha 11.

```
Digite um número:  5
1
22
333
4444
55555
```

6.10 Percorrendo arranjos de arranjos

Uma das aplicações mais comuns para laços alinhados é percorrer arranjos de arranjos. Como vimos, estes tipos de arranjos são muito utilizados para representar matrizes ou tabelas. Enquanto o laço externo percorre linhas com contador `i`, o laço mais interno percorre colunas com o contador `j`.

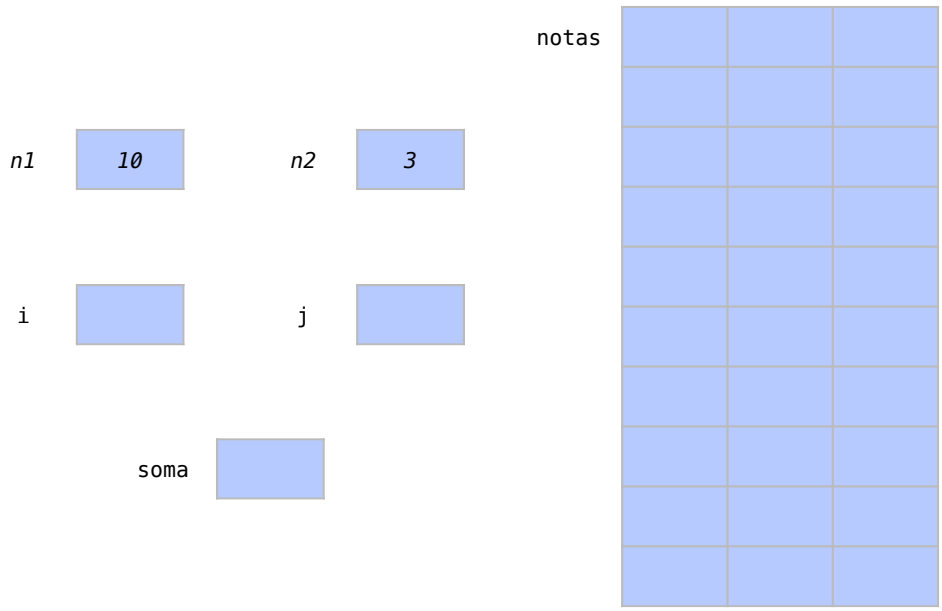
Neste próximo exemplo, criaremos um arranjo de arranjos (ou arranjo multidimensional) para representar uma tabela de notas e alunos. Cada coluna de nossa tabela representa uma prova e cada linha representa um aluno. Um laço externo percorre todas as linhas do arranjo bidimensional. Um laço interno percorre todas as colunas de uma linha.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6     const int n1 = 10;
7     const int n2 = 3;
8     int i;
9     int j;
10    double soma;
11    double notas[n1][n2];
12    for(i=0; i<n1; i++) {
13     cout << "Digite as notas do aluno "<< i << endl;
14     for (j=0; j<n2; j++){
15          cout << "Prova " << j << ": " ;
16          cin >> notas[i][j];
17     }
18    }
19    for(j=0; j<n2; j++) {
20     soma = 0;
21     for (i=0; i<n1; i++){
22            soma += notas[i][j];
23     }
24     cout << "Média na prova " << j << ": ";
25     cout << soma/n1 << endl;
26    }
```

```
27     return 0;
28 }
```

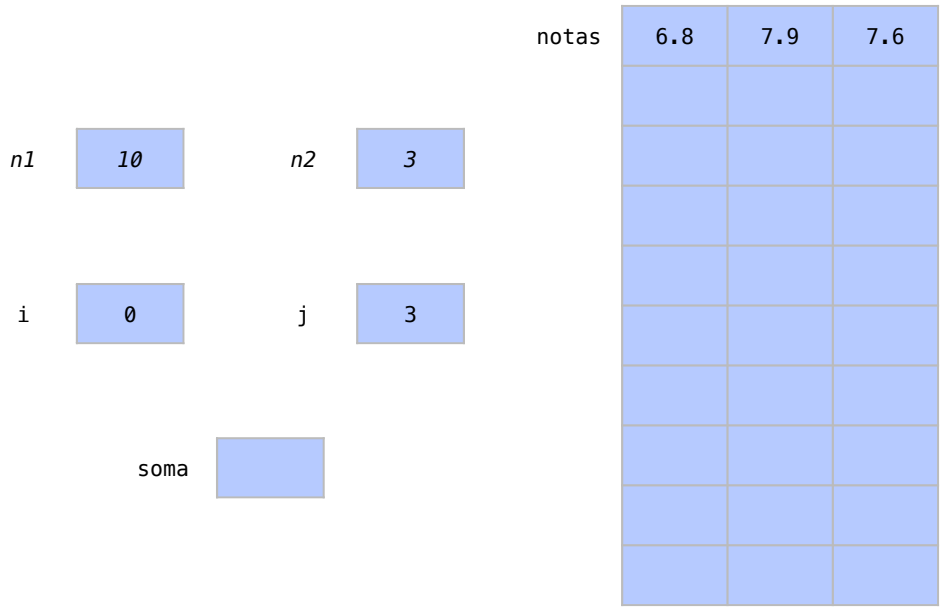
Temos um laço interno nas linhas 14 a 17 e um laço mais externo nas linhas 12 a 18. Este laço interno percorre todas as colunas `j` de uma linha `i`. A linha `i` é definida pelo laço mais externo.

Nas linhas 6 a 11, criamos e inicializamos todas as variáveis. As constantes `n1` e `n2` definem o número de linhas e colunas na tabela.



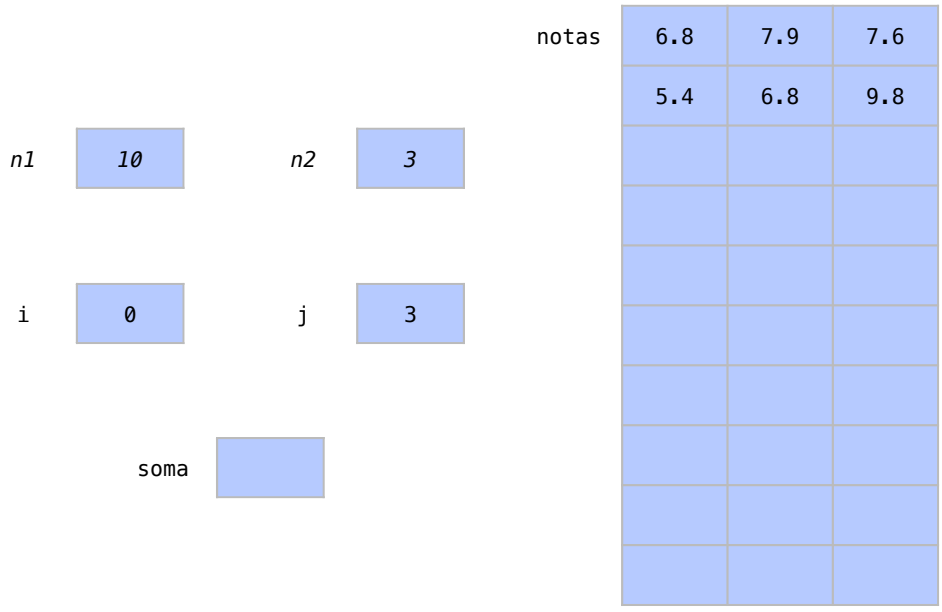
Veja que no laço mais externo, na linha 12, para cada linha `i`, que representa um aluno, imprimimos uma mensagem, na linha 13, pedindo as notas do aluno. Ainda para este aluno `i`, começamos na linha 14 um laço mais interno com o contador `j`. Este laço mais interno, nas linhas 15 e 16, pedem a nota do aluno `i` em cada prova `j`, ou seja, `notas[i][j]`. Na primeira iteração, `i` tem valor `0`. O laço mais interno utiliza `j` para percorrer então as colunas `0`, `1` e `2` deste aluno `i`.

```
Digite as notas do aluno 0
Prova 0:  6.8
Prova 1:  7.9
Prova 2:  7.6
```



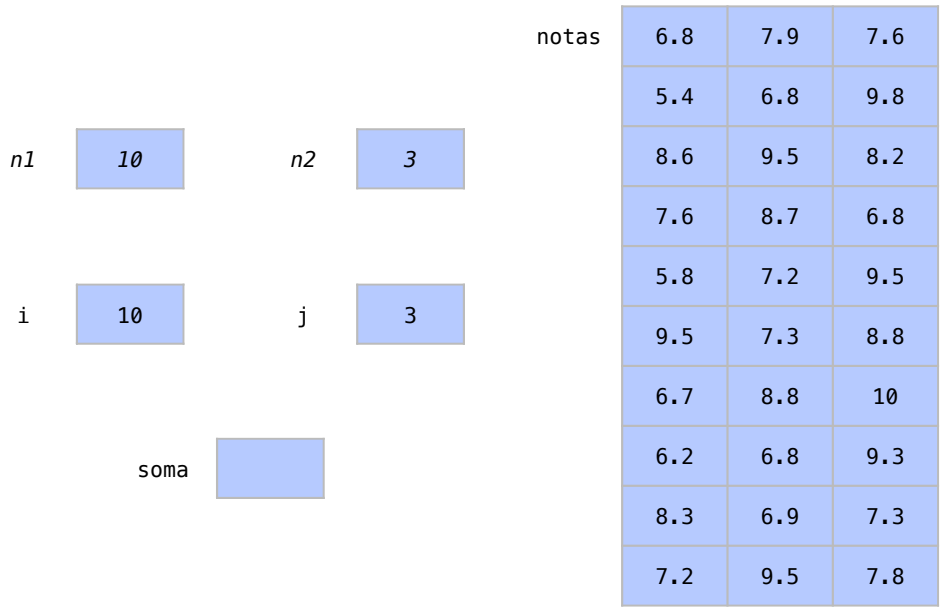
Como não acabaram os alunos, vejamos a próxima iteração, quando o valor de `i` é `1`. Essa iteração faz o mesmo para a linha seguinte da tabela.

```
Digite as notas do aluno 0
Prova 0:  6.8
Prova 1:  7.9
Prova 2:  7.6
Digite as notas do aluno 1
Prova 0:  5.4
Prova 1:  6.8
Prova 2:  9.8
```



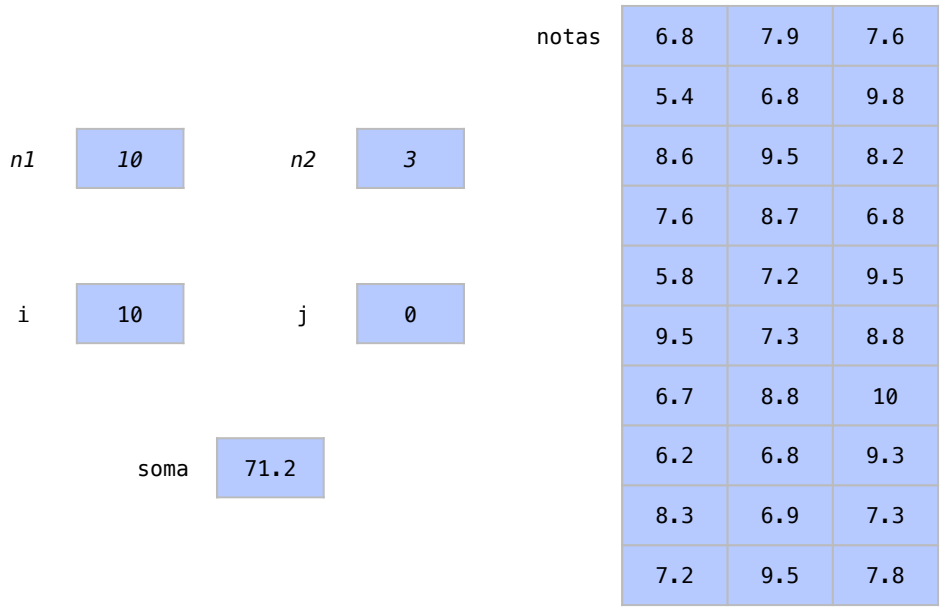
Já o conjunto de todas as iterações no laço mais externo completa a tabela e nos envia para a linha 19 do código.

```
...
Prova 1:  6.8
Prova 2:  9.3
Digite as notas do aluno 8
Prova 0:  8.3
Prova 1:  6.9
Prova 2:  7.3
Digite as notas do aluno 9
Prova 0:  7.2
Prova 1:  9.5
Prova 2:  7.8
```



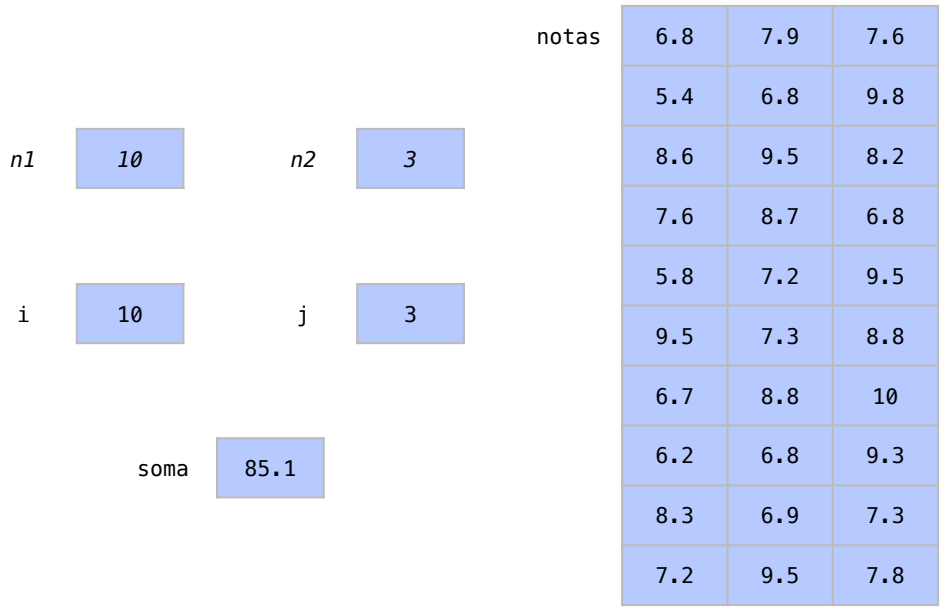
Na linha 19, temos um outro laço mais externo. O próximo laço externo utilizará `j`com contador para percorrer as colunas `j` da tabela. Para cada coluna calcularemos a sua média. A `soma` desta coluna `j` começa, então, em 0, como feito na linha 20. Nas linhas 21 a 23, para cada linha `i` desta coluna `j`, incrementamos o valor de `notas[i][j]` a `soma`. Nas linhas 24 e 25, imprimimos a média do somatório desta coluna.

```
...
Prova 1:  6.8
Prova 2:  9.3
Digite as notas do aluno 8
Prova 0:  8.3
Prova 1:  6.9
Prova 2:  7.3
Digite as notas do aluno
Prova 0:  7.2
Prova 1:  9.5
Prova 2:  7.8
Média na prova 0:  7.21
```



Na execução de todo o laço externo, fazemos o mesmo com as outras colunas. Este é o fim deste programa.

```
...
Prova 1:  6.8
Prova 2:  9.3
Digite as notas do aluno 8
Prova 0:  8.3
Prova 1:  6.9
Prova 2:  7.3
Digite as notas do aluno
Prova 0:  7.2
Prova 1:  9.5
Prova 2:  7.8
Média na prova 0:  7.21
Média na prova 1:  7.94
Média na prova 2:  8.51
```



6.11 Utilizando apenas critério de parada

É muito comum repetir comandos não apenas uma certa quantidade de vezes, como também até que uma condição ocorra. Por exemplo, quando esfregamos um prato até que ele esteja limpo. Não sabemos quantas vezes vamos esfregar, mas faremos isto até que esteja limpo.

Às vezes, queremos que um bloco de comandos seja repetido até que uma certa condição ocorra mas esta condição não está associada especificamente a um contador. Nestes casos, a instrução `for`, por depender de um contador, pode parecer inapropriada. Usamos o comando `while` (*enquanto*, em inglês), para definir a condição do laço.

Na sintaxe da instrução é a seguinte:

```
1 while (condição){
2     comandos;
3 }
```

Enquanto a condição entre parênteses for atendida, o bloco de comandos será executado. Repare que, diferentemente do comando `for`, não há comandos para inicialização ou incremento, pois o `while` não está diretamente ligado a um contador.

No exemplo seguinte, *enquanto* o número digitado for diferente de `0`, o usuário digitará um números que serão elevados ao quadrado e impressos.

```
1 int numero = 1;
2 while (numero != 0){
3     cout << "Digite␣um␣número:␣";
4     cin >> numero;
5     cout << "Número␣ao␣quadrado␣=␣";
6     cout << numero * numero << endl;
7 }
```

No exemplo desta seção, colocaremos uma estrutura deste tipo em nosso programa.

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

```
4
5  int main() {
6      int numero = 1;
7      while (numero != 0){
8          cout << "Digite um número: ";
9          cin >> numero;
10         cout << "Número ao quadrado = " << numero * numero << endl;
11     }
12     return 0;
13 }.
```

Na linha 6, criamos uma variável `numero` que inicializada com valor `1`.

A instrução `while` da linha 7 diz que repetiremos o bloco de comandos enquanto o valor de `numero` for diferente de `0`. Como `numero` é diferente de `0`, a condição de repetição é verdadeira e executaremos os comandos das linhas 8 a 10.

```
Digite um número:  7
Número ao quadrado:  49
```

Como o valor de `numero` ainda é diferente de `0`, a condição da linha 7 é verdadeira e os comandos das linhas 8 a 10 são executados novamente.

```
Digite um número:  7
Número ao quadrado:  49
Digite um número:  5
Número ao quadrado:  25
```

Não há um número definido de repetições. Este processo continua até que o usuário digite `0`, com a condição retornando `false` e saímos do laço.

```
Digite um número:  7
Número ao quadrado:  49
Digite um número:  5
Número ao quadrado:  25
Digite um número:  0
Número ao quadrado:  0
```

6.11.1 Relação entre `while` e `for`

Considere a estrutura de um `while`:

```
while (condição){
    comandos;
}
```

Para se ter o efeito de um `while` com um `for`, basta deixar em branco as condições de inicialização e de incremento deste `for`, como apresentado:

```
for ( ;condição; ){
    comandos;
}
```

Quando queríamos repetir um bloco enquanto o número fosse diferente de 0, utilizamos a seguinte estrutura `while`:

```
while (numero != 0){
    cout << "Digite um número: ";
    cin >> numero;
    cout << "Número ao quadrado = ";
    cout << numero * numero << endl;
}
```

Poderíamos ter feito o mesmo com uma instrução `for`. Neste caso, o `for` teria apenas sua condição de repetição.

```
for (;numero != 0;){
    cout << "Digite um número: ";
    cin >> numero;
    cout << "Número ao quadrado = ";
    cout << numero * numero << endl;
}
```

Por outro lado, considere a sintaxe de um `for`:

```
for (inicialização;condição;incremento){
    comandos;
}
```

Para termos o efeito de um `for` com um `while`, basta inicializar e incrementar o contador com um comandos próprios:

```
inicialização;
while (condição){
    comandos;
    incremento;
}
```

Por exemplo, se um contador vai de 0 até 9 para imprimirmos uma sequência, utilizariamos o seguinte `for`:

```
for (i = 0; i < 10; i++){
    cout << i << endl;
}
```

O mesmo efeito pode ser obtido com um `while`, onde podemos fazer a inicialização antes de iniciar a repetição. Após os comandos originais do bloco, incrementamos com um comando próprio o valor do contador.

```
1 i = 0;
2 while (i < 10){
3     cout << i << endl;
4     i++;
5 }
```

Embora seja tecnicamente possível converter um `for` em um `while` e vice-versa, a estrutura mais apropriada deve ser utilizada em cada ocasião. Utilize `for` para repetições baseadas em contadores. Utilize `while` para repetições dependentes de condições específicas de parada.

6.11.2 Laços infinitos

É necessário tomar cuidado em relação aos laços infinitos, ou seja, os laços onde a condição de repetição é sempre verdadeira e por isso nunca encerram. Isto pode ser feito propositalmente com um `while` ou `for` que tenha `true` como condição de repetição:

```
1 while (true){
2     comandos;
3 }
```

```
1 for (inicializa; true; incremento){
2     comandos;
3 }
```

É necessário garantir que a condição em algum momento será falsa.

6.12 Adiando o critério de parada

Existe ainda uma situação onde só sabemos se a condição de repetição será atendida após a primeira iteração. Este caso é bem comum em menus, onde uma das opções é sair. Para estes casos, existe a a instrução `do-while`. No `do-while`, os comandos são sempre executados uma vez antes que algum teste seja feito.

```
1 do {
2     comandos;
3 } while (condição);
```

Neste exemplo, repare que um menu é apresentado e cada opção faz uma tarefa:

```
1  int x = 5;
2  int y = 7;
3  int opcao;
4  do {
5      cout << "[1] Somar" << endl;
6      cout << "[2] Multiplicar" << endl;
7      cout << "[0] Sair do programa" << endl;
8      cout << "Digite uma opcao: ";
9      cin >> opcao;
10     if (opcao == 1){
11         cout << x + y << endl;
12     } else if (opcao == 2){
```

```
13          cout << x * y << endl;
14      }
15  } while (opcao != 0);
```

Este menu é apresentado enquanto a opção do usuário não for `0`. Esta é a condição perfeita para um `do-while`, pois é um caso onde a primeira iteração sempre deve acontecer para determinar as próximas. Só sabemos após a primeira iteração se a opção será sair.

6.12.1 Relação entre `do-while` e `while`

É possível obter o efeito de um `do-while` com um `while` se repetirmos o comando da primeira iteração antes do laço. Considere a sintaxe do `do-while`:

```
1 do {
2     comandos;
3 } while (condição);
```

O mesmo efeito pode ser obtido com a instrução `while`:

```
1 comandos;
2 while (condição) {
3     comandos;
4 }
```

Perceba que a repetição dos comandos antes do laço garante que eles serão executados pelo menos uma vez. Isto, porém, é uma solução pouco desejável pois estamos repetindo código desnecessariamente. Assim, apesar de ser tecnicamente possível usar qualquer instrução, é sempre melhor utilizar a estrutura mais apropriada em cada caso.

6.13 Exercícios

Arranjos não podem ser utilizados nestes primeiros exercícios.

Exercício 6.1 Faça um programa que apresente um menu de opções para o cálculo das seguintes operações entre dois números: adição, subtração, multiplicação e divisão. O programa deve solicitar dois números e possibilitar ao usuário a escolha da operação desejada, exibindo o resultado e a voltar ao menu. O programa só termina quando for escolhida a opção de saída.

Exemplo de saída:

```
[1] Adição
[2] Subtração
[3] Multiplicação
[4] Divisão
[0] Sair
Escolha uma opção: 3
Digite um número: 5.2
Digite outro número: 7.5
5.2 * 7.5 = 39.0

[1] Adição
[2] Subtração
[3] Multiplicação
[4] Divisão
```

```
[0] Sair
Escolha uma opção: 4
Digite um número: 4.3
Digite outro número: 6.7
4.3 * 6.7 = 0.64

[1] Adição
[2] Subtração
[3] Multiplicação
[4] Divisão
[0] Sair
Escolha uma opção: 0

Obrigado por utilizar a calculadora.
```

■

Exercício 6.2 Escrever um algoritmo que lê n valores, um de cada vez, e conta quantos destes valores são pares, escrevendo esta informação.

Exemplo de saída:

```
Quantos valores deseja ler? 7
Digite um número: 6
Digite um número: 3
Digite um número: 5
Digite um número: 4
Digite um número: 2
Digite um número: 8
Digite um número: 5
Você digitou 4 números pares e 3 números ímpares
```

■

Exercício 6.3 Escreva um programa que receba a idade de n pessoas, calcule e imprima: a quantidade de pessoas em cada faixa etária; a porcentagem de cada faixa etária em relação ao total de pessoas.

As faixas etárias são:
1 a 15 anos
16 a 30 anos
31 a 45 anos
46 a 60 anos
$\geq$ 61 anos

Exemplo de saída:

```
Quantas idades deseja digitar? 10
Digite a idade da pessoa 1: 6
Digite a idade da pessoa 2: 41
Digite a idade da pessoa 3: 33
Digite a idade da pessoa 4: 25
```

```
Digite a idade da pessoa 5: 70
Digite a idade da pessoa 6: 71
Digite a idade da pessoa 7: 11
Digite a idade da pessoa 8: 39
Digite a idade da pessoa 9: 42
Digite a idade da pessoa 10: 28
Faixa etária:
1 a 15 anos - 20.0%
16 a 30 anos - 20.0%
31 a 45 anos - 40.0%
46 a 60 anos - 0.0%
Mais de 60 anos - 20.0%
```

■

Exercício 6.4 Escreva um programa que receba um número inteiro e verifique se o número fornecido é primo ou não. O número é primo se ele tiver apenas 2 divisores: 1 e ele mesmo.

Exemplo de saída:

```
Digite um número limite: 7
7 é um número primo
```

■

Exercício 6.5 Encontre e imprima os divisores de cada número até um limite N (número inteiro) fornecido pelo usuário.

Exemplo com $N = 5$:

1:1
2:1 | 2
3:1 | 3
4: 1 | 2 | 4
5: 1 | 5

Exemplo de saída:

```
Digite um número limite: 10
Os divisores até 1 são: 1
Os divisores até 2 são: 1 2
Os divisores até 3 são: 1 3
Os divisores até 4 são: 1 2 4
Os divisores até 5 são: 1 5
Os divisores até 6 são: 1 2 3 6
Os divisores até 7 são: 1 7
Os divisores até 8 são: 1 2 4 8
Os divisores até 9 são: 1 3 9
Os divisores até 10 são: 1 2 5 10
```

■

Exercício 6.6 Construa um programa que leia vários números inteiros e mostre qual foi o maior e o menor valor fornecido.

Exemplo de saída:

```
Quantos números deseja digitar? 5
Digite o número 1: 6
Digite o número 2: 3
Digite o número 3: 5
Digite o número 4: 2
Digite o número 5: 8
O maior número digitado foi 8
O menor número digitado foi 2
```

Dica: Quando o usuário digita seu primeiro número, este é sempre tanto o maior quanto o menor número até então. ■

Arranjos devem ser utilizados nestes próximos exercícios.

Exercício 6.7 Dada uma tabela de 4 x 5 elementos, calcular a soma de cada linha e a soma de todos os elementos. Uma estrutura de repetição **deve** ser utilizada para percorrer as linhas e colunas.

Exemplo de saída:

```
Digite o elemento da linha 0 e coluna 0: 5
Digite o elemento da linha 0 e coluna 1: 7
Digite o elemento da linha 0 e coluna 2: 3
Digite o elemento da linha 0 e coluna 3: 6
Digite o elemento da linha 0 e coluna 4: 3
Digite o elemento da linha 1 e coluna 0: 3
Digite o elemento da linha 1 e coluna 1: 4
Digite o elemento da linha 1 e coluna 2: 6
Digite o elemento da linha 1 e coluna 3: 3
Digite o elemento da linha 1 e coluna 4: 2
Digite o elemento da linha 2 e coluna 0: 8
Digite o elemento da linha 2 e coluna 1: 3
Digite o elemento da linha 2 e coluna 2: 2
Digite o elemento da linha 2 e coluna 3: 6
Digite o elemento da linha 2 e coluna 4: 2
Digite o elemento da linha 3 e coluna 0: 5
Digite o elemento da linha 3 e coluna 1: 3
Digite o elemento da linha 3 e coluna 2: 7
Digite o elemento da linha 3 e coluna 3: 3
Digite o elemento da linha 3 e coluna 4: 4
O soma da linha 0 é igual a 24
O soma da linha 1 é igual a 18
O soma da linha 2 é igual a 21
O soma da linha 3 é igual a 22
A soma de todos os elementos é 85
```

■

Exercício 6.8 Dada uma matriz A[4x4], imprimir o número de linhas e o número de colunas nulas (com apenas 0s) da matriz. Uma estrutura de repetição **deve** ser utilizada para percorrer as linhas e colunas.

Exemplo de saída:

```
Digite o elemento da linha 0 e coluna 0: 0
Digite o elemento da linha 0 e coluna 1: 0
Digite o elemento da linha 0 e coluna 2: 0
Digite o elemento da linha 0 e coluna 3: 0
Digite o elemento da linha 1 e coluna 0: 3
Digite o elemento da linha 1 e coluna 1: 4
Digite o elemento da linha 1 e coluna 2: 0
Digite o elemento da linha 1 e coluna 3: 3
Digite o elemento da linha 2 e coluna 0: 0
Digite o elemento da linha 2 e coluna 1: 0
Digite o elemento da linha 2 e coluna 2: 0
Digite o elemento da linha 2 e coluna 3: 0
Digite o elemento da linha 3 e coluna 0: 5
Digite o elemento da linha 3 e coluna 1: 3
Digite o elemento da linha 3 e coluna 2: 0
Digite o elemento da linha 3 e coluna 3: 3
Digite o elemento da linha 3 e coluna 4: 4
Esta matriz tem 2 linha(s) nula(s)
Esta matriz tem 1 coluna(s) nula(s)
```

■

Exercício 6.9 Faça um programa que mostre os elementos diagonais A[i,i] de uma matriz. Uma estrutura de repetição **deve** ser utilizada para percorrer as linhas e colunas.

Exemplo de saída:

```
Digite o elemento da linha 0 e coluna 0: 5
Digite o elemento da linha 0 e coluna 1: 7
Digite o elemento da linha 0 e coluna 2: 3
Digite o elemento da linha 0 e coluna 3: 6
Digite o elemento da linha 1 e coluna 0: 3
Digite o elemento da linha 1 e coluna 1: 4
Digite o elemento da linha 1 e coluna 2: 6
Digite o elemento da linha 1 e coluna 3: 3
Digite o elemento da linha 2 e coluna 0: 8
Digite o elemento da linha 2 e coluna 1: 3
Digite o elemento da linha 2 e coluna 2: 2
Digite o elemento da linha 2 e coluna 3: 6
Digite o elemento da linha 3 e coluna 0: 5
Digite o elemento da linha 3 e coluna 1: 3
Digite o elemento da linha 3 e coluna 2: 7
Digite o elemento da linha 3 e coluna 3: 3
Os elementos da diagonal são 5 4 2 3
```

Exercício 6.10 Faça um programa que multiplique duas matrizes. Uma estrutura de repetição **deve** ser utilizada para percorrer as linhas e colunas.

```
Digite o elemento da linha 0 e coluna 0 da matriz A: 5
Digite o elemento da linha 0 e coluna 1 da matriz A: 7
Digite o elemento da linha 0 e coluna 2 da matriz A: 3
Digite o elemento da linha 1 e coluna 0 da matriz A: 3
Digite o elemento da linha 1 e coluna 1 da matriz A: 4
Digite o elemento da linha 1 e coluna 2 da matriz A: 6
Digite o elemento da linha 2 e coluna 0 da matriz A: 8
Digite o elemento da linha 2 e coluna 1 da matriz A: 3
Digite o elemento da linha 2 e coluna 2 da matriz A: 2
Digite o elemento da linha 0 e coluna 0 da matriz B: 5
Digite o elemento da linha 0 e coluna 1 da matriz B: 7
Digite o elemento da linha 0 e coluna 2 da matriz B: 3
Digite o elemento da linha 1 e coluna 0 da matriz B: 3
Digite o elemento da linha 1 e coluna 1 da matriz B: 4
Digite o elemento da linha 1 e coluna 2 da matriz B: 6
Digite o elemento da linha 2 e coluna 0 da matriz B: 8
Digite o elemento da linha 2 e coluna 1 da matriz B: 3
Digite o elemento da linha 2 e coluna 2 da matriz B: 2
A matriz C = A * B é igual a:
70 72 63
75 55 45
65 74 46
```

7. Escopo de variáveis

Como vimos em nossos exemplos, um programa é composto de blocos que estão sempre entre chaves { e }. A função principal, `main`, é composta também de um bloco. Cada estrutura de controle também contém ao menos um bloco.

Escopo de bloco Uma variável tem **escopo de bloco** quando a declaramos em um bloco. Neste caso, ela pode ser apenas utilizada neste bloco e nos blocos mais internos a este bloco. O escopo de bloco começa na declaração na variável e termina na chave que fecha o bloco.

Escopo de arquivo Uma variável tem **escopo de arquivo** quando seu identificador é declarado fora de qualquer função ou classe. Esta é uma **variável global** e é conhecida em qualquer ponto do programa.

Em variáveis com escopo de blocos, quando há um bloco aninhado mais interno e os dois blocos contém uma variável com o mesmo identificador, a variável do bloco mais externo fica oculta até que o bloco interno termine.

Isto é o que faremos neste exemplo:

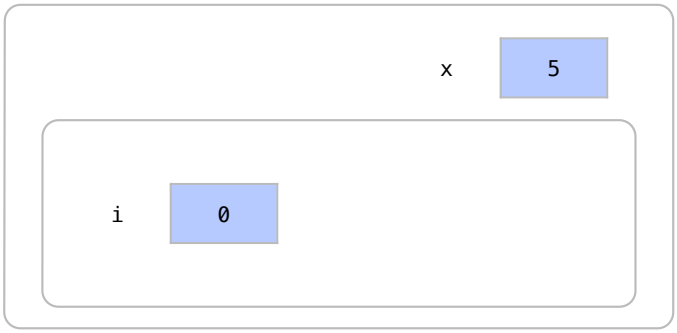
```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x;
7      x = 5;
8      cout << "x externo = " << x << endl;
9      for (int i = 0; i < 1; i++){
10         int x;
11         x = 7;
12         cout << "x interno = " << x << endl;
13         cout << "i interno = " << i << endl;
14     }
15     cout << "x externo = " << x << endl;
```

```
16     return 0;
17 }
```

Criamos e inicializamos uma variável `x`, nas linhas 6 e 7, que pertence ao bloco externo, o bloco do `main`. O valor desta variável é impresso na linha 8.

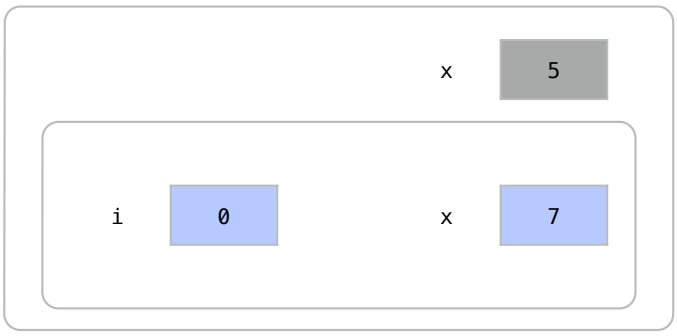
```
x externo = 5
```

Na linha 9, temos uma estrutura de repetição. Veja que não temos ainda uma variável inteira `i` para a estrutura de repetição. Isso ocorre porque a variável contadora `i` será agora criada no escopo do bloco do `for`, através da instrução `int` na linha 9. Por isso, ela só será existente enquanto estivermos no bloco do `for`.



Na linha 10, criamos uma variável `x` dentro do escopo do `for`. Como já temos uma variável `x` do bloco mais externo, isto faz com que a variável `x` do bloco externo fique oculta. Na linha 11, quando damos valor a `x`, alterações serão feitas, então, na variável visível. Veja que, na linha 12, o valor impresso foi o da variável no bloco mais interno.

```
x externo = 5
x interno = 7
```



Na linha 13 imprimimos apenas o valor de `i`, que também é do escopo do `for`.

```
x externo = 5
x interno = 7
i interno = 0
```

Após a primeira iteração o valor de `i` passar a ser `1` e a condição de repetição `i<1` não é mais atendida. Assim, o bloco do `for` é encerrado. As variáveis do bloco interno então deixam de existir e o `x` do bloco externo não está mais oculto, sendo impresso na linha 15.

```
x externo = 5
x interno = 7
i interno = 0
x externo = 5
```

Se tentássemos inserir uma linha de código que imprime o valor de `i`, no final do programa, teríamos um erro pois não existe a variável `i` neste ponto do código. Isso está representado na linha 18 do seguinte programa:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x;
7      x = 5;
8      int i;
9      cout << "x externo = " << x << endl;
10     for (i = 0; i < 1; i++){
11         int x;
12         x = 7;
13         cout << "x interno = " << x << endl;
14         cout << "i interno = " << i << endl;
15     }
16     cout << "x externo = " << x << endl;
17     // O seguinte comando causaria um erro:
18     cout << "i = " << i << endl;
19     // Não existe uma variável i neste ponto do código
20     return 0;
21 }
```

Para resolvermos este problema, portanto, precisariamos criar a variável `i` no bloco externo do programa, como feito na linha 9 do código abaixo.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x;
7      x = 5;
8      // criando variável i no bloco externo
9      int i;
10     cout << "x externo = " << x << endl;
11     for (i = 0; i < 1; i++){
12         int x;
13         x = 7;
14         cout << "x interno = " << x << endl;
15         cout << "i externo = " << i << endl;
16     }
17     cout << "x externo = " << x << endl;
18     cout << "i = " << i << endl;
19     return 0;
20 }
```

Agora, como a variável `i` foi declarada no bloco externo, ela pode ser acessada na linha 17, gerando o seguinte resultado para o programa:

```
x externo = 5
x interno = 7
i interno = 0
i = 1
```

Apesar do exemplo que utilizamos nesta seção, não é recomendada a prática de se utilizar nomes identificadores de variáveis que ocultam outras variáveis no escopo mais externo. Neste programa, isto pode ser corrigido simplesmente com um outro nome para o `x` interno:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int x;
7      int i;
8      x = 5;
9      cout << "x externo = " << x << endl;
10     for (i = 0; i < 1; i++){
11         int y;
12         y = 7;
13         cout << "y interno = " << y << endl;
14         cout << "i externo = " << i << endl;
15     }.
```

```
16      cout << "x externo = " << x << endl;
17      cout << "i = " << i << endl;
18      return 0;
19  }.
```

7.1 Exercícios

Exercício 7.1 Analise o código abaixo, que está incompleto. A linha 14 precisou ser comentada, pois causava um erro no programa. Após o `for`, queríamos imprimir o valor final do contador `i`. Conserte o código para que a linha abaixo possa ser descomentada sem causar um erro.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      // Imprime e soma as potências de 2 até 1000...
7      int somatorio = 0;
8      for (int i = 2; i < 1000; i*=2){
9          cout << i << " + ";
10         somatorio += i;
11     }
12     cout << " = " << somatorio << endl;
13     // A linha abaixo precisou ser comentada.
14     //cout << "Valor final de i = " << i << endl;
15
16     return 0;
17 }
```

■

8. Ponteiros

Vimos que os dados de cada variável contêm um endereço na memória que pode ser acessado com o operador de endereço, representado por “e"comercial (`&`). Enquanto `x` retorna o valor de uma variável `x`, o operador de endereço `&` pode ser utilizado para retornar o endereço de `x` na memória.

Ponteiros são variáveis que guardam endereços da memória em seus dados. São variáveis que além de ter seus endereços, podem guardar o endereço de outras variáveis. Quando um ponteiro guarda um endereço de um dado, dizemos que ele está apontando para aquele dado.

Suponha que criamos um número inteiro `x` de valor `5` e um ponteiro que se chamará `px`. Temos o seguinte trecho de código e a seguinte representação das variáveis:

```
1  int x;
2  x = 5;
3  int *px;
```

O operador `*` utilizado na linha 3 indica que `px` é um ponteiro para `int` e não apenas um `int`.

A Tabela 8.1 representa estas variáveis na memória de um computador. No exemplo apresentado, `x` se encontra no endereço 25 da memória. A variável `px` se encontra na posição 26 e ainda não tem um valor atribuído.

No exemplo seguinte, guardamos também o endereço de `x` em `px`, na linha 4.

```
1  int x;
2  x = 5;
3  int *px;
4  px = &x;
```

A Tabela 8.2 representa o estado destas variáveis na memória. Como `px` tem o endereço de `x`, dizemos que `px` aponta para `x`. Note que:

Nomes das variáveis	Valores das Variáveis	Endereços das Posições
		24
x	5	25
px		26
		27
		28
		29
		30
		31
		32
		33
		34
		35

Tabela 8.1: Organização de um ponteiro ainda vazio na memória. Apesar de guardar endereço, um ponteiro é uma variável com um nome identificador, valor e endereço.

- px tem endereço 26 e valor 25.
- px aponta para um dado que tem endereço 25 e valor 5.

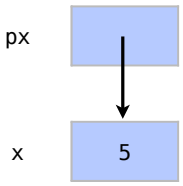
Nomes das variáveis	Valores das Variáveis	Endereços das Posições
		24
x	5	25
px	25	26
		27
		28
		29
		30
		31
		32
		33
		34
		35

Tabela 8.2: Organização de um ponteiro na memória.

O operador de endereço & aplicado a px (&px) nos retorna o endereço de px, que é 26. E o comando px, como usual, nos retorna o valor de px, que no caso é o endereço de x, ou 25.

O operador representado por um asterisco * é o **operador de desreferenciação**. Quando aplicamos este operador a px (*px), desreferenciamos px e retornamos o valor apontado por px. Nesse exemplo, este é também o valor de x, ou 5.

Sabemos que, na memória, os ponteiros fisicamente guardam outros endereços. Porém, quando representamos graficamente as variáveis isto é indicado por uma seta:

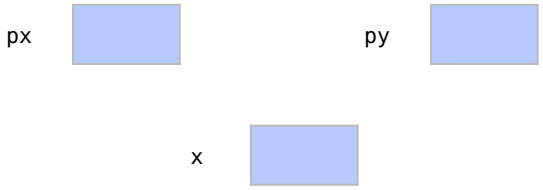


Veja a diferença entre a representação física na memória e a uma representação gráfica. A seta sai de dentro de px e aponta para o valor da variável x. Sabemos, assim, que o valor de px é o endereço de x, seja este qual for.

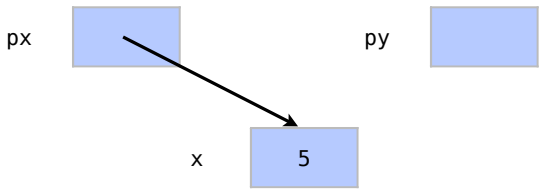
Neste exemplo, manipularemos ponteiros que apontam para outras variáveis:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x;
7      int *px;
8      int *py;
9      x = 5;
10     px = &x;
11     py = px;
12     cout << "x␣=␣" << x << endl;
13     cout << "&x␣=␣" << &x << endl;
14     cout << "px␣=␣" << px << endl;
15     cout << "*px␣=␣" << *px << endl;
16     cout << "*py␣=␣" << *py << endl;
17     cout << "&px␣=␣" << &px << endl;
18     cout << "&py␣=␣" << &py << endl;
19     return 0;
20 }
```

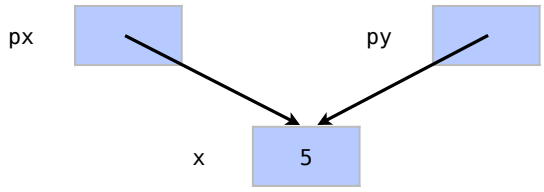
Criamos uma variável x, na linha 6, que será apontada por ponteiros. Criamos também, dois ponteiros px e py para números inteiros, nas linhas 7 e 8.



Após inicializar x com 5, na linha 9, px recebe em seguida o endereço de x na linha 10, o que o faz apontar para x.



Na linha 11, py recebe o valor de px, que é o endereço de x. Assim, py aponta também para x.



Imprimimos o valor de x e o endereço de x nas linhas 12 e 13, que é um endereço real na memória.

```
x = 5
&x = 0x7fff571abb68
```

Imprimimos em seguida, na linha 14, o valor de px, que no caso, é o endereço de x.

```
x = 5
&x = 0x7fff571abb68
&px = 0x7fff571abb68
```

Imprimimos, na linha 15, o valor apontado por px, que é o valor de x:

```
x = 5
&x = 0x7fff571abb68
&px = 0x7fff571abb68
*px = 5
```

Na linha 16, imprimimos o valor apontado por py, que também é o valor de x.

```
x = 5
&x = 0x7fff571abb68
px = 0x7fff571abb68
*px = 5
*py = 5
```

Apesar de apontarem para x, px e py são variáveis que contêm seus próprios endereços, como mostram as impressões feitas nas linhas 17 e 18.

```
x = 5
&x = 0x7fff571abb68
px = 0x7fff571abb68
*px = 5
*py = 5
&px = 0x7fff5b810b40
&py = 0x7fff5b810b38
```

Apesar de guardarem apenas endereços na memória, a sintaxe dos ponteiros nos obriga a dizer que tipo de dado ele aponta. No nosso exemplo, precisamos dizer que `px` é um ponteiro para um `int`.

Isto ocorre para que seja possível a operação que retorna o valor apontado. Esta operação depende do tipo de dado, para saber como interpretar os dados naquele endereço da memória.

8.1 Expressões com ponteiros

Se `px` aponta para `x`, o valor apontado por `px` pode ser utilizado em qualquer expressão onde `x` seria usado. Por exemplo: criamos um ponteiro que aponta para `x` e o valor apontado por `px` pode ser incrementado no lugar de `x`:

```
1 int *px;
2 int x;
3 px = &x;
4 *px = 5;
5 (*px)++;
6 cout << x << endl;
```

Ao final do exemplo, o valor de `x` será 6, assim como seria se nas linhas 4 e 5 tivéssemos incrementado `x` diretamente.

```
6
```

8.2 Aritmética com ponteiros

Podemos fazer aritmética também com ponteiros. Por exemplo, um incremento em um ponteiro `px` com `px++` faz com que ele aponte para o próximo endereço na memória para aquele tipo de dado. De modo análogo, `px + 6` retorna o endereço seis posições a frente de `px`.

É preciso tomar cuidado pois incrementar o valor apontado por `px` `(*px)++` é diferente de incrementar o endereço apontado por `px` e utilizar este valor apontado `*(px++)`.

Devemos evitar aritmética com ponteiros, pois nem sempre sabemos o que está no endereço seguinte da memória. O novo endereço pode não pertencer à área na memória reservada para o programa. Isso seria o que chamamos de uma **falha de segmentação**.

Outro problema com este tipo de aritmética é que o próximo endereço da memória pode ter um dado de outro tipo, que não o do ponteiro. Isto impede o retorno do valor apontado.

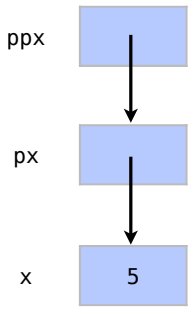
8.3 Ponteiros para ponteiros

Podemos criar ponteiros para ponteiros utilizando, mais uma vez, o operador `*`. No próximo exemplo, vamos ver como podemos criar ponteiros para ponteiros. Isso é feito utilizando mais de uma vez o operador para ponteiros.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     int x;
7     x = 5;
```

```
8     int *px;
9     int **ppx;
10    px = &x;
11    ppx = &px;
12    cout << "x = " << x << endl;
13    cout << "&x = " << &x << endl;
14    cout << "px = " << px << endl;
15    cout << "*px = " << *px << endl;
16    cout << "&px = " << &px << endl;
17    cout << "&ppx = " << &ppx << endl;
18    cout << "ppx = " << ppx << endl;
19    cout << "*ppx = " << *ppx << endl;
20    cout << "**ppx = " << **ppx << endl;
21    return 0;
22 }
```

Logo no início do programa, nas linhas 6 a 11, criamos x, um ponteiro px para dados do tipo `int`, e um ponteiro ppx para ponteiros para dados do tipo `int`. Inicializamos x com 5, px com o endereço de x e ppx com o endereço de px.



Como vimos nos exemplos anteriores, px possui seu próprio endereço e guarda o endereço de x. Assim as linhas 12 a 16 geram a seguinte saída:

```
x = 5
&x = 0x7fff55568b28
px = 0x7fff55568b28
*px = 5
&px = 0x7fff55568b20
```

Nas linhas 17 e 18, da mesma maneira, a variável ppx possui seu próprio endereço e um valor, que neste exemplo é o endereço de px, o mesmo que foi impresso com o operador de endereço em px (&px).

```
x = 5
&x = 0x7fff55568b28
px = 0x7fff55568b28
*px = 5
&px = 0x7fff55568b20
&ppx = 0x7fff55568b18
ppx = 0x7fff55568b20
```

Como valor apontado por ppx (*ppx), temos o valor de px, que é o endereço de x. Obtendo o valor apontado pelo valor apontado por ppx (**px), temos o valor apontado por px, que é o valor de x.

```
x = 5
&x = 0x7fff55568b28
px = 0x7fff55568b28
*px = 5
&px = 0x7fff55568b20
&ppx = 0x7fff55568b18
ppx = 0x7fff55568b20
*ppx = 0x7fff55568b28
**ppx = 5
```

8.4 Ponteiros e arranjos

Como vimos anteriormente, arranjos são endereços na memória onde começa um espaço alocado para uma sequência de dados do mesmo tipo. Ponteiros e arranjos estão muito relacionados, pois ponteiros que apontam para arranjos podem ter seus valores manipulados para acessar outros dados da mesma sequência de elementos.

Vamos supor um arranjo x de tamanho 5. Criaremos também um ponteiro px.

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
```

Sabemos que os valores de x estão em sequência na memória. Sabemos também que x é o endereço na memória onde começa esta sequência. Podemos então atribuir este endereço a px:

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = x;
```

Com x é o endereço onde começa a sequência de elementos. O comando acima seria equivalente a atribuir o endereço do primeiro elemento da sequência:

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = &x[0];
```

Em qualquer um dos casos, a Tabela 8.3 representa o resultado na memória destas operações. Os elementos do arranjo x estão nos endereços 25 a 29 da memória. O ponteiro px se encontra no endereço 30 e tem valor 25 que é o endereço do primeiro elemento do arranjo x.

Agora, se retornarmos o valor apontado por px, obteremos o valor de x[0]:

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = x;
4 cout << *px << endl;
```

```
5
```

Nomes das variáveis	Valores das Variáveis	Endereços das Posições
		24
x[0]	5	25
x[1]	3	26
x[2]	6	27
x[3]	7	28
x[4]	3	29
px	25	30
		31
		32
		33
		34
		35

Tabela 8.3: Um ponteiro apontando para o primeiro elemento de um arranjo.

Como x guarda elementos em posições contíguas, podemos fazer aritmética com o ponteiro para encontrar outros elementos do arranjo. Neste exemplo, retornamos o valor no endereço que está duas posições a frente de px.

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = x;
4 cout << *(px + 2) << endl;
```

```
6
```

Podemos até retornar o valor apontado por px com o operador de subscrito [e].

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = x;
4 cout << px[0] << endl;
```

```
5
```

Outros subscritos somam um número a px e procuram o que há em posições seguintes na memória.

```
1 int x[] = {5, 3, 6, 7, 3};
2 int *px;
3 px = x;
4 cout << px[3] << endl;
```

7

Os operações `px[3]` e `*(px + 3)` são então equivalentes. Elas retornam o valor no endereço 3 posições a frente de `&x[0]`. A utilização do operador de subscrito (`px[3]`), porém, é mais comum para arranjos.

É importante utilizar estes recursos com cuidado para não acessar posições onde não temos elementos.

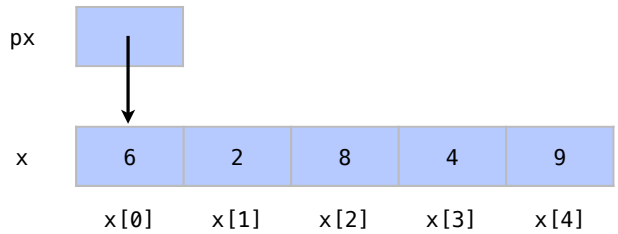
Apesar da relação entre ponteiros e arranjos, um ponteiro é uma variável que **guarda um endereço** na memória. Já um arranjo, **é um endereço** na memória, onde começa uma sequência de elementos.

Neste exemplo, vamos criar ponteiros que apontam para arranjos:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int x[5] = {6,2,8,4,9};
7      int *px;
8      px = x;
9      cout << "x = " << x << endl;
10     cout << "x[0] a x[4]: ";
11     for (int i=0; i<5; i++){
12      cout << x[i] << " ";
13     }
14     cout << endl;
15     cout << "px = " << px << endl;
16     cout << "*px = " << *px << endl;
17     cout << "px[0] = " << px[0] << endl;
18     cout << "px[3] = " << px[3] << endl;
19     cout << "*(px+3) = " << *(px+3) << endl;
20     cout << "*px++ = " << *px++ << endl;
21     cout << "px[3] = " << px[3] << endl;
22     return 0;
23 }
```

Criamos na linha 6 um arranjo de dados do tipo `int` chamado de `x` e um ponteiro para dados do tipo `int` chamado `px` na linha 7.

Na linha 8, `px` recebe o valor de `x`. Como `x` é um arranjo, e ele próprio é o endereço de seu primeiro elemento, `px` aponta para o primeiro elemento de `x`. Ou seja, `px` recebe `&x[0]`.



Se usarmos `x` diretamente, como feito na linha 9, vemos o endereço de onde começa a sequência de elementos do arranjo, ou seja, o endereço do primeiro elemento do arranjo

```
x = 0x7fff52294b10
```

O trecho de código seguinte, nas linhas 11 a 14, imprime todos os elementos do arranjo através de seus índices de 0 a 4.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
```

Na linha 15, o valor impresso de `px` é também o endereço do primeiro elemento do arranjo.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
```

Como `px` aponta para o primeiro elemento do arranjo, o valor apontado por `px` retorna `6` na linha 16.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
px = 6
```

Assim como nos arranjos, podemos retornar valores relacionados com o ponteiro com o operador de subscrito `px[i]`. O operador de subscrito com um número `i` diferente de zero, pega o elemento que estiver em `i` posições à frente do endereço apontado por `px`. Isso é o que ocorrre da linha 17 a 18.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
*px = 6
px[0] = 6
px[3] = 4
```

É preciso tomar cuidado com esta operação pois não podemos acessar elementos em posições que estão fora do arranjo.

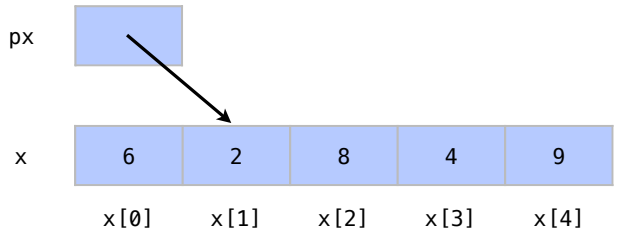
Conseguimos também o mesmo resultado com aritmética de ponteiros. Na linha 19, adicionamos ao endereço o tamanho de 3 elementos do tipo `int` e chegamos a `x[3]`.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
*px = 6
```

```
px[0] = 6
px[3] = 4
*(px + 3) = 4
```

A operação incremento em px na linha 20 (++px, faz px apontar para o próximo `int` na memória. Em seguida, o operador * retorna seu valor apontado. Veja que o valor do ponteiro px é alterado para o próximo elemento do arranjo.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
*px = 6
px[0] = 6
px[3] = 4
*(px + 3) = 4
*px++ = 6
```



Por fim, na linha 21, `px[3]` retorna o elemento 3 posições após a posição apontada por px, que agora é `x[4]`.

```
x = 0x7fff52294b10
x[0] a x[4]:  6 2 8 4 9
px = 0x7fff52294b10
*px = 6
px[0] = 6
px[3] = 4
*(px + 3) = 4
*px++ = 6
px[3] = 9
```

8.5 Exercícios

Exercício 8.1 Analise o código abaixo, que está incompleto. Coloque um `cout` antes de cada linha explicado o que foi feito neste comando.

```
1 // biblioteca padrão de entrada e saída
2 #include <iostream>
3
4 // espaço de nomes da biblioteca padrão (std)
5 using namespace std;
```

```
6
7  // função principal do programa
8  int main(){
9      int x;
10     int *px;
11     px = &x;
12
13     cout << "Digite um valor para x: " ;
14     cin >> x;
15
16     // Colocar um cout antes de cada linha
17     cout << x << endl;
18     cout << &x << endl;
19     cout << &px << endl;
20     cout << px << endl;
21     cout << *px << endl;
22
23     int v[5] = {4, 2, 3, 4, 5};
24     cout << v[0] << endl;
25     cout << v << endl;
26     cout << &v << endl;
27     cout << *v << endl;
28
29     px = v;
30     cout << px << endl;
31     cout << &px << endl;
32     cout << *px << endl;
33     cout << *(px+1) << endl;
34     cout << px[2] << endl;
35     px++;
36     cout << px[3] << endl;
37     cout << (*px)++ << endl;
38     cout << *(px++) << endl;
39
40     return 0;
41 }
```

9. Modularização de programas com funções

Veja como exemplo a letra desta música. Alguns conjuntos de estrofes são bem parecidos:

Ciranda Cirandinha

Ciranda, cirandinha
Vamos todos cirandar!
Vamos dar a meia volta
Volta e meia vamos dar

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

Por isso, dona Rosa
Entre dentro desta roda
Diga um verso bem bonito
Diga adeus e vá se embora

Ciranda, cirandinha
Vamos todos cirandar!
Vamos dar a meia volta
Volta e meia vamos dar

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

Ciranda, cirandinha
Vamos todos cirandar!
Vamos dar a meia volta
Volta e meia vamos dar

Veja agora a mesma letra de outra maneira. Os refrões que se repetem estão agrupados.

Ciranda Cirandinha

REFRÃO:

Ciranda, cirandinha
Vamos todos cirandar!
Vamos dar a meia volta

Volta e meia vamos dar

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

Por isso, dona Rosa
Entre dentro desta roda
Diga um verso bem bonito

Diga adeus e vá se embora

REFRÃO

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

REFRÃO

Se estamos seguindo a letra de uma música, é claro não cantamos a palavra “refrão"quando a vemos. Quando vemos a palavra “refrão", voltamos onde o refrão foi definido e cantamos aquela parte.

Isto nos dá um grande poder de síntese para expressar ideias. Podemos explicar a letra de uma música de maneira mais sucinta.

Outra possibilidade, é definir todas as partes no início da música. Depois, podemos cantar a música completa usando as definições que fizemos.

Ciranda Cirandinha

Partes:
REFRÃO:
Ciranda, cirandinha
Vamos todos cirandar!
Vamos dar a meia volta
Volta e meia vamos dar

Música:
REFRÃO

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

Por isso, dona Rosa
Entre dentro desta roda
Diga um verso bem bonito
Diga adeus e vá se embora

REFRÃO

O anel que tu me destes
Era vidro e se quebrou
O amor que tu me tinhas
Era pouco e se acabou

REFRÃO

É de maneira análoga que funções de um programa podem ser definidas e depois chamadas.

Funções são unidades autônomas de código, que cumprem uma tarefa particular. Os programas em C++ são formados por funções. Por enquanto, estivemos colocando todo o código em nossa função principal `main`. Porém, programas que resolvem problemas maiores são usualmente divididos em mais funções.

9.1 Funções simples

As funções são muito úteis para dividir problemas em problemas menores. Tarefas que são repetidas várias vezes podem ser colocadas em funções para simplificar o código. Para que uma função seja utilizada na função `main`, ela precisa já ter sido declarada para que seja reconhecida.

9.1.1 Minha primeira função

Neste exemplo, vamos criar uma função de exemplo e a utilizaremos em nossa função principal. Perceba que a função `exemplo` é criada antes da função `main` no seguinte exemplo:

```
1  #include <iostream>
2
3  using namespace std;
4
5  void exemplo() {
6      cout << "Olá ";
7      cout << "Mundo!";
8      cout << endl;
9  }
10
11 int main() {
12     cout << "Exemplo de função simples" << endl;
13     exemplo();
14     exemplo();
15     return 0;
16 }
```

Na linha 5, a palavra chave `void`, na frente da função `exemplo`, indica que esta função não retorna nenhum valor. Apesar disso, note que, a que a sintaxe das duas funções é muito similar: um nome de função, parênteses e um bloco de comandos delimitado por chaves.

Assim como em qualquer programa, a sua execução se inicia na função principal `main`, definida nas linhas 11 a 16. A linha 12 da função `main` imprime a seguinte mensagem:

```
Exemplo de funções simples
```

Na linha 13, a função `exemplo` é chamada. Isso faz com que a função `main` fique em espera até que os comandos da função `exemplo` sejam executados. Os comandos da função exemplo, nas linhas 5 a 9, imprimem um `Olá Mundo!` com três comandos separados, nas linhas 6, 7 e 8.

```
Exemplo de funções simples
Olá Mundo!
```

Ao fim da execução da função, voltamos para a função que a chamou. Ou seja, a função `main` sai de espera na linha 13. O próximo comando do `main`, na linha 14, é mais uma chamada à função `exemplo`. Veja que a função é executada mais uma vez.

```
Exemplo de funções simples
Olá Mundo!
Olá Mundo!
```

Retornamos à função main na linha 14. Este exemplo nos mostrou como criar novas funções com blocos de código que podem ser chamados. Isso pode nos ajudar a organizar um problema em problemas bem menores. Mas para isso, nossas funções precisam ser declaradas antes da função `main` para que possam ser reconhecidas.

9.2 Cabeçalhos de função

Por questão de organização, é usualmente melhor analisar a função `main` antes de analisar outras funções. Assim, seria bom poder definir as funções no código após a função `main`. E isso é possível através do uso de cabeçalhos.

Podemos refazer o exemplo anterior com um cabeçalho para nossa função `exemplo`:

```
1  #include <iostream>
2
3  using namespace std;
4
5  void exemplo();
6
7  int main() {
8      cout << "Exemplo de função simples" << endl;
9      exemplo();
10     exemplo();
11     return 0;
12 }
13
14 void exemplo() {
15     cout << "Olá ";
16     cout << "Mundo!";
17     cout << endl;
18 }
```

Na linha 5, o cabeçalho **declara** a função `exemplo`, de modo que a função `main` a reconheça. Após a função `main`, a partir da linha 14, **definimos** a função `exemplo`, já declarada no cabeçalho. Agora é possível usar a definição a partir da função `main` mesmo antes da definição da função aparecer no código. Por questões de organização, este é o modo mais usual de se organizar funções.

9.3 Funções e suas variáveis

Assim como a função `main`, nossas funções podem ter suas próprias variáveis. Devido ao escopo de bloco, as funções não terão acesso às variáveis de outras funções, nem mesmo às variáveis da função chamadora. Por isso, não há problema algum em se repetir identificadores usados na função chamadora. Isso facilita o reaproveitamento de código, já que não precisamos nos preocupar com estas repetições.

Neste exemplo, vamos criar uma função que tem suas próprias variáveis.

```
1  #include <iostream>
2
3  using namespace std;
4
5  void soma();
6
7  int main() {
8      int x;
9      int y;
10     x = 2;
11     y = 6;
```

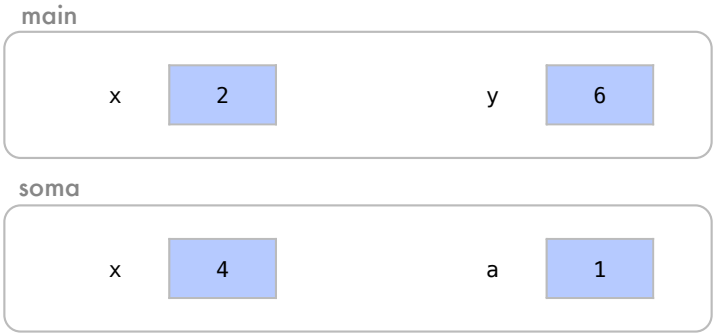
```
12      cout << "x+y␣=␣" << x + y << endl;
13      soma();
14      cout << "x+y␣=␣" << x + y << endl;
15      return 0;
16  }
17
18  void soma() {
19      int x;
20      int a;
21      x = 4;
22      a = 1;
23      cout << "x+a␣=␣" << x + a << endl;
24  }
```

Na função principal `main`, nas linhas 8 a 11, criamos duas variáveis `x` e `y`, com valores 2 e 6. A soma dessas variáveis é impressa normalmente na linha 12 da função `main`:

```
x + y = 8
```

Repare que, na linha 13, a função `main` chama a função `soma` e fica em espera até que ela termine.

A função `soma`, definida nas linhas 18 a 24, também cria duas variáveis em seu próprio escopo, nas linhas 19 e 20. Nas linhas 21 e 22, estas variáveis recebem os valores 4 e 1. Cada uma das duas funções, `main` e `soma`, têm uma variável chamada `x`. Note que não há conflito entre os identificadores pois são escopos completamente diferentes.



Ainda por causa do escopo separado das variáveis, a função `soma` não tem acesso às variáveis da função principal `main` e vice-versa. Agora, na linha 23, `soma` imprime `x + a`, de acordo com as variáveis de seu escopo, resultando em 5.

```
x + y = 8
x + a = 5
```

Retornamos então, à função `main` e as variáveis de `soma` deixam de existir.

Perceba que, na linha 14, `x + y` é novamente impresso como `8`, pois estamos nos referindo às variáveis do `main`.

```
x + y = 8
x + a = 5
x + y = 8
```

Repare como as funções podem dividir problemas em subproblemas independentes.

9.4 Retorno de valor

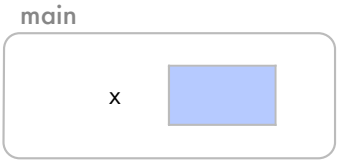
Colocamos a palavra chave `void` antes das funções que criamos até agora. Isto indica que estas funções não retornam nada. Já a palavra chave `int` antes da função `main` indica que ela retorna um número inteiro. Este retorno acontece no comando `return 0`, que finaliza o programa. O comando `return 0` é usado para indicar que o programa encerrou sem nenhum erro.

Podemos fazer com que funções retornem valores de qualquer tipo. E esse valor retornado pode ser utilizado na função chamadora como um dado daquele tipo. Isto é muito útil pois o dado pode ser diretamente utilizado ou ser guardado em uma variável daquele tipo.

Neste exemplo vamos criar uma função que retorna um dado do tipo `double`.

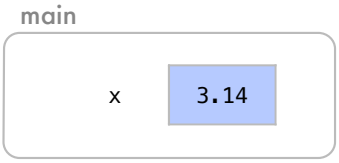
```
1  #include <iostream>
2
3  using namespace std;
4
5  double pi();
6
7  int main() {
8      double x;
9      x = pi();
10     cout << "O valor de pi é " << x << endl;
11     cout << "O valor de pi é " << pi() << endl;
12     return 0;
13 }
14
15 double pi() {
16     return 3.14;
17 }
```

Na linha 8, criamos um `double x` no escopo do `main`.



Veja que, na linha 9, `x` recebe o que for retornado pela função `pi`. Como usual, a função `main` fica em espera enquanto é executada a função `pi`.

Observe que não criamos nenhuma variável na função `pi`, que inicia na linha 15. O primeiro comando de `pi`, na linha 16, já retorna o valor 3.14. Este valor, é por sua vez capturado pela função `main`, que a chamou. Assim, a função `main` volta a ser executada na linha 9, onde o valor `3.14`, retornado por `pi`, é atribuído a `x`.



O valor retornado pela função `pi` é atribuído a `x` na linha 9 e o comando de impressão `cout` na linha 10 funciona normalmente com `x`.

```
O valor de pi é 3.14
```

Em um segundo `cout`, na linha 11, o valor de 3.14 é retornado direto da função para o comando de impressão.

```
O valor de pi é 3.14
O valor de pi é 3.14
```

Nesse exemplo, tivemos uma função que retornava o valor de π. Funções específicas permitem a outras funções chamadoras tomar este subproblema como resolvido. Assim, só precisamos utilizar a função a onde a solução do problema for esperada.

9.5 Parâmetros de função

Imagine agora, que você tem a tarefa de comprar um microondas, como na Figura 9.1. Pense nisto como uma função `comprar`.

No exemplo da Figura, se você pensa depois em comprar qualquer outro tipo de eletrodoméstico, como uma TV ou um rádio, isso é muito fácil, pois como você já comprou um micro-ondas, já sabe como ***comprar*** qualquer outro eletrodoméstico. O procedimento de compra não é diferente para outro produto. Neste caso, é como se nossa função fosse a ação de comprar algo. Ela sempre funciona da mesma maneira. Enquanto isso, nosso **parâmetro** é o que vamos comprar. Usamos a mesma função para diferentes parâmetros.

Perceba como o recurso de parâmetros aumentou o nosso poder de síntese. As funções são frequentemente utilizadas para resolver subproblemas. Mas assim como em nossos exemplos, subproblemas às vezes dependem de alguma informação sobre o problema para poderem ser resolvidos. Ou seja, na hora de comprar algo, você precisava saber o que irá comprar.

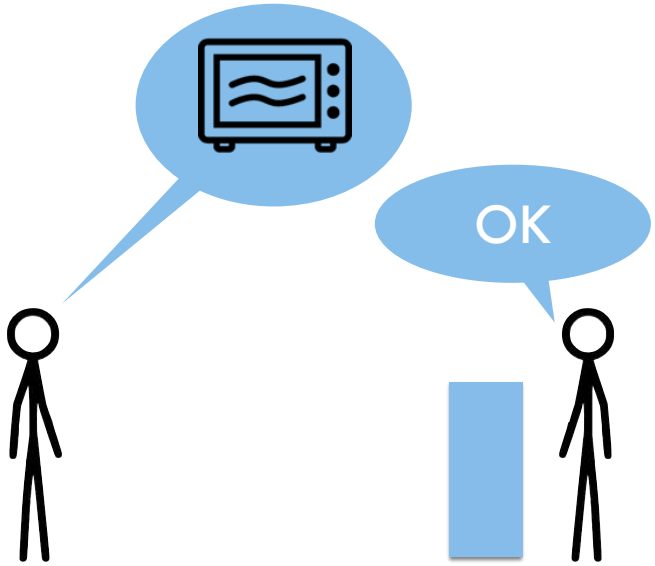


Figura 9.1: Exemplo de uma ação de *comprar*.

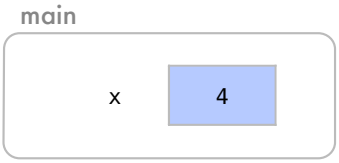
Este tipo de informação pode ser passado para as funções através dos parâmetros. Em C++, esses parâmetros são passados para a função entre parênteses `()`, após o identificador da função.

Neste exemplo, vamos criar uma função que eleva um número qualquer ao cubo. Para isto a função precisa saber qual número elevará ao cubo.

```
1  #include <iostream>
2
3  using namespace std;
4
5  double cubo(double a);
6
7  int main() {
8      int x;
9      cout << "Digite␣um␣número:␣";
10     cin >> x;
11     cout << x << "␣ao␣cubo␣é␣" << cubo(x) << endl;
12     return 0;
13 }
14
15 double cubo(double a) {
16     return a * a * a;
17 }
```

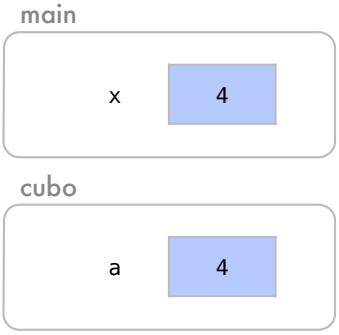
Esta função dependerá então de qual número queremos elevar ao cubo e para isso usaremos um parâmetro na função `cubo`, representado por `a` na linha 5. Na função principal, nas linhas 8 a 10, criamos e inicializamos uma variável `x` através de uma entrada do usuário.

```
Digite um número:  4
```



Imprimimos agora, na linha 11, o valor de `x` e o valor de `x` ao cubo. Não temos uma variável com o valor de `x` ao cubo, mas a função cubo tem esta capacidade de fazer este cálculo e retornar o resultado.

A função `main` fica em espera e a função `cubo(x)` é executada. A função se inicia na linha 15, onde a variável `a` é uma cópia da variável `x`, enviada por `main` ao chamar a função.



Esta função `cubo` retorna na linha 16 o valor de `a * a * a`, ou seja, `64`. Com o fim da função `cubo`, `main` volta ser executada na linha 11, e a chamada de `cubo(x)` é substituída pelo valor retornado `64`.

```
Digite um número:  4
4 ao cubo é 64
```

Note que mesmo sem ter o valor de `x` ao cubo em nenhuma variável, podemos dividir o problema em subproblemas pois sabemos que há uma função que resolve isso e retorna o que queremos. Note também que para calcular o valor de `x` ao cubo, o valor de `x` precisou ser enviado à função como parâmetro.

A função `cubo`, se iniciou com uma variável `a`. Esta variável é um parâmetro da função. O valor do parâmetro `a` foi enviado pela função chamadora, e é neste caso uma cópia de `x`. Como são dois escopos diferentes, note que não haveria problema algum se `a` também se chamasse `x`. Repare também que `a`, é nesse caso apenas uma cópia de `x` e qualquer alteração em `a` não altera o `x` original.

O único comando da função cubo retornou `a * a * a` (ou a^3). Ao término da função `cubo`, a função `main` voltar a ser executada com o valor que foi retornado. Com o valor retornado 64, essa foi a mensagem impressa acima.

Os parâmetros aumentam muito a utilidade das funções e nos ajuda mais ainda a dividir problemas em subproblemas.

É possível aumentar mais ainda a utilidade de funções utilizando mais de um parâmetro. Neste segundo exemplo, criaremos uma função que eleva um número base a um outro número que representa um expoente inteiro.

```
1 #include <iostream>
2
```

```
3  using namespace std;
4
5  double eleva(double base, int exp);
6
7  int main() {
8     double x;
9     int y;
10    cout << "Digite␣a␣base:␣";
11    cin >> x;
12    cout << "Digite␣o␣expoente:␣";
13    cin >> y;
14    cout << x << "␣elevado␣a␣" << y << "␣=␣";
15    cout << eleva(x,y) << endl;
16    return 0;
17 }
18
19 double eleva(double base, int exp) {
20    double resultado;
21    resultado = 1;
22    for (int i=0; i<exp; ++i){
23     resultado *= base;
24    }
25    return resultado;
26 }
```

Antes da função principal, na linha 5, declaramos a função `eleva`, que elevará qualquer número real `base` a qualquer expoente inteiro não negativo `exp`.

Inicializamos no `main`, nas linhas 8 a 13 as variáveis `x` e `y` onde o usuário guardará uma base e um expoente.

```
Digite a base:  4.4
Digite o expoente:  5
```

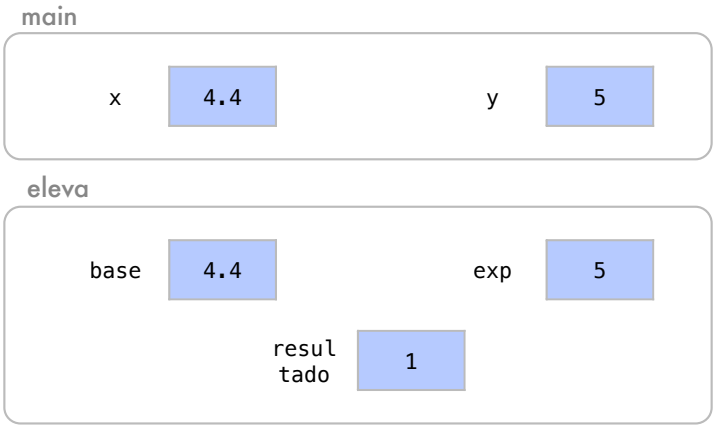
Na linha 14, uma mensagem é impressa parcialmente:

```
Digite a base:  4.4
Digite o expoente:  5
4.4 elevado a 5 =
```

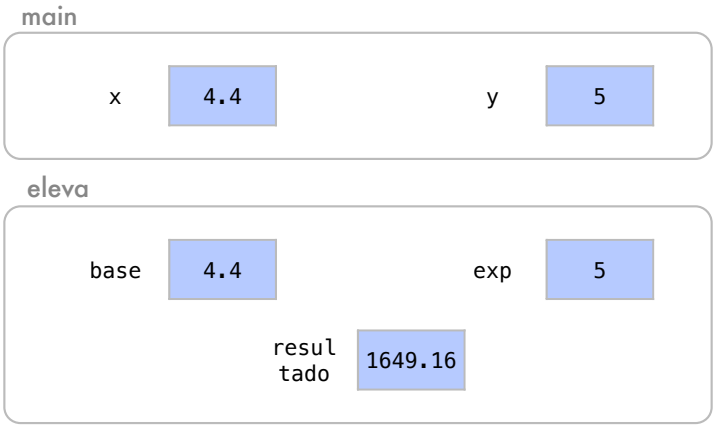
Não há um `endl` pois o resultado da expressão ainda será impresso nesta linha.

Na linha 15, o comando com `cout` precisa do resultado da função `eleva`, e por isso, a função `main` fica em espera. Veja que a função `eleva` se inicia com uma variável chamada `base`, que é uma cópia de `x`, e `exp`, que é uma cópia de `y`.

Na linha 20, criamos para a função também uma variável, `resultado`, onde será calculado o resultado. Ela se inicia com `1` na linha 21.



Nas linhas 22 a 24, temos uma estrutura de repetição. O `resultado` é, a cada iteração, multiplicado por `base` na linha 23. Com uma estrutura de repetição, fazemos com este processo se repita `exp` vezes.



Por fim, na linha 25, o valor de `resultado` é retornado por `eleva` para a função chamadora. Com isso, a função `eleva` é encerrada e a função `main` volta a executar com o valor retornado, que é `base` elevado a `exp`. Isso permite que a mensagem seja impressa na linha 15.

```
Digite a base:  4.4
Digite o expoente:  5
4.4 elevado a 5 = 1649.1622
```

9.6 Passagem de parâmetros

Existem duas maneiras de se passar parâmetros para uma função. Podemos passar nossos parâmetros por **valor** ou por **referência**.

9.6.1 Passagem de parâmetros por valor

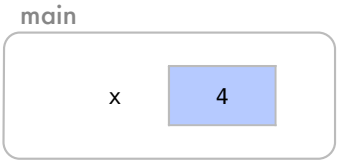
A passagem de parâmetros por valor é o modo que utilizamos até agora. Neste modo de passagem de parâmetros, a função recebe uma cópia da variável enviada pela função chamadora. No fim do processo as cópias são eliminadas.

Temos novamente um exemplo de função que calcula um número ao cubo.

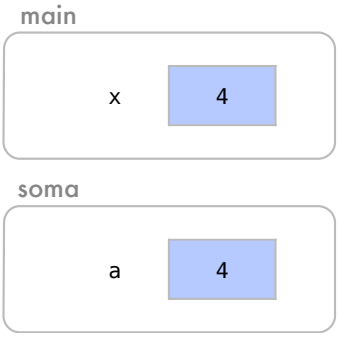
```
1  #include <iostream>
2
3  using namespace std;
4
5  double cubo(double a);
6
7  int main() {
8     double x;
9     cout << "Digite um número: ";
10    cin >> x;
11    cout << x << " ao cubo é ";
12    cout << cubo(x) << endl;
13    cout << "x = " << x << endl;
14    return 0;
15 }
16
17 double cubo(double a) {
18    a = a * a * a;
19    return a;
20 }
```

Nas linhas 8 a 10, inicializamos uma variável chamada `x` e damos um valor a ela. Na linha 11, imprimimos uma mensagem parcial, sem um `endl`:

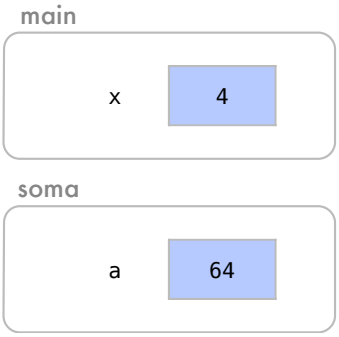
```
Digite um número:  4
4 ao cubo é
```



Como, na linha 12, `cout` depende de `cubo`, a função `main` fica em espera. Já dentro da função `cubo`, note que `a` é, neste caso, apenas uma cópia do valor de `x` e alterações em `a` não alterarão `x`.



Na linha 13, alteramos o valor de `a` pelo valor de `a * a * a`, ou `a` ao cubo. Veja como isto não influencia o valor original de `x`.



O valor de `a`, ou `64`, é então, retornado na linha 18. A função é encerrada e `main` volta à execução na linha 12 com o valor retornado.

```
Digite um número:  4
4 ao cubo é 64
```

Na linha 13, o valor de `x` é impresso novamente.

```
Digite um número:  4
4 ao cubo é 64
x = 4
```

Note como o valor de `x` não foi alterado durante a execução do programa.

9.6.2 Passagem de parâmetros por referência

Na passagem de parâmetros por referência, a função recebe apenas uma referência para a variável da função chamadora. Assim, a variável na função chamada é apenas um apelido para a variável da função chamadora. Então, neste caso, não precisamos copiar todos os dados para executar a função chamadora.

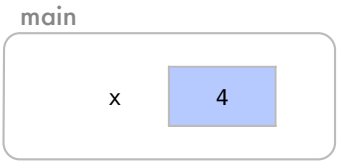
Neste caso, precisamos lembrar que alterações na variável da função alteram também a variável da função chamadora. Para indicar que o parâmetro será passado por referência, utiliza-se um e comercial (`&`).

Neste exemplo, enviaremos uma variável por referência para uma função que calcula um valor ao cubo.

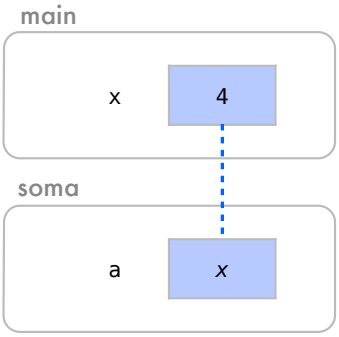
```
1  #include <iostream>
2
3  using namespace std;
4
5  double cubo(double &a);
6
7  int main() {
8     double x;
9     cout << "Digite␣um␣número:␣";
10    cin >> x;
11    cout << x << "␣ao␣cubo␣é␣";
12    cout << cubo(x) << endl;
13    cout << "x␣=␣" << x << endl;
14    return 0;
15 }
16
17 double cubo(double &a) {
18    a = a * a * a;
19    return a;
20 }
```

Ao declararmos a função `cubo`, na linha 5, o "e"comercial (`&`) indica que a função recebe a variável `a` por referência. Nas linhas 8 a 10, temos a inicialização da variável `x`, de `main`. Na linha 11, imprimimos uma mensagem parcial sem um `endl`. Como `cout` depende da função `cubo`, a função `main` fica em espera.

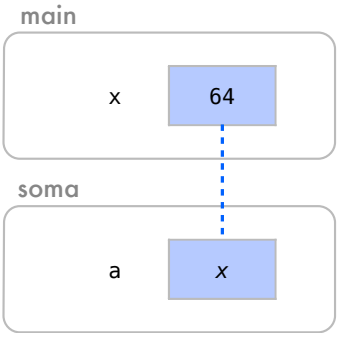
```
Digite um número:  4
4 ao cubo é
```



Já na função `cubo`, na linha 17, variável `a` foi passada por referência e por isto é agora apenas um apelido para o `x`, que tem como valor o `4` na função chamadora. Se alterarmos algo em `a`, neste momento, alteraremos também o `x`.

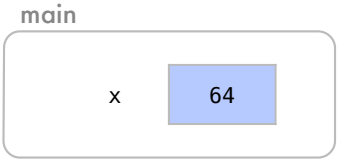


Como `a` é só um apelido para o `x` da função `main`, então na linha 18, quando `a` recebe `a * a * a`, isto ocorre diretamente em `x`.



O valor de a, na linha 19, 64, é então retornado para o `cout` da função `main`, na linha 12. A função cubo é encerrada e `main` volta à execução com o valor retornado, na linha 12.

```
Digite um número:  4
4 ao cubo é 64
```



O valor de x é impresso na linha 13. Note como durante a execução do programa o valor de x foi alterado.

```
Digite um número:  4
4 ao cubo é 64
x = 4
```

9.7 Omitindo a variável de retorno

Como a passagem de parâmetros por referência altera diretamente as variáveis da função chamadora, ela pode ser criada justamente para alterar algumas variáveis. Essas funções podem ter o tipo de retorno `void`.

Neste exemplo, usaremos uma função apenas para alterar o valor de uma variável. Após a execução da função, a variável enviada para a função terá seu valor ao cubo.

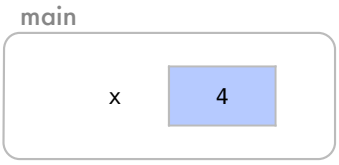
```
1  #include <iostream>
2
3  using namespace std;
4
5  void cubo(double &a);
6
7  int main() {
8     double x;
9     cout << "Digite um número: ";
10    cin >> x;
11    cout << x << " ao cubo é ";
```

```
12      cubo(x);
13      cout << x << endl;
14      return 0;
15  }
16
17  void cubo(double &a) {
18      a = a * a * a;
19  }
```

Declaramos a função `cubo` na linha 5, onde o "e"comercial (`&`) indica que a função recebe a variável `a` por referência. Dessa vez, a função não retorna nada e isto é indicado com `void`.

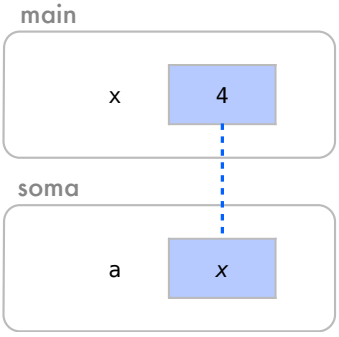
É feita a criação e inicialização da variável `x` nas linhas 8 a 10 de `main`. Na linha 11, imprimimos uma mensagem parcial:

```
Digite um número:  4
4 ao cubo é
```

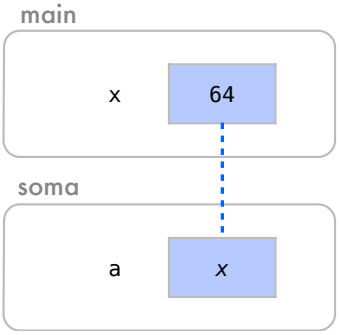


A função `cubo` é chamada na linha 12, para alterar o valor de `x`, enquanto a função `main` fica em espera. A chamada da função não pode estar em um comando de impressão pois a função não retorna valor algum.

Na função `cubo`, `a` é agora uma referência para o `x` da função chamadora e alterações em `a` farão com que `x` seja alterado, pois `a` é apenas um apelido para `x`.



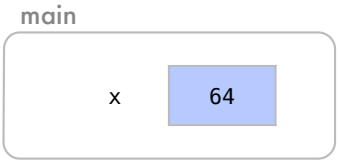
Por este motivo, na linha 18, `a` (ou `x`) tem seu valor alterado para `64`.



Por fim, a função cubo é encerrada e, apesar de não retornar nada, o valor de x foi alterado.

A função main retorna na linha 13 e repare que o novo valor de x é impresso ainda na mesma linha.

```
Digite um núnero:  4
4 ao cubo é 64
```



9.8 Retornando vários valores

Cada função pode retornar apenas um elemento de um certo tipo. No exemplo anterior, utilizamos a passagem de parâmetros por referência para influenciar a função chamadora sem precisarmos utilizar o recurso de retorno. Quando quisermos alterar mais de um elemento em uma função, este é também o recurso mais utilizado para realizarmos esta tarefa.

Neste exemplo, temos uma função maxmin retorna o mínimo e o máximo entre três números.

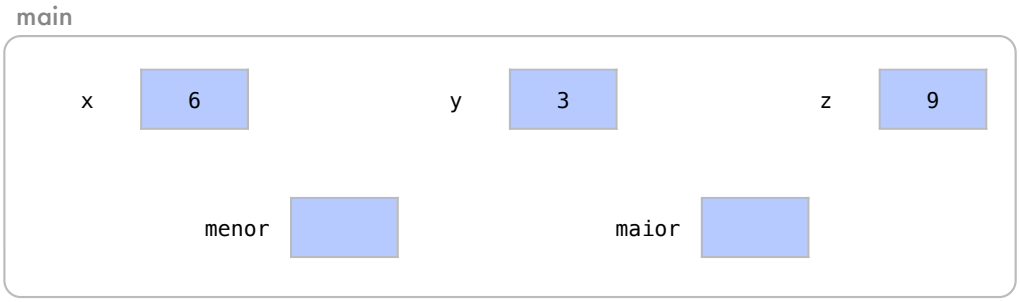
```
1  #include <iostream>
2
3  using namespace std;
4
5  void maxmin(int a, int b, int c, int &minimo, int &maximo);
6
7  int main() {
8     int x, y, z, menor, maior;
9     cout << "Digite 3 números: ";
10    cin >> x >> y >> z;
11    maxmin(x, y, z, menor, maior);
12    cout << "O menor é " << menor << endl;
13    cout << "O maior é " << maior << endl;
14    return 0;
15 }
16
17 void maxmin(int a, int b, int c, int &minimo, int &maximo) {
18    if (a > b){
19      minimo = b;
20      maximo = a;
21    }
22    else {
23      minimo = a;
24      maximo = b;
25    }
26    if (c < minimo){
27      minimo = c;
28    }
29    if (c > maximo){
```

```
30        maximo = c;
31     }
32  }
```

Na linha 5, antes da função principal `main`, declaramos então a função `maxmin`, que tem dois parâmetros passados por referência. Como queremos retornar dois valores da função, as variáveis `int minimo` e `int maximo` são passadas por referência para guardar os resultados.

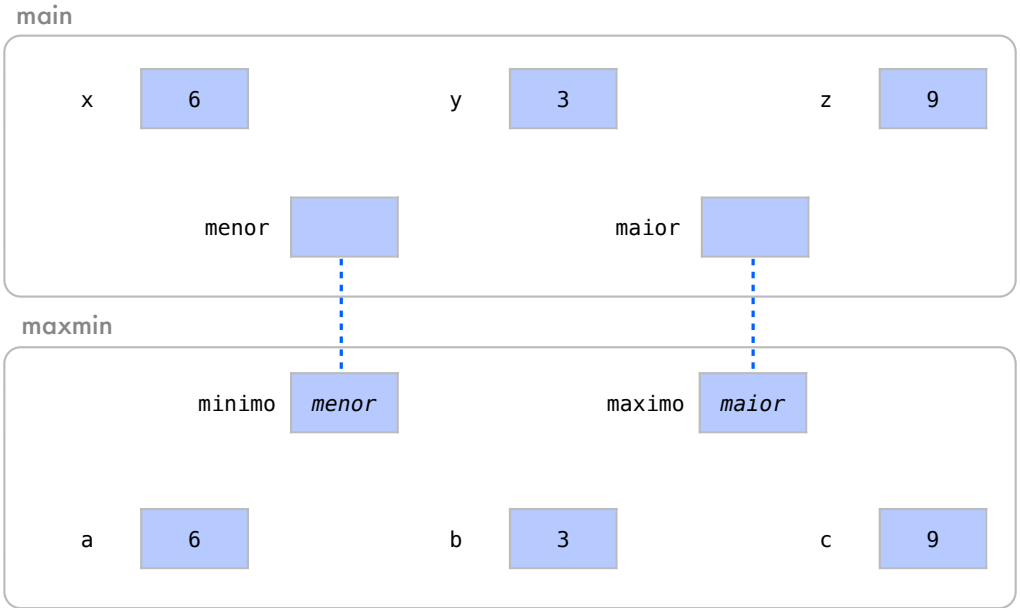
Sendo assim, na linha8, criamos 5 variáveis no `main`. Perceba que os valores de `x`, `y` e `z` são dados pelo usuário nas linha 9 e 10. As variáveis `menor` e `maior` ficam sem valores definidos por enquanto pois elas guardarão os resultados.

```
Digite 3 números:  6 3 9
```



Os valores de `menor` e `maior` serão dados pela função `maxmin`. A função `main`, na linha 11, fica então em espera até que `maxmin` seja executada.

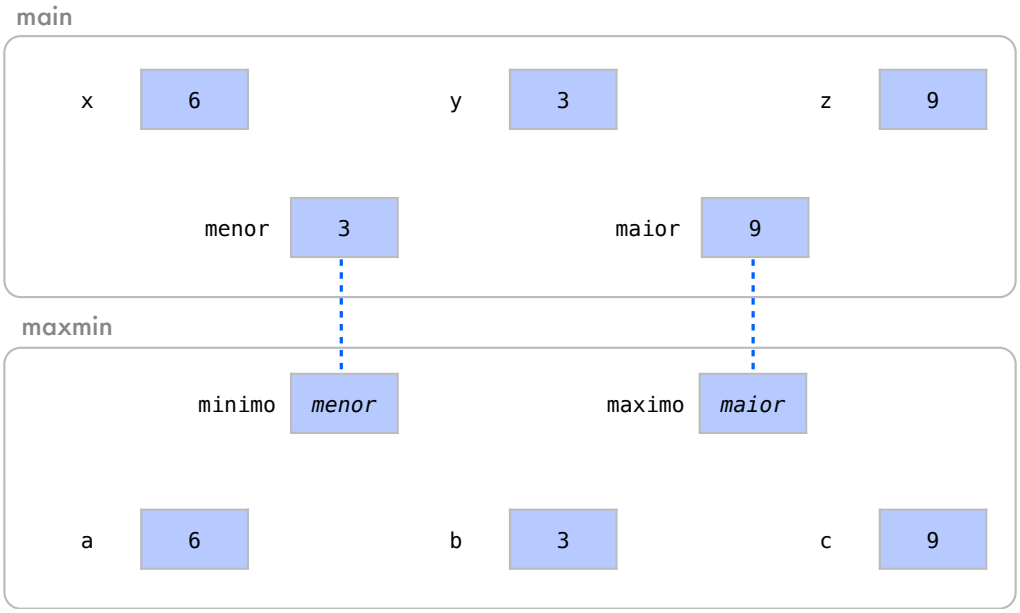
Na função `maxmin`, `a`, `b` e `c` são recebidos por valor de `x`, `y` e `z`, respectivamente. Enquanto isso `minimo` e `maximo` são referências para `menor` e `maior`, da função chamadora.



Exemplificamos agora o funcionamento da função `maxmin`. Na linha 18, testamos entre `a` e `b` qual a variável de maior valor. Como `a > b` é verdadeiro, nas linhas 19 e 20, `minimo` recebe o valor de `b` e `maximo` recebe o valor de `a`.

Nas linhas 26 e 29, comparamos `c` com o maior e menor valores já encontrados. Na linha 26, a condição `c < minimo` é falsa, o que quer dizer que `minimo` já tem o maior valor entre os três números. Mas, como na linha 29 a condição `c > maximo` é verdadeira, significa então que `c` é ainda maior que `a`, que foi o valor atribuído a `maximo` na outra condição. Portanto, `maximo`, recebe o valor de `c` na linha 30.

Neste ponto, a função `maxmin` termina sem retornar nada, mas os valores de `menor` e `maior` do `main` foram modificados corretamente pela função.



Ao retornarmos à execução da função `main`, na linha 11, os resultados são impressos nas linhas 12 e 13.

```
Digite 3 números:  6 3 9
O menor é 3
O maior é 9
```

Vimos vários exemplos onde parâmetros são passados por valor e por referência para uma função. Do ponto de vista de resultados, se uma variável não será alterada dentro da função, as passagens por valor ou por referência tem, exatamente, os mesmos resultados. Porém, do ponto de vista de eficiência, se uma variável não é alterada dentro da função, a passagem por referência é usualmente mais eficiente pois não é necessário o procedimento de cópia.

9.9 Arranjos como parâmetros

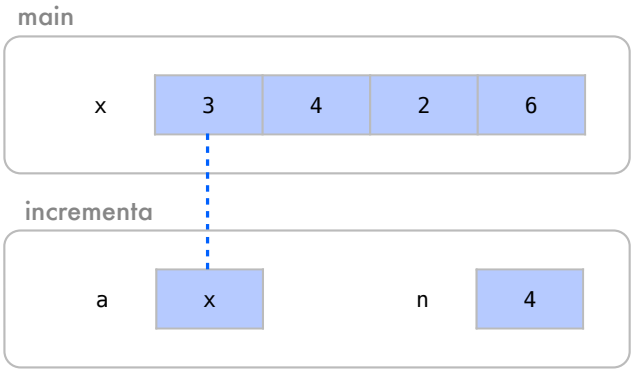
Como já vimos anteriormente, arranjos são endereços de memória onde se inicia uma sequência de valores do mesmo tipo. Por esse motivo, arranjos são sempre passados por referência já que, mesmo em caso de cópia, apenas o endereço do arranjo é passado como parâmetro, e não seus elementos. Como eles são passados sempre por referência, não é necessário utilizar o "e"comercial (`&`) para passar arranjos como referência. Se quisermos passar uma cópia de todos os valores no arranjo, uma cópia destes elementos deve ser feita explicitamente ainda na função chamadora.

Neste exemplo, criaremos uma função que incrementa em 1 cada elemento do arranjo `a`.

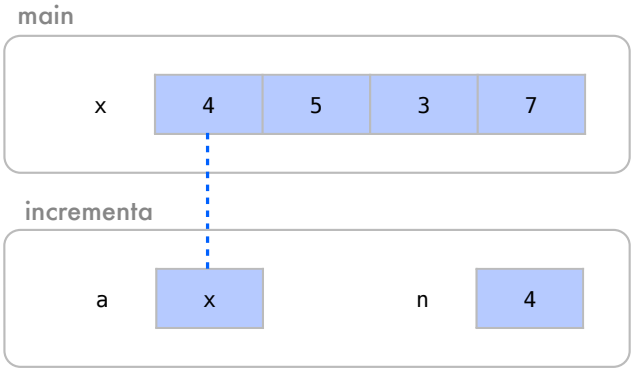
```
1  #include <iostream>
2
3  using namespace std;
4
5  void incrementa(int a[], int n);
6
7  int main() {
8     int x[] = {3,4,2,6};
9     incrementa(x,4);
10    cout << x[0] << "␣";
11    cout << x[1] << "␣";
12    cout << x[2] << "␣";
13    cout << x[3] << "␣";
14    return 0;
15 }
16
17 void incrementa(int a[], int n) {
18    for (int i = 0; i < n; ++i){
19      a[i]++;
20    }
21 }
```

Na linha 5, no cabeçalho da função, o arranjo será passado por referência, mesmo sem ter um “e"comercial (`&`). O tamanho `n` do arranjo é passado por valor.

Na função principal `main`, linha 8, criamos um arranjo de tamanho 4. Já na linha 9, a função `main` fica em espera para a execução da função `incrementa`. Veja que, na função incrementa, `a` recebe o endereço do arranjo `x` por referência e `n` recebe `4` por valor.



No laço definido pelas linhas 18 a 20 da função, cada elemento do arranjo `a` é incrementado em 1. Assim, a função `incrementa` termina sem retornar nada e o arranjo `x` agora tem seus elementos incrementados.



A função `main` volta a ser executada na linha 10. Os elementos do arranjo x são impressos no `main`, da linha 10 ate a 13.

```
4 5 3 7
```

Note como o arranjo foi passado por referência sem o "e" comercial (`&`).

9.10 Ponteiros como parâmetros

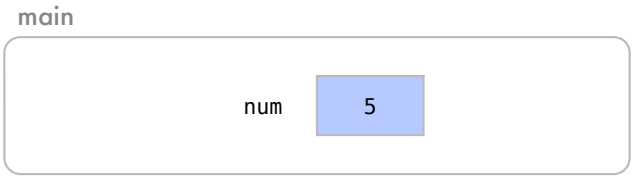
A passagem de variáveis por referência não é a única maneira de uma função alterar valores da função chamadora. Imagine um ponteiro com o endereço de uma variável da função chamadora. Se esse ponteiro é passado para uma função, ele ainda aponta na memória para a variável da função chamadora.

Neste exemplo, utilizaremos ponteiros para alterar variáveis da função chamadora.

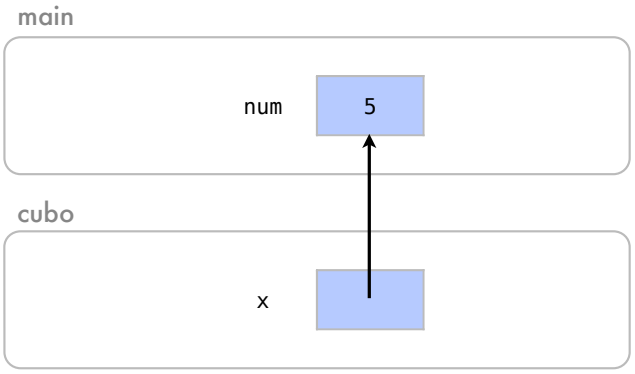
```
1  #include <iostream>
2
3  using namespace std;
4
5  void cubo(int *x);
6
7  int main() {
8      int num;
9      cout << "Digite um número: ";
10     cin >> num;
11     cout << num << " ao cubo é ";
12     cubo(&num);
13     cout << num << endl;
14     return 0;
15 }
16
17 void cubo(int *x) {
18     *x = (*x) * (*x) * (*x);
19 }
```

Declaramos a função `cubo` na linha 5, que recebe um ponteiro para um `int`. A função elevará o valor apontado ao cubo. Inicializamos, nas linhas 8 a 10 do `main`, uma variável chamada `num`. Imprimimos uma mensagem parcial na linha 11:

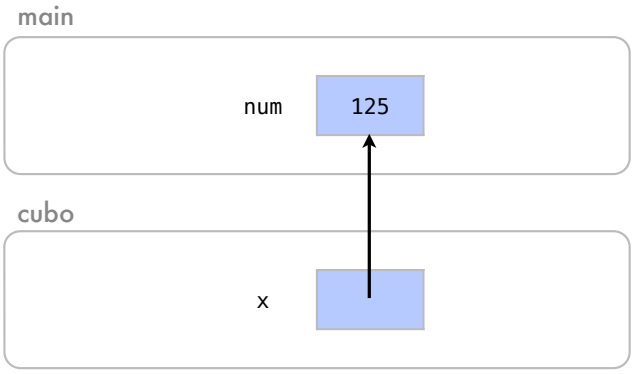
```
Digite um número:  5
5 ao cubo é
```



Logo após, veja que a função `main`, na linha 12, fica em espera até que a função `cubo` seja executada. Note que a variável `num` não pode ser enviada diretamente para `cubo`, pois `cubo` espera um ponteiro para um `int` e não um `int`. Mas, como ponteiros guardam endereços de variáveis na memória, a função `main` envia para a função `cubo` o endereço da variável `num`. Na função `cubo`, como `x` tem o endereço da variável `num`, isso faz com que `x` aponte para `num`. Isso acontece mesmo com o endereço tendo sido passado por valor.

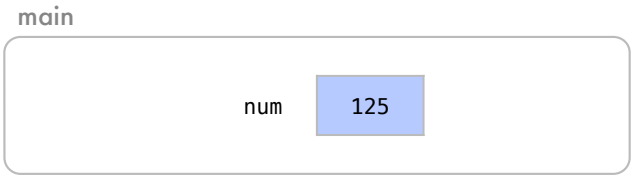


Quando acessamos o valor apontado por `x`, como na linha 18 do código, trabalhamos com a variável `num` da função chamadora, que é neste caso elevada ao cubo.



A função `cubo` encerra sem retornar nada e a função `main` retoma a execução com o valor da variável `num` alterado. Imprimimos, enfim, na linha 13, o valor alterado de `num` ainda na mesma linha.

```
Digite um número:  5
5 ao cubo é 125
```



A passagem de ponteiros por valor era utilizada especialmente em C. Isso ocorria, pois C não tinha o recurso de se passar parâmetros por referência com um &. Sendo assim, este era o único modo de se alterar valores da função chamadora. Em C++, este é um recurso que pode menos interessante já que precisamos saber na função chamadora se um ponteiro é esperado pela função chamada. De qualquer maneira, há uma grande quantidade de código ainda disponível que utiliza este recurso, que precisa ser conhecido.

9.11 Recursão

Normalmente, organizamos um programa de maneira hierárquica, onde algumas funções chamam outras. Já uma função recursiva, é uma função que direta ou indiretamente chama a si mesma.

Quando funções recursivas são chamadas, usualmente, elas só sabem resolver problemas simples, chamados de **caso base**. Se ela é chamada para um problema mais complexo, ela o divide em partes menores, partes que sabe resolver e partes que não sabe resolver. Para que a recursão funcione, a divisão deve levar sempre a um problema parecido, porém pelo menos um pouco menor. Então, a função chama uma cópia dela para trabalhar no problema menor, o que chamamos de um **passo de recursão**.

Quando uma função chama a si mesma, a função fica esperando que sua cópia seja executada para ela resolver o problema menor. O próprio passo de recursão pode levar a várias outras cópias da função com problemas menores. Essa sequência de problemas menores deve convergir, finalmente, ao caso base, que será resolvido.

Quando não resolvemos um problema recursivamente, dizemos que estamos resolvendo o problema **iterativamente**.

Por exemplo, o valor de $n!$ (n fatorial) pode ser calculado iterativamente como:

$$n! = n \times (n-1) \times (n-2) \times \ldots 3 \times 2 \times 1$$

Já **recursivamente**, o valor de $n!$ pode ser definido como:

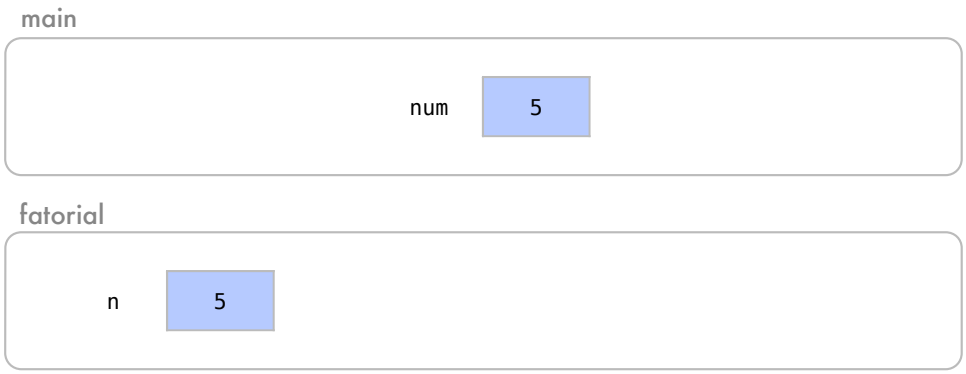
$$n! = n \times (n-1)! \text{ (Passo recursivo)} \qquad 1! = 1 \text{ (Caso base)}$$

Na definição recursiva temos um um **passo recursivo** que depende do próprio conceito de números fatoriais ($(n-1)!$) para calcular $n!$. Já no **caso base** da mesma definição recursiva, sabemos que $1!$ é igual a 1. Este caso base leva ao fim da recursão. Sem o caso base, não conseguiríamos saber o fatorial de qualquer número. Note que o passo recursivo usa a própria definição de um número fatorial, porém, para um problema menor.

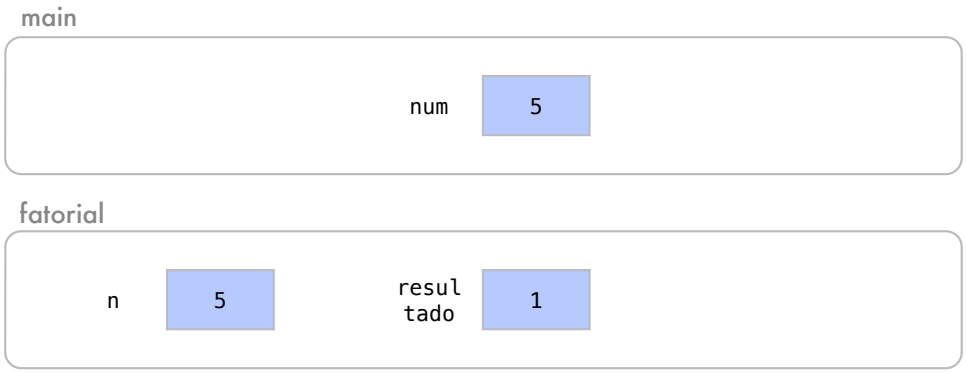
Neste primeiro exemplo, temos uma função iterativa que retorna $n!$.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int fatorial(int n);
6
7  int main() {
8     int num = 5;
9     cout << num << "!␣=␣" << fatorial(num) << endl;
10    return 0;
11 }
12
13 int fatorial(int n) {
14    int resultado = 1;
15    for (int i=n; i > 1; i--){
16      resultado *= i;
17    }
18    return resultado;
19 }
```

Inicializamos, na linha 8 do `main`, uma variável `num` que recebe o número 5. Logo em seguida, a função `main` aguarda o retorno da função `fatorial`. O parâmetro `n` na função `fatorial`, recebe o valor da variável `num` por valor.

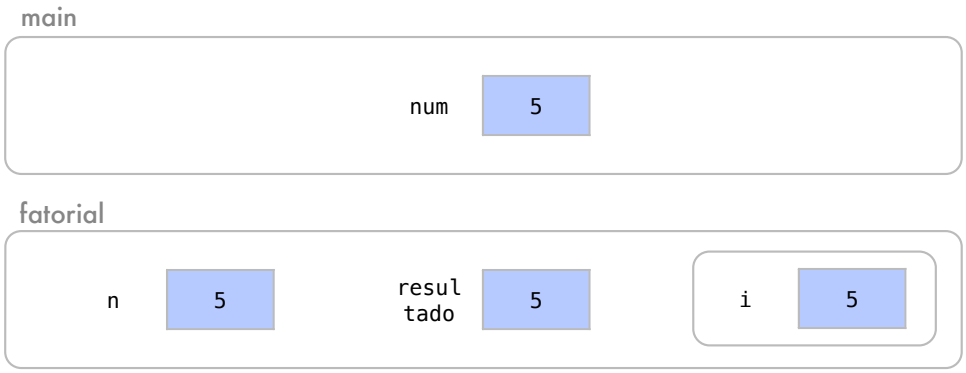


Já dentro da função `fatorial`, na linha 14, a variável `resultado` é criada e inicializada em 1.

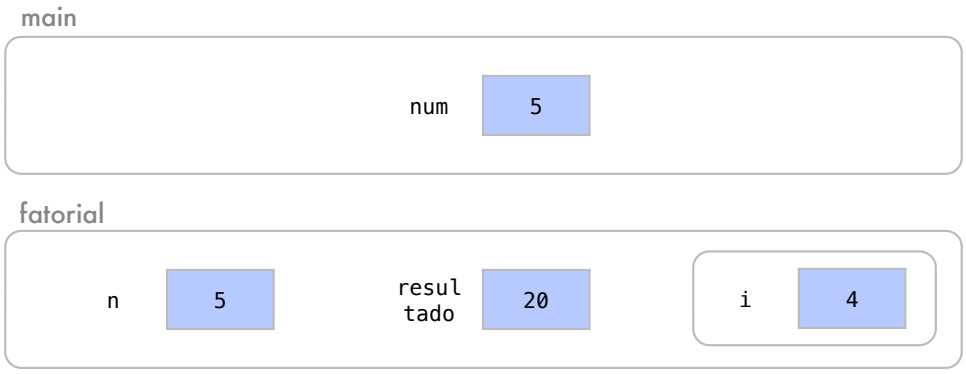


Dentro de um laço de repetição, que se inicia na linha 15, `resultado` receberá as multiplicações pelos valores de `n` até `1`. Para isso, é criado no escopo do `for` o contador `i`, que irá percorrer os valores entre `n` e 1.

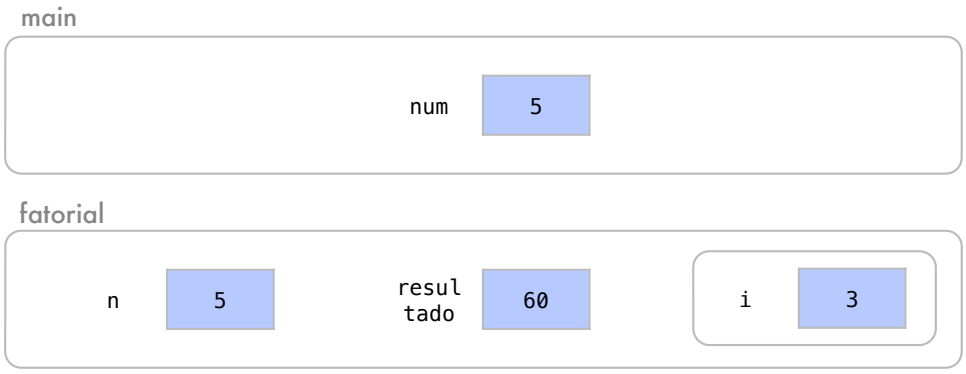
Na linha 16, enquanto `i > 1`, o `resultado` é multiplicado por `i`. Quando `i` tem valor 5:



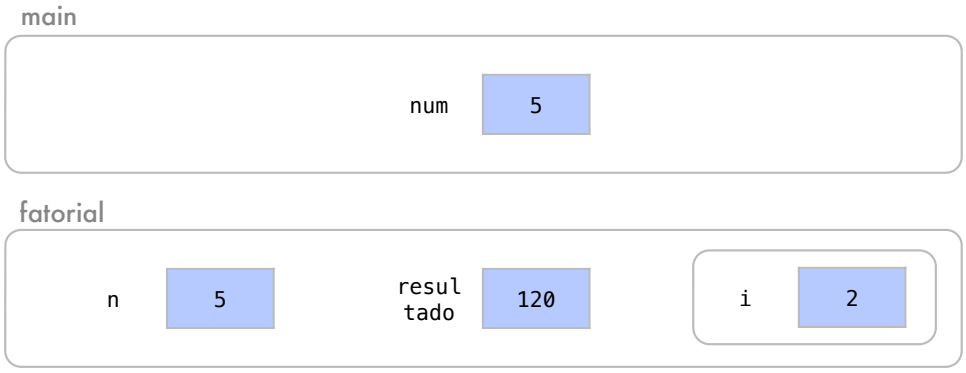
O valor de `i` é alterado em `i-` para 4, e como `i > 1`, o `resultado` é multiplicado por 4.



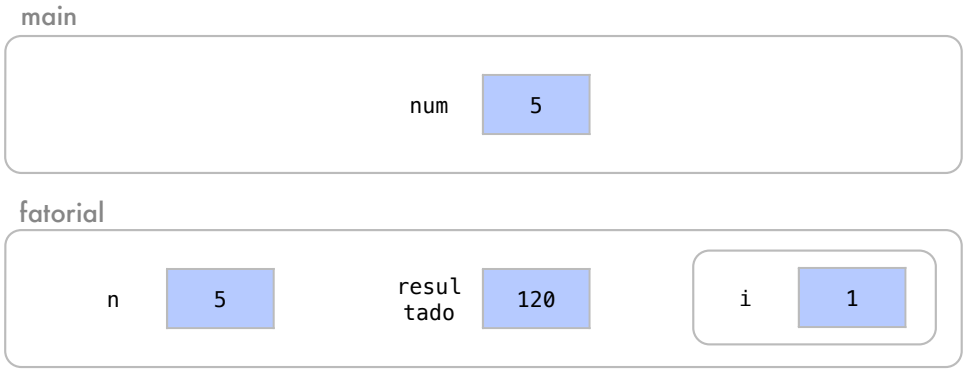
O valor de `i` é alterado em `i-` para 3, e como `i > 1`, o `resultado` é multiplicado por 3.



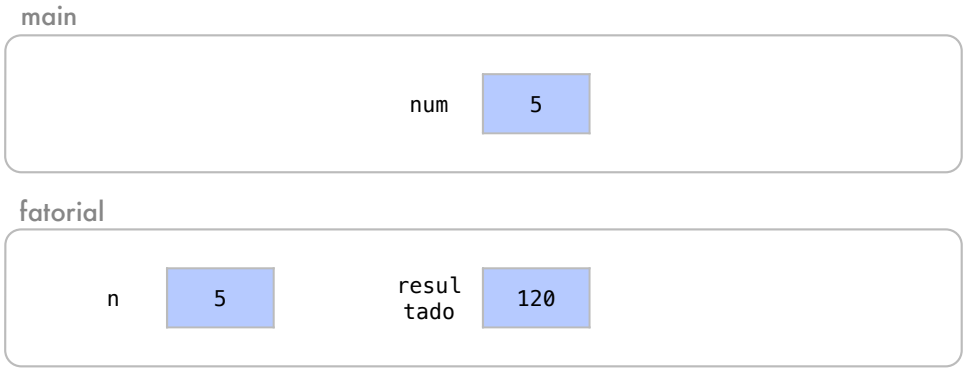
O valor de `i` é alterado em `i-` para 2, e como `i > 1`, o `resultado` é multiplicado por 2.



No fim do processo, `i` tem valor 1 e a condição do `for` passa a ser falsa.

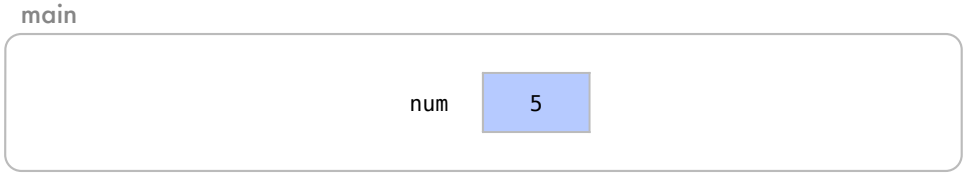


O escopo do `for` é encerrado e passamos para a linha 18 do código, onde a variável `resultado` será retornada.



A função `fatorial` é encerrada, suas variáveis são apagadas e, na linha 9, `main` retoma sua execução com o valor retornado 120.

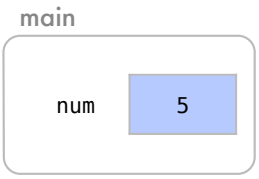
```
5!  = 120
```



Essa foi uma versão **iterativa** da função fatorial. Veja agora um segundo exemplo onde a solução do mesmo problema é encontrada com uma função **recursiva**:

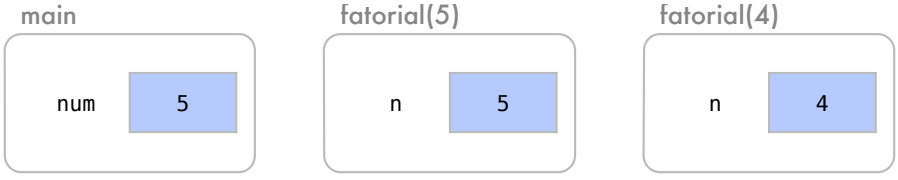
```
1  #include <iostream>
2
3  using namespace std;
4
5  int fatorial(int n);
6
7  int main() {
8     int num = 5;
9     cout << num << "!␣=␣" << fatorial(num) << endl;
10    return 0;
11 }
12
13 int fatorial(int n) {
14    if (n <= 1){
15      return 1;
16    } else {
17      return n * fatorial(n-1);
18    }
19 }
```

Declaramos uma função recursiva `fatorial` na linha 5, que retorna $n!$. Note que não há diferença no modo como a função é declarada para a versão iterativa. Na linha 8, criamos e inicializamos uma variável `num` no escopo da função `main`.



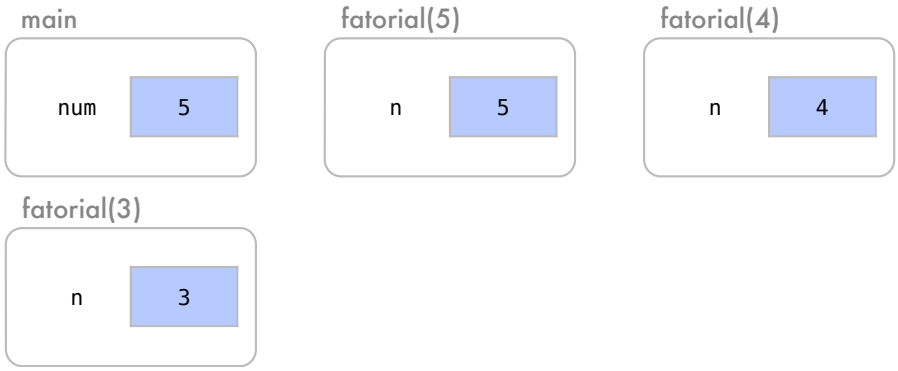
Na linha 9, nossa função principal fica em espera até que a função `fatorial` seja executada. A função `fatorial(5)`, que inicia na linha 13, recebe o valor de `num` em `n`.

Na linha 14, como `n` não é zero, a função `fatorial` tenta, na linha 17, retornar `n * fatorial(4)`. Mas ao tentar fazer este retorno, a função é colocada em espera do retorno de uma outra função `fatorial`, que recebe 4 como parâmetro.

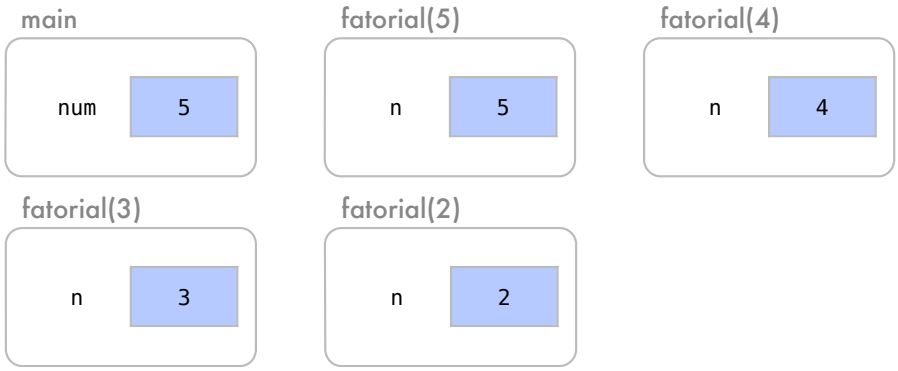


A segunda função `fatorial(4)`, por sua vez, tem um problema menor do que o problema original `fatorial(5)`. Isto é o que denominamos **passo recursivo**.

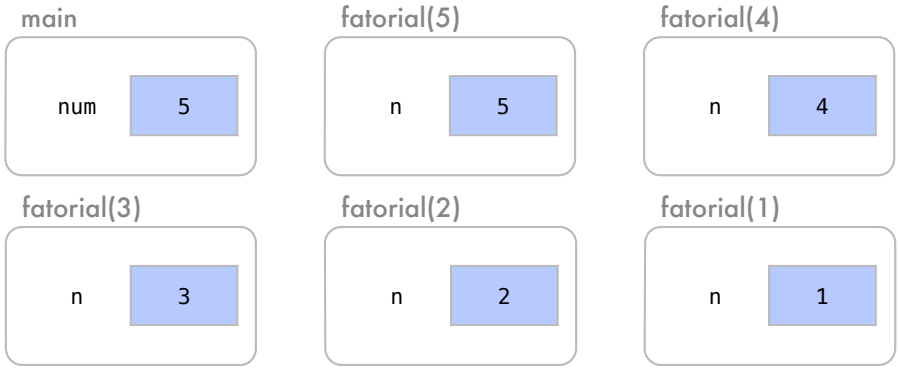
Um novo escopo para esta versão da função `fatorial(4)` é criado. Essa cópia da função tem 4 como parâmetro `num`. Na linha 17, novamente, a função `fatorial(4)` tenta retornar `4 * fatorial(3)`, mas isso faz com que ela também fique em espera de outra cópia da função `fatorial`.



Essa cópia recebe o valor 3 como parâmetro e fica em espera ao tentar retornar `3 * fatorial(2)`.



Em seguida, a função `fatorial` que recebe o valor 2, também entra em espera até o cálculo do `fatorial(1)`.



Nesta última chamada, a função `fatorial` recebe 1. Este é o nosso **caso base**, ou seja, o problema que nossa função sabe resolver sem precisar enviar um problema para outra função. Chamamos esta pilha de cópias da função, como as geradas até aqui, de **pilha de recursão**.

A função `fatorial(1)`, na linha 15, retorna 1 para a função que a chamou. Logo, `fatorial(2)` sai de espera e pode retornar para `fatorial(3)` o seu resultado 2. O `fatorial(3)`, também sai de espera e pode retornar o seu resultado 6 para `fatorial(4)`. Quando o resultado 24 de `fatorial(4)` é retornado para `fatorial(5)`, voltamos a nossa primeira chamada da função `fatorial`. Por fim, o `fatorial(5)` pode finalmente retornar seu resultado 120 para o `main`, que o havia chamado na linha 9. A função `main`, após todo este processo, sai de espera e volta a funcionar com o valor retornado, que é 120.

```
5!  = 120
```

A recursividade demonstra o poder de funções para dividir problemas em problemas menores. Alguns problemas, como os números fatoriais, podem ser definidos recursivamente de maneira bem clara.

Como vimos no exemplo, as chamadas recursivas criam uma pilha de chamadas de funções, que pode gastar muita memória. Por esse motivo nem sempre será apropriado usar uma solução recursiva para um problema.

Qualquer problema resolvido recursivamente pode ser resolvido iterativamente. Porém, a recursão pode tornar o programa mais fácil de entender ou depurar. Algumas vezes, a solução iterativa não é tão evidente. Devido a pilha de chamadas de função, evite usar recursividade em situações em que precisamos de desempenho.

É preciso tomar cuidado para garantir que as funções recursivas levem a um caso base. Caso contrário, pode ocorrer uma recursão infinita que será esgotada quando acabar a memória para a pilha de funções. Esse é um problema similar a um laço infinito em uma função iterativa e estruturas de repetição em geral.

9.11.1 Exercícios

Exercício 9.1 Analise o código abaixo, que está incompleto.

```
1  #include <iostream>
2
3  using namespace std;
4
5  // cabeçalhos
6  int cubo(int a);
7  int potencia(int a, int b);
8  int fatorial(int a);
9
10 // função principal
11 int main(){
12     int x;
13         cout << "Digite␣um␣numero:␣";
14         cin >> x;
15
16     cout << x <<  "␣elevado␣ao␣cubo␣=␣" << cubo(x) << endl;
17
18     int y;
19     cout << "Digite␣um␣valor␣para␣o␣outro␣número:␣" << endl;
20     cin >> y;
```

```
21
22      cout << x << " elevado a " << y << " = ";
23      cout << potencia(x,y) << endl;
24
25      cout << x << "! = " << fatorial(x) << endl;
26
27      return 0;
28  }
29
30  int cubo(int a){
31      return -1;
32  }
33
34  int potencia(int a, int b){
35      return -1;
36  }
37
38  int fatorial(int a){
39      return -1;
40  }
```

■

Exercício 9.2 Edite a função `cubo` (ela está definida após o `main`) para que a linha 16 passe a funcionar corretamente. ■

Exercício 9.3 Editar a função `potencia` para que o comando da linha 23 passe a funcionar. ■

Exercício 9.4 Editar a função `fatorial` para que o comando da linha 25 passe a funcionar. ■

Exercício 9.5 Analise o código abaixo, que está incompleto.

```
1   #include <iostream>
2
3   using namespace std;
4
5   // cabecalho da função com a qual vamos trabalhar
6   int potencia(int x, int y);
7   // função para ordenar arranjos
8   void ordena(int a[], int n);
9
10  int main(){
11      // a será elevado a b e guardaremos o resultado
12      int a, b, resultado;
13      do {
14          cout << "Digite a base : ";
15          cin >> a;
```

```
16          cout << "Digite␣o␣expoente␣:␣";
17          cin >> b;
18          // chamando a função que definiremos apos o main
19          resultado = potencia(a,b);
20          cout << a << "␣elevado␣a␣" << b << "␣é␣";
21          cout << resultado << endl;
22      } while (a!=-1); // programa se repete enquanto a != -1
23      return 0;
24  }
25
26  // função que eleva x a y
27  int potencia(int x, int y){
28      int produto = x;
29      int i;
30      for (i=1; i<y; i++){
31          produto*= x; // equivale a produto = produto * *x
32      }
33      return produto;
34  }
35
36  // Ordena arranjo com "método da bolha"
37  void ordena(int a[], int n)
38  {
39      int i,j,aux;
40      for( j= n-1; j>0; j--)
41          for(i=0; i<j; i++)
42          {
43              if(a[i+1] < a[i])
44              {
45                  // trocar a[i] com a[i+1]
46                  aux = a[i];
47                  a[i] = a[i+1];
48                  a[i+1] = aux;
49              }
50          }
51  }
```

■

Exercício 9.6 Troque a função `int potencia(int x, int y)` com passagem de parâmetros por valor por uma função `int potencia(int &x, int &y)` com passagem por referência.

- O que se alterou no programa do ponto de vista de resultados? Porquê?
- O que se alterou no programa do ponto de vista de sua interpretacão pelo computador?
- Em quais casos a diferença de passagem por valor e por referencia se torna relevante?

■

Exercício 9.7 Simule uma passagem de parâmetros por referência através de ponteiros passados por valor. Para isso, crie uma nova função `potencia2` para não perder a resposta das perguntas anteriores.

Como a passagem de parâmetros por referência pode ser simulada com ponteiros passados por valor? Crie uma nova função `potencia2` para não perder a resposta das perguntas anteriores.

Tendo a função `potencia2` com passagem por referência através de ponteiros passados por valor. Descreva qual a desvantagem desta abordagem do ponto de vista do programador. ■

Exercício 9.8 Crie uma nova versão da função `potencia` chamada `potencia3`. Refaça a função de modo que ela retorne `void`, a passagem de todas as variáveis seja por referência e o resultado será guardado diretamente em uma variável extra passada como argumento. ■

Exercício 9.9 Refaça esta funcao em uma função `potencia4` de modo que ela seja recursiva, ou seja, a função fará uso de si mesma. Ex: $x^0 = 1$, $x^1 = x \times x^0$, $x^2 = x \times x^1$, $x^3 = x^2 \times x$, ..., $x^n = x^{(n-1)} \times x$ ■

Exercício 9.10 Analise a função `ordena()`. Veja como ela recebe um arranjo e o tamanho deste arranjo. Crie no `main` um arranjo que a utilize.

Substitua os comandos da função `ordena`, responsáveis pela troca, por uma chamada a uma nova funcão `void troca(int &a, int &b)` que recebe como parâmetros duas variáveis e troque seus conteúdos. ■

10. Estruturas

Estruturas, ou registros, são conjuntos de elementos agrupados de dados. Essa é uma ferramenta que nos permite criar novos tipos de dados a partir dos tipos existentes. Estas estruturas são declaradas em C++ com a instrução `struct`. Os elementos de dados de uma estrutura são chamados de membros e cada membro da estrutura pode ser de um tipo diferente de dados.

Sintaxe

Temos aqui um exemplo de sintaxe para criação de um estrutura:

```
1 struct nome_do_tipo {
2     tipo_do_membro1 nome_do_membro1;
3     tipo_do_membro2 nome_do_membro2;
4     tipo_do_membro3 nome_do_membro3;
5 };
```

O `nome_do_tipo` é o nome da estrutura que queremos criar. Entre chaves { e }, temos uma lista de membros de dados. Cada membro tem um tipo de dados e um identificador. Essas estruturas completas podem ser utilizadas para criar novos tipos de dados a partir dos tipos de dados já existentes.

Por exemplo, uma estrutura como a seguinte pode ser utilizada para representar produtos:

```
1 struct produto {
2     double preco;
3     double peso;
4     string nome;
5 };
```

Com esta estrutura, declaramos um novo tipo de dados chamado `produto`. Esse tipo de dados contém 2 membros do tipo `double` e um membro do tipo `string`.

O novo tipo `produto` já pode ser utilizado para declararmos variáveis desse tipo em nosso programa. Por exemplo, uma variável `produto` que represente uma maçã ou um abacaxi.

Os membros dessas variáveis podem ser acessados diretamente com um ponto (`.`) entre o identificador da variável e o identificador do membro, como no exemplo abaixo:

```
1 produto pera;
2 pera.peso = 0.3;
3 pera.nome = "Pêra Williams";
4 cout << "Digite o preço: ";
5 cin >> pera.preco;
6 cout << "Peso da pêra: " << pera.peso << endl;
```

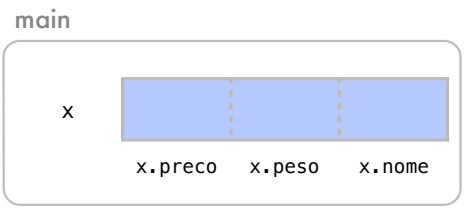
Nas linhas 1 do exemplo, criamos uma variável do tipo `produto` que com nome identificador `pera`. Nas linhas 2 e 3 definimos os membros `peso` e `nome` desta variável `pera` através do operador de atribuição. Nas linhas 4 e 5, utilizamos um `cin` para que usuário possa definir o valor do membro `peso` da variável `pera`. Na linha 6, o valor do membro `peso` da `pera` é impresso com um `cout`. O acesso a membros é sempre feito com um ponto (`.`).

Exemplo

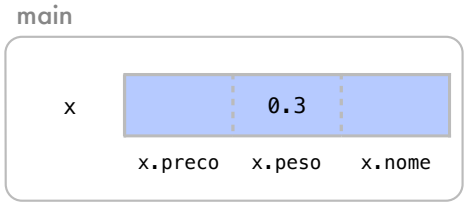
Neste exemplo, vamos criar um novo tipo de dado e utilizá-lo em nosso programa.

```
1  #include <iostream>
2
3  using namespace std;
4
5  struct produto {
6     double preco;
7     double peso;
8     string nome;
9  };
10
11 int main() {
12    produto x;
13    x.peso = 0.3;
14    x.nome = "Pêra";
15    cout << "Digite o preço: ";
16    cin >> x.preco;
17    cout << "Peso da " << x.nome << ": " << x.peso << endl;
18    return 0;
19 }
```

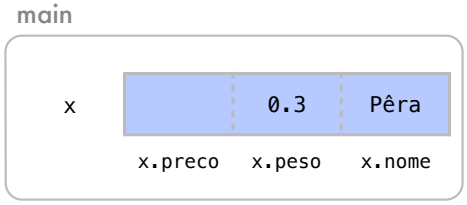
Antes da nossa função principal `main`, é declarado o novo tipo de dados `produto` nas linhas 5 a 9. Na linha 12, é criada uma variável `x` do tipo `produto` no escopo de `main`. Observe que `x`, apesar de ser apenas uma variável, tem 3 membros: `preco`, `peso` e `nome`.



Com um ponto (`.`) entre o nome da variável e o nome do membro, atualizamos o valor do `peso` de `x` (`x.peso`) na linha 13. O `peso` deste produto recebe 0.3.

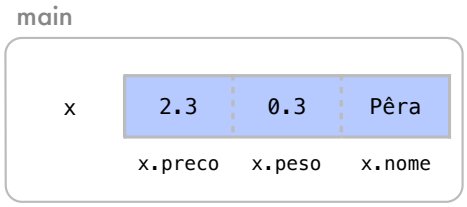


De maneira similar, na linha 14, o nome de x recebe a `string` "`Pera`".



Nas linhas 15 e 16, o usuário atribui um valor a `x.preco` através do fluxo de entrada, com o número `2.3`.

```
Digite o preço:  2.3
```



Então, na linha 17, imprimimos os membros `nome` e `peso` do objeto `x`.

```
Digite o preço:  2.3
Peso da pêra:  0.3
```

10.1 Arranjos de estruturas

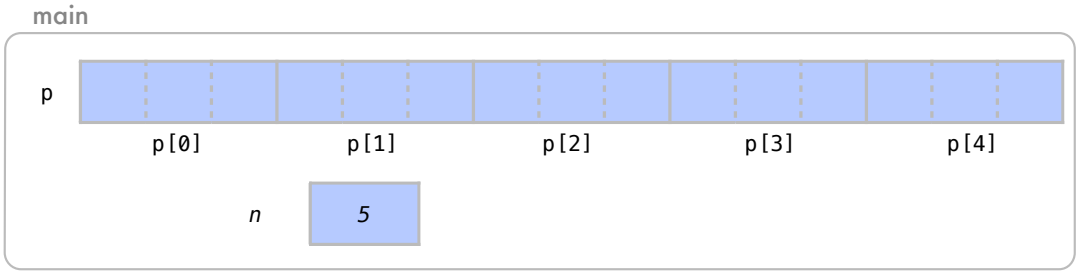
Vimos então que as estruturas servem para definir novos tipos de dados a partir dos tipos já conhecidos. Como estruturas são novos tipos de dados, elas também podem ser utilizadas como tipos de arranjos.

Neste exemplo, criaremos um arranjo da dados do tipo `produto`.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 struct produto{
7    double preco;
```

```
8     double peso;
9     string nome;
10 };
11
12 int main() {
13     const int n = 5;
14     produto p[n];
15     for (int i=0; i<n; ++i){
16      cout << "Produto " << i << endl;
17      cout << "Digite o preço: ";
18      cin >> p[i].preco;
19      cout << "Digite o peso: ";
20      cin >> p[i].peso;
21      cout << "Digite o nome: ";
22      cin >> p[i].nome;
23     }
24     for (int i=0; i<n; ++i){
25      cout << p[i].nome << "\t";
26      cout << "R$" << p[i].preco << "\t";
27      cout << p[i].peso << "kg" << endl;
28     }
29     return 0;
30 }
```

Antes de nossa função principal, nas linhas 6 a 10, declaramos o novo tipo de dados `produto`. Já na função principal, na linha 13, definimos a constante `n` com valor `5` e, na linha 14, criamos um arranjo de produtos `p`, que tem capacidade para `n` produtos.

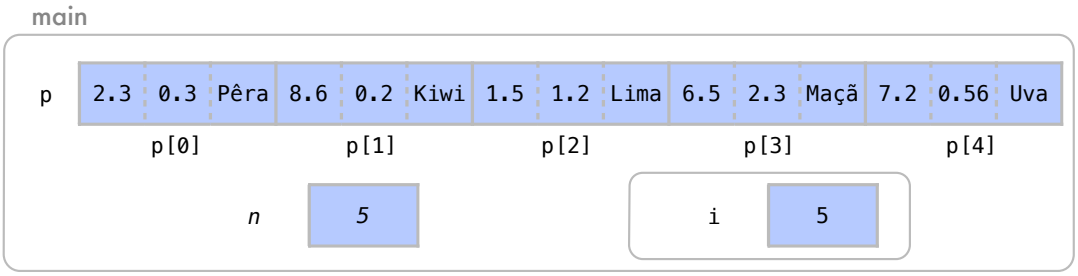


Note como cada elemento do arranjo tem espaço para os três membros de um produto: preço, peso e nome. Na primeira iteração do `for` definido nas linhas 15 a 23, o usuário atribui valores aos membros do primeiro produto. Para isto, indicamos uma posição do arranjo e o membros desta posição, com `p[i].preco`, `p[i].peso` e`p[i].nome`.

```
Produto 0
Digite o preco:  2.3
Digite o peso:  0.3
Digite o nome:  Pêra
```

No conjunto de todas as iterações, passando por todos os valores de `i`, teremos dado valores a todos os membros de todos os produtos.

```
...
Digite o nome:  Lima
Produto 3
Digite o preço:  6.5
Digite o peso:  2.3
Digite o nome:  Maçã
Produto 4
Digite o preço:  7.2
Digite o peso:  0.56
Digite o nome:  Uva
```



Já no outro conjunto de iterações, definido nas linhas 24 a 28, imprimimos todos os produtos do arranjo de maneira análoga.

```
...
Produto 4
Digite o preço:  7.2
Digite o peso:  0.56
Digite o nome:  Uva
Pêra R$2.3 0.3kg
Kiwi R$8.6 0.2kg
Lima R$1.5 1.2kg
Maçã R$6.5 2.3kg
Uva R$7.2 0.56kg
```

10.2 Estruturas e ponteiros

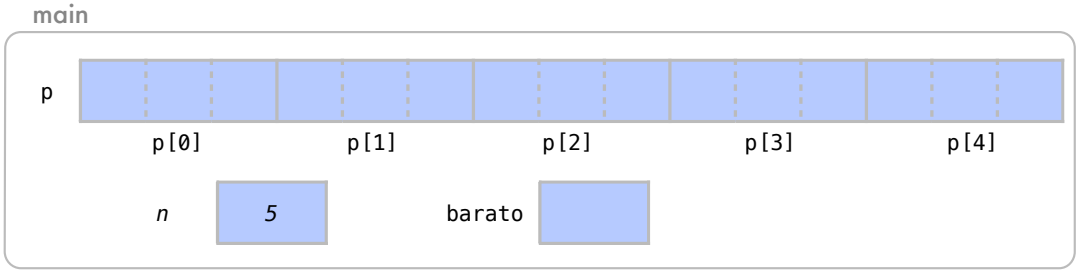
Quando temos um ponteiro `px` apontando para uma estrutura `x`, podemos acessar os membros de `x` através do valor apontado por `px`, ou seja, `(*px).membro`). Porém, podemos fazer o mesmo com o operador de seta `->`, como em `px->membro`. Este operador de seta retorna o membro do elemento apontado por `px`.

Nesse exemplo, criaremos ponteiros para produtos que estão em um arranjo. Temos no exemplo, na linha 16, um ponteiro para um `produto` que se chama `barato`. Ele irá apontar para o produto mais barato do arranjo.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
```

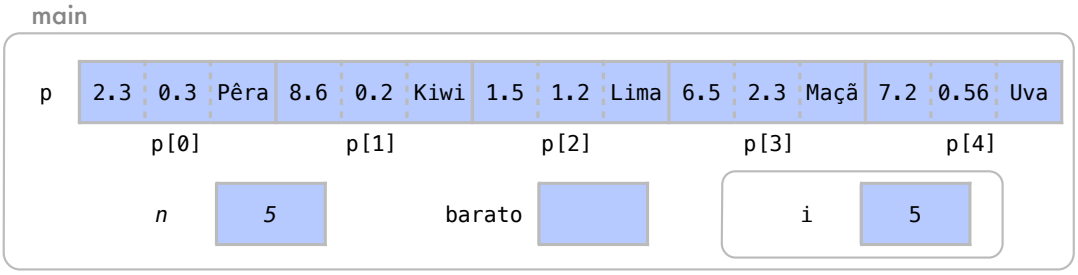
```
6
7  struct produto{
8     double preco;
9     double peso;
10    string nome;
11 };
12
13 int main() {
14    const int n = 5;
15    produto p[n];
16    produto *barato;
17    for (int i=0; i<n; ++i){
18     cout << "Produto␣" << i << endl;
19     cout << "Digite␣o␣preço:␣";
20     cin >> p[i].preco;
21     cout << "Digite␣o␣peso:␣";
22     cin >> p[i].peso;
23     cout << "Digite␣o␣nome:␣";
24     cin >> p[i].nome;
25    }
26    barato = &p[0];
27    for (int i=1; i<n; ++i){
28     if (p[i].preco < barato->preco){
29         barato = &p[i];
30     }
31    }
32    cout << "Produto␣mais␣barato:" << endl;
33    cout << barato->nome << "\t";
34    cout << "R$" << barato->preco << "\t";
35    cout << barato->peso << "kg" << endl;
36    return 0;
37 }
```

É declarado no código, na linha 6, o novo tipo de dados `produto`. Nas linhas 14 a 16, criamos a constante `n`, um arranjo `p` de tamanho `n` para dados do tipo `produto` e um ponteiro `barato`, também para dados do tipo `produto`.

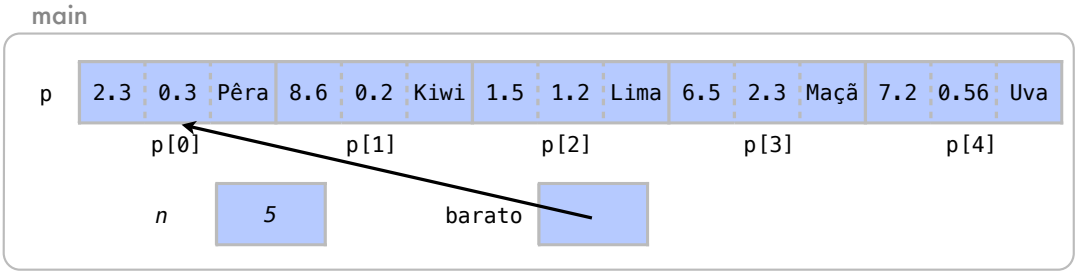


Exatamente como no exemplo anterior, a partir da linha 16, damos valores a todos os produtos de nosso arranjo.

```
...
Dite o nome:  Lima
Produto 3
Digite o preço:  6.5
Digite o peso:  2.3
Digite o nome:  Maçã
Produto 4
Digite o preço:  7.2
Digite o peso:  0.56
Digite o nome:  Uva
```

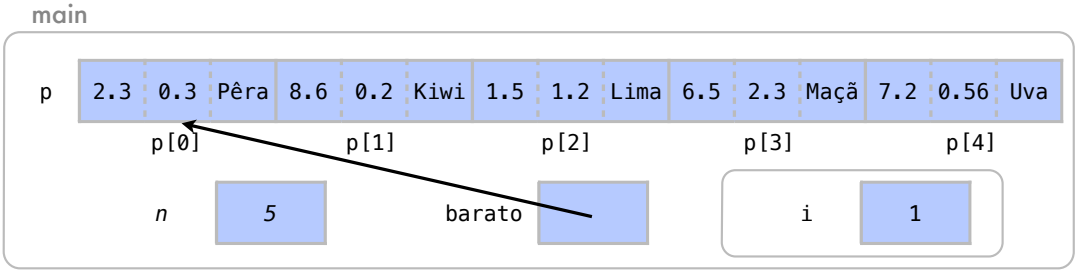


Em seguida, na linha 26, o ponteiro `barato` aponta para o primeiro elemento do arranjo. O endereço do primeiro elemento do arranjo pode ser acessada tanto com `&p[0]` quanto com `p`.

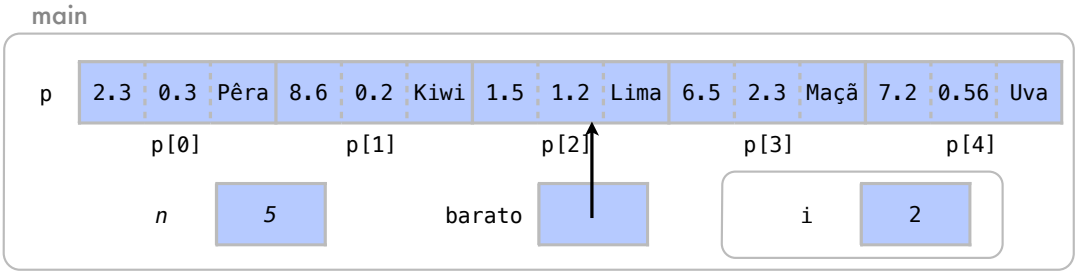


Temos entre as linhas 27 e 31 um laço que procura o elemento mais barato do arranjo. O ponteiro `barato` aponta para o primeiro elemento e começaramos a busca a partir do segundo elemento `p[1]`.

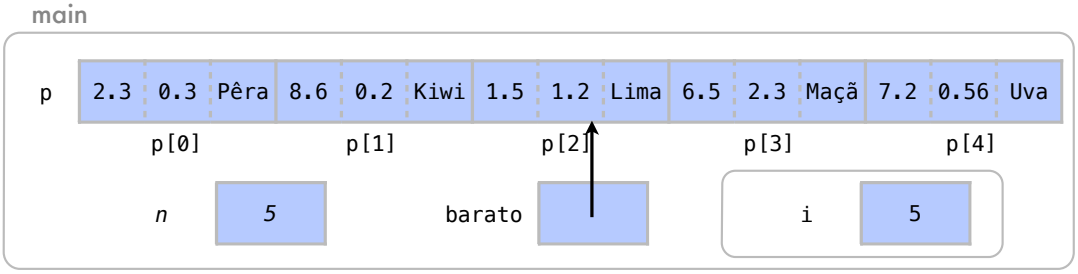
Na linha 28, na primeira iteração do laço, se o elementop`[1]` fosse mais barato que o apontado por `barato`, `barato` apontaria para `p[1]`. Repare como o operador de seta está sendo usado na linha 28.



Na próxima iteração, quando `i` tem valor 2, como `p[2]` é mais barato que o elemento apontado por `barato`, então ele passa a ser o elemento apontado.

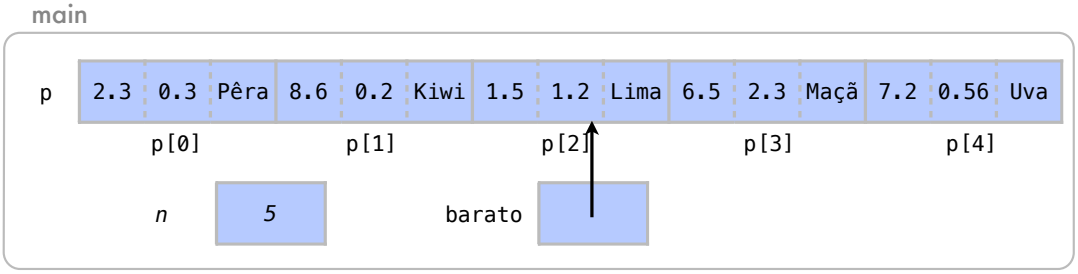


Ao fim do `for`, teremos testado isso para todos os elementos, porém nenhum foi mais barato que `p[2]` neste exemplo.



Após o fim do laço, utilizamos novamente o operador de seta, nas linhas 32 a 35, para imprimir os dados sobre elemento mais barato do arranjo.

```
...
Digite o preço:  6.5
Digite o peso:  2.3
Digite o nome:  Maçã
Produto 4
Digite o preço:  7.2
Digite o peso:  0.56
Digite o nome:  Uva
Produto mais barato:
Lima R$1.5 1.2kg
```



11. Alocação de memória

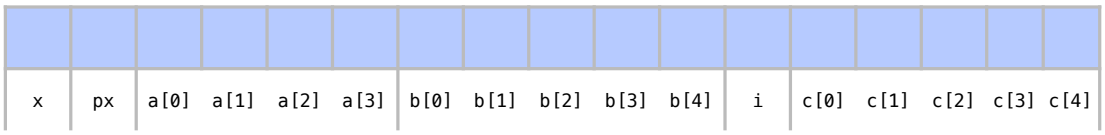
11.1 Alocação automática de memória

Para cada variável de nosso programa, há uma quantidade de memória que precisa ser alocada para nossas variáveis. Quando executamos uma função ou bloco de código, o sistema operacional aloca a memória necessária para todas as suas variáveis. Essa é a memória automaticamente alocada. Por isso, por exemplo, é importante dizermos o tamanho dos arranjos em nosso programa, diretamente ou através de constantes.

Neste exemplo de código, criamos algumas variáveis:

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main() {
7     int x = 4;
8     int *px = &x;
9     int a[] = {2,4,5,3};
10    int b[5];
11    int i;
12    for (i=0; i<5; ++i){
13     b[i] = a[0] + i;
14    }
15    const int n = 5;
16    int c[n];
17    for (i=0; i<n; ++i){
18     c[i] = a[1] + i;
19    }
20    return 0;
21 }
```

Antes de iniciar o `main`, o sistema operacional já aloca memória para 16 inteiros e um ponteiro para inteiro. Essa é a memória alocada de maneira automática para o `main`.



Repare na linha 15 que `n` é uma constante. Valores de constantes não podem ser alterados durante o programa. Repare também que, por ele ser constante, não existe memória alocada para ele. Não é necessário alocar memória para constantes pois sabemos que seu valor nunca se altera. É simplesmente como se todas as aparições de `n` no código fossem substituídas por seu valor. Assim, os dois códigos abaixo são equivalentes:

```
1 const int n = 5;
2 int c[n];
3 for (i=0; i<n; ++i){
4     c[i] = a[1] + i;
5 }
```

```
1 int c[5];
2 for (i=0; i<5; ++i){
3     c[i] = a[1] + i;
4 }
```

De maneira análoga, esse trecho de código pode levar a uma mensagem de erro:

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main() {
7     int n;
8     cout << "Digite␣o␣tamanho␣desejado␣de␣vetor:␣";
9     cin >> n;
10    int x[n];
11    for (int i=0; i < n; ++i){
12     x[i] = i+1;
13    }
14    return 0;
15 }
```

Isso acontece porque na linha 10, usamos uma variável para definição do tamanho do arranjo. Assim, o sistema não consegue determinar quanta memória será necessária para a função antes de iniciar `main`.

Apenas constantes podem definir o tamanho de arranjos alocados automaticamente.

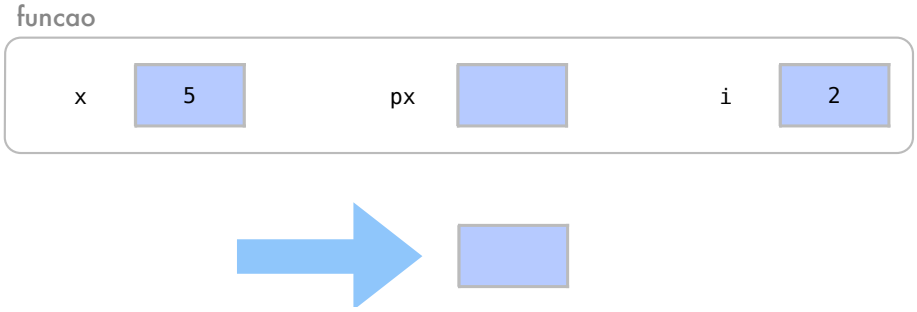
11.2 Alocação dinâmica de memória

Em várias situações, não sabemos a quantidade de memória necessária para fazermos tudo o que precisamos antes de iniciar o programa. Isso é comum quando a quantidade de memória depende de uma resposta do usuário. Neste caso, é necessário pedir ao sistema para alocar dinamicamente mais memória do que aquela inicialmente reservada para o programa. Esta memória

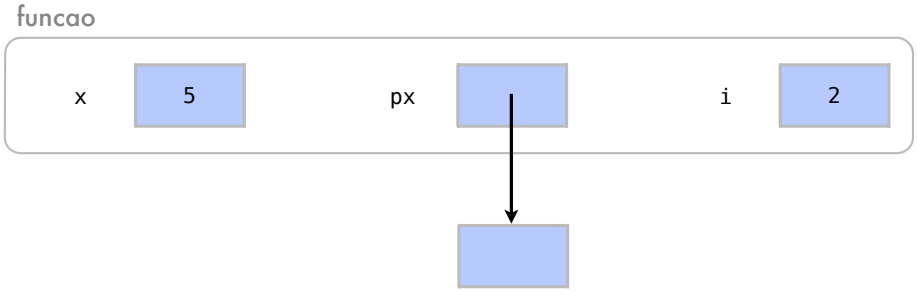
é alocada *dinamicamente* pelo sistema durante a execução dos comandos, diferentemente da memória alocada de maneira *automática*, antes do bloco de comandos se iniciar.

Os operadores `new` e `delete` (*novo* e *deletar*, em inglês), nos permitem fazer a alocação dinâmica de memória. Suponha agora, que temos uma função com três variáveis alocadas automaticamente. `x` e `i` são variáveis do tipo `int`. Enquanto isso, `px` é um ponteiro para dados do tipo `int`, que até o momento não aponta para ninguém:

O operador `new` pede ao sistema para alocar dinamicamente memória para mais uma variável do tipo`int`. Essa variável é alocada fora do escopo da função pois não está sendo alocada automaticamente.



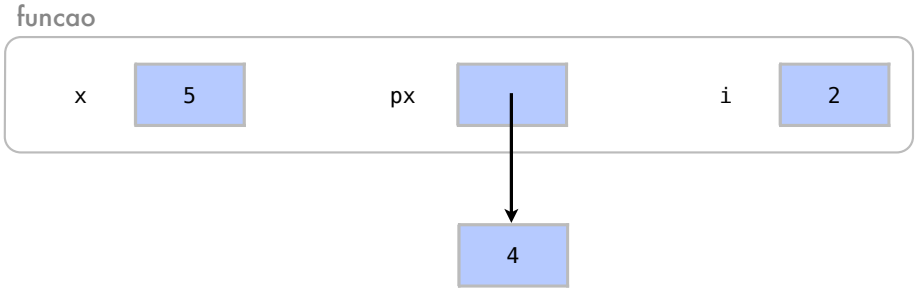
Um problema com esse `int` alocado é que não podemos acessá-lo ainda pois não existe um nome identificador para ele. Para resolver este problema, o operador `new` retorna também o endereço onde foi alocada a memória para o `int`. Assim, este endereço pode ser atribuido a um ponteiro, como em `px = new int`), onde o ponteiro `px` recebe então o endereço deste novo `int`.



Agora sim podemos acessar essa memória através desse ponteiro, como exemplo abaixo:

```
1 int x = 5;
2 int i = 2;
3 px = new int;
4 *px = 3;
5 (*px)++;
6 cout << *px << endl;
```

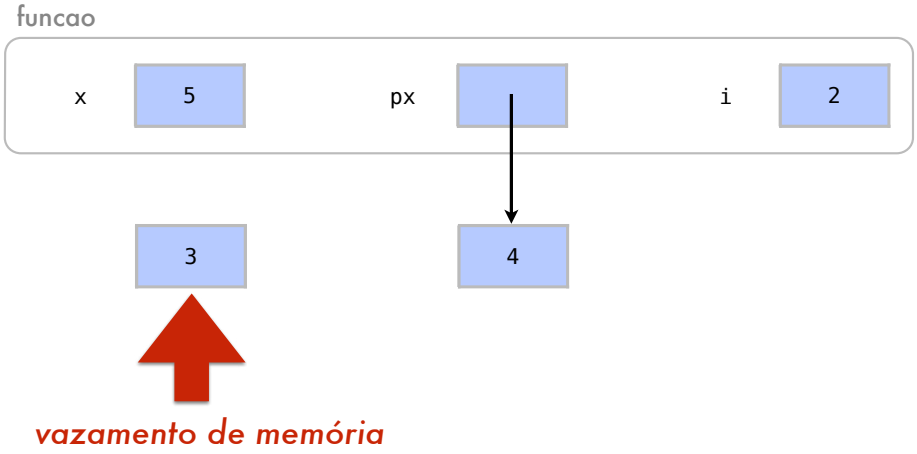
```
4
```



Considere agora este exemplo onde utilizamos o comando `new` duas vezes.

```
1 int x = 5;
2 int i = 2;
3 px = new int;
4 *px = 3;
5 px = new int;
6 *px = 4;
```

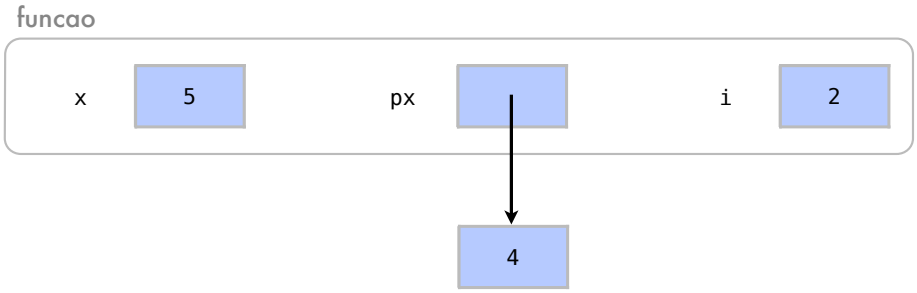
Temos novamente um problema, pois como `px` aponta ao final do código para o segundo `int`, não temos mais como acessar a memória alocada para o primeiro `int`.



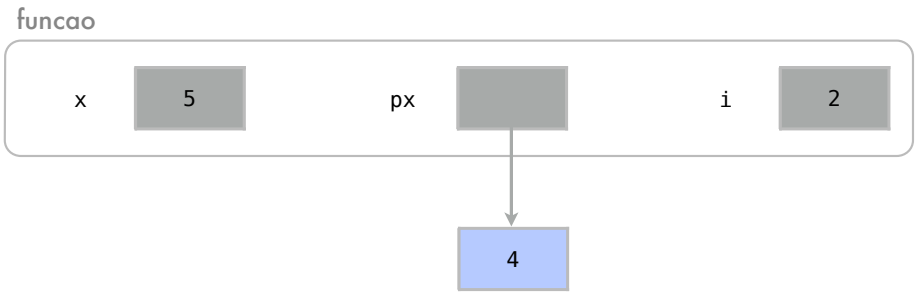
Quando uma memória fica alocada porém não temos como acessá-la, dizemos que houve um **vazamento de memória**. Esse é um problema sério em programas, pois nossa memória é um recurso finito.

Considere agora esta função e o estado de suas variáveis:

```
1 void funcao(){
2     int x = 5;
3     int i = 2;
4     int *px;
5     px = new int;
6     *px = 4;
7 }
```

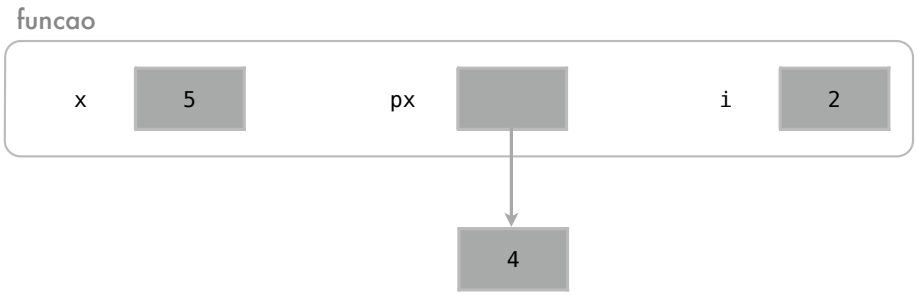


Quando chegamos ao fim da função, as variáveis do escopo da função deixam de existir. Assim, px deixa também de existir e não há nenhum ponteiro para a memória alocada. Isso causa novamente um vazamento de memória.



É muito comum que a memória alocada seja utilizada apenas durante um período de tempo. Por isso, o comando `delete` pode ser utilizado para liberar essa memória para outros pedidos de alocação, como no exemplo abaixo:

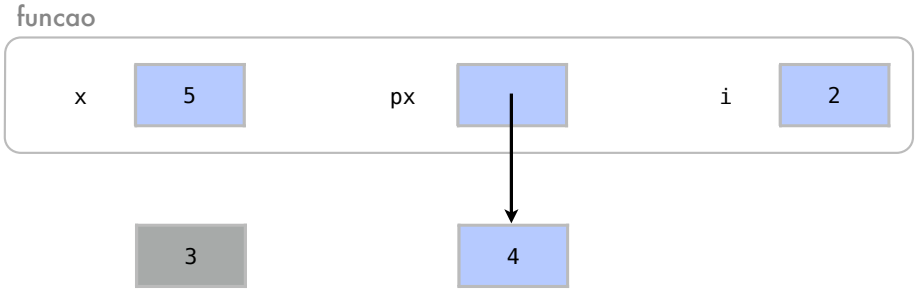
```
1 void funcao(){
2     int x = 5;
3     int i = 2;
4     int *px;
5     px = new int;
6     *px = 4;
7     delete px;
8 }
```



Repare que o comando `delete` libera a memória apontada por px e não apaga a variável px. Apagar a variável px nem seria possível, pois px foi automaticamente alocada.

Em outro exemplo da utilização do `delete`, px aponta para um inteiro de valor 3, e na linha 4 ele é apagado para apontar para outro inteiro de valor 4:

```
1 int *px;
2 px = new int;
3 *px = 3;
4 delete px;
5 px = new int;
6 *px = 4;
```

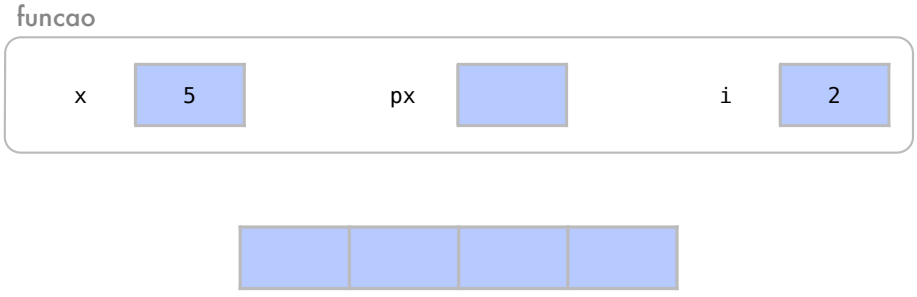


Como o dado inicialmente apontado por px foi desalocado, ele pode apontar para outro endereço na memória sem que houvesse vazamento de memória.

11.3 Alocação dinâmica de arranjos

O operador new[] pede ao sistema para dinamicamente alocar memória para várias variáveis do mesmo tipo, através de um arranjo:

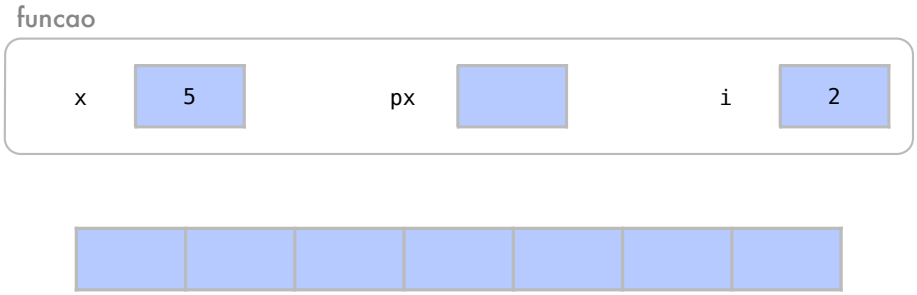
```
1 int n = 4;
2 new int[n];
```



Note que n agora não precisa mais ser uma constante, pois a memória está sendo dinamicamente alocada. Isso nos permite, até mesmo, associar o tamanho do arranjo a uma resposta do usuário, como ocorre no código abaixo:

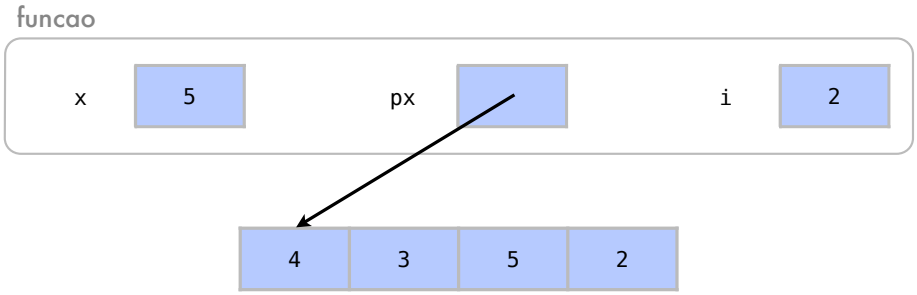
```
1 int n;
2 cout << "Quantos␣elementos␣deseja␣ter␣no␣arranjo?␣";
3 cin >> n;
4 new int[n];
```

```
Quantos elementos deseja ter no arranjo?  7
```



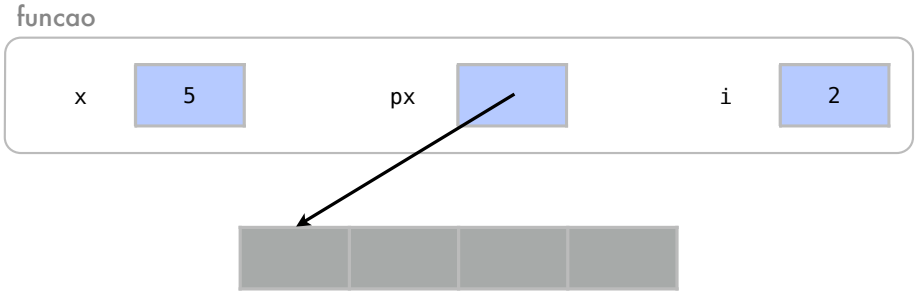
De maneira análoga a `new`, `new[]` retorna o endereço do primeiro elemento do arranjo alocado. Com um ponteiro para um arranjo, podemos acessar seus elementos com o operador para retornar o valor apontado (∗). Como vimos na Seção 8.4, sobre ponteiros e arranjos, podemos também usar o operador de subscrito `[]`, representado por colchetes, como em um arranjo automaticamente alocado:

```
1 int n = 4;
2 px = new int[n];
3 *px = 4; // mesmo que px[0]
4 px[1] = 3; // mesmo que *(px+1)
5 px[2] = 5;
6 px[3] = 2;
```



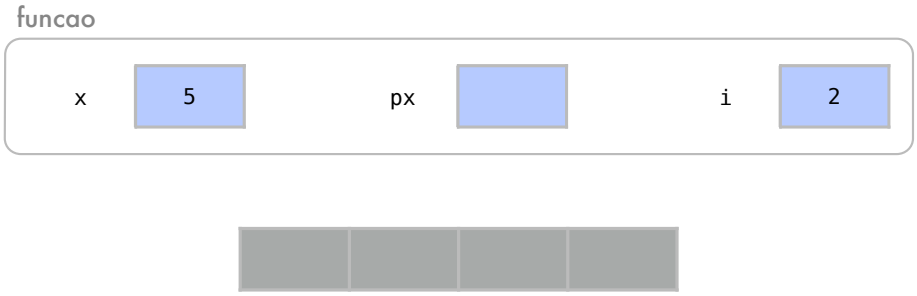
No código acima, ao fim da utilização dessa memória, seria importante liberá-la para que não haja vazamento de memória. Para apagar arranjos, usamos `delete[]` . Mas não confunda este operador com com o `delete` simples. O `delete` simples desaloca a memória de apenas um elemento:

```
1 int n = 4;
2 px = new int[n];
3 delete[] px;
```



É fundamental desalocar toda a memória após utilizá-la. Após liberar a memória, para que `px` pare de apontar para um endereço onde não há memória alocada, o fazemos apontar para `nullptr`, que representa um **ponteiro nulo**. Isso faz com que o ponteiro não aponte para nenhum endereço e previne erros futuros. Isso pode ser visto na última linha do código abaixo:

```
1 int n = 4;
2 px = new int[n];
3 delete[] px;
4 px = nullptr;
```



No exemplo abaixo, vamos alocar um arranjo dinamicamente com um tamanho escolhido pelo usuário.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int n;
7      int *x;
8      cout << "Digite o tamanho desejado de arranjo: ";
9      cin >> n;
10     x = new int[n];
11     for (int i=0; i < n; ++i){
12      x[i] = i+1;
13     }
14     delete[] x;
15     x = nullptr;
16     return 0;
17 }
```

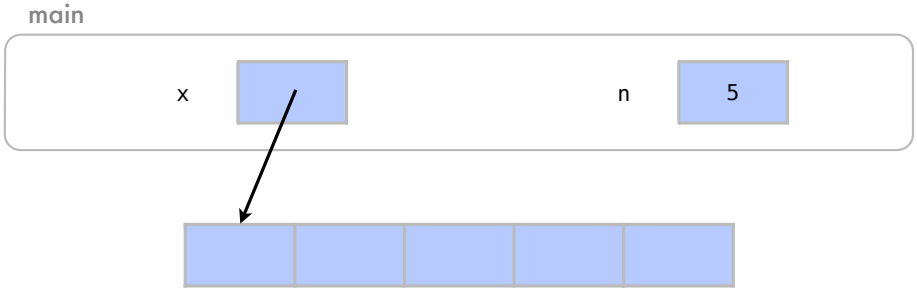
Antes mesmo de iniciar a função `main`, ainda na linha 5, o compilador sabe que ela terá duas variáveis `n` e `x`. Essas variáveis não existem ainda, mas o espaço para elas já foi alocado automaticamente.

Sendo assim, nas linhas 6 e 7, as variáveis `n` e `x` passam a existir na memória já alocada automaticamente para elas.

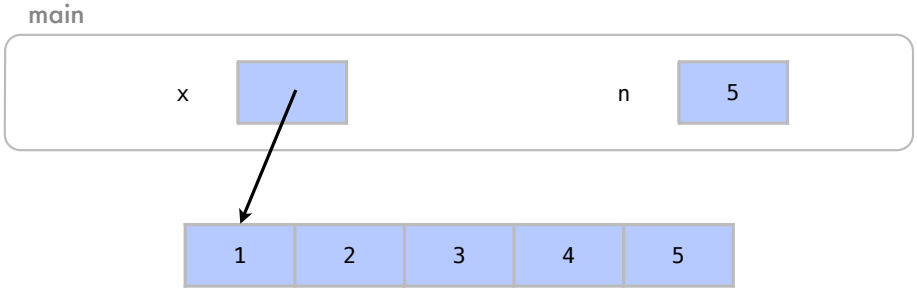
Nas linhas 8 e 9, o usuário, por sua vez, atribui o valor que deseja para `n`, no caso, `5`.

```
Digite o tamanho desejado de arranjo:  5
```

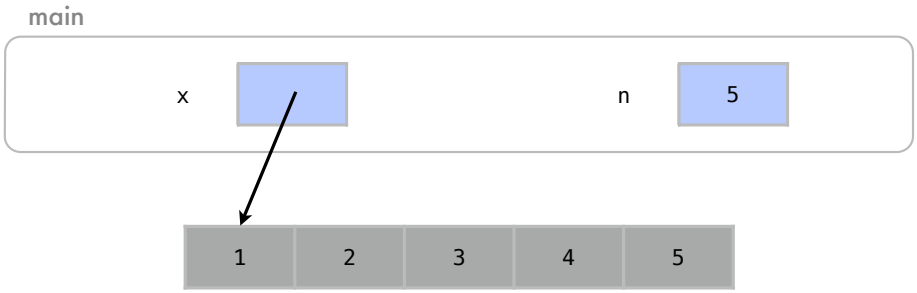
Na linha 10, é alocada memória para um arranjo de tamanho `5` e `x` aponta para ele. Repare que a memória foi alocada no `main`, mas ela não pertence a seu escopo. Apenas conseguimos acessar o arranjo através do ponteiro que está em `main`.



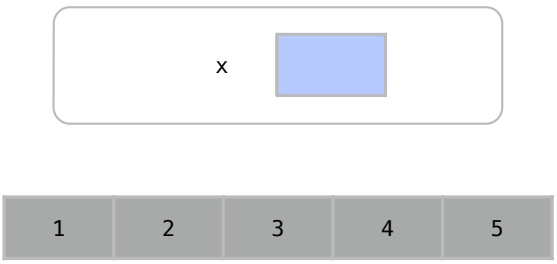
Nas linhas 11 a 13, em um laço de repetição, o arranjo recebe os valores de `1` até `n`. Neste trecho de código poderíamos ter qualquer outra operação sobre arranjos. Veja que `x` é acessado através de subscritos em `x[i]`.



Após isto, na linha 14, chegamos em um ponto onde não precisamos mais do arranjo alocado. Assim, o apagamos com a instrução `delete[]`.



Na linha 15, fazemos com que x pare de apontar para aquela posição de memória também, já que não temos mais o arranjo naquele endereço.

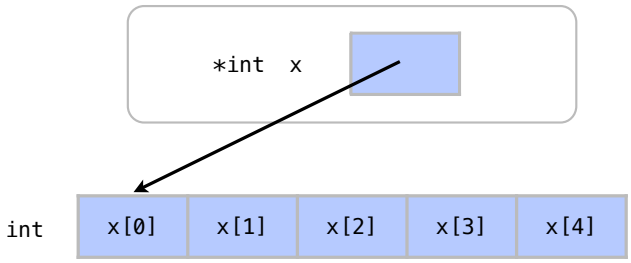


11.4 Alocação dinâmica de arranjos multidimensionais

Como vimos na Seção 5.5, sobre arranjos multidimensionais, os arranjos de várias dimensões são representados por arranjos de arranjos. Esta característica é importante ao alocar dinamicamente arranjos multidimensionais.

Considere o código abaixo que aloca dinamicamente um arranjo de 5 elementos:

```
1  int *x;
2  x = new int[5];
3  // operações com o arranjo
4  delete[] x;
```



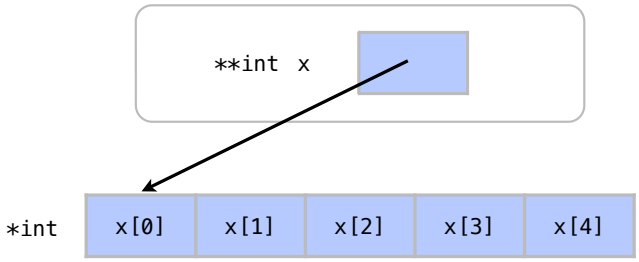
Neste programa, `x` é um ponteiro para um arranjo de dados do tipo `int` e `x[i]` acessa a posição `i` deste arranjo. Ao fim do código, a memória é desalocado com o comando `delete[]`.

No exemplo abaixo alocamos dinamicamente um arranjo multidimensional como um arranjo de arranjos:

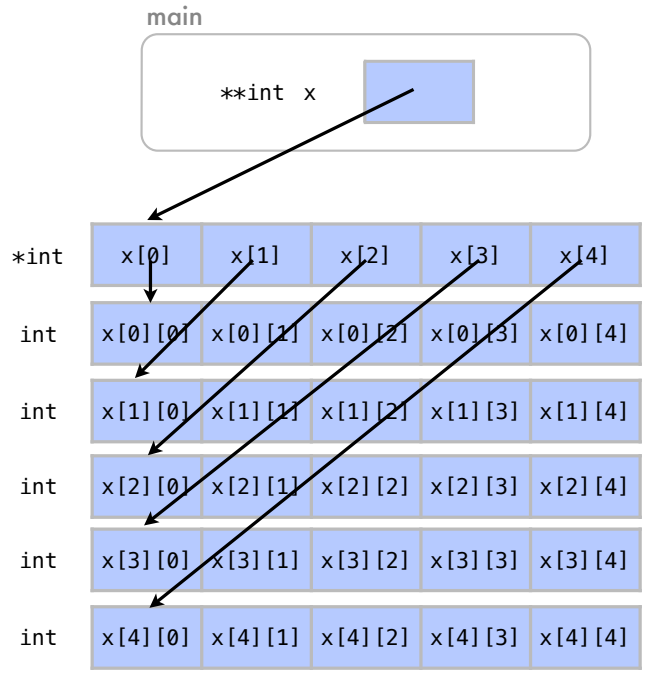
```
1  // ponteiro para ponteiros de int
2  int **x;
3  // alocando arranjo de ponteiros para int
4  x = new *int[5];
5  // alocando arranjos
6  for (int i=0; i<5; i++){
7      x[i] = new int[105
8  }
9  // utilizar o arranjo
10 // liberando arranjos
11 for (i=0; i<5; i++){
12     delete[] x[i];
```

```
13 }
14 delete[] x;
```

Na linha 2, criamos uma variável `x` que é um ponteiro para ponteiros de dados do tipo `int`. Na linha 4, `x` aponta para um arranjo de tamanho 5. Neste exemplo, note que o comando cria um arranjo de dados do tipo `*int[5]`. Ou seja, apesar da representação gráfica similar, o arranjo criado é um arranjo de ponteiros para números inteiros, e não um arranjo de números inteiros. Cada posição `x[i]` do primeiro arranjo é um ponteiro que deve apontar então para um outro arranjo de número inteiros.



Nas linhas 6 a 8, temos uma estrutura de repetição que faz com que cada `x[i]` aponte para um segundo arranjo de tamanho 5, como representado a seguir:



Assim como arranjos multidimensionais alocados automaticamente, podemos utilizar o operador de subscrito `x[i][j]` para acessar os elementos da linha `i` e coluna `j`. Nas linhas 11 a 12 temos uma estrutura de repetição de libera a memória apontada pelos elementos `x[i]` antes de liberarmos o arranjo apontado por `x`, na linha 13.

12. Processamento de arquivos sequenciais

Todos os dados que guardamos nas variáveis em nossos programas são temporários. Sendo assim, todos eles deixam de existir logo após o fim do programa. Os arquivos em unidades externas de armazenamento são utilizados para guardar esses dados de modo permanente em outros dispositivos.

Os objetos `cin` e `cout`, são criados quando `<iostream>` é incluído. O fluxo desses objetos criam um canal de comunicação do programa com dispositivos de entrada e saída, como o monitor e o teclado. Com a inserção de `<fstream>`, podemos criar objetos que possibilitam a comunicação do programa com arquivos. Isto é representado na Figura 12.1.

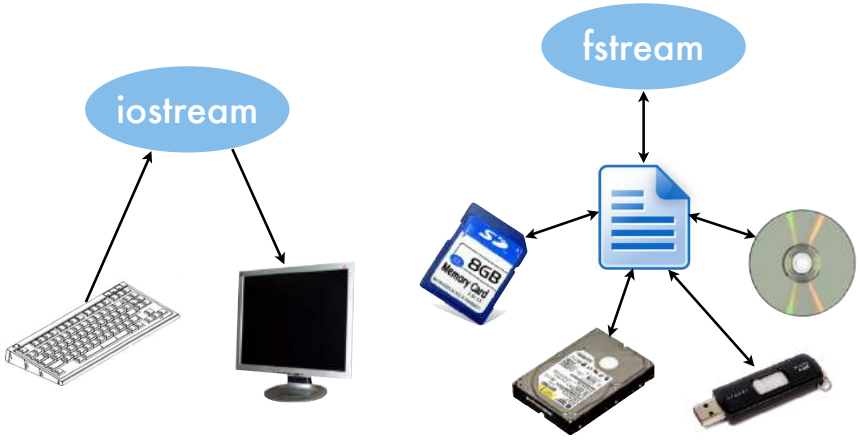


Figura 12.1: Bibliotecas `iostream` e `fstream` para fluxo de entrada e saída. A biblioteca `fstream` faz fluxo de entrada e saída através de arquivos.

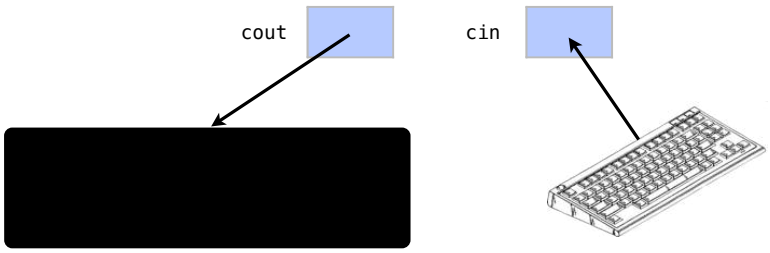
Os objetos de `<fstream>` podem ser manipulados assim como os de `<iostream>`. Os objetos dos tipos `<ofstream>` ou `<ifstream>` funcionam de maneira similar aos objetos `cout`

e `cin`: o operador de inserção (<<) insere dados no arquivo; o operador de extração (>>) lê os dados do arquivo.

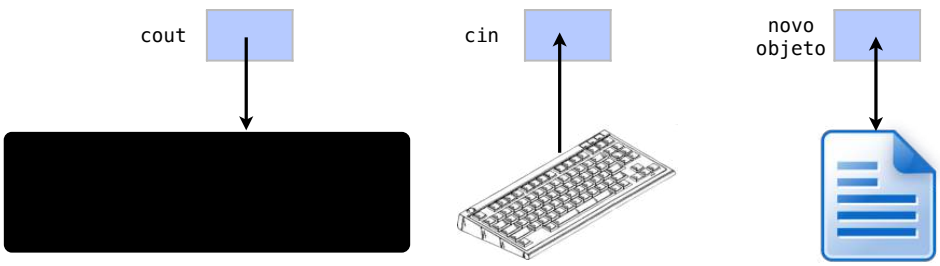
Neste exemplo, faremos um programa que guarda dados em um arquivo:

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  int main() {
8     ofstream fout("pessoas.txt");
9     string nome;
10    int idade;
11    int opcao;
12    do {
13     cout << "Digite um nome: ";
14     cin >> nome;
15     cout << "Digite a idade de" << nome << ": ";
16     cin >> idade;
17     fout << nome << " " << idade << endl;
18     cout << "Deseja continuar? (0 = Não, 1 = Sim): ";
19     cin >> opcao;
20    } while (opcao!=0);
21    fout.close();
22    return 0;
23 }
```

Como mencionado, a inserção de `<iostream>`, na linha 1 do código, cria os objetos `cin` e `cout`. Este objetos fazem o fluxo de saída e entrada no programa. Apesar de não representá-los graficamente, estes objetos estão constantemente disponíveis ao longo de nosso programa:

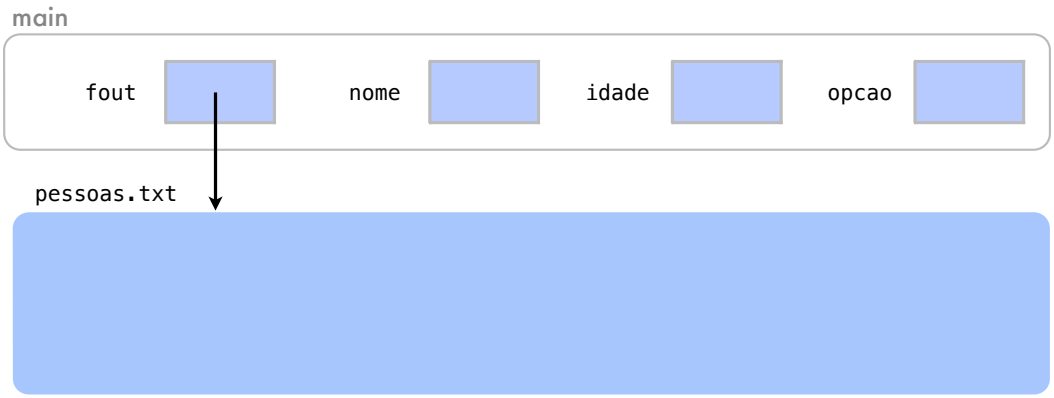


Com a inserção de `<fstream>`, na linha 2 do programa, podemos criar outros objetos que façam fluxo de saída e entrada com arquivos do computador.



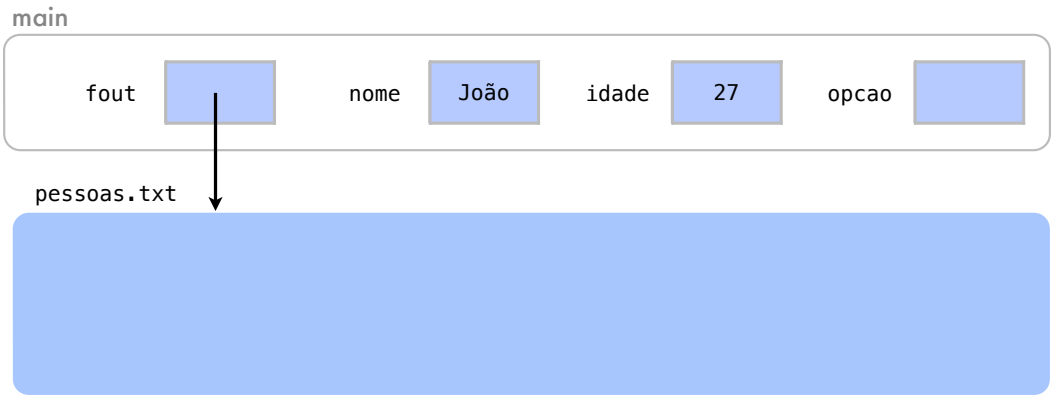
Veja que o fluxo de relacionamento com o arquivo pode ser nas duas direções, de acordo com a seta.

Na linha 8 do código, em `main`, criamos um objeto `fout` do tipo `ofstream`, para saída de dados. Esse objeto é conectado a um arquivo chamado `pessoas.txt`. Criamos também, as variáveis `nome`, `idade` e `opcao` nas linhas 9 a 11.

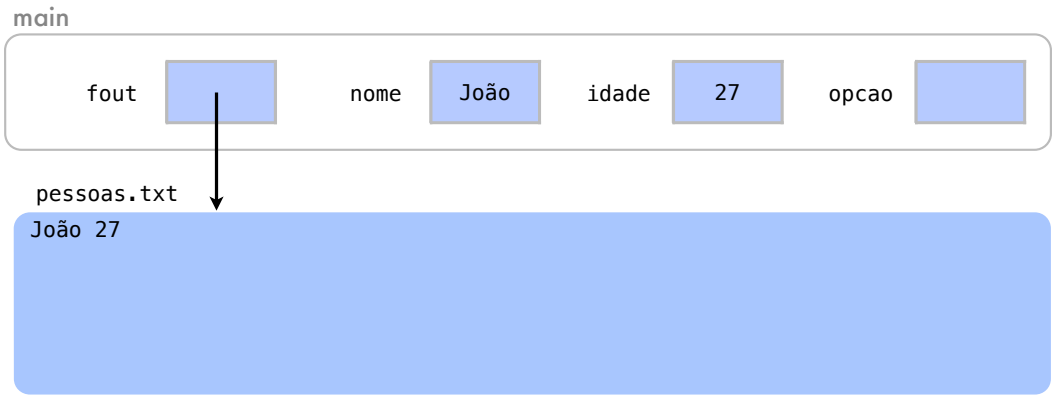


Nas linhas 12 a 20, temos uma estrutura de repetição na qual colocaremos dados no arquivo através de `fout`. Nas linhas 13 a 16, já dentro do laço, perguntamos ao usuário um nome e uma idade e atualizamos as variáveis. Isso é feito através dos fluxos `cin` e `cout`.

```
Digite um nome:  João
Digite a idade de João:  27
```



Colocamos agora, na linha 17, os valores de `nome` e `idade` em nosso arquivo através do objeto `fout`. Os dados são enviados para o arquivo de texto do mesmo modo que seriam enviados para a tela, caso `cout` fosse utilizado.

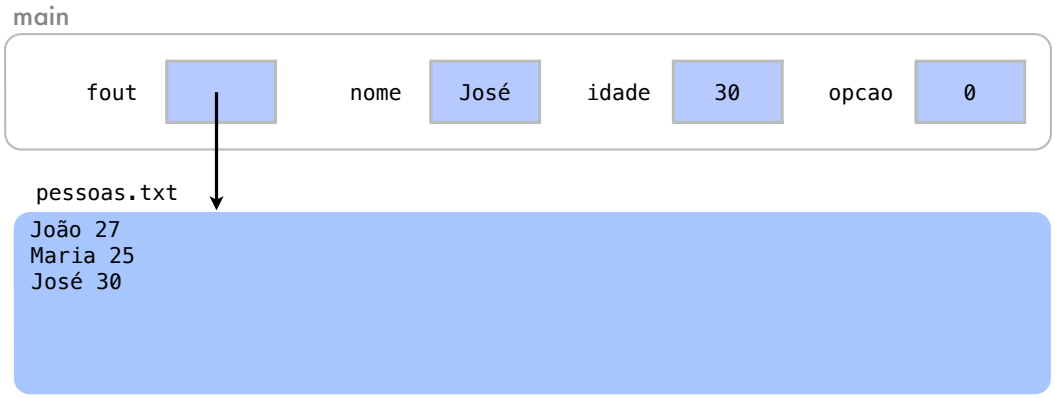


O usuário, nas linhas 18 e 19, escolhe continuar com uma opção diferente de `0`, que é a condição do laço.

```
Digite um nome:  João
Digite a idade de João:  27
Deseja continuar?  (0 = Não, 1 = Sim):  1
```

Repetimos o processo por mais algumas vezes. Em outras iterações do laço, o usuário insere mais dois nomes até que decide sair.

```
Digite um nome:  João
Digite a idade de João:  27
Deseja continuar?  (0 = Não, 1 = Sim):  1
Digite um nome:  Maria
Digite a idade de Maria:  25
Deseja continuar?  (0 = Não, 1 = Sim):  1
Digite um nome:  José
Digite a idade de José:  30
Deseja continuar?  (0 = Não, 1 = Sim):  0
```



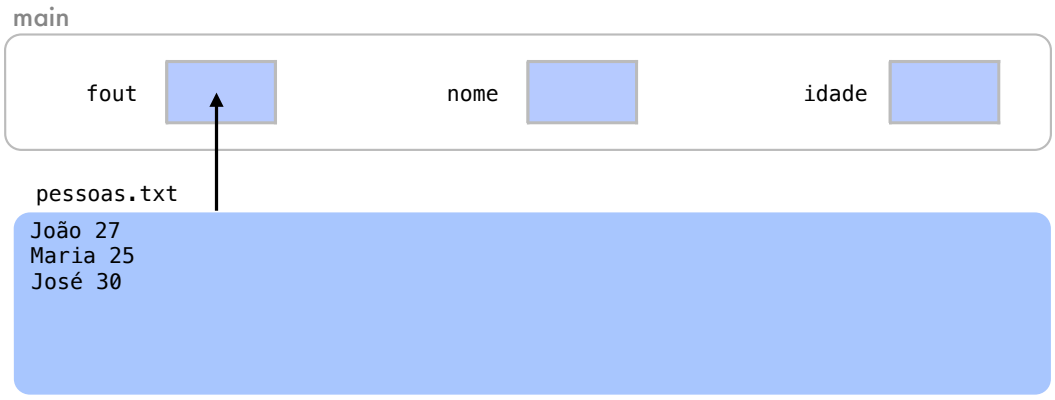
Como a opção do usuário foi `0` na último iteração, saímos do laço. Na linha 21, após o laço, fechamos o arquivo `pessoas.txt` e salvamos os dados. A função `close()` pertence ao próprio objeto `fout`. A função `open("arquivo.txt")` poderia ser utilizada para associar `fout` a outro

arquivo. Em nossa memória secundária, na pasta onde foi executado o programa, temos agora um arquivo `pessoas.txt`. No arquivo temos todos os nomes e idades digitados.

Em nosso segundo exemplo, criaremos um fluxo para obter informações de um arquivo.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  using namespace std;
6
7  int main() {
8     ifstream fin("pessoas.txt");
9     string nome;
10    int idade;
11    while (!fin.eof()){
12     fin >> nome;
13     fin >> idade;
14     cout << nome << "␣" << idade << endl;
15    }
16    fin.close();
17    return 0;
18 }
```

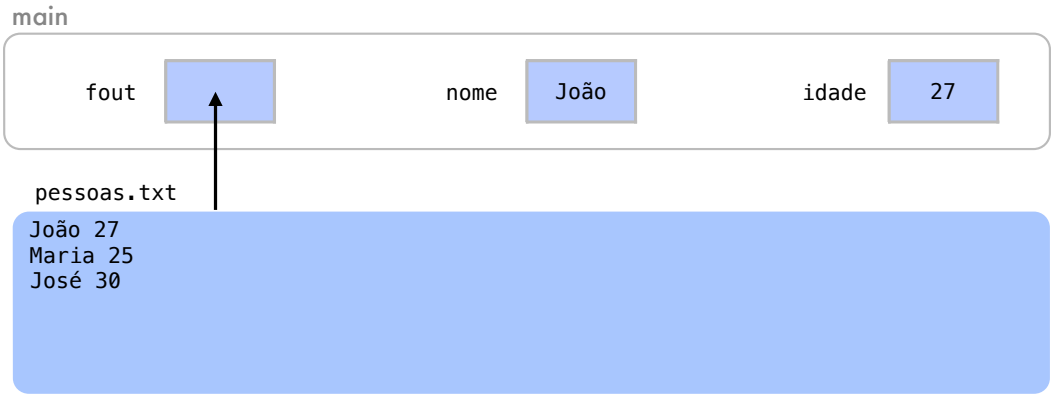
Para ler um arquivo, na linha 8, primeira do `main`, criamos um objeto do tipo `ifstream` para um fluxo de entrada através de arquivo. O arquivo já contém as informações que serão lidas. Criamos também, nas linhas 9 e 10, as variáveis `nome` e `idade` para armazenar temporariamente as informações do arquivo.



Na linha 11, A função `eof()` é uma abreviação para *end of file* (ou *fim de arquivo*, em inglês). Esta função retorna um `bool` indicando se já estamos no fim do arquivo. No momento ela retornará `false` pois acabamos de abrir o arquivo e estamos ainda no seu início. Na linha 11, em outras palavras, a expressão completa do `while` diz que repetiremos o bloco de comandos enquanto não estivermos no fim do arquivo.

Nas linhas 12 e 13, usamos o objeto `fin` para ler um nome do arquivo para a variável `nome` e depois um número para outra variável `idade`. Apesar dos dados virem do arquivo, estes comandos funcionam da forma muito similar a um `cin`. Em seguida, na linha 14, imprimimos os valores lidos e guardados nas variáveis.

```
João 27
```



De acordo com a condição da linha 11, como estamos dentro de um laço de repetição e ainda não estamos no fim do arquivo, vamos ler o próximo nome e a próxima idade. Imprimimos com `cout`.

```
João 27
Maria 25
```

Mas este ainda não é o fim do arquivo. Lemos o outro nome, outra idade, e imprimimos o resultado:

```
João 27
Maria 25
José 30
```

Chegamos ao fim do arquivo e por isso o bloco de comandos não se repetirá.

Na linha 16, fechamos então o arquivo antes de encerrar o programa. O comando `open(‘‘arquivo.txt’’)` também poderia ser utilizado para associar `fin` a outro arquivo. O arquivo `pessoas.txt` continua em nossa memória secundária e não foi alterado.

12.1 Exercícios

Exercício 12.1 Analise o código abaixo, que está incompleto.

```
1  // biblioteca para manipulacao de arquivos
2  // (entrada e saida por arquivos)
3  #include <fstream>
4  #include <iostream>
5
6  using namespace std;
7
8  int main(){
9      double n1, n2, n3;
10     // declaração do arquivo de entrada
11     ifstream arq_entrada;
12     arq_entrada.open("teste.xyz");
```

```
13      // enquanto não chega ao fim do arquivo
14      while (!arq_entrada.eof()){
15          // coloca dados em n1, n2 e n3
16          arq_entrada >> n1 >> n2 >> n3;
17          // imprime n1, n2 e n3
18          cout << n1 << "␣" << n2 << "␣" << n3 << endl;
19      }
20      // fechamento do arquivo
21      arq_entrada.close();
22      return 0;
23  }
```

■

Exercício 12.2 Utilize um editor de textos para criar um arquivo "`teste.xyz`" na pasta onde será executado o programa. O arquivo deverá conter três números reais em cada linha separados por espaço, em um formato como o apresentado abaixo:

```
5.2 8.5 4.6
8.3 3.2 6.9
7.8 9.8 1.2
```

Execute o programa. ■

Exercício 12.3 Se o programa for executado corretamente, execute-o novamente, agora para um arquivo "`pontos.xyz`" com milhares de linhas. Estes pontos serão considerados dados de levantamento topográfico de uma região. ■

Exercício 12.4 Defina a estrutura Ponto com os seguintes membros

```
1  struct Ponto {
2      double x,y,z;
3  };
```

■

Exercício 12.5 Altere o programa para que os dados dos pontos sejam armazenados em um arranjo cujos elementos sejam do tipo `Ponto`. O arranjo deve ter tamanho 2000. O número de elementos do arranjo preenchidos com elementos válidos deve ser contado. ■

Exercício 12.6 Escreva uma função para encontrar o ponto mais alto da região. O protótipo da função deve ser:

`int pontoMaisAlto(Ponto pontos[], int n);`

onde `n` é o número de pontos no arranjo. A função deverá retornar o índice do ponto encontrado no arranjo. ■

Exercício 12.7 Escreva uma função para determinar os limites, no eixos x e y, da região ocupada pelos pontos. O protótipo da função deve ser:

```
   void limites(Ponto pontos[], int n, double &minX, double &maxX, double
&minY, double &maxY);
```
■

Exercício 12.8 Refaça o programa de maneira que a tamanho do arranjo para os pontos seja alocado dinamicamente com a instrução `new`.

O arquivo deve ser lido uma vez apenas para se fazer a contagem do número de pontos. A alocacão será feita então com o número exato de pontos e os dados serão então inseridos no arranjo.

Para se retornar a leitura para o inicio do arquivo, use os comandos:

```
arq_entrada.clear();
arq_entrada.seekg(0, ios::beg);
```

■

13. Resumo de comandos

Nesta seção apresentamos um resumo de comandos em C++. Estes comandos podem ser utéis para recordar os conceitos deste capítulo ou como referência para leitores que já são programadores porém não estão acostumados com a linguagem C++.

13.1 Estrutura base de programas

```
#include <iostream>
using namespace std;
int main(){
    // ...
    return 0;
}
```

13.2 Tipos fundamentais de dados

```
// Dado lógico
bool nome;
// Números inteiros
short nome;
int nome;
long nome;
// Números reais
float nome;
double nome;
// Caracteres
char nome;
```

13.3 Cadeias de caracteres

```
// Adicionar cabeçalho para strings
```

```
#include string
// Cadeia (tipo de dado não fundamental)
string nome("texto");
```

13.4 Operadores básicos

```
int x = valor;
// operadores aritméticos
x = a + b; // soma
x = a - b; // subtração
x = a / b; // divisão
x = a * b; // multiplicação
x = a % b; // resto
// operadores de incremento e decremento
x++; // incremento
x--; // decremento
// operadores de atribuição aritméticos
x += a;
x -= a;
x *= a;
x /= a;
x %= a;
```

13.5 Operadores relacionais

```
// maior que
valor1 > valor2
// menor que
valor1 < valor2
// maior ou igual que
valor1 >= valor2
// menor ou igual que
valor1 <= valor2
// igual
valor1 == valor2
// diferente
valor1 != valor2
```

13.6 Operadores lógicos

```
// e lógico
dadológico1 && dadológico2
// ou lógico
dadológico1 || dadológico2
// negação
!dadológico1
```

13.7 Estruturas condicionais

```
// seleção única
if (condição){
    ...
```

```
}
// seleção dupla
if (condição){
    ...
} else {
    ...
}
// seleção múltipla
if (condição){
    ...
} else if (condição2) {
    ...
} else if (condição3) {
    ...
} else {
    ...
}
```

```
// operador ternário
x = condição?expressão1:expressão2;
```

```
// switch para seleção múltipla
switch(variável){
    case constante1:
        ...
        break;
    case constante2:
        ...
        break;
    case ...
    ...
    default:
        ...;
}
```

13.8 Estruturas de Repetição

```
// for (para repetições baseadas em contador)
for (inicializa; condição; incremento){
    ...
}
```

```
// while (para repetições baseadas em critério de parada)
while (condição) {
    ...
}
```

```
// do-while (executa o laço uma vez antes de testar a condição)
do {
    ...
} while (condição);
```

13.9 Entrada e saída

```
// cout para saída na tela
cout << "Mensagem na tela" << endl;
// cout sem endl para não fazer quebra de linha
cout << "Digite um valor:";
// Entrada pelo teclado
cin >> variavel;
```

13.10 Arranjos (Alocação automática, de tamanho fixo)

```
Tipo nome_do_vetor[tamanho_do_vetor];
nome_do_vetor[posição] = valor;
```

13.11 Arranjos multidimensionais (Alocação automática, de tamanho fixo)

```
Tipo nome[tamanho1][tamanho2];
nome[posição1][posição2] = valor;
```

13.12 Ponteiros

```
Tipo *px; // ponteiro para o tipo Tipo
px = &x; // ponteiro = endereço de x
*px ou p[0] // Valor de x
px // Valor de (endereço salvo em) px
&nomep // Endereço do ponteiro px
*(px+1) ou p[1] // Elemento salvo após x
*px+1 // Mesmo que x + 1
```

13.13 Funções

```
// cabeçalho da função
tipo nome(tipo parametro, tipo parametro, ...);
// ...
// função principal
int main(){
    ...
}
// definição da função
tipo nome(tipo parametro, tipo parametro,...){
    // ... definição
}
```

13.14 Passagem por valor e referência

```
Tipo funcao(tipo por_valor, tipo &por_referência);
```

13.15 Estruturas ou Registros

```
struct Nome{
    tipo variavel1;
    tipo variavel2;
    ...
};
```

```
// ...
Nome x; // Cria variável do tipo Nome
x.variavel1 = valor; // Atribuição
Nome *px = x; // Ponteiro para registro
px->variavel1 = valor; // Atribuição
```

13.16 Alocação dinâmica de arranjos

```
cin >> n; // n é o tamanho do vetor
tipo *v = nullptr; // v aponta para nada
v = new tipo[n]; // v aponta arranjo alocado com tamanho n
//...
delete [] v; // deleta o arranjo
v = nullptt; // v aponta para nada de novo
```

13.17 Entrada e saída de arquivos

```
#include <fstream>
// ...
ofstream arquivo_saida("saida.txt");
istream arquivo_entrada("entrada.txt");
arquivo_saida << variavel;
arquivo_entrada >> variavel;
arquivo_saida.close();
arquivo_entrada.close();
```

II Comparando Algoritmos

14. Análise de Algoritmos

14.1 Algoritmos

Um algoritmo é um procedimento de passos para cálculos e o mesmo é composto de instruções que definem uma função. Até o momento vimos apenas códigos em C++ para descrever ideias. Um algoritmo pode descrever ideias para pessoas programarem em outras linguagens de programação. Abaixo temos a mesma ideia descrita com um código em C++ e com um algoritmo genérico:

```
1  int max(int[] a, int n){
2      int i, m;
3      m = a[0];
4      for (i = 1; i < n; i++){
5          if (m < a[i]){
6              m = a[i];
7          }
8      }
9      return m;
10 }
```

Entrada: Conjunto de números inteiros A
Saída: Maior elemento deste conjunto

$m \leftarrow$ primeiro elemento de A, ou A_1
Para todos os outros elementos A_i de A
    se o elemento A_i for maior que m
        $m \leftarrow A_i$
retorne o valor de m

Veja que temos duas descrições de uma sequência de passos para encontrar o maior elemento de um conjunto. No algoritmo à direita, os passos são descritos de maneira geral, de modo que possam ser utilizadas por qualquer pessoa. No código à direta, o conjunto de números está representado concretamente através de arranjos.

No código em C++ a entrada é o conjunto de números inteiros representado pelo arranjo `a` e a saída é o maior elemento desse conjunto. A variável `m` guarda o primeiro número do arranjo na linha 3. Uma estrutura de repetição nas linhas 4 a 8 compara se há algum elemento `a[i]` maior que `m`.

De maneira análoga, no algoritmo à direita, a variável m é então o primeiro elemento de A, ou A_1. Comparamos todos os outros elementos A_i para $i = 2, 3, 4, \ldots, n$ com m. O valor de m é substituído se $A_i > m$.

Os algoritmos são expressões mais gerais de regras para sequências de passos que resolvem problemas. Apesar de ser um conjunto de regras que define uma sequência de passos, não existe uma definição formal e geral de algoritmo. Existem até hoje muitas dúvidas sobre a definição formal de um algoritmo:

- Algoritmos que não fazem cálculos são algoritmos?
- Um algoritmo precisa sempre parar? Um programa que não para é um algoritmo?
- Programas que só param com intervenção do usuário são algoritmos?
- Programas que dependem do usuário são algoritmos?
- Algoritmos podem ter componentes aleatórios?

Os algoritmos podem ser descritos de várias maneiras. A própria descrição de um código em C++ é uma maneira de descrever um algoritmo. Podemos expressar um algoritmo em uma linguagem natural (em português), com um pseudo-código (uma listagem das operações matemáticas), com diagramas (usando quadrados e setas) ou com as próprias linguagens de programação (como C++, Java, Pascal, etc).

14.2 Modelos de Comparação

Estudamos algoritmos em geral em vez de códigos em linguagens de programação porque queremos conclusões amplas sobre as análises que fazemos dos algoritmos. Em programação, essa análise mais ampla permite ajudar pessoas que utilizam diferentes linguagens.

Nem sempre dois algoritmos que resolvem o mesmo problema são igualmente eficientes. Podemos comparar algoritmos sob vários critérios como velocidade, gasto de memória e qualidade das soluções.

A **medida de custo real** é quando executamos dois algoritmos na prática e comparamos seus resultados lado a lado. Esta é usualmente uma medida inadequada visto que depende muito do sistema computacional que temos para os experimentos. Os computadores podem variar em *hardware* ou quantidade disponível de memória. O compilador ou a linguagem de programação podem ter alguma característica que altera o desempenho de um código em relação ao outro.

Por outro lado, uma medida de custo real pode ser vantajosa quando dois algoritmos tem comportamento muito similar e os custos reais das operações são então considerados em ambientes mais práticos.

Para contornar os problemas relacionados a medidas de custo real, utilizamos um **medida por modelos matemáticos** para comparar diferentes algoritmos. Nestas medidas especificamos um conjunto de operações e seus custos. Usualmente, apenas as operações mais importantes do algoritmo são consideradas, como comparações, atribuições e chamadas de função.

14.3 Complexidade dos algoritmos

Para mensurar o custo de um algoritmo, definimos uma **função de custo** ou **função de complexidade**, denominada $f(n)$. Esta função $f(n)$ pode medir o tempo ou memória necessários para executar um algoritmo.

Considere este algoritmo que retorna o maior elemento de um arranjo `a` com `n` elementos:

```
1 int max(int a[], int n){
2     int i, temp;
3     temp = a[0];
4     for (i = 1; i < n; i++){
5         if (temp < a[i]){
6             temp = a[i];
7         }
```

```
8       }
9       return temp;
10  }
```

Na linha 3, ele guarda o valor do primeiro elemento do arranjo `temp`. A variável `temp` guarda o valor do maior elemento encontrado até então. Na linha 5, ele compara o maior elemento encontrado até então `temp` com todos os outros elementos `a[i]` do arranjo. O `for` definido na linha 4 varia os valores de `i` entre `1` e `n-1`.

Para um arranjo de n elementos e uma função $f(n)$ do número de comparações, temos que o custo deste algoritmo é $f(n) = n - 1$. Todas estas comparações são feitas na linha 5 do algoritmo.

Repare que a operação relevante desse algoritmo é o número de comparações. No caso geral, não é interessante medir o número de operações de atribuição pois elas só ocorrem quando a comparação da linha 5 retorna `true`.

14.4 Melhor caso, pior caso e caso médio

Suponha agora um computador no qual as operações de atribuição são muito custosas. Estamos então interessados no número de operações de atribuição como medida de tempo.

No algoritmo que acabamos de ver, se definirmos o número de atribuições em como medida relavante de custo $f(n)$, o **melhor caso** em relação a esta medida é quando a comparação da linha 5 é sempre `false`. Assim, temos apenas uma operação de atribuição na linha 3, já que o comando da linha 6 nunca será executado.

Este melhor caso ocorre quando o primeiro elemento do conjunto já é o maior elemento. No melhor caso, podemos dizer que $f(n) = 1$, pois sempre teremos apenas uma operação de atribuição.

Na situação contrária, se a comparação da linha 5 é sempre `true`, temos nosso **pior caso** em relação ao número de atribuições, pois o comando da linha 6 sempre será executado.

Isso ocorre quando os elementos do arranjo estão em ordem crescente. Nesse pior caso, podemos dizer que temos custo $f(n) = n$ em relação ao número de atribuições, pois teremos 1 operação de atribuição na linha 3 e teremos mais $n-1$ operações de atribuição para cada outro elemento do arranjo na linha 6.

Já no **caso médio**, ou caso esperado, é a média de todos os casos possíveis. O caso médio de um algoritmo é sempre muito mais difícil de ser obtido pois depende da nossa estimativa de probabilidades de cada entrada. Para fins didáticos, suponha que a probabilidade de uma comparação ser `true` na linha 5 é de 50%. Neste caso, podemos dizer que nosso custo no caso médio é $f(n) = 1 + (n-1)/2 = (n+1)/2$.

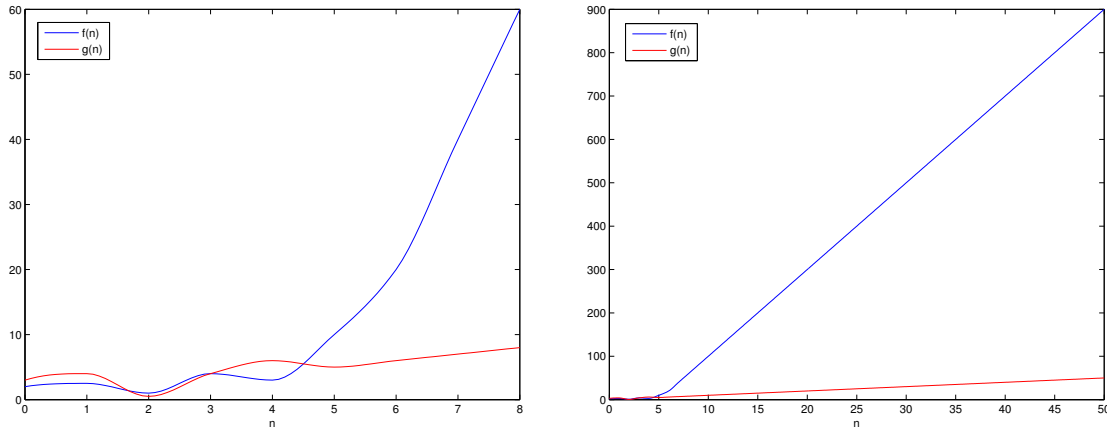
Porém, note que nossa suposição de que 50% das comparações serão `true` é pouco provável na prática. Para arranjos aleatórios, a probabilidade de uma comparação `true` na linha 5 cai à medida que `temp` guarda valores maiores.

14.5 Notação O e dominação assintótica

A notação O é utilizada para descrever a tendência de uma função. Ela é muito utilizada em computação para classificar algoritmos de acordo com a taxa de crescimento de suas funções de custo.

Na Figura 14.1 temos a gráficos representando duas funções $f(n)$ e $g(n)$. Veja que no gráfico da Figura 14.1(a), que quando $n < 4,5$, a função $g(n)$ pode estar acima de $f(n)$. Porém, para qualquer $n > 4,5$, $f(n)$ está sempre acima de $g(n)$. Isso pode ser visto principalmente na Figura 14.1(b), onde são apresentados os valores de 0 a 50 para n. Como $f(n)$ domina assintoticamente $g(n)$, temos que, *para valores grandes de n*, $f(n)$ está sempre acima de $g(n)$. Neste caso,

qualquer valor de n acima de $4,5$ pode ser considerado um "valor grande". Este valor $4,5$ a partir do qual $f(n)$ domina $g(n)$ é uma constante que chamaremos de m.



(a) $0 < n < 8$ (b) $0 < n < 50$

Figura 14.1: Uma função $f(n)$ que domina assintoticamente a função $g(n)$. Cada gráfico mostra uma faixa de valores.

Do ponto de vista de programação, suponha que $f(n)$ e $g(n)$ são funções de custo de algoritmos A e B. Se para valores grandes de n, $f(n) > g(n)$, sabemos que o algoritmo A consome mais recursos que B.

Veja na Figura 14.2 o efeito de se multiplicar $f(n)$ por uma constante c que pode ter diversos valores. No gráfico, $f(n)$ foi multiplicado pelas constantes $c = \{1;2;3;0,5;0,2\}$. Veja como mesmo quando multiplicamos $f(n)$ por uma constante, $f(n)$ ainda está acima de $g(n)$ *para valores grandes* de n. É claro que ao alterar c, temos uma alteração no valor m, que define o que é um valor grande o suficiente. Contudo, $f(n)$ continua dominando assintoticamente $g(n)$. De fato, se acharmos qualquer constante c para que $f(n)$ esteja acima de $g(n)$, podemos dizer que $f(n)$ domina $g(n)$ assintoticamente.

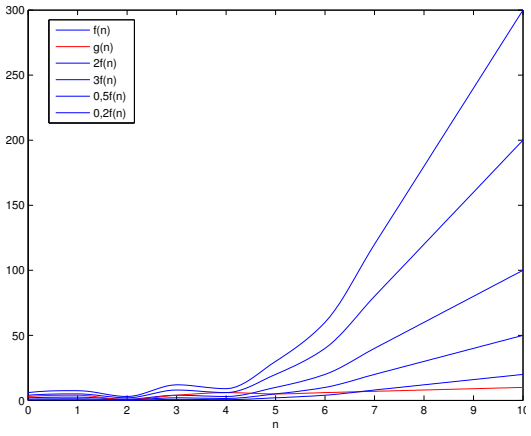


Figura 14.2: Os valores de $f(n)$ multiplicado por diferentes constantes. A função $f(n)$ domina assintoticamente a função $g(n)$ mesmo quando multiplicada por uma constante.

Do ponto de vista de programação, esta informação é muito relevante. Suponha novamente que $f(n)$ e $g(n)$ são funções de custo dos algoritmos A e B. Sabemos agora que para valores grandes de n, não só $f(n) > g(n)$ mas também $cf(n) > g(n)$ para alguma constante c. Como $f(n) > g(n)$, poderíamos pensar que o algoritmo A seria mais eficiente que B se tivessemos um

computador 1000 vezes mais rápido para A. Porém, se $0,001f(n) > g(n)$ para valores grandes n, o algoritmo A ainda seria pior que B. Ainda, se $cf(n) > g(n)$ para valores grandes n e para pelo menos alguma constante c, o algoritmo A ou é pior que B ou pode ser pior que B, caso B seja executado em um computador mais rápido.

De modo formal, o conceito de dominação assintórica pode ser definido como abaixo:

Definição 14.5.1 — Dominação Assintótica. Dizemos que uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos que $|g(n)| \leq c \times |f(n)|$. Ou seja, **para números grandes** $n > m$, $cf(n)$ será sempre maior que $g(n)$, para alguma constante c.

14.5.1 Notação O

A notação O é utilizada para representar dominação assintótica. Quando uma função $g(n) = O(f(n))$, dizemos que:

- $g(n)$ é "O" de $f(n)$
- $f(n)$ domina $g(n)$ assintoticamente

> ! Quando $g(n) = O(f(n))$ não é correto dizer que $g(n)$ *é igual a "O" de* $f(n)$, pois a notação O não indica uma relação de igualdade, e sim de dominação. Dizemos apenas que $g(n)$ *é "O" de* $f(n)$.

É importante comparar os algoritmos em notação O pois não estamos interessados em funções exatas de custo mas sim no comportamento geral da função, principalmente à medida que temos valores maiores de n.

Vejamos o exemplo das seguintes funções:

- $g(n) = (n+1)^2$
- $f(n) = n^2$

Estas duas funções *se dominam assintoticamente*, ou seja, $g(n) = O(f(n))$ e $f(n) = O(g(n))$. $g(n) = O(f(n))$ por que $g(n) \leq 4f(n)$ para $n > 1$. Por outro lado, $f(n) = O(g(n))$ pois $f(n) \leq g(n)$ para qualquer $n > 0$.

Vejamos agora um segundo exemplo:

- $g(n) = (n+1)^2$
- $f(n) = n^3$

A função $f(n)$ domina a função $g(n)$. Temos que $g(n) = O(f(n))$ pois $g(n) \leq cf(n)$ para um c grande o suficiente. Porém, a $g(n)$ não domina $f(n)$ pois dizer que $f(n) \leq cg(n)$ a partir de qualquer n seria uma afirmação falsa para qualquer c. Não importa qual valor escolhemos para c, $f(n)$ **sempre** será maior que $g(n)$ se n for grande o suficiente.

14.5.2 Limites fortes

Temos que $g(n) = 3n^3 + 2n^2 + n = O(n^3)$, pois $g(n) \leq 6n^3$, para $n > 0$. Este é considerado um limite forte para a função $g(n)$. É claro que também podemos dizer que $g(n) = O(n^4)$, porém esta seria uma afirmação mais fraca. Se já sabemos que n^3 domina $g(n)$, é óbvido que n^4 também a dominará. Para analisar comportamentos de algoritmos, estamos interessados em limites fortes pois são deles que podemos tirar melhores conclusões sobre a eficiência de algoritmos.

14.5.3 Operações com notação O

Vejamos agora algumas operações e propriedades da representação de funções através de notação O. Estas propriedades deixam claro porque a notação O é útil para representar de

maneira simples o comportamento de funções e, no nosso caso de interesse, de programas e algoritmos.

Uma função $f(n)$ sempre se domina pois basta a multiplicar por uma constante $c > 1$ para que $cf(n) > f(n)$:

$$f(n) = O(f(n))$$

O mesmo vale para qualquer função $O(f(n))$ multiplicada por uma constante c, pois basta multiplicarmos $f(n)$ por uma constante c_2 grande o suficiente para que $c_2 > c(f(n))$:

$$cO(f(n)) = O(f(n))$$

A soma de duas funções dominadas por $f(n)$ é ainda dominada por $f(n)$ pois esta diferença ainda pode ser compensada por uma constante:

$$O(f(n)) + O(f(n)) = O(f(n))$$

É desta última propriedade que podemos facilmente calcular a ordem de complexidade de vários algoritmos. Por exemplo, se temos uma função de custo $g(n) = n^3 + n^2 + n$, podemos identificar rapidamente que $g(n) = O(n^3)$ já que n^3 é o maior termo entre os elementos somados em $g(n)$.

Se uma função $O(O(f(n))$ (ou seja, uma função que é dominada por uma função dominada por $f(n)$), a primeira função é também dominada por $f(n)$:

$$O(O(f(n))) = O(f(n))$$

A soma de duas funções, uma dominada por $f(n)$ e outra dominada por $g(n)$, será dominada pela maior função que as domina:

$$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

A multiplicação de duas funções será dominada pela multiplicação das funções que as dominavam:

$$O(f(n))O(g(n)) = O(f(n)g(n)) \qquad f(n)O(g(n)) = O(f(n)g(n))$$

14.6 Classes de algoritmos

Com as funções de complexidade de cada algoritmo, podemos compará-los em relação a suas funções de custo. No caso geral, se tivermos limites fortes e para valores grandes de n, se sabemos que um programa leva o tempo $O(n)$, ele é melhor que um outro programa que leva tempo $O(n^2)$.

A Tabela 14.1 apresenta algumas classes importantes de algoritmos de acordo com suas complexidades. A Tabela mostra a ordem da função de complexidade, como estes algoritmos são chamados e que tipo de algoritmo usualmente tem este tipo de complexidade. Em todos os exemplos, n é considerado o tamanho do problema e c é considerada uma constante qualquer.

Notação	Complexidade	Algoritmo Típico
$O(1)$	Constante	Procurar e alterar uma posição de um arranjo
$O(\log)$	Logarítmica	Dividem o problema em dois a cada passo
$O(n)$	Linear	Fazer uma operação em cada elemento de um conjunto
$O(n \log n)$	Log-linear	Dividem o problema e uma operação por elemento
$O(n^2)$	Quadrática	Comparações par-a-par em conjuntos de elementos
$O(n^3)$	Cúbica	Comparações par-a-par repetidas
$O(n^c), c > 1$	Polinomial	Comparações par-a-par repetidas
$O(2^n)$	Exponencial	Força-bruta em combinações
$O(c^n), c > 1$	Exponencial	Força-bruta em combinações de c elementos
$O(n!)$	Fatorial	Força-bruta em permutações

Tabela 14.1: Classes Importantes de Algoritmo por Complexidade.

Os algoritmos de custo $O(1)$, por exemplo, têm complexidade constante e eles possuem sempre o mesmo desempenho independente do valor de n. Já nos algoritmos $O(n)$, de complexidade linear, o tempo aumenta de acordo com o tamanho do problema. Os algoritmos $O(n \log n)$, na prática costumam ter desempenho próximo aos algoritmos $O(n)$, a não ser para problemas extremamente grandes.

A partir dos algoritmos $O(n^2)$, porém, o tamanho do problema por ser um entrave em nossos programas. O tempo para resolver problemas começa a crescer em proporção maior do que o tamanho do problema. Problemas que podem ser resolvidos com qualquer algoritmo que custe menos que $O(n^c), c > 1$ usualmente são chamados de problemas polinomiais.

Os algoritmos de custo $O(2^n)$, $O(c^n), c > 1$ e $O(n!)$ são usualmente algoritmos que usam força bruta para resolver problemas testando todas as soluções possíveis até que se encontre a certa. Estes algoritmos não custumam ser utilizados na prática pois um computador poderia levar séculos para resolver problemas pequenos.

14.7 Exercícios

Exercício 14.1 O que significa dizer que uma função $g(n)$ é $O(f(n))$? ■

Exercício 14.2 Explique se há alguma diferença entre $O(1)$ e $O(2)$. Justifique. ■

Exercício 14.3 Indique se as afirmativas são verdadeiras ou falsas:

1. $2^{n+1} = O(2^n)$
2. $2^{2n} = O(2^n)$
3. $f(n) = O(u(n))$ e $g(n) = O(v(n)) \rightarrow f(n) + g(n) = O(u(n) + v(n))$

■

Exercício 14.4 Se os algoritmos A e B levam tempo $a(n) = n^2 - n + 549$ e $b(n) = 49n + 49$. $a(n) = O(b(n))$? $b(n) = O(a(n))$? Para quais valores A leva menos tempo para executar do que B? ■

Exercício 14.5 Apresente o esboço de uma algoritmo para obter o maior e o segundo maior elementos de um conjunto. Apresente uma análise do algoritmo. Ele é eficiente? Porquê? ■

Exercício 14.6 Considere um algoritmo para inserir um elemento em um arranjo ordenado de elementos. Qual o número mínimo de passos para resolver este problema? Qual o melhor caso? Qual o pior caso? Qual o caso médio? ■

Exercício 14.7 Do ponto de vista assintótico, temos a seguite relação de precedência entre funções:

$1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}$

Onde $k \geq 1$ e $0 < \varepsilon < 1 < c$

Indique se A é O de B para os pares abaixo:

A	B	O
$A = \log^{k} n$	$B = n^{\varepsilon}$	
$A = n^{k}$	$B = c^{n}$	
$A = \sqrt{n}$	$B = n^{\sin n}$	
$A = 2^{n}$	$B = 2^{n/2}$	
$A = n^{\log m}$	$B = m^{\log n}$	
$A = \log(n!)$	$B = \log(n^{n})$	

■

15. Busca em arranjos

Como estudamos na Seção 5, arranjos são uma maneira de se armazenar muita informação em série. Esta informação, em aplicações práticas, é usualmente dividida em registros do C++ com o recurso `struct`, que estudamos na Seção 10. Quando temos os dados organizados desta maneira, precisamos de estratégias para encontrar dados nestes arranjos.

Usualmente, cada `struct` do arranjo tem um membro que não se repete, que chamamos de *chave*. As chaves são importantes para diferenciar os itens de um conjunto de elementos. Por exemplo, suponha um arranjo de dados do tipo `Aluno`:

```
1 struct Aluno{
2     string nome;
3     double nota_prova1;
4     double nota_prova2;
5     int matricula;
6 }
```

Nesta estrutura, cada `Aluno` terá um `nome`, uma nota para cada prova (`nota_prova1` e `nota_prova2`) e um número de matrícula (`matricula`). Como os números de matrícula não se repetem, eles são bons candidatos a serem o **membro chave** dos alunos.

Neste caso prático, sendo a chave de um aluno o seu número de matrícula, o problema da busca em arranjo consiste em encontrar em qual posição de um arranjo `a` está o aluno com número de matrícula `x`. Veja o protótipo de uma função para busca em arranjos na Figura 15.1. Uma função que faz busca em arranjos deve receber um valor de `chave`, um arranjo de elementos `a` e o tamanho `n` deste arranjo. O problema está em fazer com que a função retorne em qual posição do arranjo está o elemento com a chave pedida.

Na Figura 15.2 apresentamos o protótipo e exemplo de uso de uma função para busca em arranjos de dados do tipo `int`. Considerando que o próprio número representa sua chave, a função `busca` deve retornar em qual posição do arranjo `a` (que tem tamanho `n`) está o número `x`, que é `9`. No caso, como o elemento `9` se encontra na posição `a[4]`, a função deverá retornar `4`.

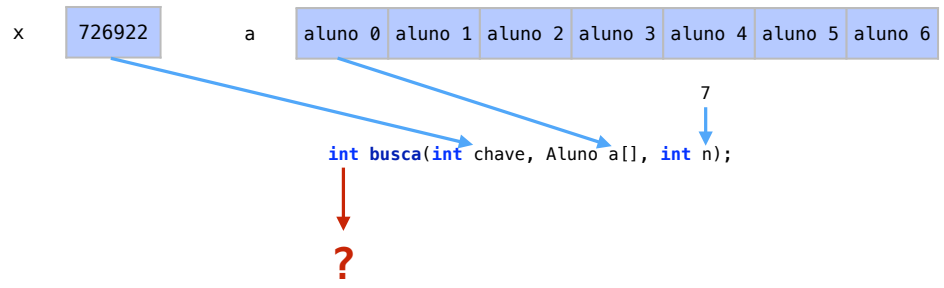


Figura 15.1: Protótipo de uma função de busca em arranjos.

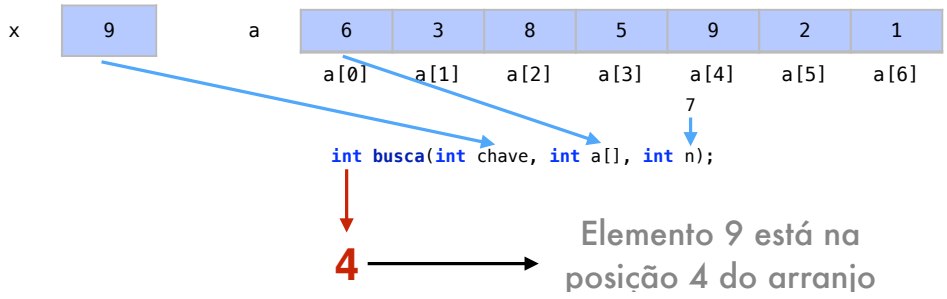


Figura 15.2: Protótipo de uma função de busca em arranjos de dados do tipo `int`.

15.1 Busca sequencial

O método mais simples para busca é, a partir do primeiro elemento, pesquisar os elementos sequencialmente um-a-um até encontrar a chave desejada. Veja a função que procura o elemento `chave` no arranjo `a`, que possui tamanho n:

```
1 int buscaSequencial(int chave, int a[], int n){
2     for (int i=0; i<n; i++){
3         if (a[i] == chave){
4             return i;
5         }
6     }
7     return -1;
8 }
```

A função irá retornar um `int` que dirá em qual posição do arranjo `a` está o elemento `chave`. A estrutura de repetição definida nas linhas 2 a 6 ocorre com `i` de `0` até `n-1` para percorrer da primeira até a última posição `a[i]` do arranjo `a`.

Na linha 3, comparamos cada elemento do arranjo com o elemento `chave`. Se o elemento `chave` foi encontrado, o valor de `i`, que tem sua posição nesta iteração da repetição, é então retornado. Já se o elemento `chave` não foi encontrado e a comparação retornou `false`, passamos para a próxima iteração de repetição.

Na linha 7, se em nenhuma das iterações o elemento foi encontrado, saímos do for e retornamos `-1`. O retorno de `-1` é um modo de se avisar para a função chamadora que o elemento não foi encontrado.

15.1.1 Análise

Temos como operação relevante da busca em arranjo o número de comparações para se encontrar o elemento `chave`. Em relação ao número de comparações, o *melhor caso* acontece quando o elemento que procuramos é o primeiro do arranjo. Neste caso, faremos apenas uma comparação e já retornaremos a posição `i`. Sendo assim, teríamos uma função de custo $f(n) = 1$ no melhor caso. Em notação assintótica, $f(n) = O(1)$.

Já o *pior caso* ocorre quando o elemento que procuramos é o último do arranjo ou não está no arranjo. Para um arranjo de n elementos, teríamos de fazer n comparações para encontrar o último elemento ou descobrir que o elemento não está no arranjo. Assim, temos uma função de custo $f(n) = n$ no pior caso. Em notação assintótica, $f(n) = O(n)$.

No caso médio, precisamos de uma suposição de probabilidades. Vamos supor que o elemento chave sempre está no arranjo e que ele tem a mesma probabilidade $1/n$ de estar em qualquer uma de suas posições. Se (i) encontrar o elemento da posição i tem custo $f(n) = i$ e (ii) a probabilidade da chave estar em uma posição i é $1/n$, teremos que nosso *caso médio* tem custo definido pela seguinte equação:

$$f(n) = 1\frac{1}{n} + 2\frac{1}{n} + 3\frac{1}{n} \ldots (n-1)\frac{1}{n} + n\frac{1}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$$

Assim, em notação assintótica, temos um caso médio onde $f(n) = O(n)$.

Na Tabela 15.1 temos um resumo do custo do algoritmo de busca sequencial em seus principais casos. Tanto o pior caso quando o caso médio têm custo $O(n)$ para encontrar um elemento.

Caso	Descrição	Custo
Pior caso	`chave` é o último elemento ou pesquisa sem sucesso	$O(n)$
Caso médio	Probabilidade $1/n$ de ter a `chave` em cada posição	$O(n)$
Melhor caso	`chave` é o primeiro elemento do arranjo	$O(1)$

Tabela 15.1: Custo do algoritmo de busca sequencial em suas situações mais comuns

15.2 Busca binária

A busca binária é um método mais eficiente que a busca sequencial. Porém, a busca binária requer que os elementos estejam mantidos em ordem dentro do arranjo. A busca binária é feita com um algoritmo onde a cada iteração de repetição, pesquisamos o elemento do meio. Se a `chave` é maior que este elemento, repetimos o processo na primeira metade do arranjo. Senão, repetimos esse passo na segunda metade do arranjo. A busca binária é um método interessante pois remete a como buscamos palavras em um dicionário.

A Figura 15.3 apresenta um exemplo da aplicação deste algoritmo. Pesquisaremos o arranjo `a` pela `chave` de valor `8`. Diferentemente da busca sequencial, não iniciaremos procurando na primeira posição `i` do arranjo `a`. O índice `i` se inicia na posição que representa a metade do arranjo (`6/2`, ou `3`).

Quando comparamos a `chave` com o elemento `a[i]`, ou `a[3]`, percebemos que a `chave` é maior que este elemento. Assim, sabemos que o elemento procurado só pode estar entre as posições `4` e `6`, ao lado direito do arranjo. Assim, os elementos `a[0]` a `a[3]` são desconsiderados e repetimos o procedimento.

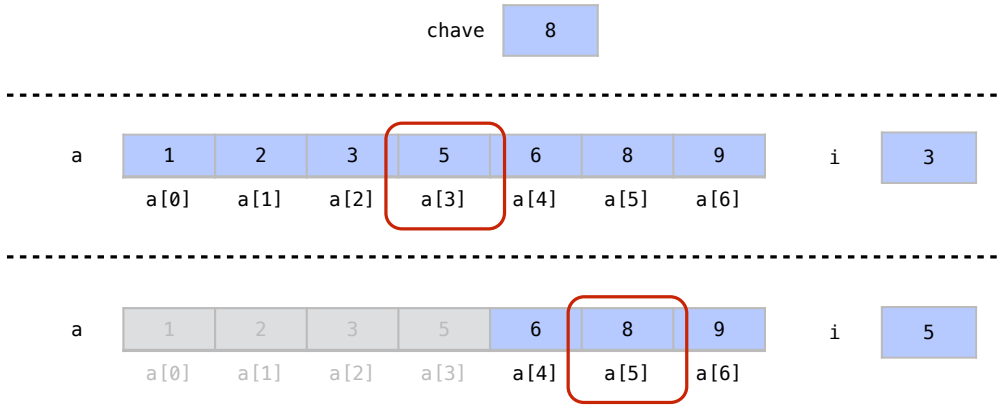


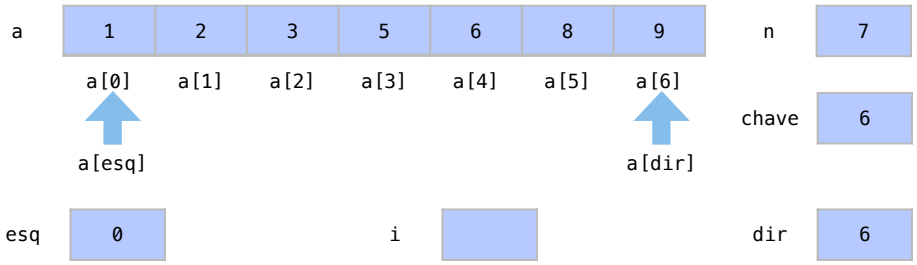
Figura 15.3: Exemplo da aplicação de um algoritmo de busca binária.

No segundo passo do algoritmo, o elemento do meio entre as posições consideradas é `a[4+6/2]`, ou `a[5]`. Como o elemento é igual à `chave` o algoritmo retorna o valor de `i`. Veja como utilizamos apenas dois passos para encontrar um elemento com a busca binária.

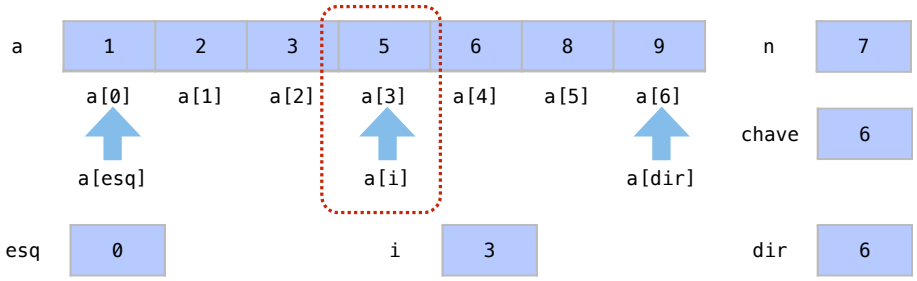
Temos abaixo o código de uma busca binária:

```
1  int buscaBinaria(int chave, int a[], int n){
2      int i;
3      int esq = 0;
4      int dir = n-1;
5      do {
6          i = (esq + dir)/2;
7          if (chave > a[i]){
8              esq = i + 1;
9          } else {
10             dir = i - 1;
11         }
12     } while ((chave != a[i]) && (esq <= dir));
13     if (chave == a[i]){
14         return i;
15     } else {
16         return -1;
17     }
18 }
```

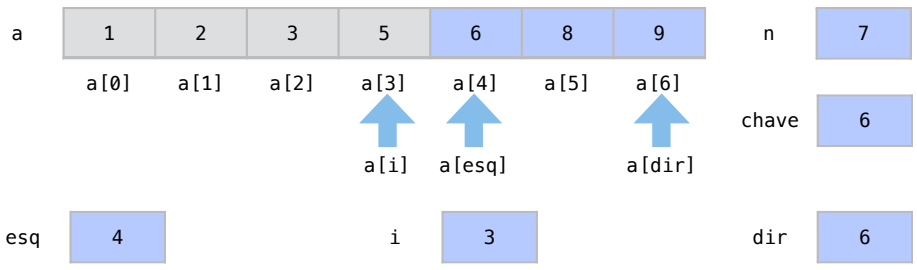
Na linha 1, similarmente à busca sequencial, a função procura o elemento `chave` do arranjo `a`, que tem tamanho `n`. O índice `i`, declarado na linha 2, marcará o elemento sendo comparado, assim como no `for` da busca sequencial. Os índices `esq` e `dir`, definidos nas linhas 3 e 4 por sua vez, marcarão os limites de onde a busca está sendo feita. Inicialmente, faremos a busca entre as posições `0` e `n-1`, ou seja, entre todos os elementos no arranjo `a`. Temos abaixo um exemplo destes passos iniciais em um arranjo `a`, no qual procuramos o elemento `6`:



Em seguida, entramos em uma estrutura de repetição definida entre as linhas 5 e 11, onde a cada iteração o elemento `i` será comparado. Na primeira iteração, com a execução da linha 6, o índice `i` indica o elemento do meio do arranjo.



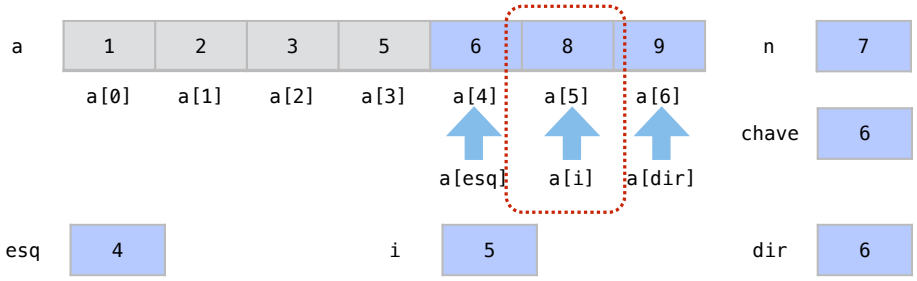
Na linha 7, comparamos a `chave` com o elemento `a[[i]` e temos que a `chave` é maior que `a[i]`. Por este motivo, pesquisaremos agora apenas na metade à direita de `i` e, a partir de agora, a metade à esquerda será desconsiderada pelo algoritmo. Para isto, atualizamos o valor de `esq` na linha 8. Em nosso exemplo, apesar os elementos entre `a[4]` e `a[6]` serão considerado a partir de agora:



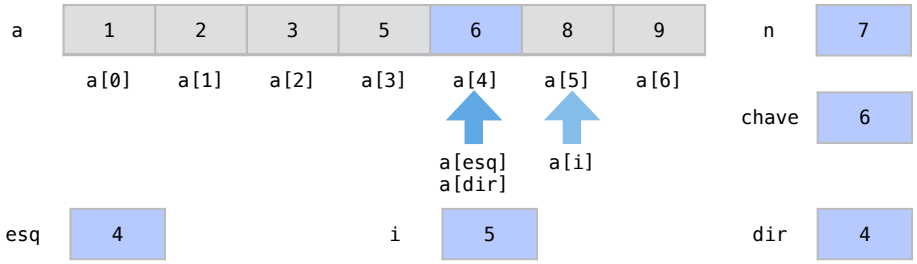
Se a `chave` fosse menor que a `a[i]`, o contrário ocorreria, e a metade à direita passaria a ser desconsiderada com a atualização de `dir` na linha 10.

Na linha 12, temos duas condições para continuar a estrutura de repetição. A primeira condição é que `chave != a[i]`, ou seja, que o elemento pesquisado `i` ainda não seja a chave sendo procurada. A segunda condição para continuar a repetição é que `esq <= dir`, ou seja, se ainda há elementos para se pesquisar. Os índices `esq` e `dir` vão se movendo a medida que pesquisamos mais elementos. Se o índice `esq` é menor ou igual a `dir`, a repetição deve se continuar pois isso indica que o arranjo inteiro não foi pesquisado.

Em nosso exemplo, como as condições do laço foram atendidas, na nova iteração do laço, na linha 6, o índice `i` indica a posição o elemento do meio entre os que ainda estão sendo considerados, ou seja, `(esq + dir)/2` = 5:

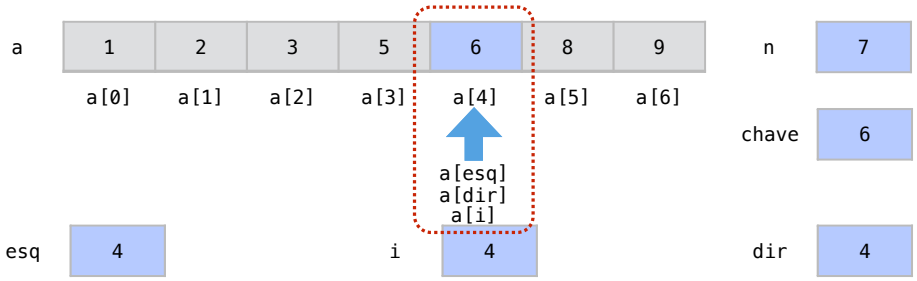


Na linha 7, como `a[5]`, ou 8, é maior que a chave 6 que procuramos, alteramos `dir`, na linha 10, para restringir a busca ao lado esquerdo do arranjo:

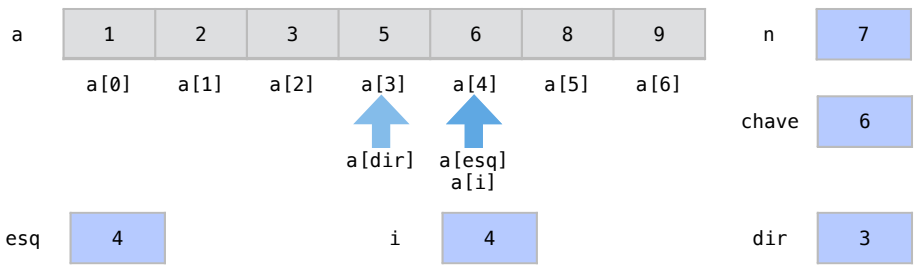


Nas condições de repetição do laço, na linha 12, o elemento `a[i]` ainda não é a chave que procuramos e os índices `esq` e `dir` não se cruzaram indicando que todo o arranjo já foi percorrido.

Assim, na próxima iteração, na linha 6, o elemento do meio indicado por `i` (`(esq + dir)/2` = 4) é o único elemento ainda considerado do arranjo.



No teste da linha 7, como a `chave > a[i]` retorna `false`, deslocamos o índice `dir` mais uma vez. O índice `dir` estar à esquerda do índice `esq` indica que não que não há mais elementos a se pesquisar.



Nas condições de parada, na linha 12, nenhuma das duas condições são atendidas. A condição `chave != a[i]` é `false` pois o índice `i` já aponta para o elemento chave e a condição `esq <= dir` também é `false` pois os índices `dir` e `esq` já se cruzaram, indicando não há mais elementos a se pesquisar.

Nas linhas 13 a 17, testamos por qual motivo o laço foi encerrado e a função faz o retorno. O primeiro teste da linha 13, verifica se `chave == a[i]`, ou se a chave foi encontrada. Se a chave foi encontrada ao fim do laço, retornamos o valor de `i` na linha 14. Em nosso exemplo, teríamos retornado 4.

Se o elemento `a[i]` não fosse igual à chave, seria porque o elemento não estava no arranjo. Neste caso, retornaríamso `-1` na linha 16 do código.

15.2.1 Análise

A busca sequencial é um método interessante de busca pois remete a como buscamos palavras em um dicionário. Quando pesquisamos em um dicionário, não passamos palavra por palavra. Procuramos uma palavra no meio do dicionário e verificamos se a palavra que procuramos estaria a frente ou atrás da palavra que encontramos.

Em relação ao número de comparações temos um cenário similar à busca sequencial para a descrição dos casos. O melhor caso ocorre quando o elemento que procuramos é o primeiro que testamos, ou seja, o elemento no meio do arranjo. Sendo assim, nosso melhor caso seria $f(n) = O(1)$.

Já o pior caso acontece quando o elemento que procuramos é o último comparado ou quando o elemento não está no arranjo. Vejamos o número máximo de comparações que podemos fazer em uma busca binária. Se no primeiro passo de nossa busca consideramos n elementos, temos $n/2$ elementos considerados no segundo passo, $n/4 = n/2^2$ elementos considerados no terceiro passo e assim por diante. Ou seja, no passo k do algoritmo, consideraríamos $n/2^k$ elementos em nossa busca, até que o elemento fosse encontrado. No pior caso, faríamos estes passos até que apenas 1 elemento seja considerado.

$$\frac{n}{1} \to \frac{n}{2} \to \frac{n}{2^2} \to \frac{n}{2^3} \to \cdots \to 1$$

Como consideramos $n/2^k$ elementos no passo k e apenas 1 elemento no último passo do pior caso, nosso último passo seria aquele no qual $n/2^k = 1$. Invertendo a equação temos que o número de passos k é:

$$k = \log n$$

Isso nosso pior caso teria $O(k) = O(\log n)$ passos no máximo. Se considerarmos que os elementos estão distribuídos de uma forma não tendenciosa no arranjo, nosso caso médio também será $f(n) = O(\log n)$.

Na Tabela 15.2 temos um resumo do custo do algoritmo de busca binária em seus principais casos. Tanto o pior caso quando o caso médio têm custo $O(\log n)$ para encontrar um elemento.

Em notação O, mesmo os piores casos são $O(\log n)$ pois, a cada passo, o algoritmo elimina metade do problema. Veja que isto coincide com o comportamento típico de algoritmos de complexidade logarítmica, como listado na Seção 14.6. Isto o deixa muito mais eficiente em relação à busca sequencial $O(n)$. Veja por exemplo que se $n = 1.000.000$, temos apenas que $\log n = 6$.

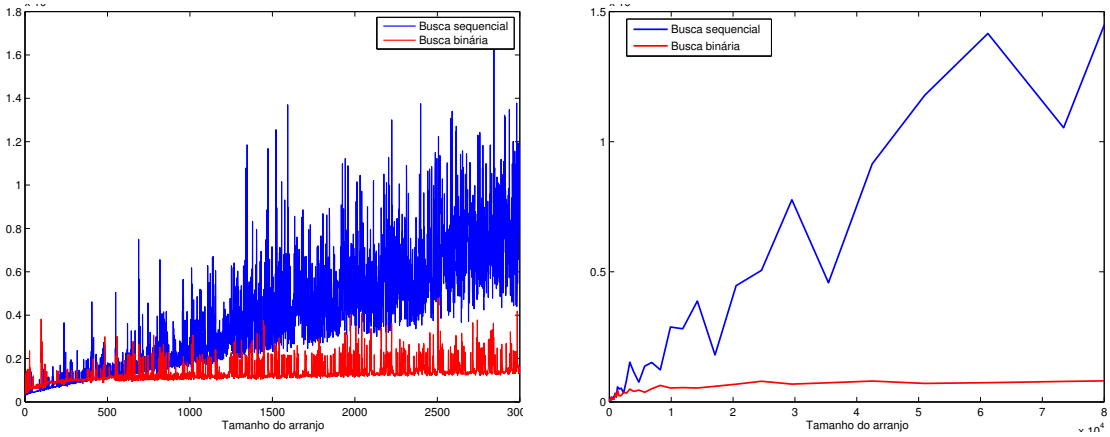
Uma desvantagem do algoritmo de busca binária é que precisamos manter o arranjo ordenado. Esta operação pode ter um custo muito alto para algumas aplicações práticas. Isso torna este

Caso	Descrição	Custo
Pior caso	`chave` é o último elemento pesquisado ou pesquisa sem sucesso	$O(\log n)$
Caso médio	Itens distribuídos de forma uniforme pelo arranjo	$O(\log n)$
Melhor caso	`chave` é o elemento do meio do arranjo	$O(1)$

Tabela 15.2: Custo do algoritmo de busca binária em suas situações mais comuns

método mais vantajoso para aplicações pouco dinâmicas, como é justamente o caso de dicionários. Estudaremos nas Seções seguintes métodos para os arranjos ordenados.

A Figura 15.4 apresenta o tempo gasto pelos algoritmos de busca sequencial e busca binária para se encontrar um elemento em um arranjo. A Figura 15.4(a) apresenta resultados para arranjos com até 3000 elementos enquanto a Figura 15.4(b) apresenta resultados para arranjos com até 80 mil elementos. Como a busca sequencial tem custo $O(n)$ e a busca binária tem custo $O(\log n)$, vemos que à medida que os arranjos ficam maiores, a busca binária se torna cada vez mais vantajosa. Porém, é possível que para arranjos pequenos, a busca sequencial seja mais vantajosa. Nestes experimentos, isto ocorre para arranjos com menos de 350 elementos.



(a) Arranjos menores (b) Arranjos maiores

Figura 15.4: Tempo necessário pelos algoritmos de busca para se encontrar um elemento em arranjos de vários tamanhos.

15.3 Exercícios

Exercício 15.1 Analise o código abaixo, que está incompleto.

```
1  #include <iostream>
2
3  using namespace std;
4
5  // busca posição de x no arranjo a de tamanho n
6  int buscasequencial(int a[], int n, int x);
7  // ordena o vetor a de tamanho n
8  void ordena(int a[], int n);
9
10 int main(){
```

```
11          // Tamanho do arranjo
12      int n;
13      cout << "Digite␣o␣tamanho␣do␣arranjo:␣";
14      cin >> n;
15      // ponteiro para o arranjo na memória
16      int *v;
17      // Alocado arranjo tamanho n
18      v = new int[n];
19      for (int i = 0; i < n; i++){
20          cout << "Digite␣o␣valor␣do␣elemento␣v[" << i << "]:
                ␣";
21          cin >> v[i];
22      }
23      int chave, posicao = 0;
24      do {
25          cout << "Digite␣o␣elemento␣que␣deseja␣buscar:␣";
26          cin >> chave;
27          posicao = buscasequencial(v, n, chave);
28          if (posicao == -1){
29              cout << "O␣elemento␣não␣existe" << endl;
30          } else {
31              cout << "Elemento␣na␣posicão␣v[" << posicao <<
                    "]" << endl;
32          }
33      } while (chave!=-1);
34      // Laço se repete até que a chave pesquisada seja -1
35      return 0;
36  }
37
38  // Procura a posição de x no arranjo a de tamanho n
39  int buscasequencial(int a[], int n, int x)
40  {
41      for (int i = 0; i<n; i++){
42          if (a[i]==x){
43              return i;
44          }
45      }
46      return -1;
47  }
48
49  // Ordenação - Método da Bolha
50  void ordena(int a[], int n)
51  {
52      int i,j,aux;
53      for( j= n-1; j>0; j--)
54          for(i=0; i<j; i++)
55          {
56              if(a[i+1] < a[i])
57              {
```

```
58                  // trocar a[i] com a[i+1]
59                  aux = a[i];
60                  a[i] = a[i+1];
61                  a[i+1] = aux;
62              }
63          }
64  }
```

■

Exercício 15.2 Crie uma função de busca binária, que busca o elemento `chave` em um arranjo ordenado. Para ordenar o arranjo, use a funcao `ordena(int a[], int n, int x)`. ■

Exercício 15.3 Altere todas as funções para que elas calculem também o número de passos necessários para se encontrar (ou não) o elemento. ■

Exercício 15.4 Altere o programa `main` para que sempre que um elemento chave seja digitado, ele seja pesquisado atraves de busca sequencial e busca binaria. ■

Exercício 15.5 Use a alteração para que logo em seguida seja impresso o número de passos necessários para se encontrar o elemento com cada um dos métodos, incluindo o passo de ordenação. ■

16. Ordenação de arranjos

É comum em programação a necessidade de se ordenar um arranjo. Um motivo razoável para isto pode ser inclusive ordenação dos dados para facilitar sua visualização. Veja na Tabela 16.1 como é muito visualizar e analisar uma lista nomes ordenados.

Desordenado	**Ordenado**
Pedro	Alberto
João	Barbosa
Maria	Joaquim
Roberto	João
Manuel	José
José	Manuel
Barbosa	Maria
Joaquim	Pedro
Alberto	Roberto

Tabela 16.1: A visualização e análise de um conjunto de elementos ordenado é muito mais fácil.

Além disto, como vimos na Seção 15.2, buscas binárias em arranjos são mais eficientes que buscas sequenciais. Porém, estas buscas só podem ser feitas em arranjos ordenados. Imagine por exemplo fazer uma pesquisa em um catálogo telefônico onde temos os nomes das pessoas em ordem.

As estratégias para se ordenar um arranjo são diversas. Veremos algumas dessas estratégicas ao longo deste livro. O processo de ordenação pode ser para colocar os itens em ordem crescente ou decrescente.

Como no caso das buscas em arranjos, os algoritmos são usualmente estendidos em situações práticas para ordenar `struct`s por seus membros-chave. Porém, para fins didáticos, os algoritmos de ordenação deste capítulo serão apresentados para ordenação de arranjos de tipos de dados fundamentais.

A Figura 16.1 mostra um exemplo de um arranjo `a` com dados do tipo `int` sendo ordenado.

O protótipo da função recebe um arranjo `a` e o tamanho do arranjo `n`. Como o arranjo `a` é sempre passado por referência, os dados no próprio arranjo `a` estarão em ordem ao fim do processo.

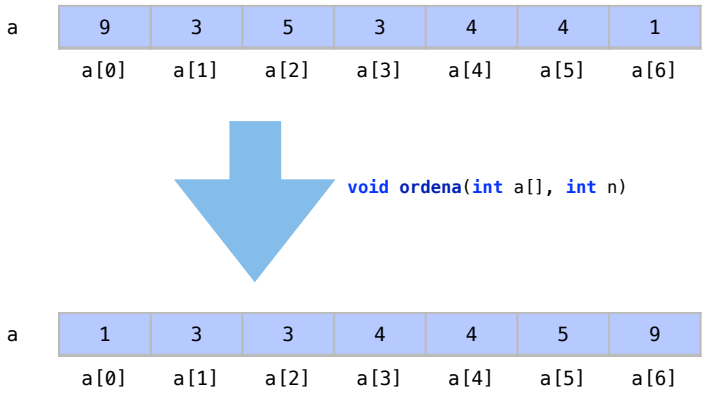


Figura 16.1: Exemplo de ordenação de arranjos. Neste caso temos um arranjo de dados do tipo `int`.

16.1 Conceitos de ordenação de arranjos

16.1.1 Estabilidade de ordenação

Uma ordenação estável é aquela que mantém a ordem relativa dos elementos antes da ordenação. Ou seja, se dois elementos são iguais, o elemento que aparece antes no arranjo desordenado deve ainda aparecer antes no arranjo ordenado.

Suponha a ordenação já apresentada na Figura 16.1. Agora, Figura 16.2 mostra um exemplo de ordenação estável. Note que o resultado do processo de ordenação é o mesmo. Repare a procedência dos elementos 3 e 4. Note através das setas azuis como os elementos 3 mantiveram sua ordem relativa. Note através das setas verdes como os elementos 4 mantiveram sua ordem relativa.

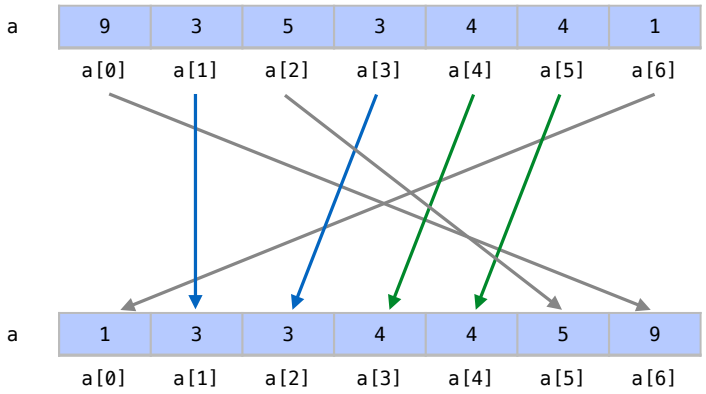


Figura 16.2: Exemplo de ordenação estável.

De outro modo, a Figura 16.3 mostra um exemplo de ordenação instável. Note que, apesar de termos o mesmo resultado, não houve procupação para que os elementos iguais mantivessem sua ordem relativa.

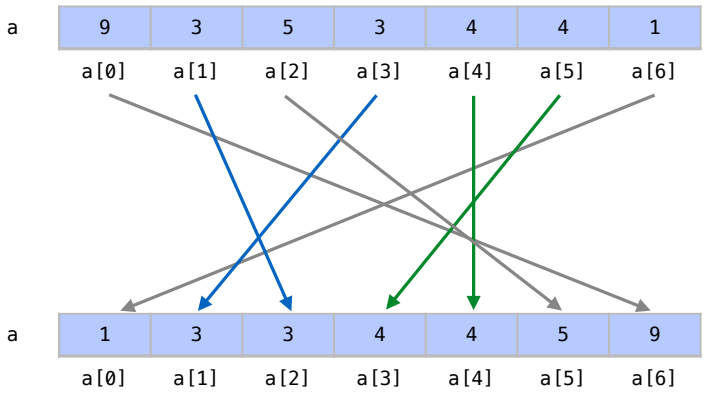
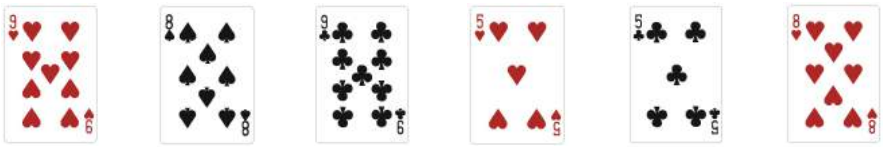


Figura 16.3: Exemplo de ordenação estável.

Motivação

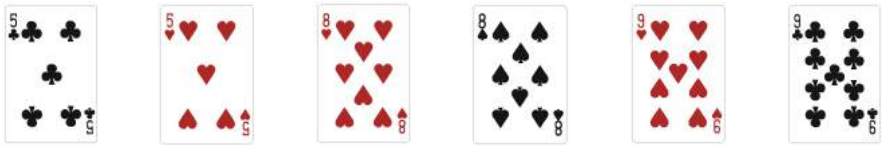
As vantagens de uma ordenação estável ficam claras quando fazemos a ordenação de elementos através de seus membros-chave. Suponha que queremos ordenar um arranjo com as seguintes cartas:



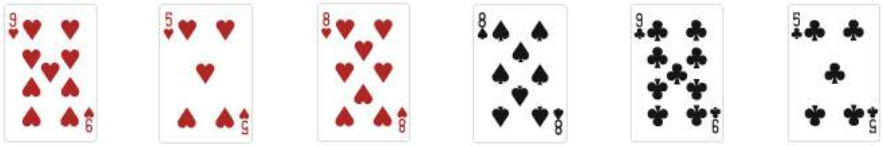
As cartas deste arranjo podiam ser facilmente representadas pelo tipo de dados definido pela seguinte estrutura:

```
1 struct Carta{
2     int valor;
3     int naipe;
4 }
```

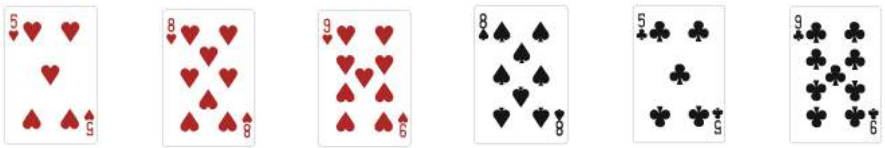
Nesta estrutura, o valor seria representado por um número de 1 a 13 e o naipe seria representado por um número de 1 a 4. Utilizando então uma ordenação qualquer pelos números, podemos ter um resultado como este:



As cartas estão ordenadas agora pelos números, mas a ordem dos naipes é arbitrária. Vamos supor agora que queremos que as cartas estejam ordenadas também de acordo com seu naipe. Precisamos então ordenar mais uma vez as cartas, agora pelo naipe. Veja o que pode acontecer se fizermos a segunda ordenação com um algoritmo instável:



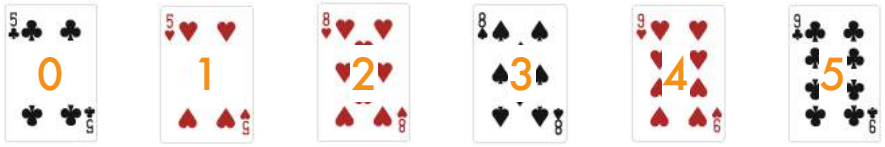
Perdemos a ordem de valor das cartas ao ordenar pelos naipes. Claramente, isto não é o que queríamos. Veja o que aconteceria se houvéssemos ordenado as cartas com um algoritmo estável:



Agora temos as cartas ordenadas por seus naipes mas também mantivemos a ordenação anterior por seus valores dentro de capa naipe.

Forçando estabilidade

Alguns métodos mais eficientes de ordenação não são estáveis. Porém, podemos forçar um método não estável a se comportar de forma estável. Para forçar essa estabilidade, adicionamos um índice a cada elemento do arranjo. O índice indica a posição do elemento na posição original.



Agora, podemos utilizar o índice como fator de desempate caso os elementos tenham a mesma chave (ou o mesmo naipe, no nosso exemplo).

Apesar de ser possível, forçar um método não estável a se tornar estável diminui a eficiência dos algoritmos e faz com que o algoritmo gaste memória extra para armazenar todos os índices. Como o índice quer dizer um membro a mais para cada elemento, este custo extra de memória é $O(n)$.

16.1.2 Complexidade de tempo

Outro fator a se considerar é a complexidade de tempo dos algoritmos de ordenação. A medida mais relevante de tempo é o número de comparações $C(n)$ feitas por um algoritmo para ordenar o arranjo. Em geral os métodos de ordenação simples requerem $O(n^2)$ comparações e são apropriados para arranjos pequenos. Os métodos mais eficientes requerem $O(n \log n)$ comparações e são adequados para arranjos grandes.

Uma medida secundária de comparação entre estes algoritmos é o número $M(n)$ de movimentações, ou operações de atribuição. Alguns algoritmos conseguem fazer apenas $O(n)$ movimentações enquanto a maioria dos algoritmo fará $O(n \log n)$ movimentações.

Porém, é possível forçar um algoritmo a fazer apesar $O(n)$ dos elementos principais do arranjo através da ordenação de índices. Ordenamos os índices do arranjo em vez dos elementos do arranjo propriamente ditos. Ao fim, reorganizamos os elementos, como indicado pelos índices. Esta estratégia implica em um custo $O(n)$ extra de memória para armazenamento dos índices.

16.1.3 Complexidade de memória

Mais um fator relevante é a memória extra que estes algoritmos utilizam para conseguir ordenar o arranjo. Os métodos de ordenação que não precisam de memória extra são métodos $O(1)$ por utilizarem memória constante para qualquer tamanho de arranjo. Estes métodos são chamados de métodos *in place*, ou *in situ*.

Há algoritmos que têm um gasto $O(\log n)$ de memória para ordenar arranjos. Este é ainda considerado um custo baixo, quase equivalente a algoritmos *in place*. Outros algoritmos menos eficientes deste ponto de vista, têm um gasto de memória $O(n)$.

16.2 Algoritmos simples de ordenação

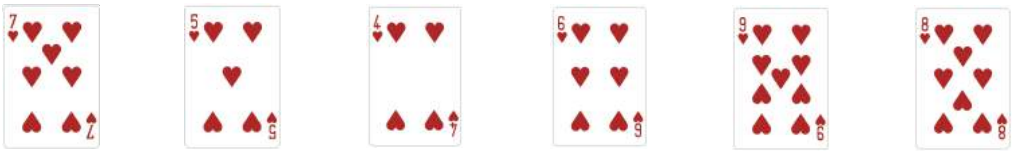
Apresentaremos nesta seção métodos simples de ordenação. Estes métodos costumam ter custo $O(n^2)$ em relação ao número de comparações.

16.2.1 Ordenação por seleção

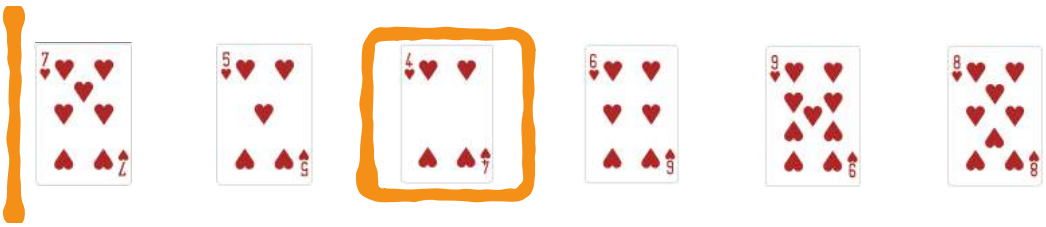
A ordenação por seleção (ou *selection sort*) é um dos algoritmos mais simples de ordenação. É um algoritmo onde a cada passo de repetição se escolhe o menor elemento do arranjo e o troca com o elemento da primeira posição. Repetimos este procedimento para os outros $n-1$ elementos.

Exemplo

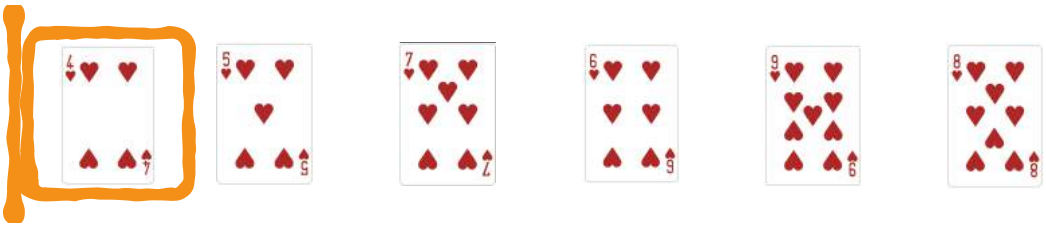
Considere um arranjo de cartas com os seguintes elementos:



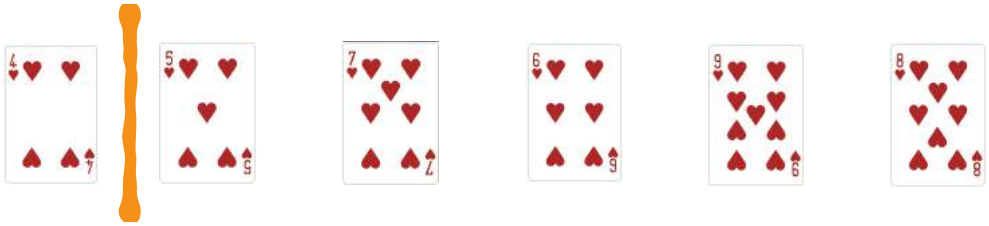
A partir do primeiro elemento, procuramos o menor elemento do arranjo, que no caso é 4.



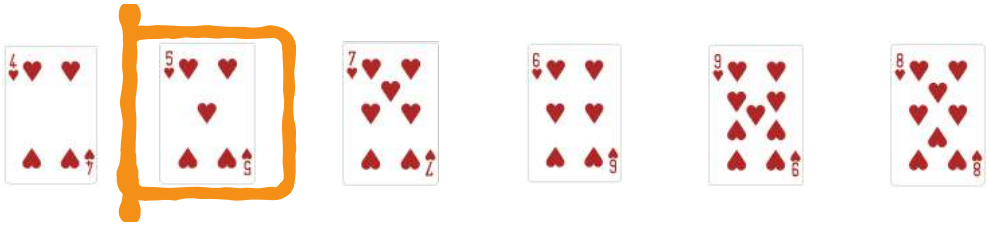
Este menor elemento é trocado com o elemento 7, da primeira posição:



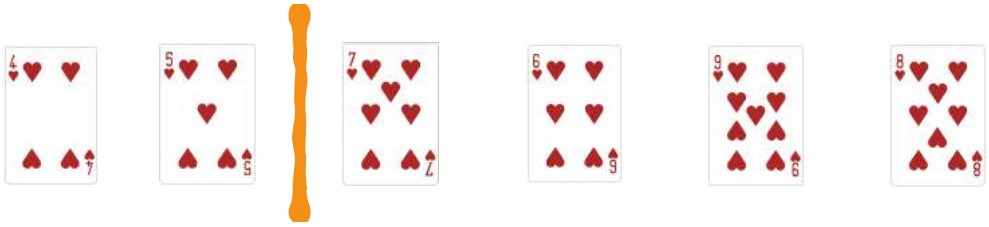
Sabemos agora que o arranjo está ordenado até o primeiro elemento.



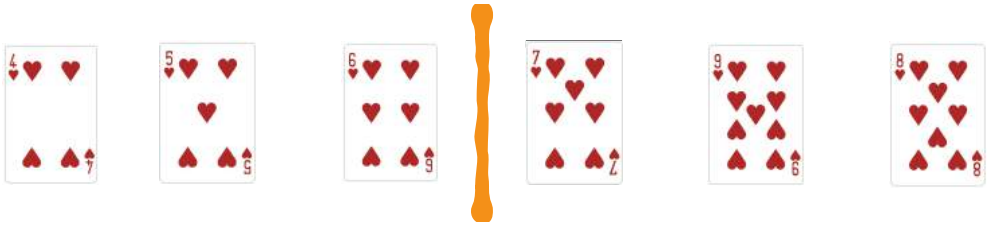
A partir do segundo elemento, procuramos o menor elemento do arranjo.



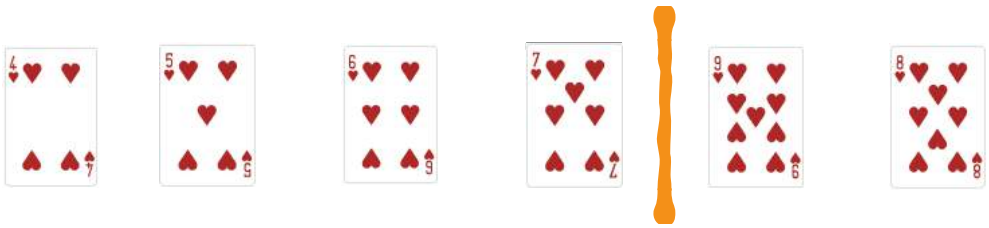
Este é colocado na segunda posição e sabemos que os dois primeiros elementos do arranjo já estão ordenados.



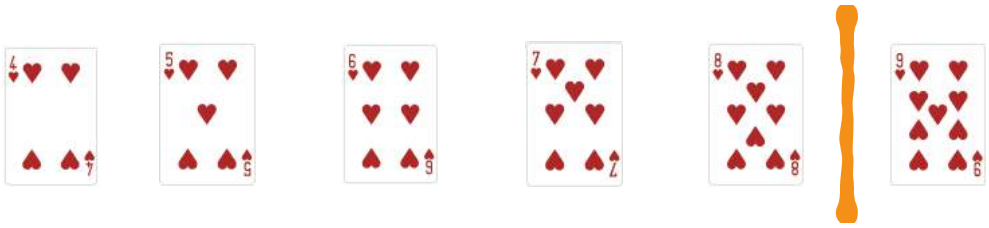
A partir do terceiro elemento, procuramos o menor elemento e o colocamos na terceira posição.



A partir do quarto elemento, procuramos o menor elemento e o colocamos na quarta posição.



A partir do quinto elemento, procuramos o menor elemento e o colocamos na quinta posição.



Como há apenas um elemento no subarranjo não ordenado à direita, já sabemos que ele está em sua posição correta então podemos encerrar o método.

A Figura 16.4 apresenta de forma resumida os passos deste processo de ordenação por seleção. Temos um subarranjo ordenado à esquerda, representado em amarelo, e um subarranjo desordenado à direita, representado em azul. A cada passo, trocamos o primeiro elemento do arranjo desordenado com o menor elemento deste arranjo. Trocas estão representadas pelos valores em vermelho. Assim, cresce a cada passo o subarranjo ordenado à esquerda, representado em amarelo.

Arranjo	7	5	4	6	9	8
Passo 1	4	5	7	6	9	8
Passo 2	4	5	7	6	9	8
Passo 3	4	5	6	7	9	8
Passo 4	4	5	6	7	9	8
Passo 5	4	5	6	7	8	9
Arranjo	4	5	6	7	8	9

Troca Desordenado Ordenado

Figura 16.4: Exemplo de ordenação por seleção.

Algoritmo

A ordenação por seleção para um arranjo de inteiros é dada pelo seguinte código:

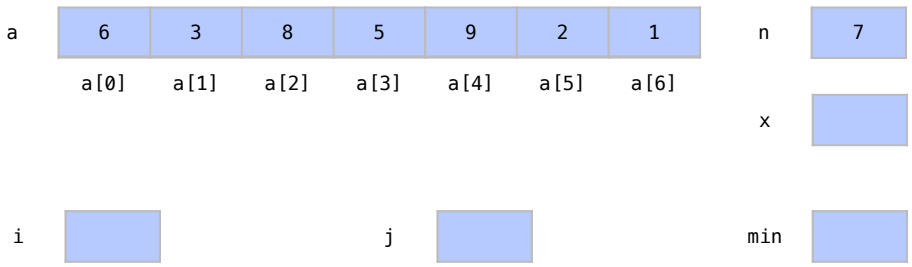
```
1  void selecao(int a[], int n){
2      int i, j, min; // índices
3      int x; // elemento
4      // para cada posição
5      for (i = 0; i < n - 1; i++){
6          // procuramos o menor entre i+1 e n e colocamos em i
7          min = i; //mínimo é o i
8          for (j = i + 1; j < n; j++){
9              if (a[j] < a[min]){
10                 min = j; //mínimo é o j
11             }
12         }
13         // troca a[i] com a[min]
14         x = a[min];
15         a[min] = a[i];
16         a[i] = x;
17     }
18 }
```

Na linha 1 do código, como sabemos que arranjos são passados por referência em C++, a ordenação ocorrerá no arranjo `a` original e não precisamos retornar nada desta função, o que é indicado com `void`. Os índices `i`, `j` e `min`, declarados na linha 2, serão utilizados para percorrermos o arranjo e procurar o menor elemento a cada passo do algoritmo.

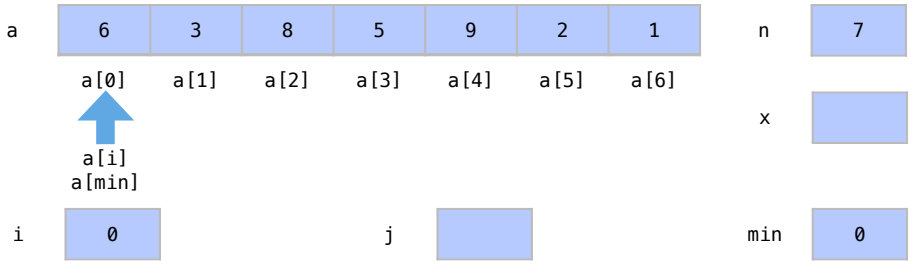
A variável `x`, declarada na linha 3, deve ser do mesmo tipo dos elementos do arranjos pois será uma variável temporária para fazer as trocas entre elementos.

Nas linhas 5 a 17, temos um `for` que, para cada posição do arranjo, colocará em seu lugar o elemento mínimo entre os elementos posteriores. Temos abaixo um exemplo de inicialização

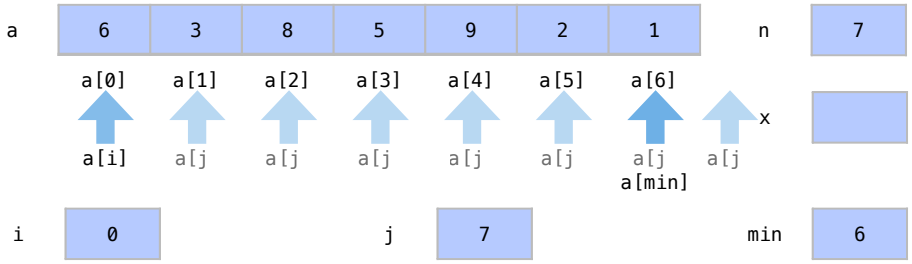
destas variáveis para um arranjo a:



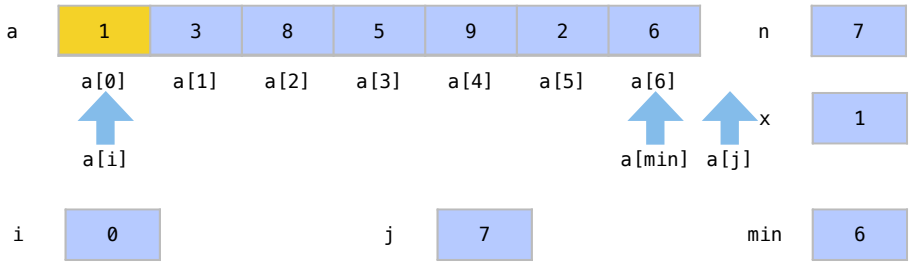
Na primeira iteração do `for` definido na linha 5, quando `i` tem valor `0` procuramos pela posição do menor elemento nas linhas 6 a 12. Para isto supomos na linha 7 que o menor elemento deste arranjo está na posição `i`. Ou seja, no nosso exemplo, supomos por enquanto que o menor elemento entre as posições `0` e `n-1` está na posição `0`. Assim, `a[i]` e `a[min]` indicam o primeiro elemento do arranjo ainda desordenado.



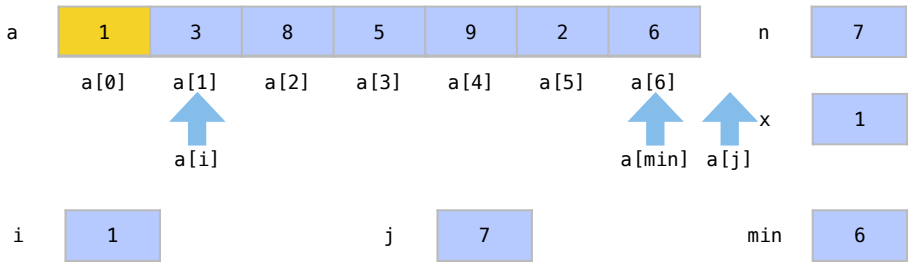
No `for` definido nas linhas 8 a 12, utilizamos `j` como contador para percorrer os elementos entre `i+1` e `n-1` atualizando a posição do menor elemento. Na linha 9, para cada elemento `a[i]`, se este é menor do que o menor que já conhecemos `a[min]`, atualizamos a posição indicada por `min`. No nosso exemplo, `j` percorrerá as posições `1` a `6` finalizando com valor `7` e descobriremos que o menor elemento está em `a[6]`.



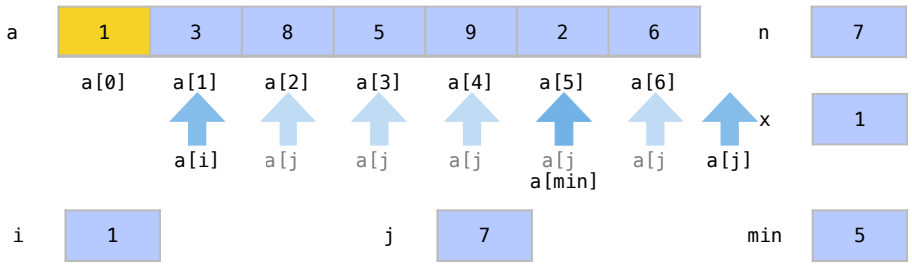
Nas linhas 13 a 16, trocamos então os elementos da posição `a[i]` e `a[min]`, ou `a[0]` e `a[6]` em nosso exemplo. O `x` é utilizado como uma variável auxiliar para fazer esta troca. Com isto, temos o fim do primeiro passo. Neste passo, procuramos então o menor elemento entre `a[0]` e `a[n-1]` e colocamos no lugar de `a[0]`. Ao fazer esta troca, sabemos agora que o primeiro elemento já representa um arranjo ordenado, representado aqui em amarelo.



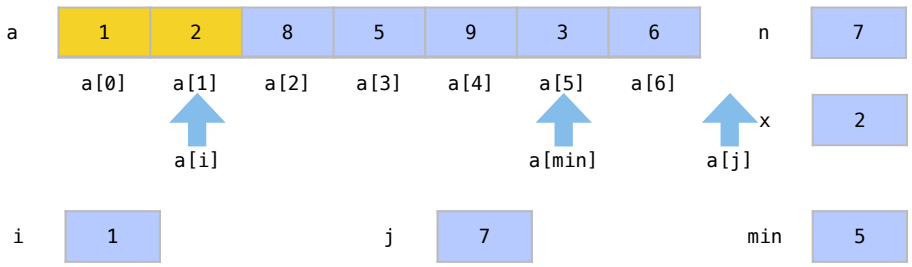
Na próxima iteração do `for` mais externo, atualizamos `i` para 1 e fazemos mais um passo. Sabemos que o arranjo até `a[i]` já está ordenado.



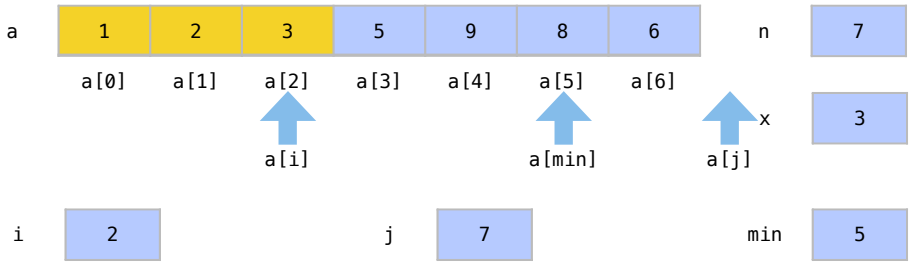
Esta iteração, nas linhas 7 a 12, procura o menor elemento entre `a[i]` e `a[n-1]`, variando o valor de `j`. O valor de `min`, em nosso exemplo, será 5.



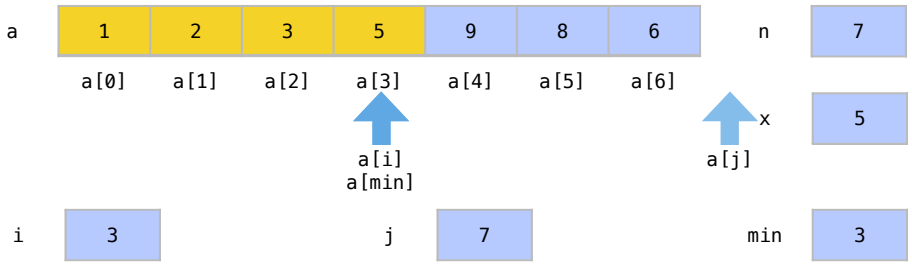
Este elemento `a[min]` é trocado com `a[i]` nas linhas 13 a 16, e sabemos agora que o arranjo até `a[i]` (ou `a[1]`) já está ordenado.



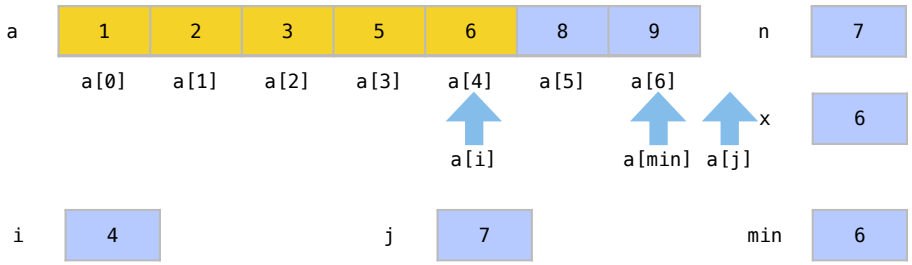
Na próxima iteração do `for` mais externo, atualizamos novamente `i` para 2 e sabemos agora que o arranjo até `a[i]` já está ordenado. Achamos o menor entre `a[i]` e `a[n-1]`, nas linhas 6 a 12, e trocamos `a[i]` com `a[min]`, nas linhas 13 a 16. Sabemos que o arranjo está ordenado até a posição `a[i]`.



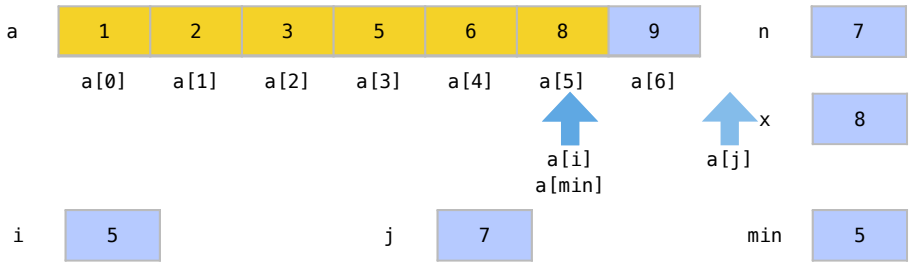
Na próxima iteração fazemos o mesmo, tendo `i` valor 3. Temos o arranjo ordenado entre as posições `a[0]` e `a[3]`.



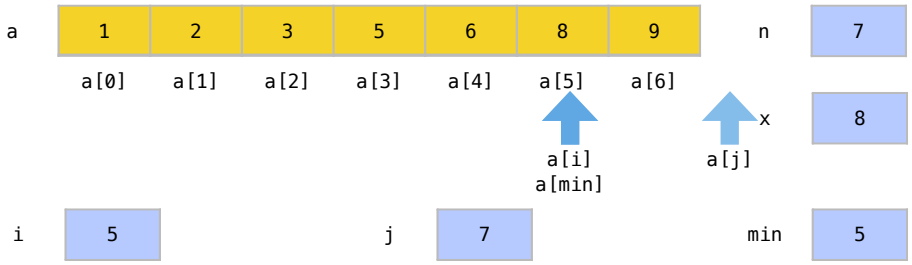
Na iteração seguinte, colocamos o menor elemento em `a[4]`.



Na última iteração, colocamos o menor elemento em `a[5]`.



Veja que não precisamos de uma iteração para o último elemento. Se há apenas um elemento, sabemos que ele é o menor. Sabemos então que mesmo encerrando o `for` sem alterar o valor das variáveis, todo os arranjo está ordenado.



Assim, terminamos a função com todos os elementos do arranjo `a` ordenados.

Análise

Analisaremos primeiro o laço interno interno do algoritmo, definido nas linhas 8 a 12. Sempre que procuramos o menor elemento entre i e n, fazemos $n-i$ comparações.

Já no laço mais externo, o valor de i varia entre 0 e $n-1$. Assim, estas $n-i$ comparações são feitas para $i = 0, 1, 2, \ldots, n-2$ elementos do arranjo desordenado, o que pode ser representado com o seguinte somatório:

$$f(n) = \sum_{i=0}^{n-2} n - i = O(n^2)$$

Assim, sabemos que o número de comparações feitas pelo algoritmo é $O(n^2)$. Isto ocorre para qualquer caso.

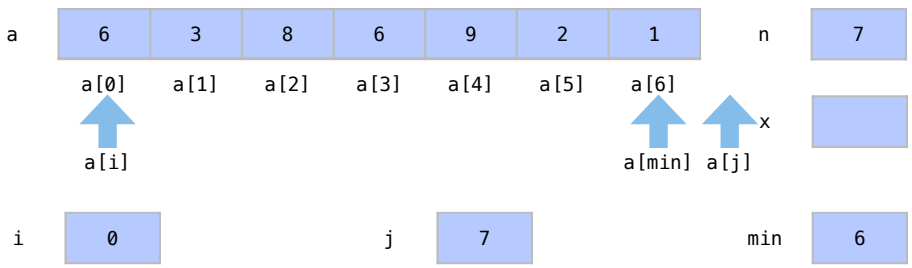
Em relação ao número de atribuições, cada troca, definida nas linhas 13 a 16, envolve 3 atribuições. Como são feitas $n-1$ trocas, temos $f(n) = 3(n-1) = O(n)$ atribuições de movimentação.

Além das atribuições de movimentação, se considerarmos todas as atribuições, temos a atualização da posição menor, na linha 10. Esta ocorre em média $O(n \log n)$ vezes ao longo de todo algoritmo, sendo $O(\log n)$ vezes por iteração. Se considerarmos estas atribuições, o custo total de atribuições é $O(n \log n)$. Note contudo, que as operações de atribuição para movimentação são muito mais custosas em situações práticas.

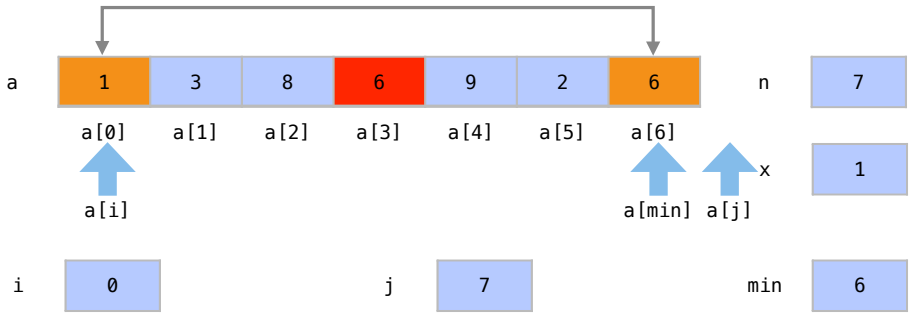
A grande vantagem da ordenação por seleção é que em relação ao custo de movimentação, temos um custo linear $O(n)$. É então um bom algoritmo onde estas movimentações por algum motivo custem muito. Em relação ao número de comparações, que é usualmente a operação mais relevante, o algoritmo é interessante apenas para arranjos pequenos, já que tem um custo $O(n^2)$.

Uma desvantagem do algoritmo é que se o arranjo já estiver ordenado, isto não ajuda o algoritmo, pois o custo de comparações continua $O(n^2)$. Isto quer dizer que o algoritmo não tem **adaptabilidade**. Outra desvantagem da ordeção por seleção é que o algoritmo não é estável pois a troca entre elementos pode destruir a ordem relativa dos mesmos.

Suponha este exemplo de passo do algoritmo de ordenação por seleção onde trocaremos o elemento `a[i]` com o elemento `a[min]`, como fazemos usualmente nas linhas 13 a 16 do algoritmo. Neste caso, os elementos a serem trocados são 6 e 1.



Ao fazermos esta troca, o elemento `a[i]`, de valor 6, perde sua ordem relativa entre qualquer qualquer elemento entre `a[i+1]` e `a[min-1]`. Neste caso, o elemento `a[i]` perdeu sua ordem relativa com o elemento `a[3]`, que tem o mesmo valor.

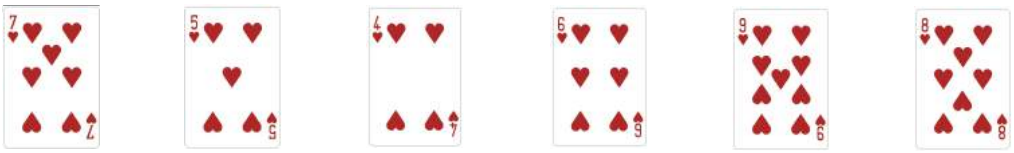


16.2.2 Ordenação por inserção

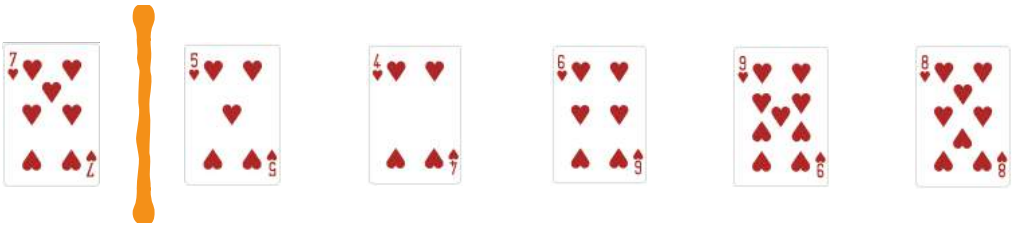
A ordenação por inserção o algoritmo mais utilizado por jogadores de cartas. É um algoritmo onde a cada passo de repetição temos um arranjo ordenado até um certo ponto, pegamos o próximo elemento do arranjo desordenado e colocamos o na posição correta entre o primeiro e ele mesmo.

Exemplo

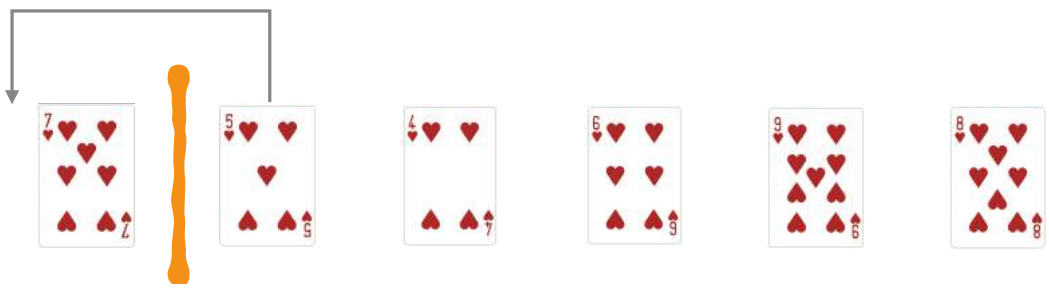
Considere um arranjo de cartas com os seguintes elementos:



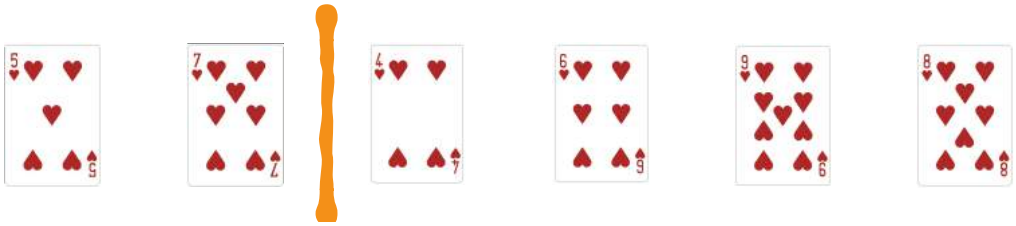
Já sabemos inicialmente que o arranjo até o segundo elemento já está ordenado pois um arranjo de apenas um elemento está sempre ordenado.



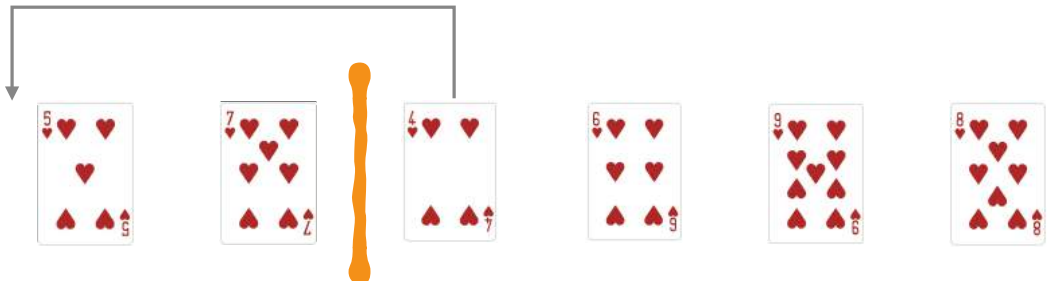
Colocamos então o próximo elemento do subarranjo desordenado na posição correta do subarranjo ordenado.



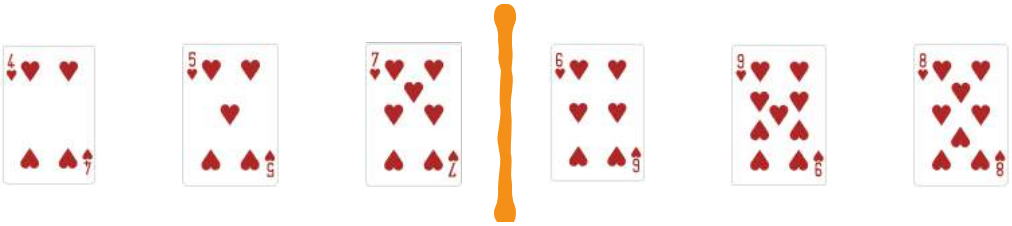
Sabemos agora então que os dois primeiros elementos estão ordenados.



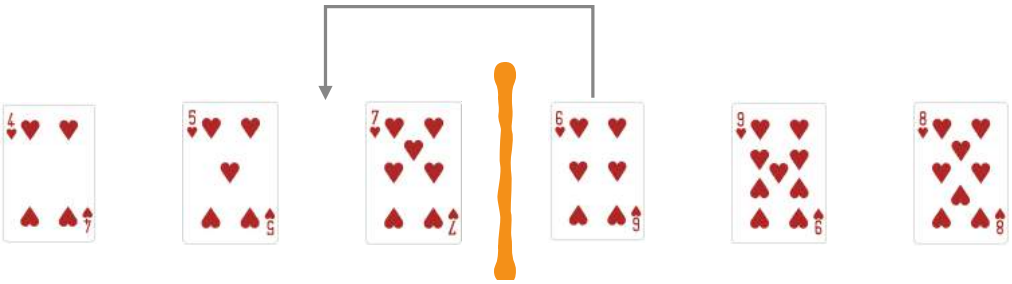
O terceiro elemento deverá ser colocado em sua posição correta.



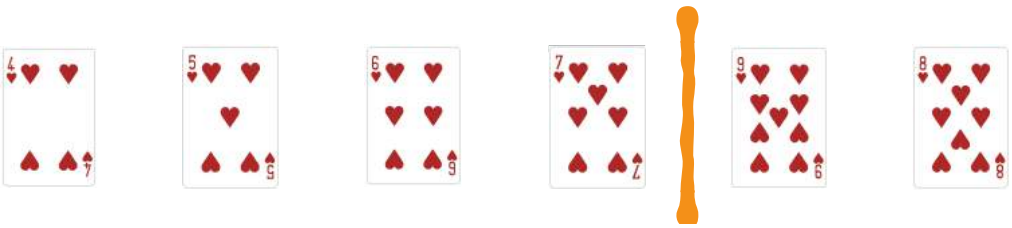
Sabemos agora que os três primeiros elementos estão ordenados.



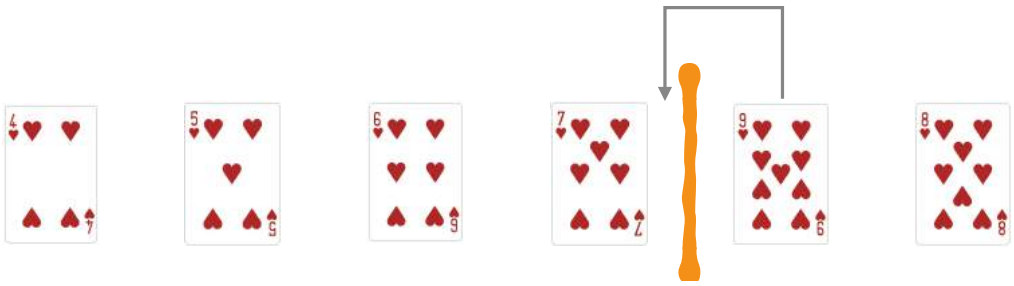
O quarto elemento deverá ser colocado em sua posição correta.



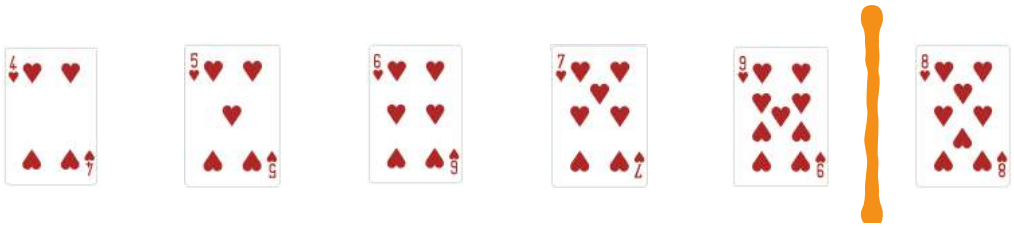
Sabemos agora que os quatro primeiros elementos estão ordenados.



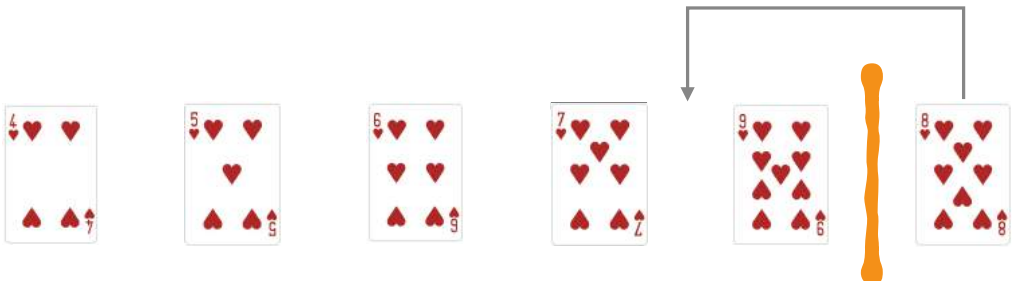
O quinto elemento deverá ser colocado em sua posição correta.



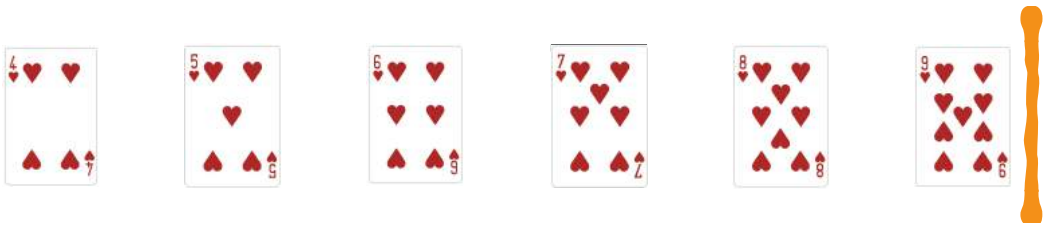
Sabemos agora que os cinco primeiros elementos estão ordenados.



O último elemento deverá ser colocado em sua posição correta.



Sabemos agora que todos os elementos estão ordenados.



A Figura 16.5 apresenta de forma resumida os passos deste processo de ordenação por inserção. Logo no passo 1, sabemos que temos um subarranjo ordenado à esquerda já que um subarranjo de apenas 1 elemento é um subarranjo ordenado. O subarranjo ordenado está representado em amarelo, e um subarranjo desordenado à direita, representado em azul. A cada passo, tiramos o primeiro elemento do arranjo desordenado e o inserimos nas posição correta do subarranjo ordenado que temos até o momento. As trocas estão representadas pelos valores em vermelho. Veja como pode ser necessário movimentar vários elementos em um passo para colocar um elemento em sua posição correta. Neste processo, cresce em um elemento a cada passo o subarranjo ordenado à esquerda, representado em amarelo.

Arranjo	7	5	4	6	9	8
Passo 1	7	5	4	6	9	8
Passo 2	5	7	4	6	9	8
Passo 3	4	5	7	6	9	8
Passo 4	4	5	6	7	9	8
Passo 5	4	5	6	7	9	8
Arranjo	4	5	6	7	8	9

Movimentação Desordenado Ordenado

Figura 16.5: Exemplo de ordenação por insercao.

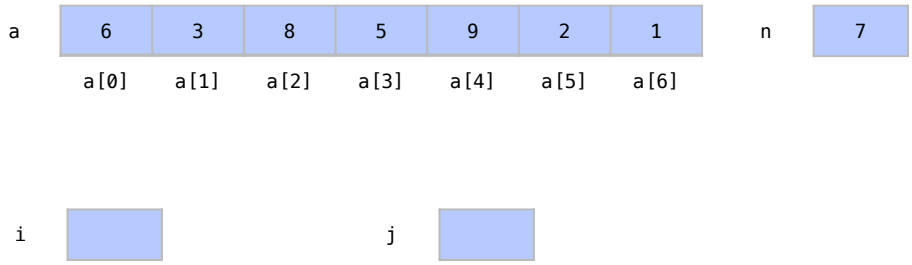
Algoritmo

A ordenação por inserção para um arranjo de inteiros é dada pelo seguinte código:

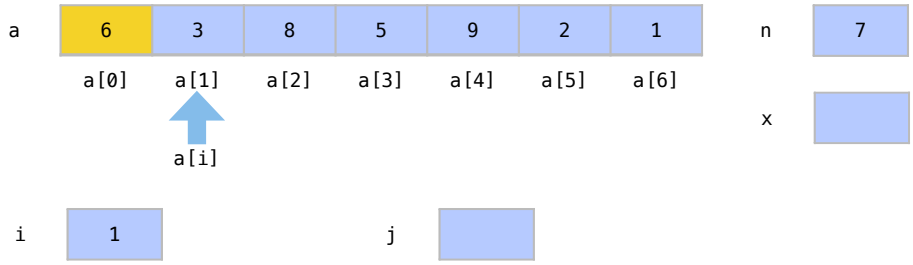
```
1  void insercao(int a[], int n){
2      int i, j; // índices
3      int x; // elemento
4      // para cada posição a partir de i = 1
5      for (i = 1; i < n; i++){
6          // coloca o elemento a[i] na posição correta
7          x = a[i];
8          j = i - 1;
9          while ((j >= 0) && (x < a[j])){
10             a[j+1] = a[j];
11             j--;
12         }
13         a[j+1] = x;
14     }
15 }
```

No protótipo da função, definido na linha 1, a função ordena o arranjo `a`, que contém n elementos. Como sabemos que arranjos são passados por referência em C++, não precisamos retornar nada desta função.

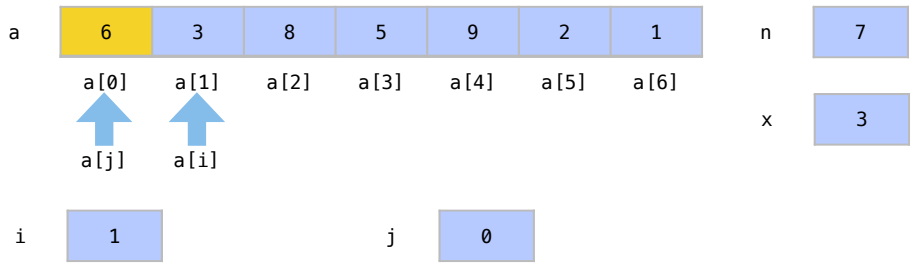
Os índices `i` e `j`, definidos na linha 2, são utilizados para percorrermos o arranjo em um `for` aninhado e colocar cada elemento `a[i]` em sua posição correta no arranjo ordenado. A variável `x`, definida na linha 3, deve ser do mesmo tipo dos elementos do arranjos pois será uma variável temporária para fazer as trocas entre elementos.



Entramos em um `for` na linha 5 que, para cada posição do arranjo, colocará o elemento`a[i]` em seu lugar correto do arranjo ordenado de `a[0]` até `a[n-1]`. Na primeira iteração, quando `i` é 1, já sabemos antes de mais nada que o arranjo até `a[0]`, de apenas um elemento, já está ordenado.

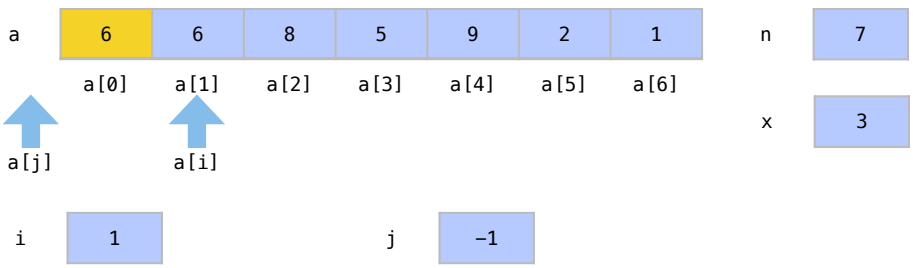


Ao executar o bloco de repetição, a variável x recebe um cópia do elemento`a[i]` na linha 7. Esta cópia é guardada para ser colocada em sua posição correta no arranjo ordenado. O índice `j`, recebe `i-1` na linha 8. É a partir de a`a[i-1]` que procuraremos a posição correta de `a[i]`.



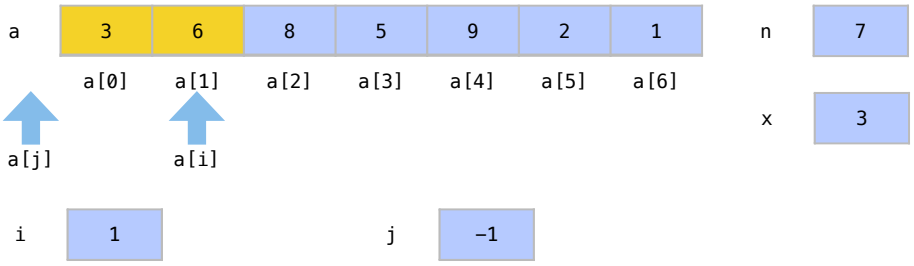
No `while` aninhado, definido na linha 9, deslocaremos para a direita todos os elementos anteriores a maiores que `a[i]`, agora guardado em `x` para permitir o deslocamento. Temos duas condições no `while` que garantem que faremos este deslocamento. A primeira condição `j >= 0` garante simplesmente que façamos os deslocamentos enquanto `j` estiver apontando para uma posição válida do arranjo, maior ou igual a 0. A segunda condição `x <= a[j]` garante que faremos o deslocamento apenas enquanto os itens forem maiores que `x`.

Na primeira iteração do `while`, o valor 6 é deslocado em uma posição, na linha 10, e `j` é deslocado para a esquerda, na linha 11.

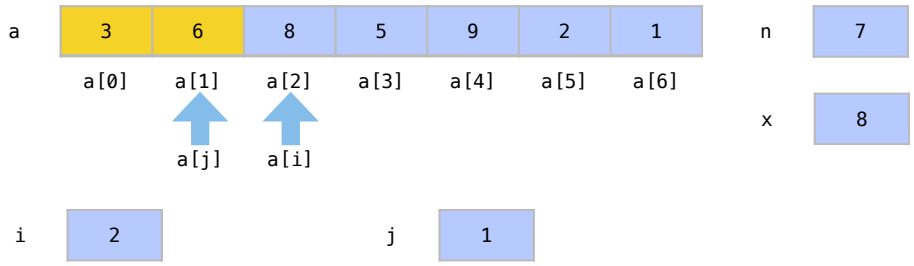


Isso acorre até encontrarmos a posição de `x` ou até que todos os elementos tenham sido deslocados. Na primeira iteração temos que `a[0]` é deslocado e não temos mais elementos para deslocar. Assim, encerramos o `while` pois `j >= 0` é `false`. O valor original de `a[i]`, porém, ainda está guardado em `x`.

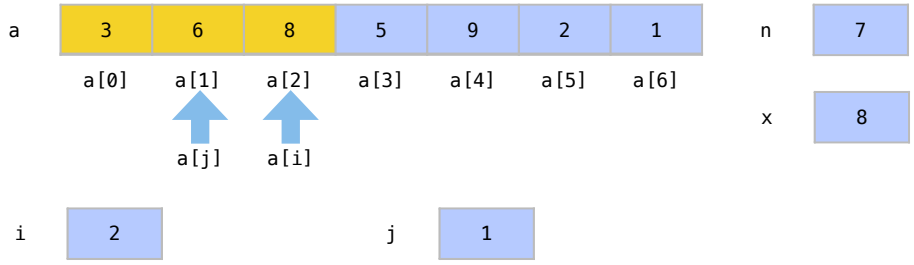
Saímos deste loop pois todos os elementos foram deslocados. Na linha 13, por fim, o elemento `x`, que saiu de `a[i]` e está guardado em `x`, é então inserido em sua posição correta `a[j+1]`, posição do último elemento deslocado para a direita. Com o fim desta operação sabemos que os elementos `a[0]` até `a[i]` formam agora um subarranjo ordenado.



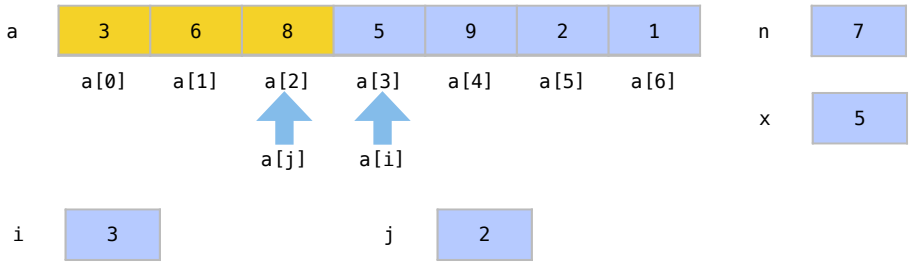
Na próxima iteração do `for`, incrementamos `i` para 2, e sabemos agora que o arranjo até `a[1]` já está ordenado. Repetindo o processo, `x` guarda uma cópia do elemento `a[i]` na linha 7. A partir de `a[i-1]`,procuraremos no `while` a posição correta para o elemento `x`, deslocando elementos para a direita. Para isto `j` é definido como `i-1` na linha 8.



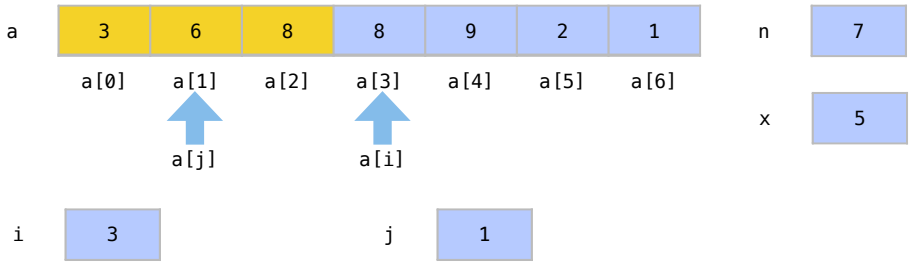
Como o elemento guardado `x` não é menor que o elemento `a[j]`, já achamos a posição correta para ele sem executar nenhuma iteração do `while`. Na linha 13, o elemento `x` é colocado na posição `a[j+1]`, que já era sua posição original. Sabemos que o subarranjo de `a[0]` até `a[2]` agora está ordenado.



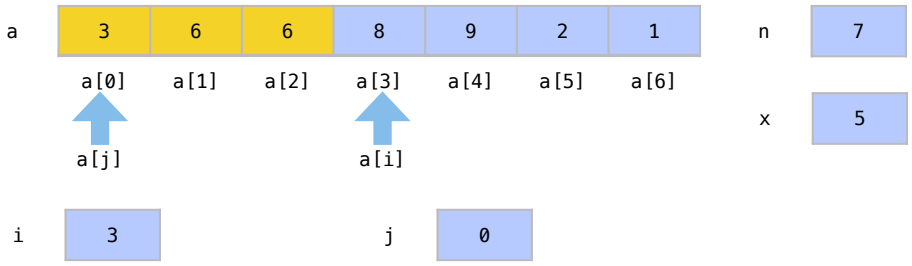
Incrementamos `i` para 3 na iteração seguinte do `for`. Na linha 7, a variável `x` recebe uma cópia dea`[i]`. O índice `j` marca que procuraremos a posição a partir de `a[i-1]`, na linha 8.



Deslocamos então, o elemento `a[j]` em uma posição na linha 10 e decrementamos `j` na linha 11 para testar o próximo elemento.

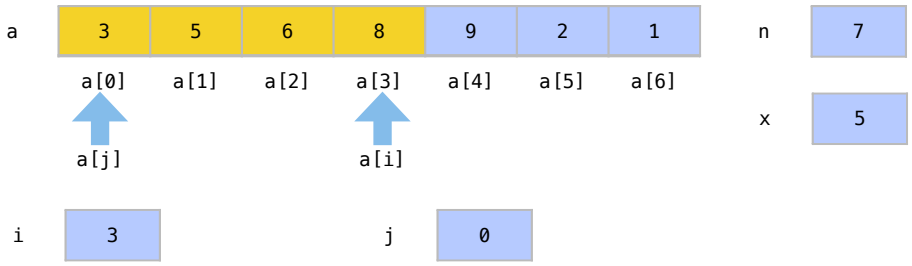


O `while` terá mais uma iteração pois ainda não achamos a posição correta de `x` e nem deslocamos todos os elementos. Deslocamos o elemento `a[j]` em uma posição na linha 10 e decrementamos `j` na linha 11 para testar a próxima posição.

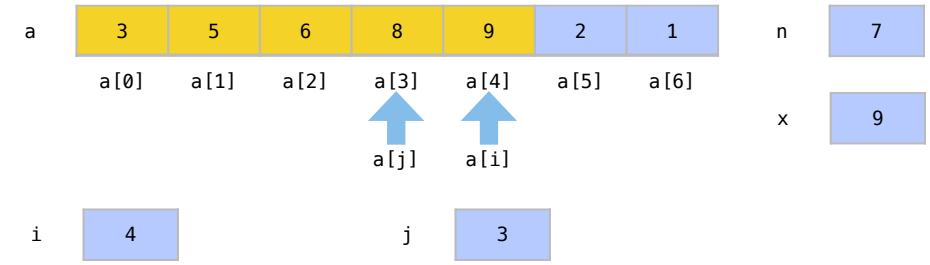


Como o elemento `x` não é menor que `a[j]`, encontramos sua posição correta e saimos do `while` devido à segunda condição.

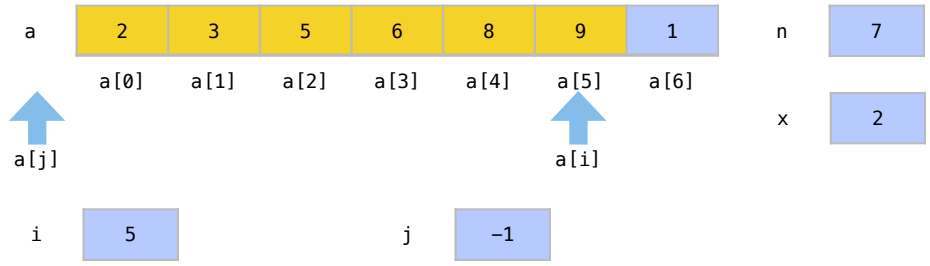
Na linha 13, o elemento guardado em `x` é então colocado em sua posição correta `a[j+1]`. Sabemos que o arranjo está ordenado até `a[3]`.



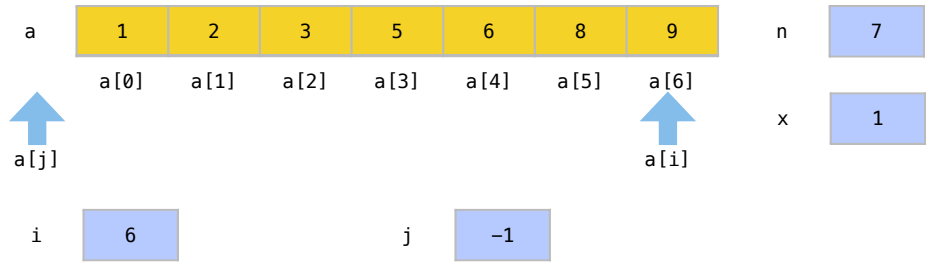
Na iteração seguinte do `for`, incrementamos `i` para 4. Assim como nas outras iterações, nas linhas 6 a 13, elementos maiores que `a[i]` serão deslocados no arranjo ordenado para a direita e `a[i]` será inserido em sua posição correta. Sabemos que o arranjo está ordenado até `a[4]`.



Em seguida, incrementamos `i` para 5 e, novamente nas linhas 6 a 13, inserimos `a[i]` em sua posição correta no arranjo ordenado. Elementos maiores que `a[i]` serão deslocados no arranjo ordenado e `a[i]` será inserido em sua posição correta. Sabemos que o arranjo está ordenado até `a[5]`.



Na última iteração, onde `i` tem valor 6. Inserimos `a[i]` novamente em sua posição correta do arranjo e sabemos agora que todo o arranjo está ordenado.



Atingimos assim nosso critério de parada quando `i` chega a 7, encerrando assim a função.

Análise

Existem duas condições que terminam o laço mais interno do algoritmo, definido na linha 9:

- Termos analisado todos os elementos
- Termos encontrado a posição correta do elemento

Assim, em relação aos casos possíveis para o laço interno temos o melhor caso quando é feita apenas uma comparação, pois o elemento já está em sua posição correta $O(1)$. Temos o pior caso, quando i comparações são feitas até testarmos todas as posições, levando a custo $O(i)$ para comparar e deslocar i elementos. E o caso médio, se supormos a mesma probabilidade de um elemento estar em qualquer posição, temos um custo $(1/i)(1+2+3\Delta\Delta\Delta+i) = (i+1)/2 = O(i)$.

Já no laço mais externo, definido na linha 5, temos sempre $n-1$ iterações que incluem uma execução do laço interno. Assim, no melhor caso, executaremos $n-1$ vezes o melhor caso do laço interno $O(1)$ e teremos uma laço externo com custo $(n-1)O(1) = O(n)$. Se tivéssemos sempre o pior caso $O(i)$ no laço interno, o algoritmo tem um custo $\sum_{i=1}^{n-1} O(i) = O(n^2)$. De modo análogo, no caso médio, que também tem custo $O(i)$ no laço interno, o algoritmo também teria um custo total $O(n^2)$.

No algoritmo de ordenação por inserção, o número de movimentações é proporcional ao número de comparações. Isso porque os elementos são movimentados até que uma comparação indique que a posição correta foi enfim, encontrada. Sendo assim, em relação ao número de movimentações, temos também um melhor caso $O(n)$, pior caso $O(n^2)$ e caso médio $O(n^2)$.

O algoritmo tem um melhor caso que ocorre quando os elementos já estão ordenados. Sendo assim, uma ordenação prévia é aproveitada pelo algoritmo pois não fazemos tantos deslocamentos. Quando isso acontece, dizemos que o método é **adaptável**.

Assim, a ordenação por inserção é um bom método para se adicionar alguns poucos elementos a um arranjo ordenado, pois terá um custo baixo em um arranjo "quase ordenado".

A ordenação por inserção é também um algoritmo de ordenação é *estável*, pois a ordem relativa dos elementos é mantida no deslocamentos feitos para se encontrar a posição de um elemento.

Apesar de pouco provável, na maior parte das aplicações práticas, o custo máximo do algoritmo ocorre quando os elementos estão em ordem reversa, pois sempre faremos i deslocamenos de elemento por iteração. Neste pior caso, se o número de movimentações é o fator mais relevante, ele se torna muito ineficiente em relação à ordenação por seleção.

Na Tabela 16.2 temos um resumo do custo do algoritmo de ordenação por inserção em seus principais casos. Tanto o pior caso quando o caso médio têm custo $O(n^2)$ para ordenar o arranjo.

Caso	Descrição	Custo
Pior caso	Arranjo desordenado	$O(n^2)$
Caso médio	Arranjo com elementos em ordem aleatória	$O(n^2)$
Melhor caso	Arranjo ordenado	$O(n)$

Tabela 16.2: Custo do algoritmo de ordenação por inserção em suas situações mais comuns

16.2.3 Comparação entre algoritmos simples de ordenação

Nesta seção apresentamos dois algoritmos simples para ordenação: a ordenação por seleção e ordenação por inserção. Na Tabela 16.3 temos uma comparação do comportamento deste algoritmos. Os dois algoritmos têm o mesmo custo médio e custo de pior caso $O(n^2)$ mas existem vantagens e desvantagens.

Para a ordenação por seleção, não existem diferenças entre os melhores e piores casos. A medida que nos aproximamos do melhor caso da ordenação por inserção, que é quando o arranjo está ordenado, temos um custo assintótico $O(n)$ para a função, pois ela apenas confere se o arranjo já está ordenado.

Em relação ao número de movimentações, a ordenação por seleção sempre faz apenas n trocas de elementos, enquanto a ordenação por inserção faz um número de movimentações proporcional ao número de comparações no deslocamento dos elementos.

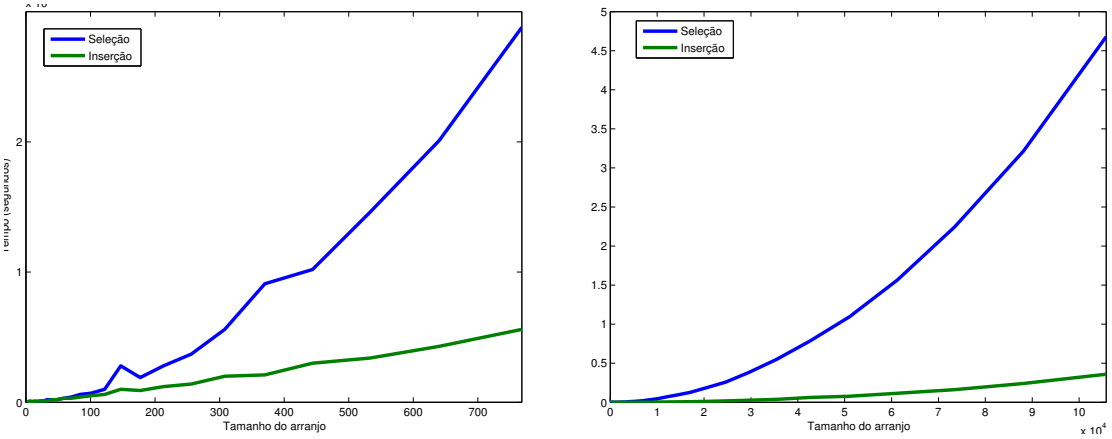
Por fim, a ordenação por inserção tem duas características vantajosas. Ela é um método de ordenação estável, ou seja, que não desfaz a ordem entre elementos que tenham a mesma chave. Ela também tem um custo menor para arranjos que já estejam "quase ordenados", o que chamamos de adaptabilidade.

Como os dois algoritmos têm um custo $O(n^2)$ no caso médio, é importante fazer uma comparação com testes reais com estes algoritmos. Para estes testes, implementamos versões eficientes dos dois algoritmos e medimos o tempo para ordenar arranjos de diversos tamanhos. A Figura 16.6 apresenta o tempo necessário por cada algoritmo para ordenar um arranjo de elementos em ordem aleatória. No caso médio, a ordenação por inserção é muito mais eficiente que a ordenação por seleção. À medida que o tamanho dos arranjos cresce, a diferença entre os

	Algoritmo	
	Seleção	**Inserção**
Comparações		
Caso médio	$O(n^2)$	$O(n^2)$
Pior caso	$O(n^2)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$
Movimentações		
Caso médio	$O(n)$	$O(n^2)$
Pior caso	$O(n)$	$O(n^2)$
Melhor caso	$O(n)$	$O(n)$
Características		
Estabilidade	Não	Sim
Adaptabilidade	Não	Sim

Tabela 16.3: Comparação de custo em termos de número de comparações para algoritmos simples de ordenação. Os dois algoritmos têm um custo médio $O(n^2)$

dois algoritmos se torna ainda mais discrepante.



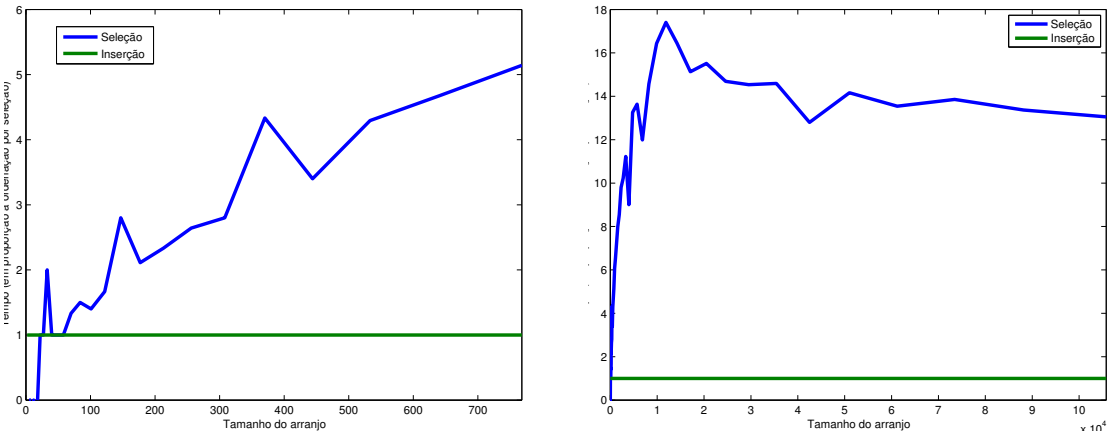
(a) Arranjos menores (b) Arranjos maiores

Figura 16.6: Tempo necessário para se ordenar um arranjo com ordem inicial aleatória (caso médio).

Na Figura 16.7 temos a proporção entre o tempo necessário para se ordenar um arranjo com cada um dos algoritmos. A linha verde, com valor 1, representa o tempo gasto pela ordenação por inserção. A linha azul, representa quantas vezes mais tempo levou uma ordenação por seleção. Em proporção, para alguns tamanhos de arranjo, a ordenação por seleção chega a ser quase 18 vezes mais lenta que a ordenação por inserção.

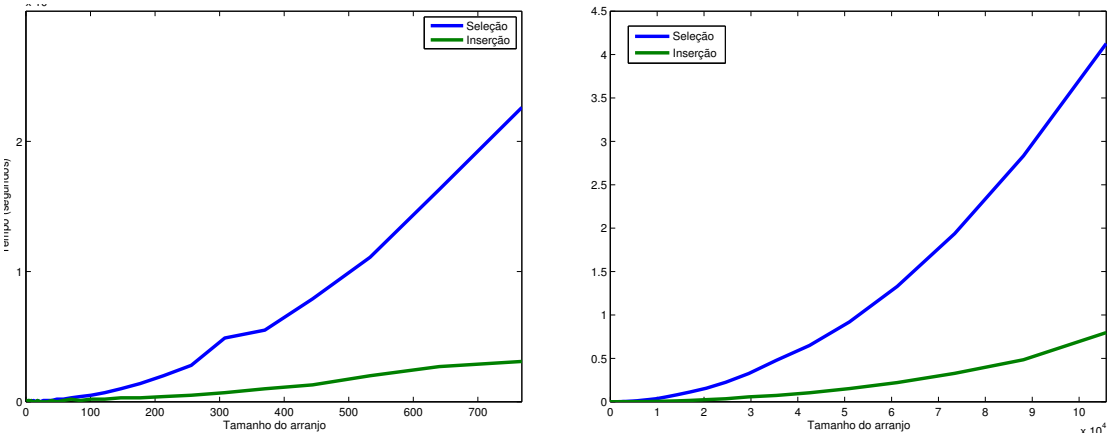
Já o pior caso da ordenação por inserção ocorre quando os arranjos estão em ordem descrescente, por temos nestes casos mais deslocamentos de elementos. A Figura 16.8 apresenta o tempo necessário por cada algoritmo para ordenar um arranjo de elementos em ordem decrescente. Mesmo nestes casos específicos, a ordenação por inserção consegue ser mais vantajosa que uma ordenação por seleção, apesar da diferença de desempenho entre os algoritmos ser menor.

Na Figura 16.9 temos a proporção entre o tempo necessário para se ordenar um arranjo em ordem decrescente com cada um dos algoritmos. A linha verde, com valor 1, representa o tempo gasto pela ordenação por inserção. A linha azul, representa quantas vezes mais tempo levou uma



(a) Arranjos menores (b) Arranjos maiores

Figura 16.7: Proporção entre tempo necessário para se ordenar um arranjo com ordem inicial aleatória (caso médio).



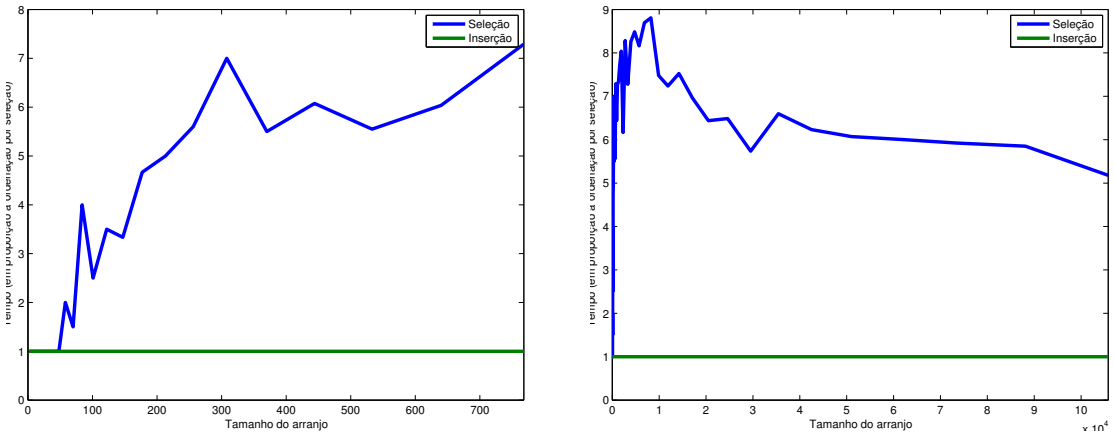
(a) Arranjos menores (b) Arranjos maiores

Figura 16.8: Tempo necessário para se ordenar um arranjo com ordem decrescente (pior caso para a ordenação por inserção).

ordenação por seleção. Em proporção, para alguns tamanhos de arranjo, mesmo no pior caso da ordenação por inserção, a ordenação por seleção chega a ser quase 9 vezes mais lenta que a ordenação por inserção.

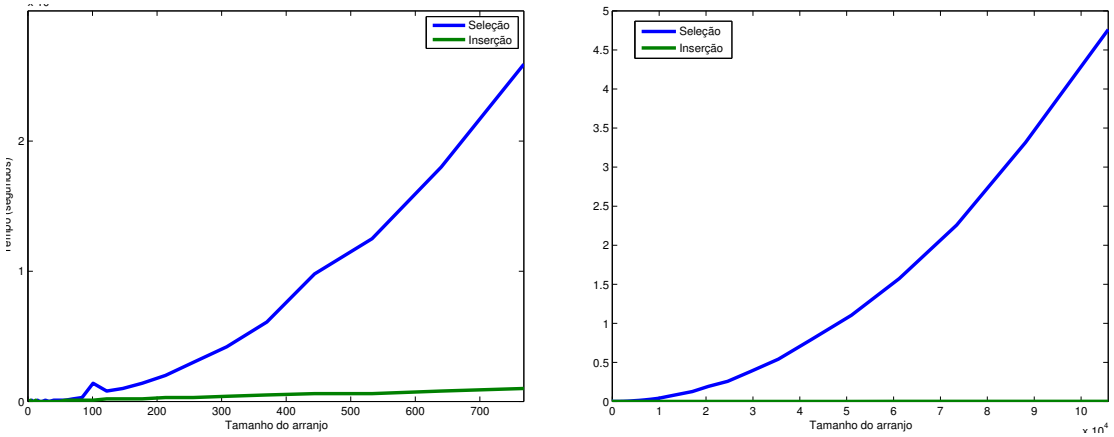
Por fim, o melhor caso da ordenação por inserção ocorre quando os arranjos estão em ordem crescente, pois nestes casos não temos deslocamentos de elementos. O algoritmo apenas confere a ordenação do arranjo e encerra. A Figura 16.10 apresenta o tempo necessário por cada algoritmo para ordenar um arranjo de elementos em ordem decrescente. O desempenho da ordenação por inserção nestes casos é muito melhor.

Na Figura 16.11 temos a proporção entre o tempo necessário para se ordenar um arranjo em ordem crescente com cada um dos algoritmos. A linha verde, com valor 1, representa o tempo gasto pela ordenação por inserção. A linha azul, representa quantas vezes mais tempo levou uma ordenação por seleção. Em proporção, para alguns tamanhos de arranjo apresentados, a ordenação por inserção em seu melhor caso chega a ser quase 2.500 vezes mais rápida que a ordenação por seleção. Esta diferença é maior a medida que os arranjos sejam maiores.



(a) Arranjos menores (b) Arranjos maiores

Figura 16.9: Proporção entre tempo necessário para se ordenar um arranjo com ordem decrescente (pior caso para a ordenação por inserção).



(a) Arranjos menores (b) Arranjos maiores

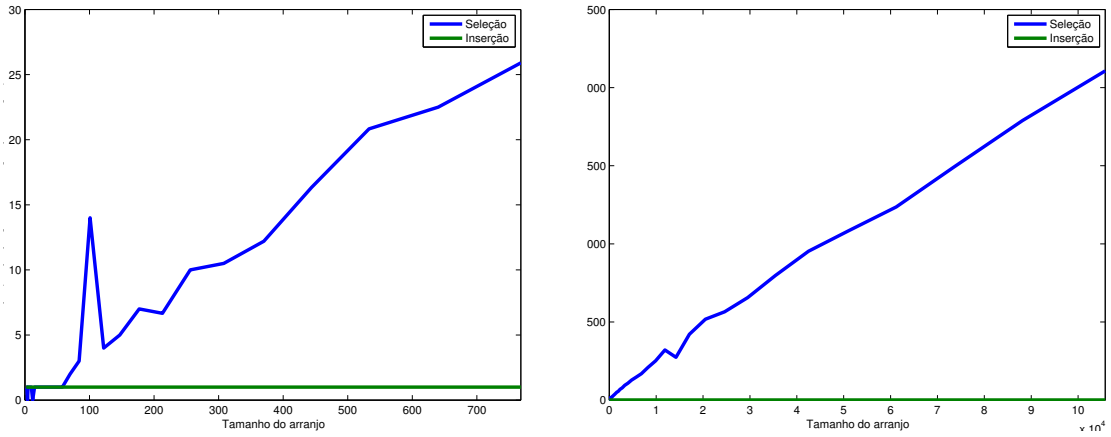
Figura 16.10: Tempo necessário para se ordenar um arranjo com ordem decrescente (melhor caso para a ordenação por inserção).

Vemos então como a diferença real entre algoritmos para a mesma tarefa pode ser significativa em aplicações práticas. Como previmos com nossa medida por modelos matemáticos que o algoritmo por inserção teria um melhor caso $O(n)$, já era esperado que houvesse grande diferença de desempenho em relação à ordenação seleção, que tem custo $O(n^2)$ para todos os casos.

16.2.4 Exercícios

Exercício 16.1 Analise o código abaixo, que está incompleto.

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <time.h>
4
5 using namespace std;
6
7 void learranjo(int a[], int n);
```



(a) Arranjos menores (b) Arranjos maiores

Figura 16.11: Proporção entre tempo necessário para se ordenar um arranjo com ordem crescente (melhor caso para a ordenação por inserção).

```
8  void imprimearranjo(int a[], int n);
9  void embaralhaarranjo(int a[], int n);
10 // ordenação pelo método da bolha
11 void ordena(int a[], int n);
12 // ordenação pelo método 1
13 void ordena1(int a[], int n);
14 // ordenação pelo método 2
15 void ordena2(int a[], int n);
16
17 int main() {
18     // gerador de números aleatórios
19     srand(time(NULL));
20
21     // tamanho do arranjo
22     int n;
23     cout << "Digite␣o␣tamanho␣do␣arranjo:␣";
24     cin >> n;
25     // ponteiro para arranjo que será alocado com tamanho n
26     int *v;
27     v = new int[n];
28     learranjo(v,n);
29     imprimearranjo(v,n);
30
31     cout << "Embaralhando" << endl;
32     embaralhaarranjo(v,n);
33     imprimearranjo(v,n);
34     cout << "Ordenando␣pelo␣método␣da␣bolha␣" << endl;
35     ordena(v,n);
36     imprimearranjo(v,n);
37
38     cout << "Embaralhando" << endl;
```

```
39      embaralhaarranjo(v,n);
40      imprimearranjo(v,n);
41      cout << "Ordenando␣pelo␣metodo␣???␣" << endl;
42      ordena1(v,n);
43      imprimearranjo(v,n);
44
45      cout << "Embaralhando" << endl;
46      embaralhaarranjo(v,n);
47      imprimearranjo(v,n);
48      cout << "Ordenando␣pelo␣metodo␣???␣" << endl;
49      ordena2(v,n);
50      imprimearranjo(v,n);
51
52      return 0;
53  }
54
55  void embaralhaarranjo(int a[], int n) {
56      int pos,i,aux;
57      for (i=0; i<n; i++){
58          pos = rand() % n + 0; // posicao entre 0 e n-1
59          aux = a[i];
60          a[i] = a[pos];
61          a[pos] = aux;
62      }
63  }
64
65  void learranjo(int a[], int n)
66  {
67      for (int i = 0; i < n; i++){
68          cout << "Digite␣o␣elemento␣v[" << i << "]:␣";
69          cin >> a[i];
70      }
71  }
72
73
74  void imprimearranjo(int a[], int n)
75  {
76      cout << "v␣=␣" ;
77      for (int i = 0; i < n; i++){
78          cout << a[i] << "␣";
79      }
80      cout << endl;
81  }
82
83  void ordena(int a[], int n)
84  {
85      int i,j,aux;
86      for( j= n-1; j>0; j--)
87          for(i=0; i<j; i++)
```

```
88          {
89              if(a[i+1] < a[i])
90              {
91                  // trocar a[i] com a[i+1]
92                  aux = a[i];
93                  a[i] = a[i+1];
94                  a[i+1] = aux;
95              }
96          }
97  }
98
99  void ordena1(int a[], int n)
100 {
101     int i, j, pos_min, aux;
102     for (i = 0; i < n - 1; i++)
103     {
104         pos_min = i;
105         for (j = i + 1; j < n; j++)
106         {
107             if (a[j] < a[pos_min])
108             {
109                 pos_min = j;
110             }
111         }
112         // troca a[i] com a[min]
113         aux = a[pos_min];
114         a[pos_min] = a[i];
115         a[i] = aux;
116     }
117 }
118
119 void ordena2(int a[], int n)
120 {
121     int i, j, aux;
122     for (i = 1; i < n; i++)
123     {
124         aux = a[i];
125         j = i - 1;
126         while (aux < a[j] && j >= 0)
127         {
128             a[j+1] = a[j];
129             j--;
130         }
131         a[j+1] = aux;
132     }
133 }
```

■

Exercício 16.2 Renomeie as funcoes `ordena1` e `ordena2` para o nome correto dos métodos de ordenação. ■

Exercício 16.3 Crie um `struct` do tipo `Aluno` com `nome` e numero de `matricula` de cada aluno. ■

Exercício 16.4 Faça com que as funções de ordenação ordenem agora arranjos de elementos do tipo `Aluno` de acordo com seus números de matrícula. ■

Exercício 16.5 Faça com que as funções de ordenação retornem o número de comparações realizadas em vez de `void`. ■

Exercício 16.6 Compare o número de comparações realizadas por cada método de ordenação para diferentes tamanhos de arranjo. Para que isto seja possível em arranjo grandes, faça com que os elementos sejam atribuídos automaticamente ao arranjo. Utilize a função `rand()`, apresentada na função `embaralhaarranjo`. ■

16.3 Algoritmos eficientes de ordenação

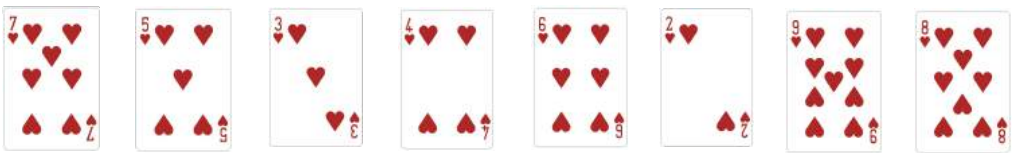
Apresentaremos nesta seção métodos simples de ordenação. Estes métodos costumam ter custo $O(n \log n)$ em relação ao número de comparações.

16.3.1 Merge sort

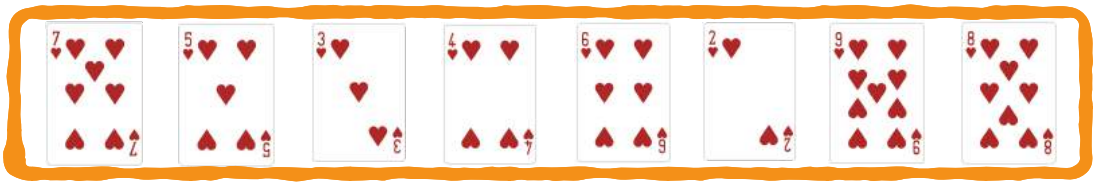
O *merge sort* (ordenação por intercalação), é um método que utiliza uma técnica de *dividir para conquistar*. É um algoritmo onde a cada passo, se divide o arranjo em dois subarranjos menores até que tenhamos vários arranjos de tamanho 1. Então, iniciamos uma repetição onde, dois subarranjos são fundidos em um subarranjo maior ordenado até que tenhamos o arranjo original ordenado.

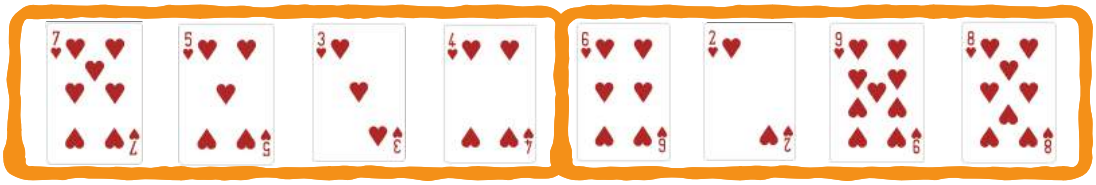
Exemplo

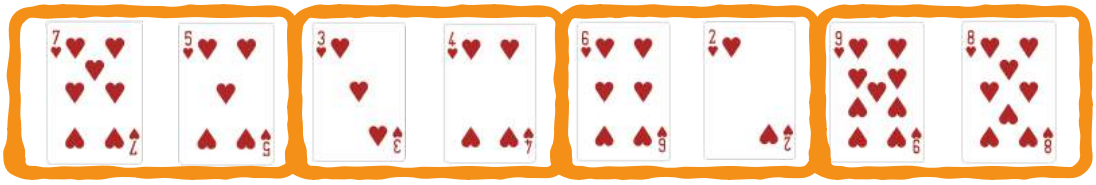
Considere um arranjo de cartas com os seguintes elementos:

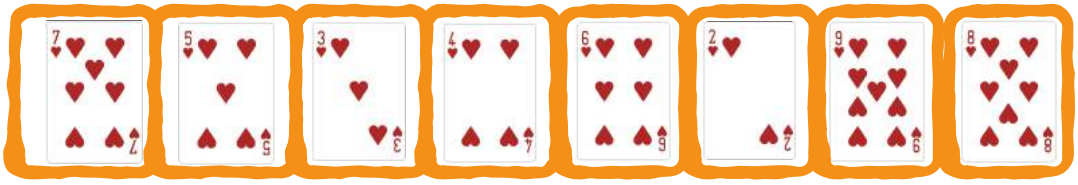


Inicialmente, consideramos que temos um subarranjo desordenado de 8 elementos. Vamos iniciar assim a etapa de ***divisão***. Vamos dividir cada subarranjo na metade. Esse processo continua até que tenhamos subarranjos de um elemento cada.

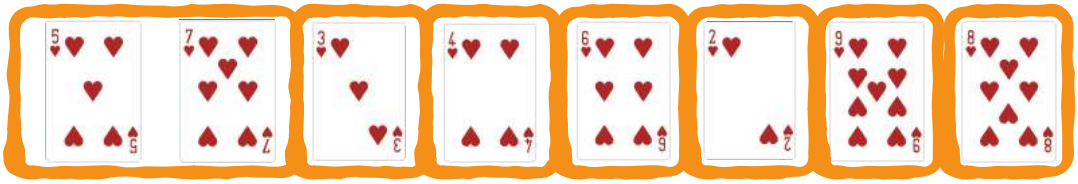




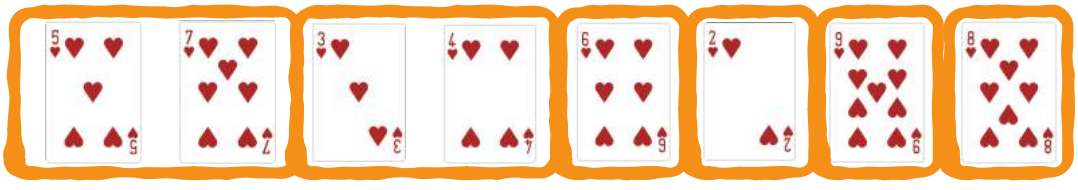


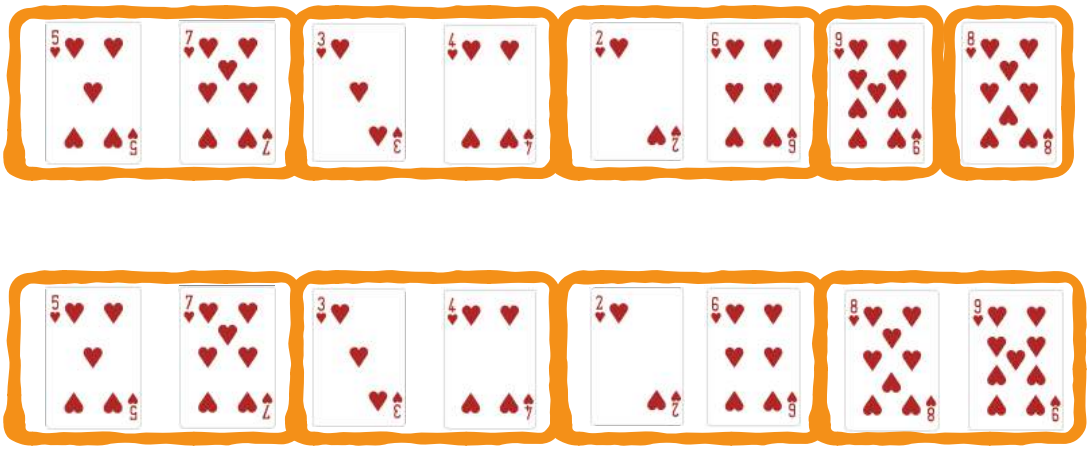


Por definição, sabemos que cada um destes arranjos de 1 elemento é um arranjo ordenado. Iniciaremos agora a fase de ***intercalação*** destes arranjos ordenados. Os dois primeiros arranjos ordenados são intercalados em um novo arranjo ordenado.

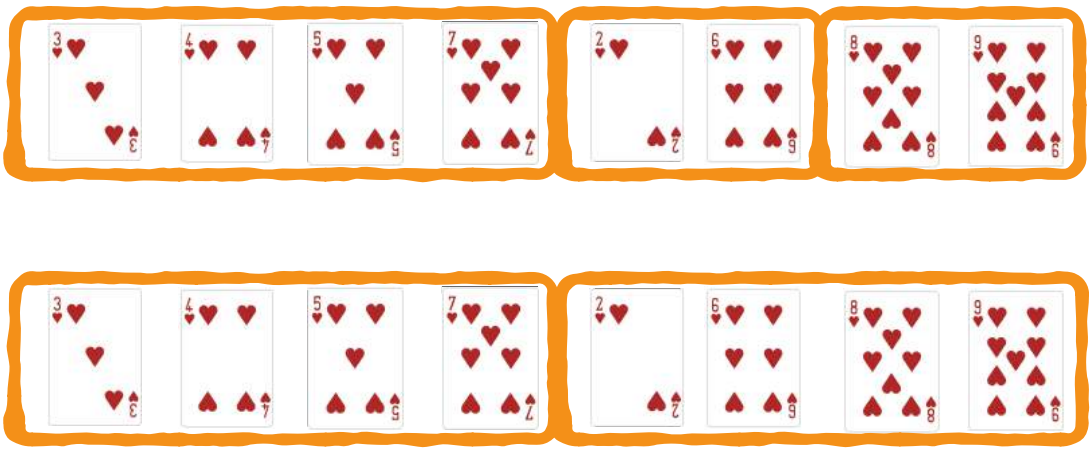


Fazemos isto com todos os arranjos menores.

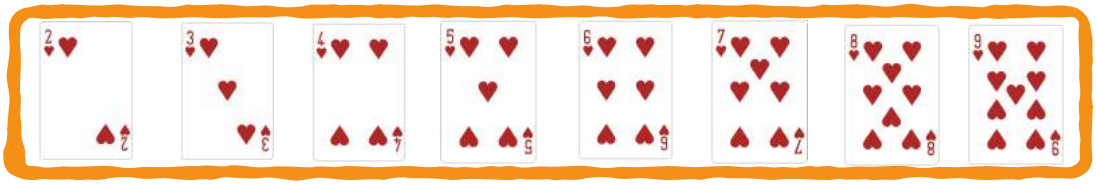




O interessante é que o custo de se intercalar dois arranjos *ordenados* de tamanho $n = O(n)$ em um arranjo ordenado de tamanho $2n = O(n)$ é apenas $O(n)$, um custo menor do que seria o custo de se colocar dois arranjos *desordenados* em um arranjo ordenado (o que teria o mesmo custo de uma ordenação). Fazemos a intercalação para os arranjos de tamanho 2:



Na última etapa de intercalação, temos todo o arranjo original ordenado.



A Figura 16.12 apresenta de forma resumida os passos deste processo de ordenação com o merge sort. Na etapa de divisão, os arranjos desordenados são divididos até que formem arranjos ordenados de apenas 1 elemento. Estes arranjos ordenados são representados em amarelo. A cada passo de intercalação, ou conquista, intercalamos arranjos ordenados em novos arranjos ordenados maiores.

Figura 16.12: Exemplo de ordenação com o merge sort.

Algoritmo de intercalação

Em sua implementação mais usual, a etapa de intercalação de dois arranjos requer um arranjo auxiliar, onde serão colocados os itens. Os elementos são intercalados em um arranjo auxiliar e então são transferidos de volta para o arranjo de onde vieram: o arranjo sendo ordenado. Isto é apresentado na Figura 16.13

Assim, para se intercalar n elementos, precisamos de uma memória extra $O(n)$ para o arranjo auxiliar.

Temos abaixo o algoritmo de intercalação:

```
1  void intercala(int a[], int n) {
2      // arranjo temporário para intercalação
3      int *temp = new int[n];
4      // índice que marca o meio do arranjo
5      int meio = n / 2;
6      // índices para a estrutura de repetição
7      int i, j, k;
8      // índice i para itens do primeiro arranjo
9      i = 0;
10     // índice j para itens do segundo arranjo
11     j = meio;
12     // índice k para itens no arranjo temporário
13     k = 0;
14     // enquanto i e j não chegam ao fim dos arranjos
15     while ((i < meio) && (j < n)){
```

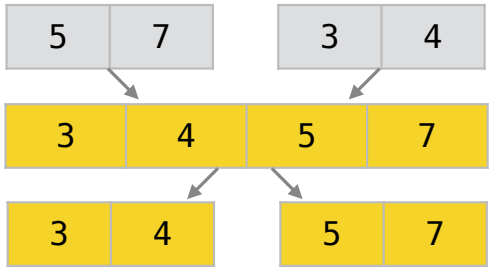


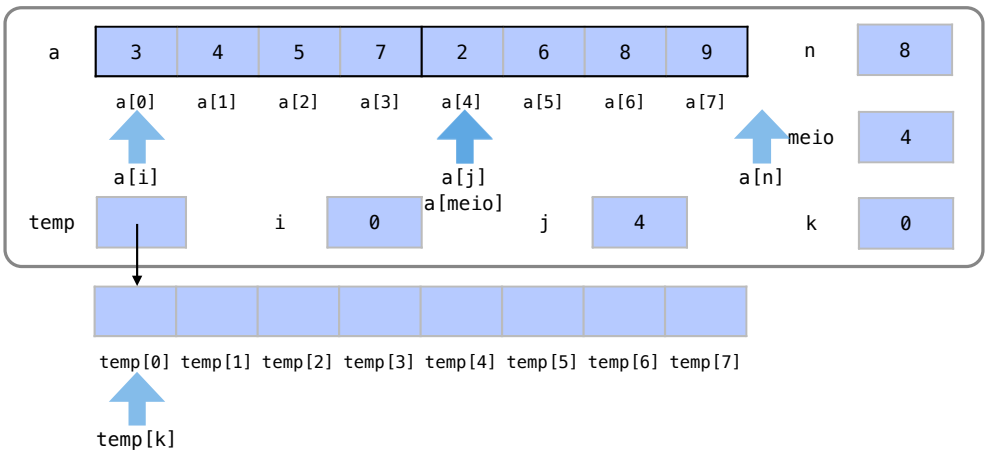
Figura 16.13: Etapa de intercalação de um merge sort.

```
16          // o menor entre a[i] e a[j] vai temp
17          if (a[i] < a[j]){
18              temp[k] = a[i];
19              i++;
20          } else {
21              temp[k] = a[j];
22              j++;
23          }
24          k++;
25      }
26      // se o índice i chegou ao fim de seu arranjo
27      if (i == meio){
28          // copiamos o restante do segundo arranjo para temp
29          while (j < n){
30              temp[k] = a[j];
31              j++;
32              k++;
33          }
34      }
35      // se foi j quem chegou ao fim de seu arranjo
36      else {
37          // copiamos o restante do primeiro arranjo para temp
38          while  (i < meio){
39              temp[k] = a[i];
40              i++;
41              k++;
42          }
43      }
44      // temp agora tem todos os elementos intercalados
45      // copiamos os elementos de volta para a[]
46      for (i = 0; i < n; i++){
47          a[i] = temp[i];
48      }
49      // o arranjo temporário pode ser então apagado
50      delete[] temp;
51  }
```

O algoritmo de intercalação intercala a primeira metade ordenada do arranjo `a`, que é `a[0]` a `a[(n/2)-1]`, com sua segunda metade ordenada `a[n/2]` a `a[n-1]`. O algoritmo é organizado

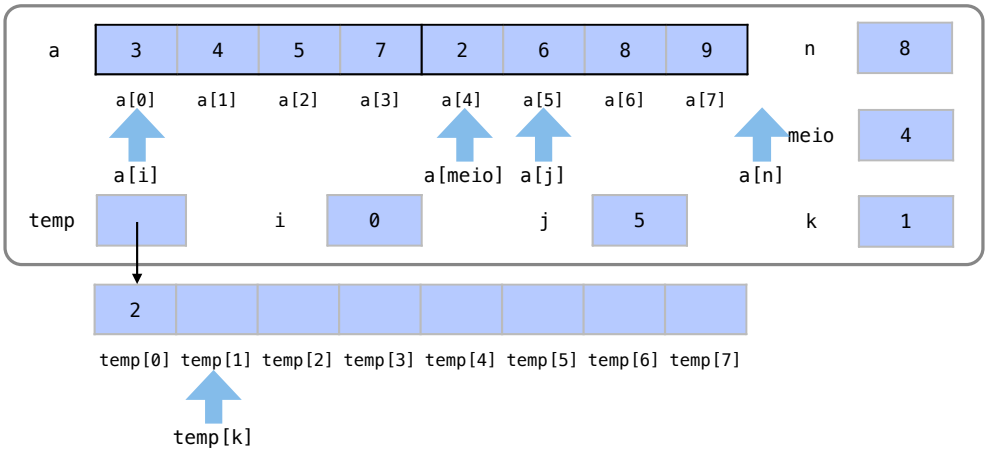
em 3 etapas principais. Entre as linhas 2 e 13, criamos um arranjo temporário e inicializamos os índices. Entre as linhas 14 e 43, os elementos de `a` são intercalados nesse arranjo temporário `temp`. Na linhas 44 a 50, os elementos intercalados no arranjo temporário são transferidos de volta para o arranjo `a`.

Temos aqui um exemplo de intercalação. Na linha 3, criamos um ponteiro `temp` que aponta para um arranjo de dados do tipo `int`. Note que o arranjo é alocado dinamicamente e, por isto, está fora do escopo da função. O índice `meio`, declarado na linha 5, marca onde está o meio do arranjo e não se alterará ao longo da função. Elementos entre `a[0]` e `a[meio-1]` representam representam o primeiro subarranjo ordenado. Elementos entre `a[meio]` e `a[n-1]` representam representam o segundo subarranjo ordenado. Por fim, nas linhas 8 a 13, o índice `i` marcará o início do primeiro subarranjo ordenado, o índice `j` marcará o início do segundo subarranjo ordenado e o índice `k` marcará o início do arranjo temporário.

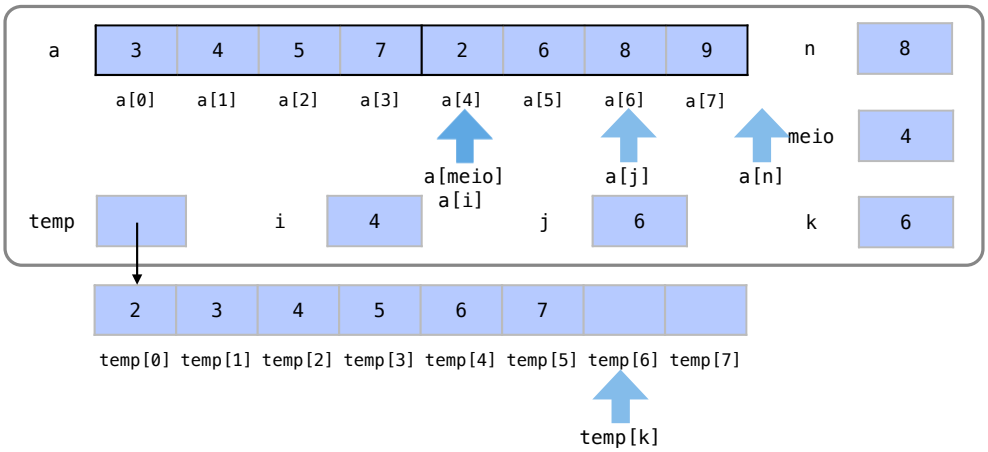


Na função os elementos de a[0] até a[3] (ou `a[meio-1]`) serão intercalados com os elementos de a[4] (ou `a[meio]`) a a[7] (ou `a[n-1]`). Por isto a condição de repetição na linha 15 é que `i` ainda não tenha chegado a `meio` e `j` não tenha chegado a `n`.

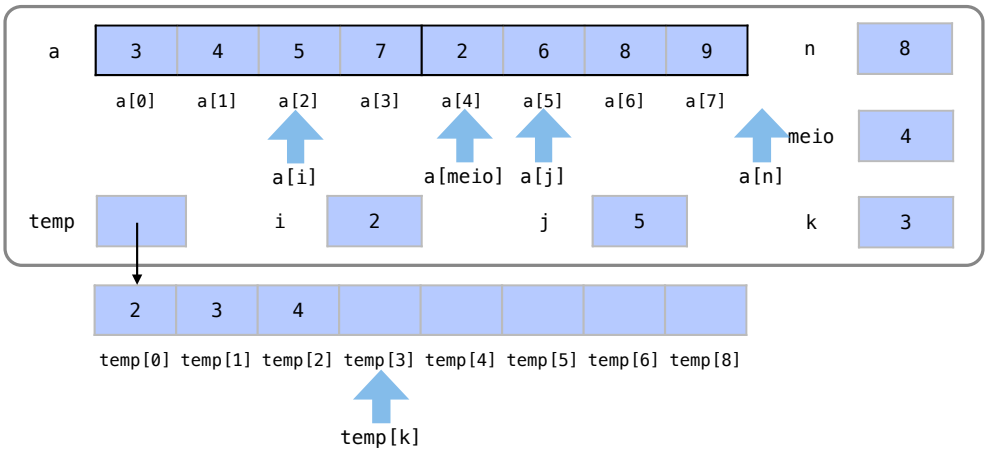
Na linha 17, sempre escolhemos o menor elemento entre `a[i]` e `a[j]` para guardarmos em `temp[k]`. Nas linhas 18 e 19, guardamos o valor do primeiro subarranjo em `temp` e incrementamos `i` para marcar qual o próximo elemento a ser copiado deste arranjo. Nas linhas 21 e 22, guardamos o valor do segundo subarranjo em `temp` e incrementamos `j`. O índice `k` é sempre incrementado para marcar a próxima posição do arranjo temporário. Na primeira iteração, por exemplo, como `a[j]` é menor que `a[i]`, copiaremos `a[j]` para `temp[k]` e incrementaremos os índices `j` e `k`.



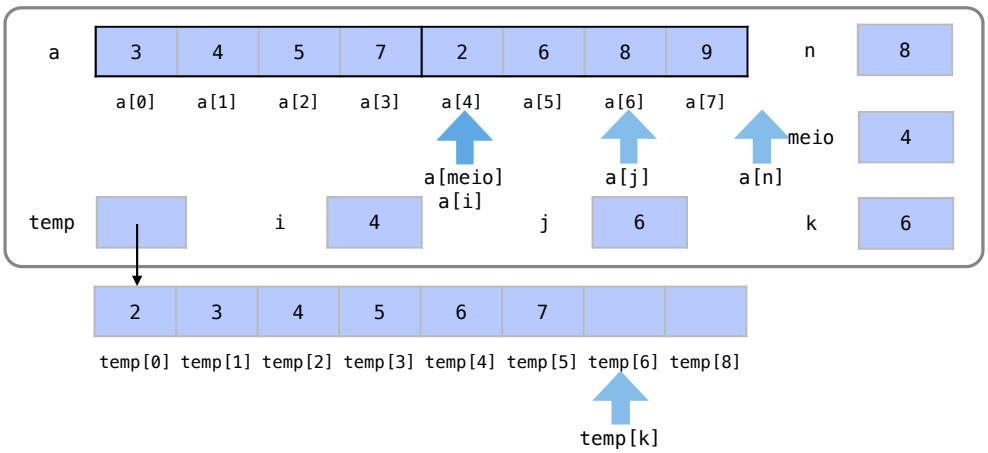
Este processo das linhas 16 a 24 continua se repetindo até que `i` atinja `meio` ou `j` atinja `n`. O valor 3, por ser menor que 6, é inserido no arranjo temporário `temp`.



O valor 4, por ser menor que 6, é inserido no arranjo temporário `temp`.

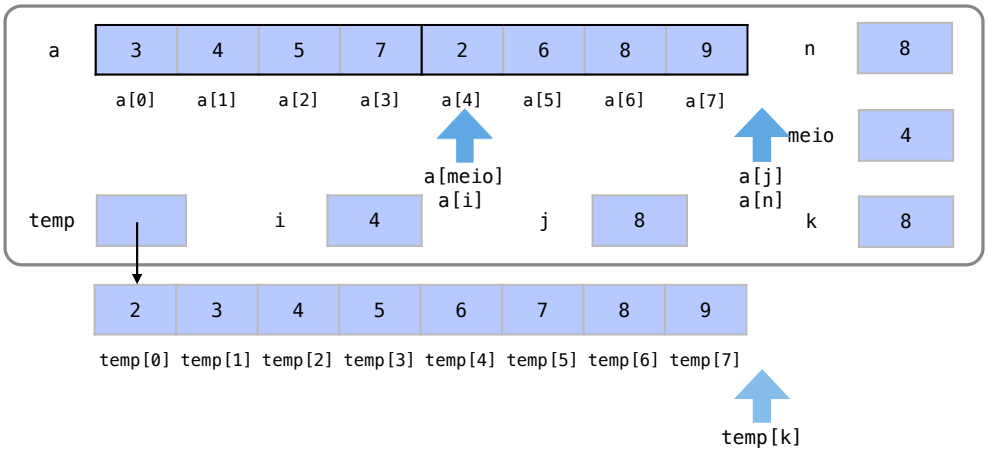


Em nosso exemplo, a repetição se encerrará quando `i` atinjir o valor de `meio`, indicando que todos os elementos do primeiro subarranjo ordenado já estão em `temp`.



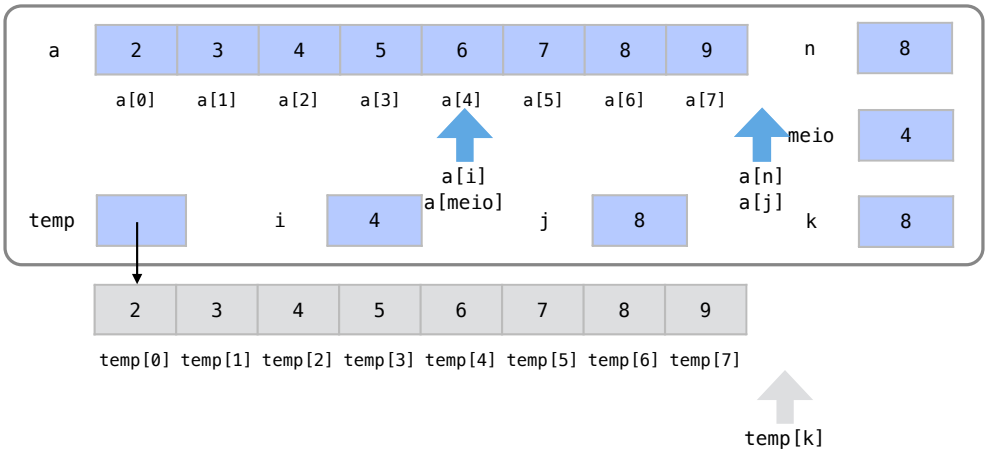
Ao sair do laço de repetição, temos um teste `i == meio` na linha 27. Este teste nos diz se a condição de repetição encerrou porque o primeiro subarranjo foi todo percorrido por `i` (`i == meio`) ou porque o segundo subarranjo foi todo percorrido por `j` (`j == n`). Em nosso exemplo, a primeira condição é verdadeira, o que nos leva para o bloco de comandos entre as linhas 28 e 33. Isso quer dizer que os outros elementos do segundo arranjo ainda precisam ser copiados.

Para copiar os elementos restantes do outro subarranjo, temos uma estrutura de repetição `while` que percorrerá todas as posições `a[j]` guardando estes elementos em `temp[k]`. Ao fim do processo, `i` sempre será `meio` e `j` sempre será `n`.



Note que se o segundo subarranjo tivesse sido todo copiado primeiro, os comandos entre as linhas 36 e 43 seriam executados para copiar os elementos restantes do primeiro subarranjo de maneira análoga.

Ao chegar neste ponto, o vetor `temp` tem todos os elementos intercalados. Ao fim do código, nas linhas 44 a 50, copiamos os elementos de temp de volta para `a` e desalocamos o arranjo apontado por `temp`.



Note que é importante desalocar o arranjo apontado por `temp`. Se o arranjo não for desalocado, temos um erro de vazamento de memória, pois o arranjo foi alocado dinamicamente e não pertence ao escopo da função.

Nos nossos exemplos de alocação dinâmica de arranjos, na Seção 11.3, fizemos com que os ponteiros apontassem para `nullptr` após desalocar a memória apontada por eles. Neste caso, não precisamos fazer com que `temp` receba `nullptr` pois `temp` deixará de existir logo na próxima linha, ao fim da função.

Algoritmo de ordenação

A utilidade de funções é justamente quebrar problemas em problemas menores. Usando a função apresentada de intercalação como uma função auxiliar, podemos definir de maneira simples a função de ordenação com o merge sort. Essa função de ordenação pode ser facilmente expressa em termos recursivos.

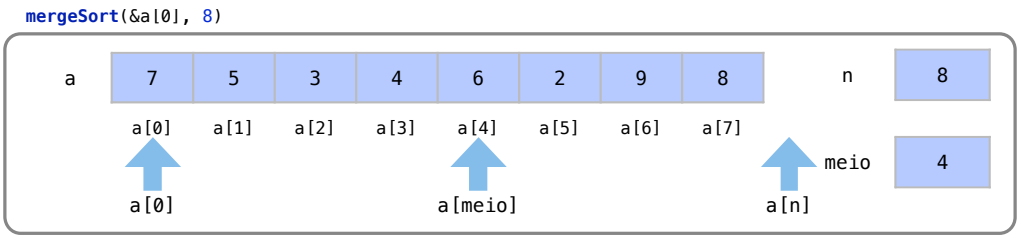
Utilizando a função de intercalação como uma função auxiliar, esta função recursiva ordena um arranjo com o método Merge Sort.

```
1 void mergeSort(int a[], int n) {
2     int meio;
3     if (n > 1) {
4         meio = n / 2;
5         mergeSort(a, meio);
6         mergeSort(a + meio, n - meio);
7         intercala(a, n);
8     }
9 }
```

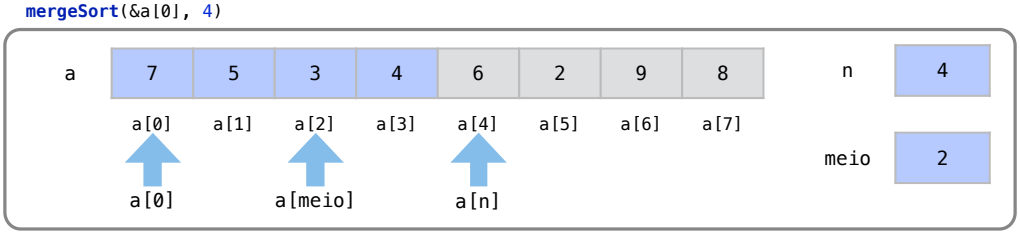
Na linha 4, dividimos o arranjo ao meio. Na linha 5, usamos o próprio método recursivamente para ordenar o arranjo até a metade. Na linha 6, agora usamos o próprio método recursivamente para ordenar o arranjo da metade até o final. Na linha 7, intercalamos os dois subarranjos ordenados em a com a função de intercalação que já deve estar declarada.

Como vimos na Seção 9.11, sobre recursão, as funções recursivas precisam de um *passo recursivo* e um *caso base*. O caso base garante que não entraremos em uma recursão infinita. Veja que o passo recursivo acontece sempre que chamamos a própria função `mergeSort` para um arranjo menor que o original. Já a condição da linha 3 garante que o caso base ocorra. Esta condição garante que o passo recursivo só será executado para arranjos maiores que `1`. Para arranjos de tamanho `0` ou `1`, temos nosso caso base, pois o arranjo já está ordenado e basta não executar instrução alguma.

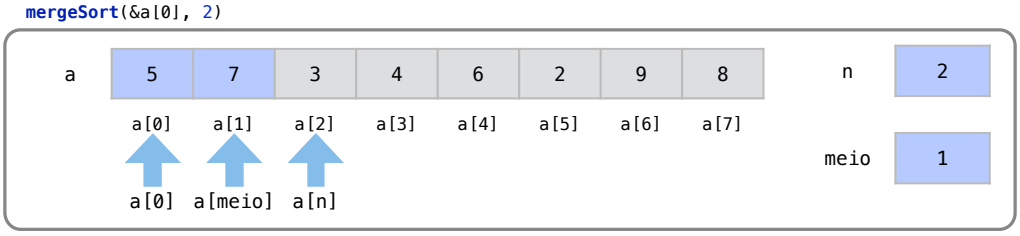
Em nosso exemplo, a função será executada então nas primeira metade do arranjo `a[0]` a `a[3]` e na segunda metade `a[4]` a `a[8]`.



Na linha 5 da função `mergeSort(&a[0],8)`, é chamada uma cópia da função para ordenar o arranjo nas posições `a[0]` a `a[3]`. Lembre-se que um arranjo nada mais é que um endereço onde se inicia uma série de elementos alocados na memória. Assim, a função chamada ordenará o arranjo de elementos que começa no endereço de `a` e vai até 4 posições a frente. A função original `mergeSort(&a[0],8)` fica na pilha de funções e fica em espera.

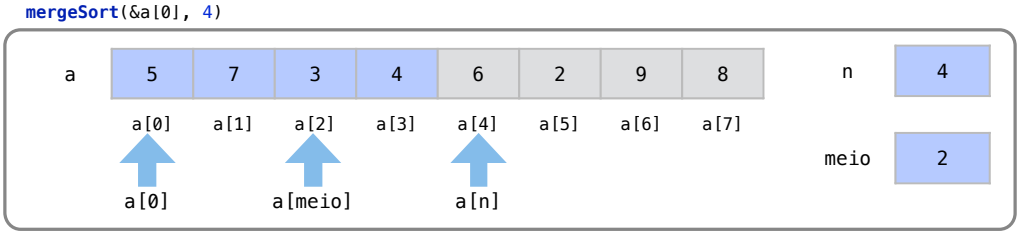


Na linha 5 da função `mergeSort(&a[0],4)`, é chamada uma cópia da função para ordenar o arranjo nas posições `a[0]` a `a[1]`. A função `mergeSort(&a[0],4)` fica na pilha de funções e fica em espera.

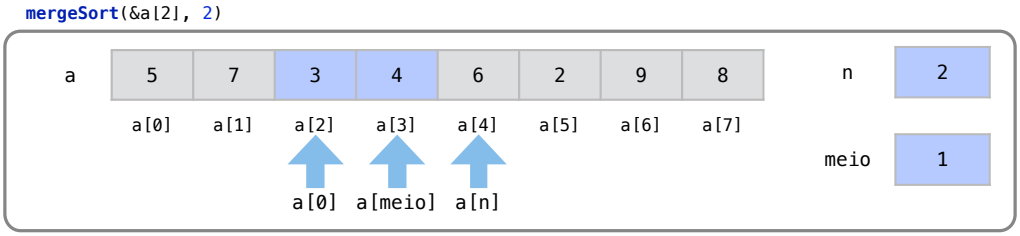


Na função `mergeSort(&a[0],2)` as duas chamadas de função das linhas 5 e 6 não executarão nenhum comando, pois elas terão arranjo de tamanho 1, que é o caso base da função. Assim, iremos para a linha 7 da função `mergeSort(&a[0],2)`. A linha 7 utiliza a função `intercala`, que temos definida anteriormente. Assim, os elementos 7 e 5 são intercalados de maneira correta.

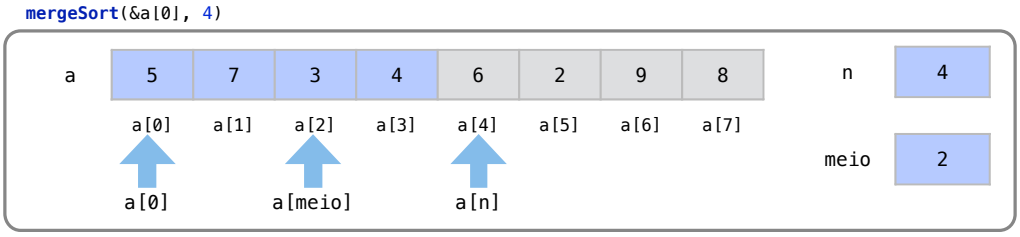
A função `mergeSort(&a[0],4)`, que havia chamado `mergeSort(&a[0],2)` volta a ser executada.



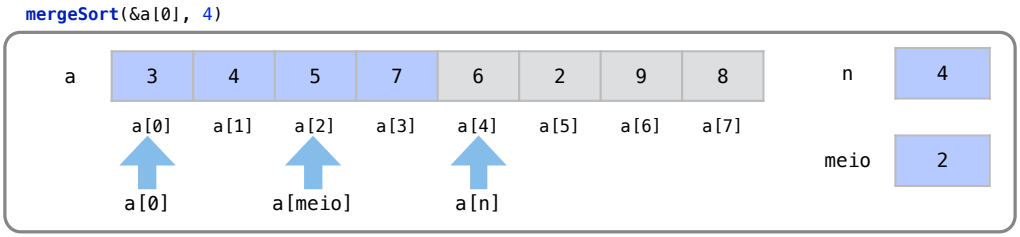
Na linha 6, esta função chama mergeSort para o arranjo que se inicia no endereço `a + meio` e tem tamanho `n-meio`.



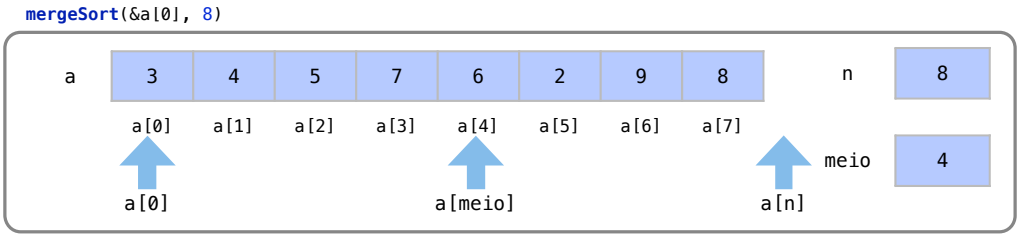
Na função `mergeSort(&a[2],2)`, as duas operações de ordenação das linhas 6 e 7 cairão no caso base, pois há apenas um elemento. A operação de intercalação manterá os elementos como estão já que eles já estão em ordem. Assim retornaremos à função `mergeSort(&a[0],4)`, que a chamou.



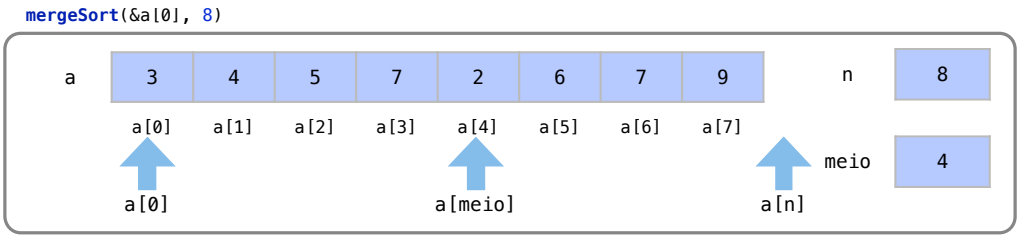
A chamada `mergeSort(&a[0],4)` da função já executou recursivamente assim as linhas 5 e 6, que ordenam os elementos nas duas metades do arranjo. Na linha 7, os elementos `a[0]` e `a[1]`, da primeira metade do arranjo, serão intercalados com os elementos `a[2]` e `a[3]`, do segundo subarranjo ordenado.



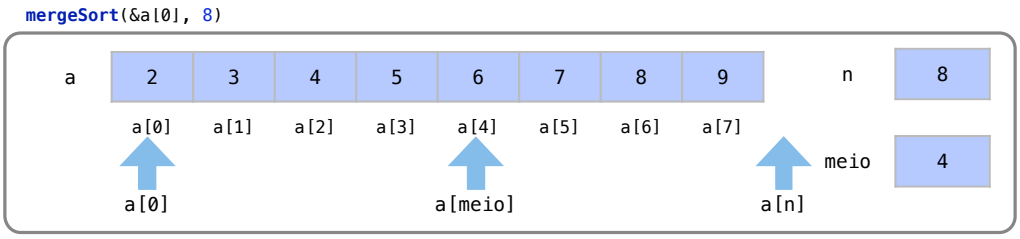
Com o fim da intercalação, a função `mergeSort(&a[0],4)` retorna e voltamos a executar sua função chamadora `mergeSort(&a[0],8)`.



A execução da linha 5 já ordenou recursivamente a primeira metade do arranjo. Como já vimos ser possível, na linha 6, a função `mergeSort(&a[4],4)` ordenará a segunda metade do arranjo, que começa no endereço `&a[4]` e tem 4 elementos.



Basta agora intercalar as duas metades ordenadas do arranjo, o que é feito na linha 7.



Temos assim o arranjo original ordenado e finalizamos a função.

Análise

Uma peça fundamental do algoritmo é a intercalação de dois arranjos. Esta intercalação tem um custo $O(n)$ pois passamos 1 vez por cada elemento o colocando-o no arranjo temporário.

Como cada etapa de intercalação custa $O(n)$, o custo total do algoritmo merge sort depende então do número de intercalações feitas. Para isto, analisemos todos os passos de intercalação apresentados na Figura 16.14.

No exemplo note que houve $k = 3$ passos de intercalação para $n = 8$ elementos sendo ordenados. Em cada passo de intercalação, temos um custo $O(n)$. Assim, o custo total do algoritmo é $O(nk)$. Nos resta saber o número k de passos.

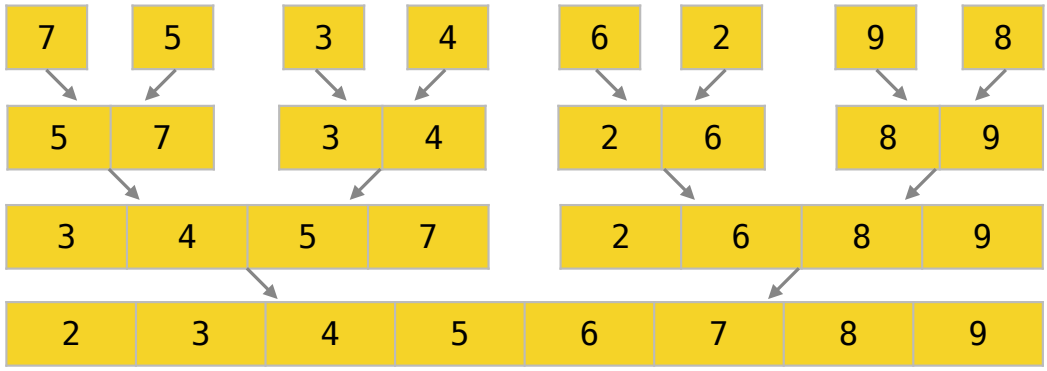


Figura 16.14: Representação de todas as etapas de intercalação de um merge sort.

Como a cada passo dobramos o tamanho do arranjo intercalado, em k passos de intercalação conseguiremos ordenar k^2 elementos. Se quisermos saber o número de passos para ordenação de n elementos, podemos inverter a equação e termos:

$$2^k = n \qquad\qquad k = \lg n$$

Sendo assim, precisamos de $k = \lg n$ passos para encerrar o algoritmo com n elementos. Assim, o custo total do algoritmo merge sort é $O(nk) = O(n \log n)$ para qualquer caso.

Como vimos na Seção 14.6, sobre classes de algoritmo, o algoritmo merge sort é um algoritmo típico da classe loglinear, pois é um algoritmo que quebra o problema em problemas menores, faz uma operação para cada um dos elementos e depois combina os resultados.

Sendo assim, a complexidade do método é $O(n \log n)$. Uma vantagem do método é sua complexidade constante para o pior caso e melhor caso. Sua maior desvantagem é a necessidade de memória extra $O(n)$ na fase de intercalação e nas chamadas recursivas da função. Dessa forma, esse é o algoritmo a ser usado quando queremos uma ordem de complexidade baixa, constante, mas que a memória não seja um problema. Ele é também um algoritmo estável, pois a fase de intercalação não desfaz a ordem relativa entre elementos de mesma chave.

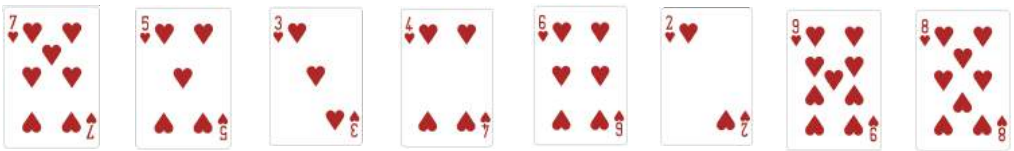
16.3.2 Quicksort

Para uma ampla variedade de situações, o **quicksort** é o método mais rápido que se conhece. Assim como o merge sort, o quicksort também é definido de maneira recursiva. A cada passo do Quicksort, o problema de ordenação é dividido em dois problemas menores que são ordenados de maneira independente.

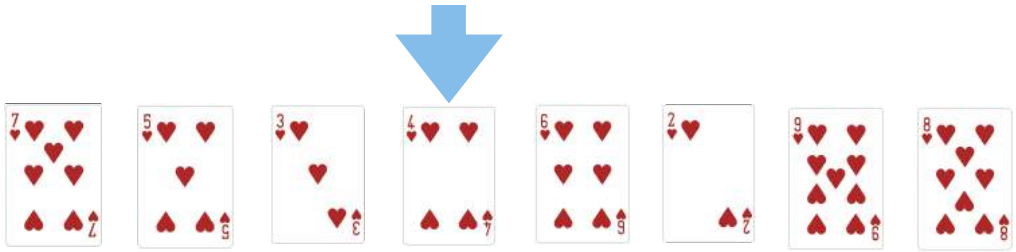
A parte mais complexa do método é a partição do problema em problemas menores. Esse processo de partição é feito a partir da escolha de um pivô `x`. Assim que escolhido o pivô, um arranjo é dividido em duas partes: (i) o subarranjo da esquerda, com elementos menores ou iguais a `x`, e (ii) o subarranjo da direita com elementos maiores ou iguais a `x`.

Exemplo

Considere um arranjo de cartas com os seguintes elementos:

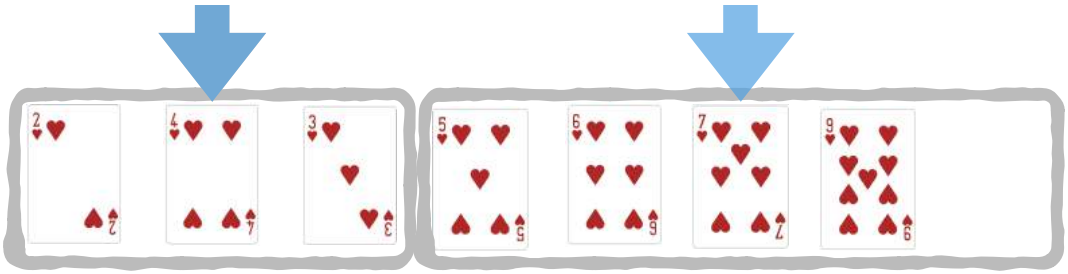


Escolheremos um pivô x, que neste exemplo será o elemento do meio do arranjo, 4.

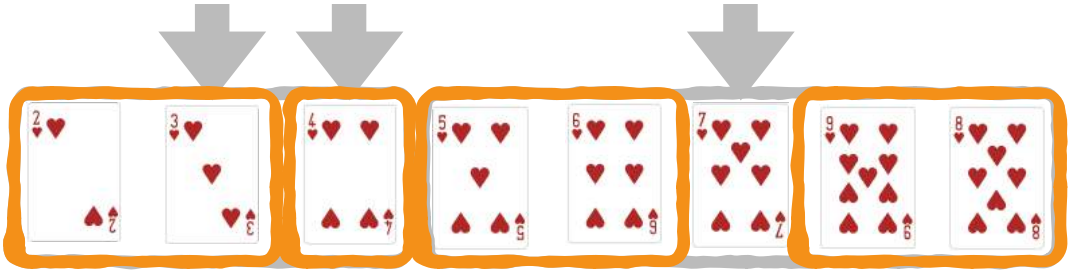


Particionamos o arranjo de modo que elementos menores que o pivô fiquem à esquerda e elementos maiores fiquem à direita.

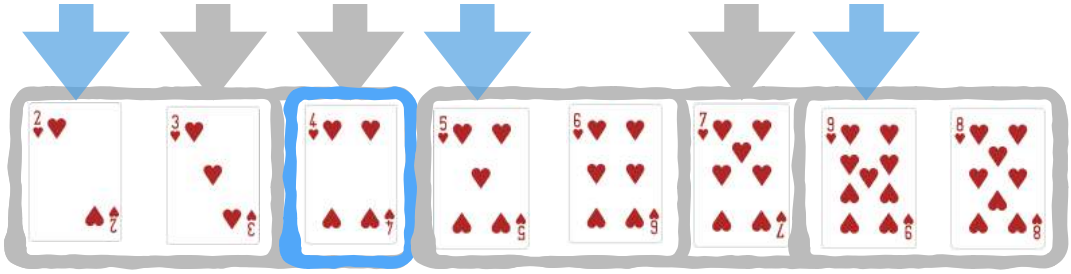
Pelo mesmo princípio recursivo, selecionamos um pivô para cada subarranjo.



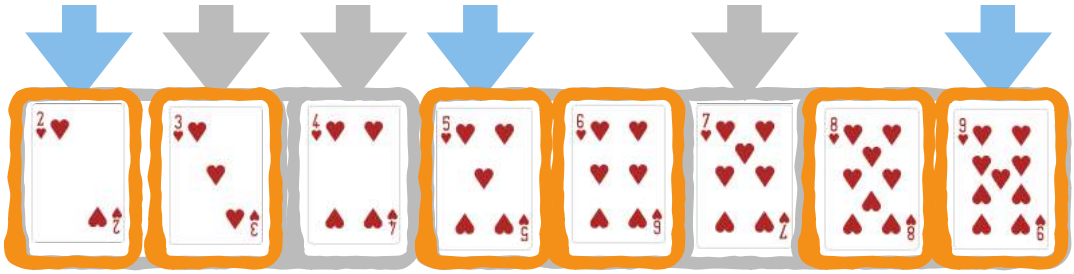
Particionamos novamente os subarranjos com os maiores à direita e menores à esquerda



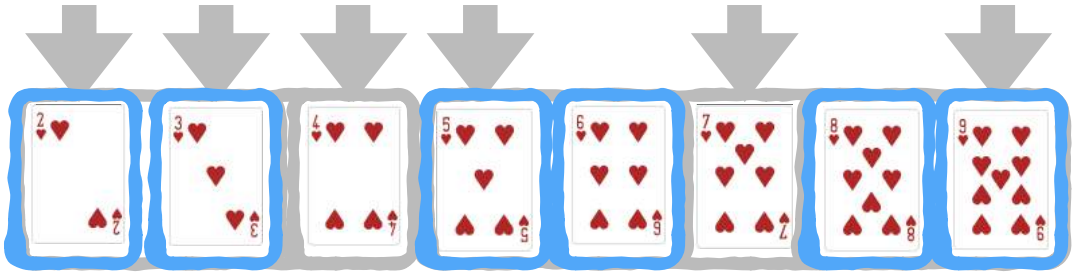
Repetiremos o processo para cada subproblema. Porém, os arranjos de tamanho 1 já são considerados como ordenados. Este é o caso base.



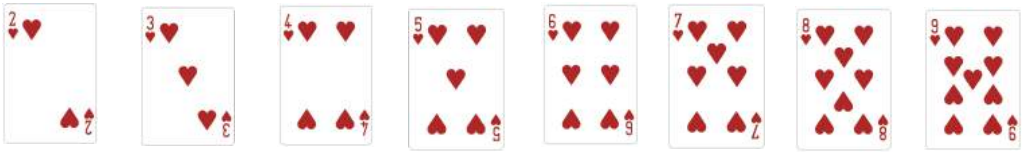
Este é o resultado do novo processo de partição.



Todos os subarranjos gerados de tamanho 1 podem ser então considerados ordenados.



Com todos os subarranjos ordenados, sabemos que o arranjo original está ordenado.

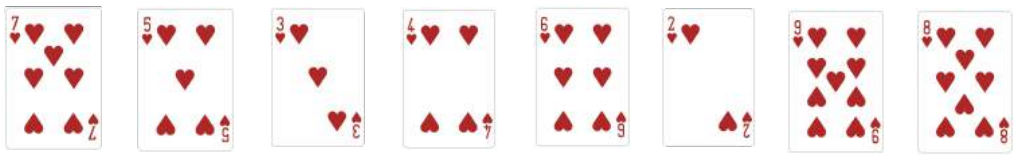


Particionamento

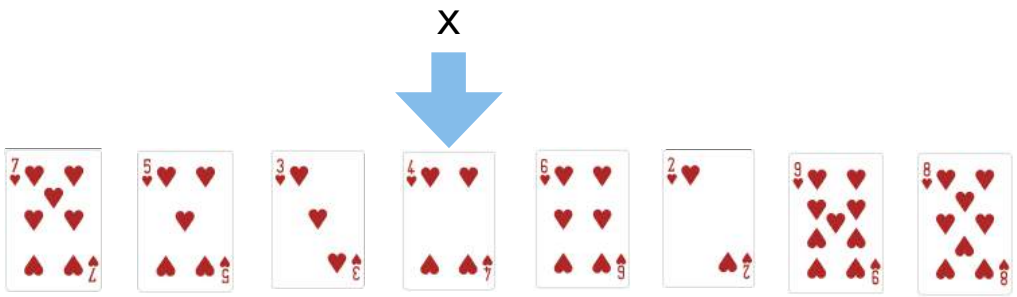
Para fazer o particionamento de um arranjo no quicksort:

1. Escolhemos um pivô `x`
2. Percorremos o arranjo a partir da esquerda até que `a[i] >= x`
3. Percorremos o arranjo a partir da direita até que `a[j] >= x`
4. Se `i` e `j` não tiverem cruzado, trocamos os elementos de a[i] com a[j]
5. Incrementamos `i`, decrementamos `j` e continuamos do passo 2 até que i e j se cruzem

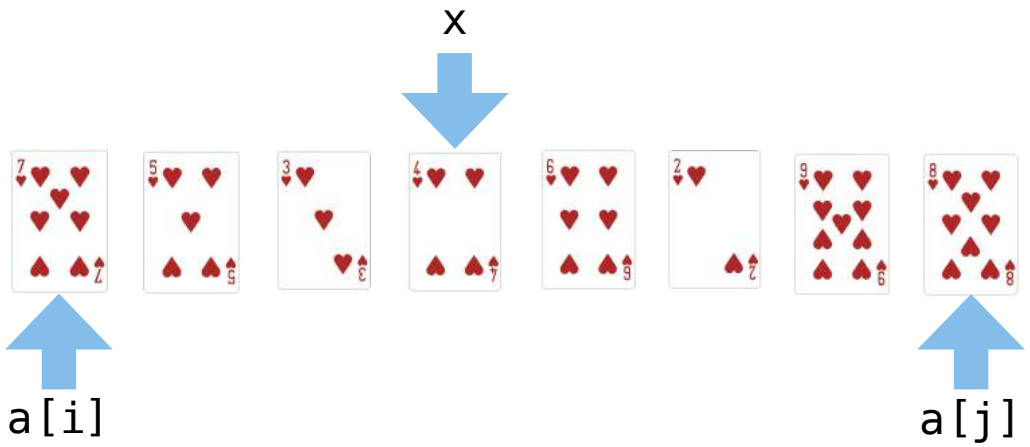
Como exemplo, considere novamente o arranjo de cartas com os seguintes elementos:



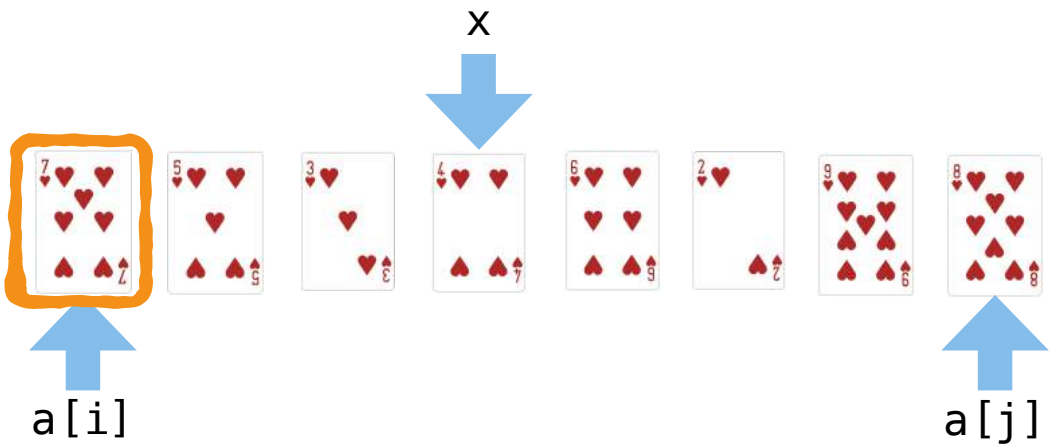
Escolhemos um elemento como pivô.



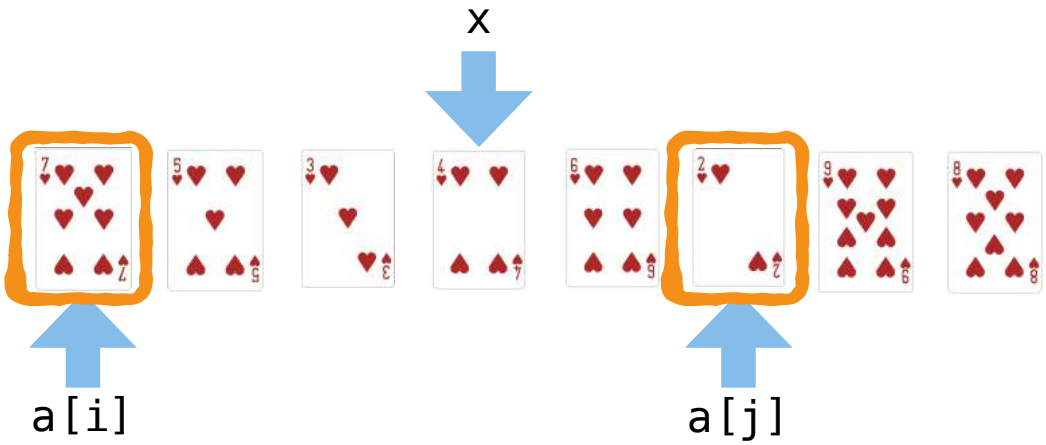
Definimos índices `i` e `j` que marcam o ínicio e o fim do arranjo



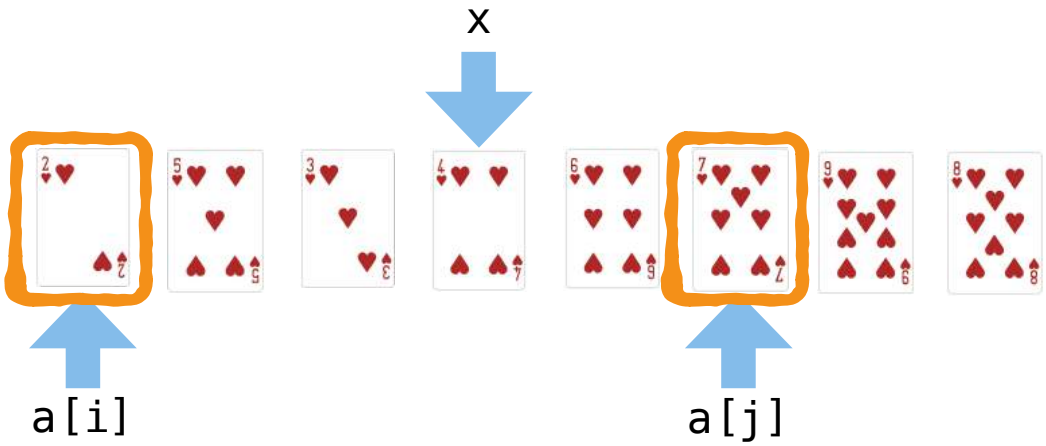
Movimentamos `i` para a direita até encontrar um elemento maior ou igual a `x`. Neste caso, o elemento é o próprio 7.



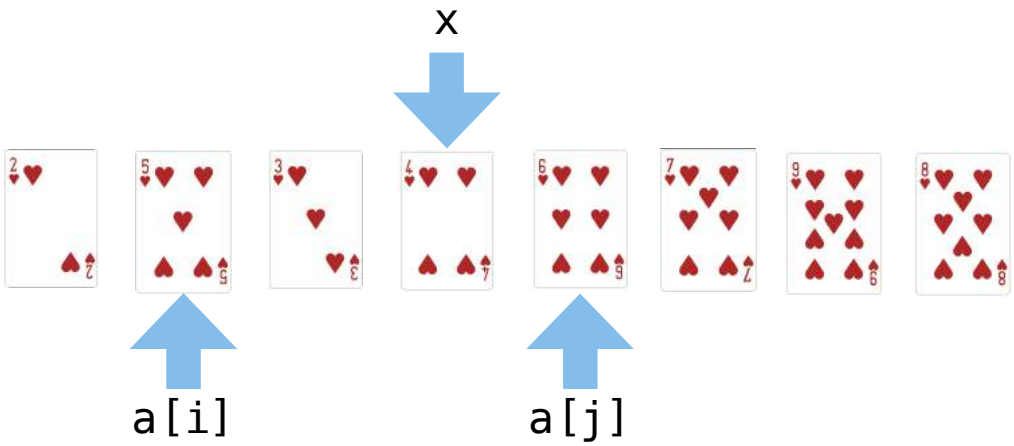
Movimentamos `j` para a esquerda até encontrar um elemento menor ou igual a `x`. O primeiro elemento nesta condição é o 2.



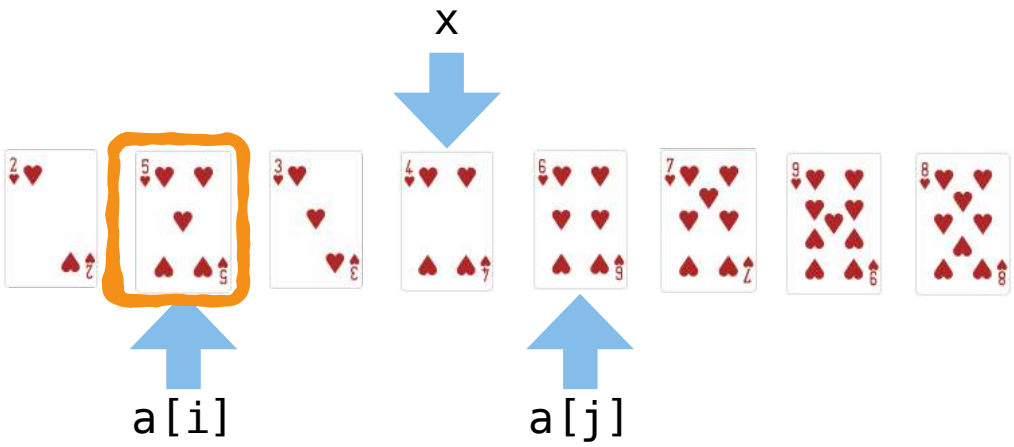
Trocamos a[i] com a[j].



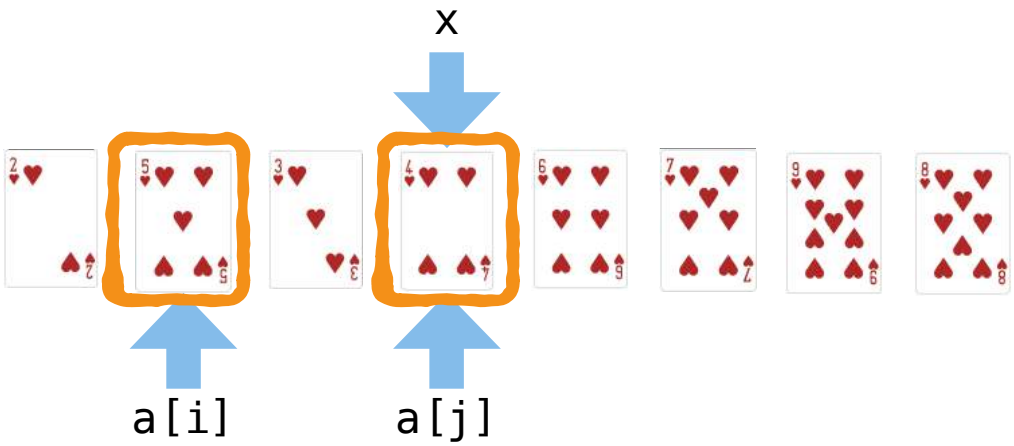
Deslocamos `i` e `j` e continuaremos o processo.



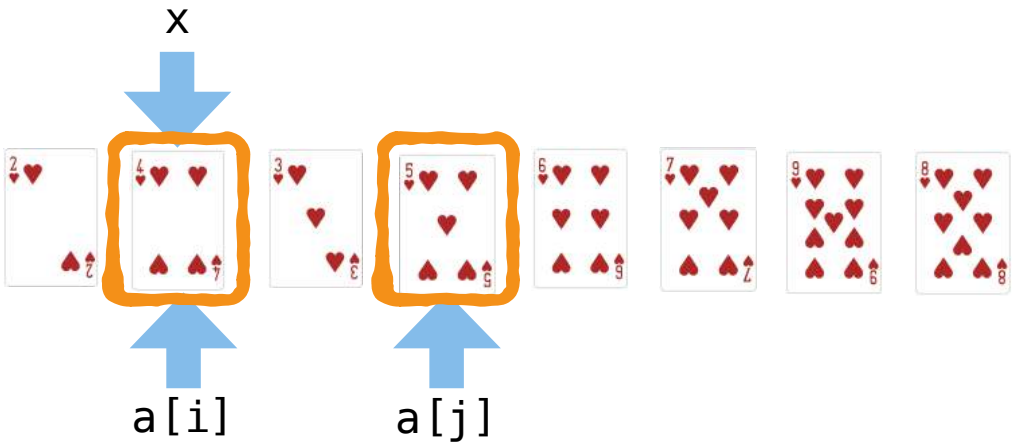
Deslocamos `i` até encontrar um elemento `a[i]` maior ou igual a `x`. Este elemento já é o 5.



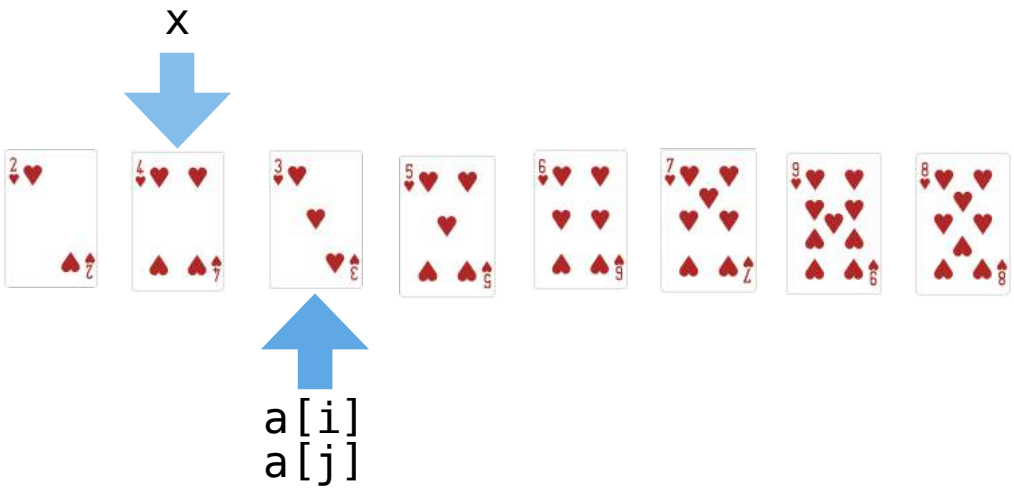
Deslocamos `j` até encontrar um elemento `a[j]` menor ou igual a `x`. Este elemento é o 4.



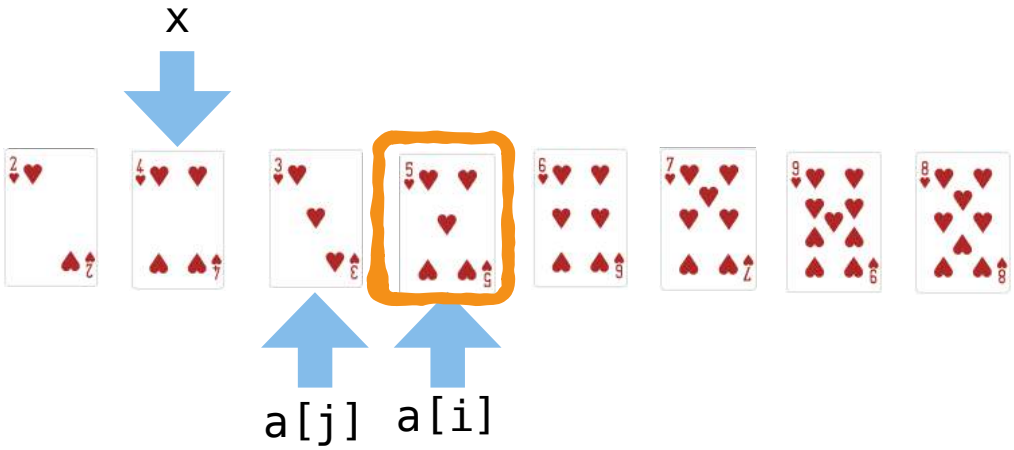
Trocamos `a[i]` com `a[j]`.



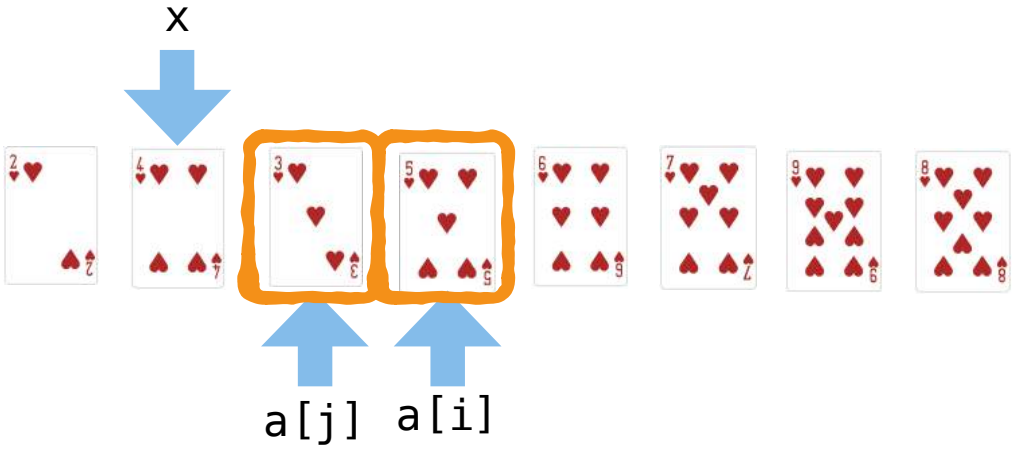
Deslocamos `i` e `j` e continuaremos o processo.



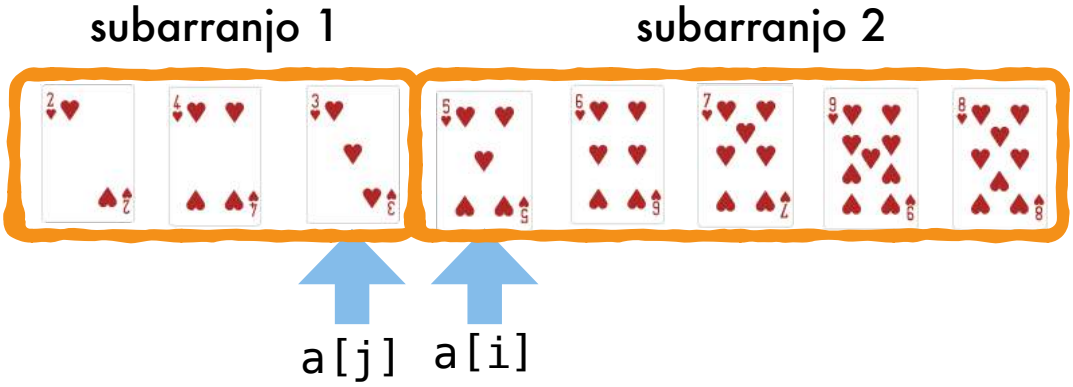
Deslocamos `i` até encontrar um elemento `a[i]` maior ou igual a `x`. Este elemento é o `5`.



Deslocamos `j` até encontrar um elemento `a[j]` menor ou igual a `x`. Este elemento já é o `3`.



Desta vez, porém, os valores de `i` e `j` se cruzaram e por isto não faremos a troca e encerramos a função. Os itens da esquerda até o elemento `a[j]` formam um subarranjo com elementos menores ou iguais ao pivô. Os itens da direita a partir do elemento a[i] formam um subarranjo com elementos maiores ou iguais ao pivô.



Em todos os exemplos até o momento, o pivô foi escolhido como o elemento na posição `(i+j)/2`.

A Figura 16.15 apresenta de forma resumida os passos deste processo de partição, que coloca os elementos menores à esquerda e elementos maiores à direita. A cada passo percorremos da esquerda para a direita e da direita para a esquerda procurando elementos que são maiores e menores que o pivô. Trocamos estes elementos até que os índices se cruzem. Veja que os dois subarranjos formados não são sempre do mesmo tamanho.

Passo	7	5	3	4	6	2	9	8
1	2	5	3	4	6	7	9	8
2	2	4	3	5	6	7	9	8
3	2	4	3	5	6	7	9	8

Movimentação Pivô

Figura 16.15: Exemplo de partição para ordenação com o método quicksort.

A Figura 16.16 apresenta de forma resumida os passos de um processo de ordenação com o quicksort. A cada passo, os arranjos desordenados são divididos em dois outros subarranjos, com elementos menores e maiores que o pivô. O processo se repete até que se formem arranjos de apenas 1 elemento, que são, por definição, ordenados. Estes arranjos ordenados são representados em amarelo.

Algoritmo de partição

Este é o algoritmo que particiona o arranjo `a[esq]...a[dir]` nos subarranjos `a[esq]...a[j]` e `a[i]...a[dir]`.

```
1 void particionar(int esq, int dir, int &i, int &j, int a[]) {
2     int x, temp;
3     i = esq;
```

Figura 16.16: Exemplo de ordenação com o quicksort.

```
4       j = dir;
5       x = a[(i + j) / 2];
6       do  {
7           while (x > a[i]){
8               ++i;
9           }
10          while (x < a[j]){
11              --j;
12          }
13          if (i <= j){
14              temp = a[i];
15              a[i] = a[j];
16              a[j] = temp;
17              ++i;  --j;
18          }
19      } while (i <= j);
20  }
```

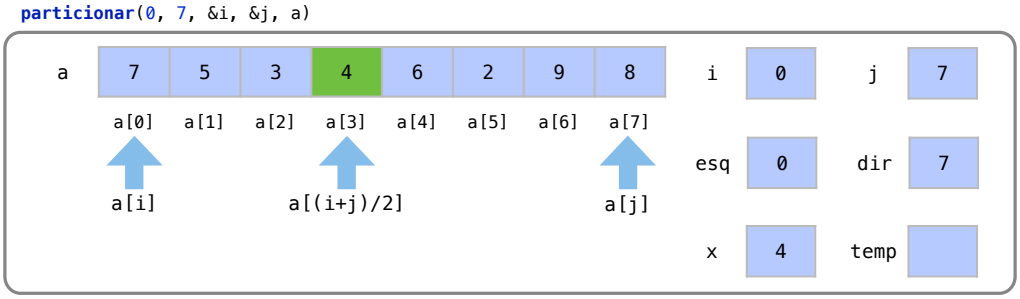
O protótipo da função recebe o arranjo `a` e as posições `esq` e `dir` entre as quais o arranjo deve ser particionado. Os índices `i` e `j` são passados por referência pois ao final da função deverão marcar onde começam e terminam os subarranjos criados.

O laço interno do algoritmo de partição, definido entre as linhas 7 e 18, é muito simples. Por isto, o algoritmo *quicksort* é tão rápido.

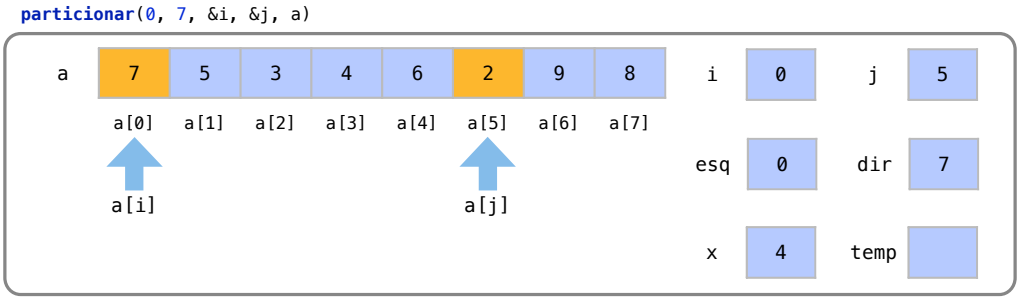
Na linha 1, a função de partição recebe vários parâmetros. Os parâmetros `esq` e `dir` indicam qual subarranjo de `a[]` queremos particionar. Os parâmetros `i` e `j` são passados por referência. Eles pertencem originalmente à função chamadora do quicksort. Ao fim do algoritmo, estas referências `i` e `j` dirão à função chamadora quais são os sub-arranjos particionados. Todo o arranjo `a[]` é também enviado à função. Como sabemos, arranjos são endereços na memória e por isto são apenas enviados por referência.

Na linha 2, criamos então a variável `x`, para guardar o elemento pivô, e uma variável temporária `temp` para fazer trocas. Nas linhas 3 e 4, os índices `i` e `j` são inicializados nos extremos do arranjo a ser particionado. O elemento do meio é então escolhido como pivô na linha 5. De acordo com a estratégia, outro pivô poderia ter sido escolhido.

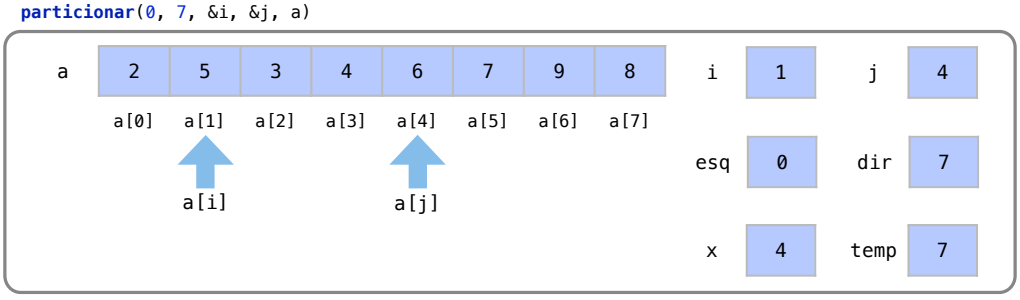
Neste exemplo, particionaremos todo um arranjo de `a[0]` a `a[7]`.



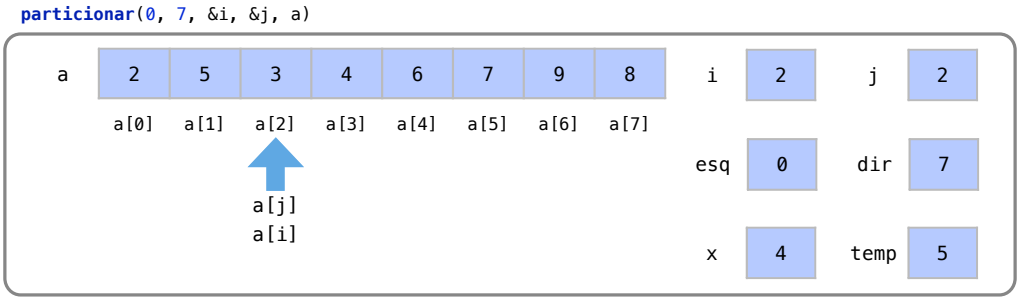
Neste grande laço entre as linhas 6 e 19, vamos fazer as trocas enquanto os índices `i` e `j` não tenham se cruzado. Nas linha 7 a 9, deslocamos o índice `i` até encontrar o primeiro elemento `a[i]` maior ou igual ao `x`. Nas linhas 10 a 12, deslocamos o índice j até encontrar o primeiro elemento `a[j]` menor ou igual a x.



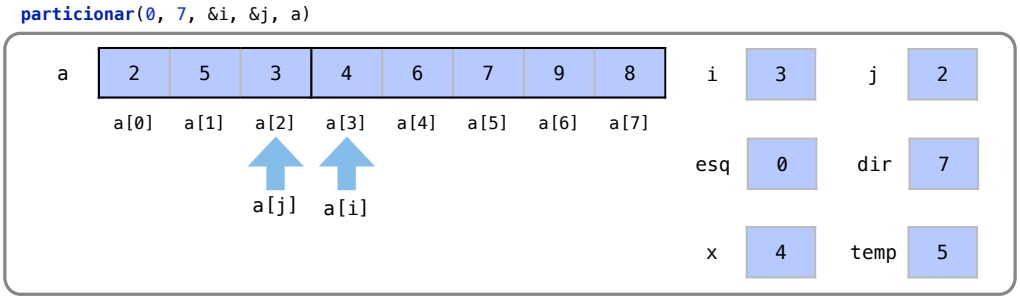
Na linha 13, como nesse deslocamento os índices não se cruzaram, as linhas 14 a 16, trocam `a[i]` com `a[j]`. Na linha 17, deslocamos `i` e `j` em mais uma posição.



Todo o laço externo se repete enquanto os índices não se cruzarem, encontramos os elementos trocados (linhas 7 a 12), trocamos os elementos (linhas 14 a 16) e deslocamos então, os índices `i` e `j` (linhas 17).



Os índices `i` e `j` estão na mesma posição mas ainda não se cruzaram. Nas linhas 7 a 9, o primeiro `a[i]` maior ou igual a `x` é `a[3]`. Nas linhas 10 a 12, o elemento `a[j]` já é menor ou igual a `x`. Como os índices já se cruzaram, a condição da linha 13 não é atendida e a troca não é feita.



Na linha 19, a segunda constatação de que os índices já se cruzaram encerra a função. Como resultado, temos dois subarranjos. Um de a[0] até a[2] (ou `a[esq]` a `a[j]`) e outro de a[3] até a[7] (`a[i]` a `a[dir]`).

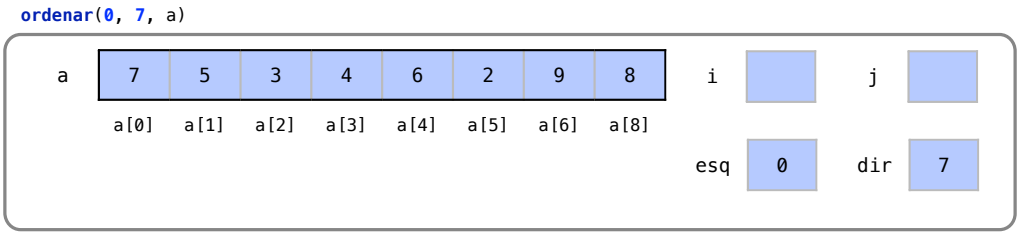
Como `i` e `j` foram passados por referência, a função chamadora tem acesso aos valores calculados de `i` e `j`.

Algoritmo de ordenação

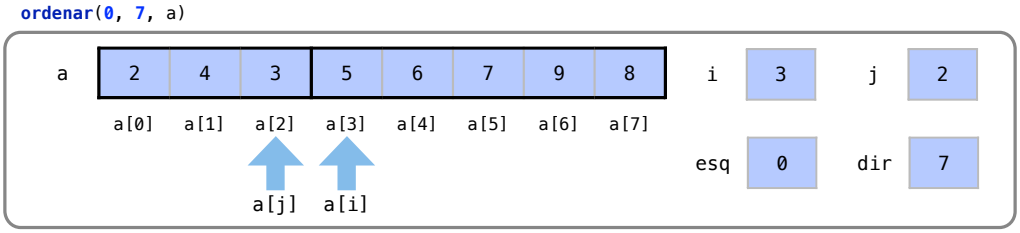
Tendo pronta a função para o processo de partição, o algoritmo de ordenação do quicksort se torna conceitualmente muito simples.

```
1 void ordenar(int esq, int dir, int a[]) {
2     int i, j;
3     particionar(esq, dir, i, j, a);
4     if (esq < j)
5         ordenar(esq, j, a);
6     if (i < dir)
7         ordenar(i, dir, a);
8 }
```

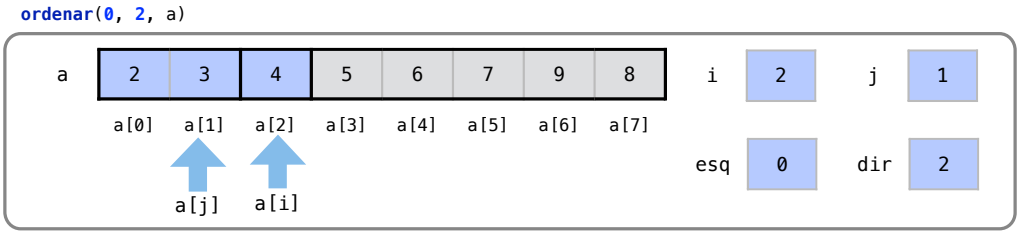
No protótipo da função, da linha 1, esta função ordena o arranjo `a[]` entre as posições `esq` e `dir`. Para ordenar todo arranjo, chamamos: `ordenar(0, 7, a)`. Na linha 2, criamos então os índices `i` e `j`, que indicarão quais são os sub-arranjos após a partição.



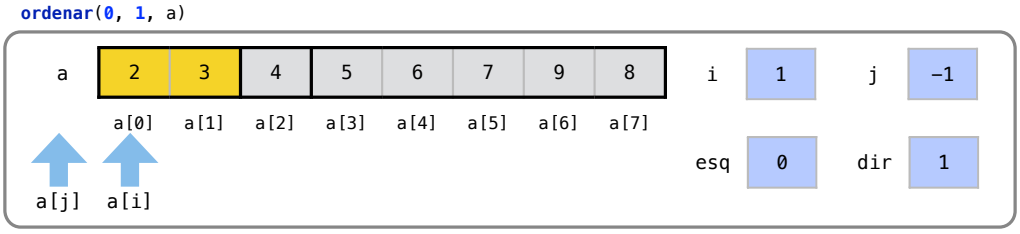
Na linha 3 particionamos o arranjo entre as posições `a[esq]` e `a[dir]`. Os índices `i` e `j` indicam onde termina o primeiro sub-arranjo e onde começa o segundo.



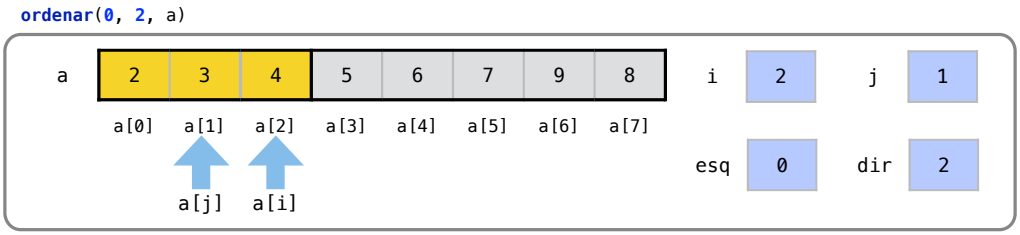
Se o valor de `j` atingir o valor de `esq`, como testado na linha 4, o sub-arranjo tem apenas um elemento e não precisa ser ordenado. Como o sub-arranjo de `a[0]` a `a[2]` tem mais de 1 elemento, usamos a própria função `quicksort(0, 2, a)` para ordená-lo. A função chamadora vai para a pilha de funções e executamos a função que ordenará `a[]` das posições 0 a 2. Os índices são criados e o sub-arranjo é particionado.



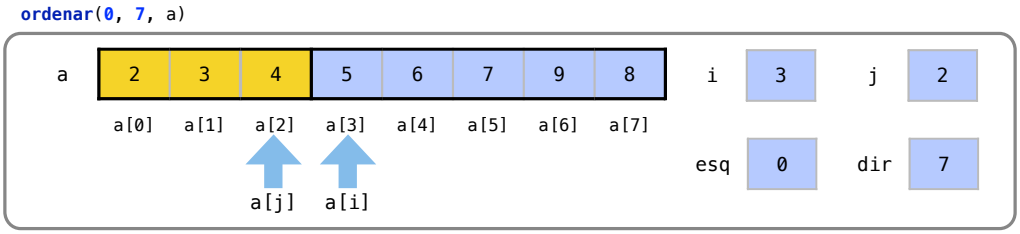
Na linha 5, esta função é jogada na pilha e chamamos a função de ordenação para o subarranjo de `a[esq]` até `a[j]`. Os índices são criados e o sub-arranjo é particionado. Como o subarranjo da esquerda não tem nenhum elemento, não executamos esta ordenação. E como o subarranjo da direita tem também apenas 1 elemento, damos este sub-arranjo como ordenado. Ao fim desta função, temos o sub-arranjo de 0 até 1 ordenado e voltaremos a com a função do topo da pilha.



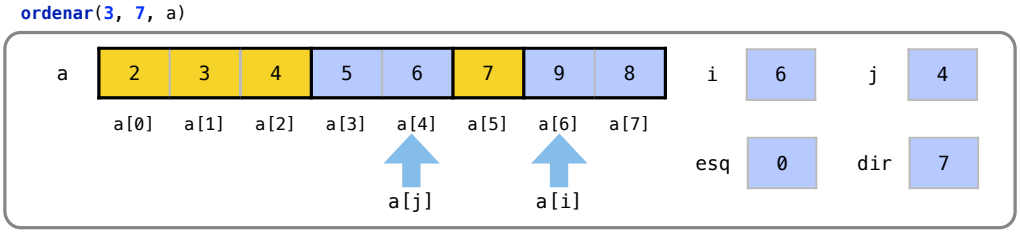
O próximo comando desta função `ordena(0, 2, a)` seria ordenar o sub-arranjo da direita, mas isto não é necessário pois este sub-arranjo tem apenas 1 elemento. Assim, encerramos a função pois sabemos que os elementos entre `a[0]` e `a[2]` estão ordenados.



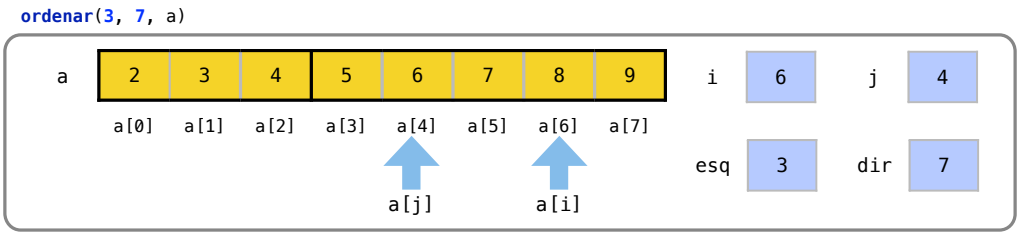
Voltamos para a função `ordena(0, 7, a)` do topo da pilha. Esta função, na linha 7, chama então a ordenação dos elementos de `a[3]` até `a[7]`.



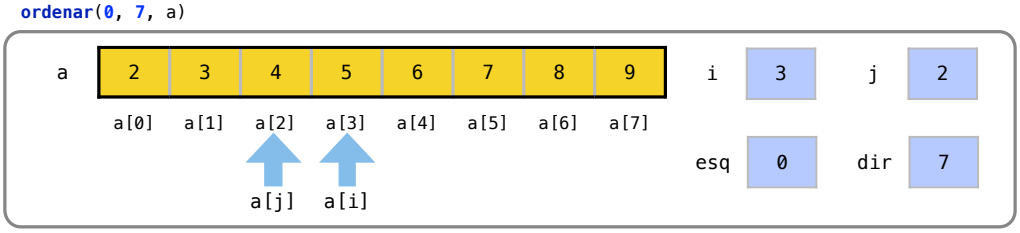
Criamos os índices e particionamos este subarranjo na linha 3 da função `ordena(3, 7, a)`. Entre os subarranjos de `a[3]` até `a[4]` e de `a[6]` até `a[7]` a função de partição deixou isolado o elemento `a[5]`, que por estar sozinho, está também, por definição, ordenado.



Com uma chamada recursiva, ordenamos `a[3]` e `a[4]`. Criamos os índices e particionamos. Como nenhum dos subarranjos têm mais de um elemento, damos todos os elementos estão ordenados. A função que sai da pilha ainda precisa ordenar os elementos de `a[6]` até `a[7]`. Esta função também leva a dois subarranjos de tamanho 1, que por isto já estão ordenados, que por isso já estão ordenados. Sendo assim, a função corrente sai da pilha.



A função original também sai da pilha, finalizando o método com o arranjo original ordenado.



Um inconveniente deste método é que precisamos passar para a função três parâmetros quando queremos ordenar todo o arranjo. Para não precisamos fazer isto, criamos um método auxiliar que é chamado para ordenar todo o arranjo, iniciando de a[0].

```
1 void quicksort(int a[], int n) {
2     ordenar(0, n-1, a);
3 }
```

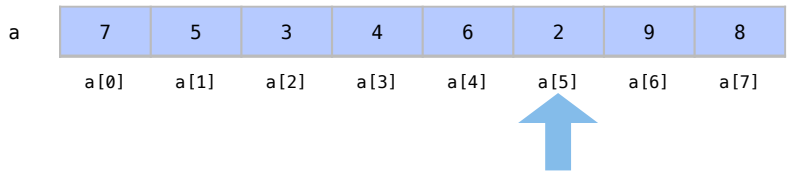
Deste modo, mantemos o padrão com os outros métodos de ordenação.

Análise

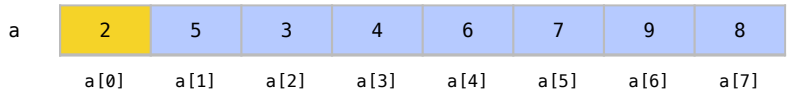
O *quicksort* é também um típico exemplo de algoritmo $O(n \log n)$, ou loglinear. Algoritmos típicos desta classe são os que dividem um problema em dois problemas menores e depois os juntam fazendo uma operação em cada um dos elementos.

Análise do pivô

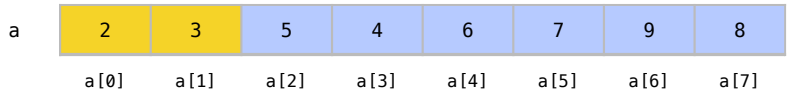
A escolha do pivô é um fator determinante no desempenho do algoritmo pois dependemos dele para realmente dividir o arranjo em dois subarranjos de tamanho similar. Imagine um algoritmo onde o pivô escolhido é sempre o menor elemento do conjunto de n elementos.



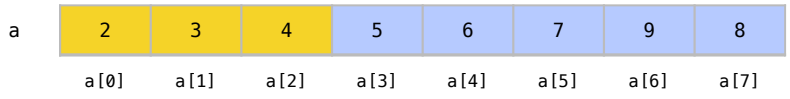
Ao dividir o arranjo em dois arranjos, teremos apenas 1 elemento no subarranjo da esquerda e teremos que ordenar o subarranjo da direita que ainda tem $n-1$ elementos.



Se em todos os passos escolhermos o pior pivô, não conseguiremos dividir o problema pela metade a cada passo.



Em 2 dois passos de partição ordenamos apenas 2 elementos.



Apesar de ser uma situação muito rara, com a pior escolha de pivô a cada passo, precisaríamos de n passos de partição para ordenar todo o arranjo. Como a cada passo estamos achando o menor elemento do arranjo e o colocando em sua posição correta, este pior caso do quicksort equivale exatamente à ordenação por seleção, que tem custo $O(n^2)$.

Algumas estratégias podem ser utilizadas para se escolher um melhor pivô, melhorando a eficiência do algoritmo e deixando o pior caso ainda menos provável.

O **pivô ótimo** para um passo de repartição seria a mediana dos valores no arranjo. Um pivô igual à mediana dividiria sempre o arranjo em dois subarranjos de tamanho idêntico. Porém, obter a mediana de um arranjo desordenado seria um processo caro (mais especificamente $O(n)$).

Escolher a cada passo um elemento aleatório do arranjo ou escolher três elementos aleatórios do arranjo e usar a mediana dos três como pivô são exemplos de estratégias comuns para se obter melhores pivôs.

Análise geral

O quicksort um método muito eficiente para ordenar dados. Devido à pilha de funções precisa de apenas um espaço extra muito pequeno, que é $O(\log n)$. Em termos práticos, este custo de memória é usualmente considerado muito próximo de $O(1)$.

O algoritmo requer apenas $O(n \log n)$ comparações em seu caso médio. Apesar da mesma ordem de complexidade média, é usualmente mais rápido que o Merge sort no caso médio.

O quicksort tem um pior caso $O(n^2)$ quando o pior pivô possível é sempre selecionado. A probabilidade de ocorrência deste pior caso é extremamente baixa e não chega a alterar o caso médio. Contudo, considerar este pior caso pode ser um fator importante em aplicações críticas, mesmo que com uma baixa probabilidade.

O quicksort também não é um algoritmo estável pois as trocas na fase de partição não consideram a ordem relativa dos elementos.

Na Tabela 16.4 temos um resumo do custo do quicksort em seus principais casos. Apesar de ter um pior caso $O(n \log n)$, o algoritmo tem custo $O(n \log n)$ no caso médio e no melhor caso.

Caso	Descrição	Custo
Pior caso	pivô sempre maior ou menor elemento	$O(n^2)$
Caso médio	pivô aleatório	$O(n \log n)$
Melhor caso	pivô sempre elemento mediano	$O(n \log n)$

Tabela 16.4: Custo do algoritmo quicksort em suas situações mais comuns

16.3.3 Exercícios

Exercício 16.7 Analise o código abaixo, que está incompleto.

- Veja que incluímos as bibliotecas `stdlib.h` e `time.h` neste código. Elas permitem geração de números aleatórios e medir tempo.
- Veja que criamos um arranjo dinamicamente com o tamanho pedido. (`new int[n]`)
- Veja que a função `rand()` gera números aleatórios que são inseridos no arranjo.
- Veja que temos algumas funções que não são utilizadas ainda.

```
1  #include <iostream>
2  // Geração de números aleatórios
3  #include <stdlib.h>
4  // Funções relacionadas a tempo
5  #include <time.h>
6  #include <chrono>
7
8  void intercala(int a[], int n);
9  void particionar(int esq, int dir, int &i, int &j, int a[])
       ;
10
11 using namespace std;
```

```
12
13 int main(){
14         // gerador de números aleatórios
15     srand(time(NULL));
16         // tamanho do arranjo
17     int n;
18     cout << "Digite␣o␣tamanho␣do␣arranjo:␣";
19     cin >> n;
20     // ponteiro para o arranjo na memória
21     int *v;
22     v = new int[n];
23
24     for (int i=0; i<n; ++i){
25         // numero entre 1 e n*3
26         v[i] = rand() % (n*3) + 1;
27     }
28
29     for (int k=0; k<n; ++k){
30         cout << v[k] << "\t";
31     }
32     cout << "<-␣Desordenado" << endl;
33
34     // Começa a medir tempo
35     chrono::high_resolution_clock::time_point inicio;
36     inicio = chrono::high_resolution_clock::now();
37
38     // Chame a sua função de ordenação aqui
39     // ordena(v,n);
40
41     // Termina de medir tempo
42     chrono::duration<double> duracao;
43     duracao = chrono::duration_cast<chrono::duration<double
          >>(chrono::high_resolution_clock::now() - inicio);
44
45     cout << "Ordenar␣o␣arranjo␣demorou␣" << duracao.count()
          << "␣segundos" << endl;
46
47     for (int k=0; k<n; ++k){
48         cout << v[k] << "\t";
49     }
50     cout << "<-␣Ordenado" << endl;
51
52     return 0;
53 }
54
55 void intercala(int a[], int n)
56 {
57         int *tmp = new int[n];
58     int meio = n / 2;
```

```
59          int i, j, k;
60      i = 0;
61          j = meio;
62          k = 0;
63          // Enquanto os índices i e j não tenham chegado ao
               fim de seus arranjos
64          while (i < meio && j < n){
65                  // colocamos o menor item entre a[i] e a[j]
                       no arranjo temporário
66                  if (a[i] < a[j]){
67                          tmp[k] = a[i];
68                          ++i;
69                  } else {
70                          tmp[k] = a[j];
71                          ++j;
72                  }
73                  ++k;
74          }
75          // se o índice i chegou ao fim de seu arranjo
               primeiro
76          if (i == meio) {
77                  // os outros elementos do segundo arranjo v
                       ão para o arranjo temporário
78                  while (j < n) {
79                          tmp[k] = a[j];
80                          ++j;
81                          ++k;
82                  }
83                  // se foi o índice j que chegou ao fim de
                       seu arranjo primeiro
84          } else {
85                  // os outros elementos do primeiro arranjo
                       vão para o arranjo temporário
86                  while (i < meio) {
87                          tmp[k] = a[i];
88                          ++i;
89                          ++k;
90                  }
91          }
92          // neste ponto, o arranjo temporário tem todos os
               elementos intercalados
93          // estes elementos são copiados de volta para o
               arranjo int a[]
94          for (i = 0; i < n; ++i){
95                          a[i] = tmp[i];
96          }
97          // o arranjo temporário pode então ser desalocado
               da memória
98          delete [] tmp;
```

```
99  }
100
101 void particionar(int esq, int dir, int &i, int &j, int a[])
        {
102     int x, temp;
103     i = esq;
104     j = dir;
105     x = a[(i + j) / 2];
106     do
107     {
108         while (x > a[i]){
109             ++i;
110         }
111         while (x < a[j]){
112             --j;
113         }
114         if (i <= j){
115             temp = a[i];
116             a[i] = a[j];
117             a[j] = temp;
118             ++i;
119             --j;
120         }
121     } while (i <= j);
122 }
```

■

Exercício 16.8 Veja a função `intercala`:

- Ela recebe um arranjo que começa no endereço `a` e tem tamanho `n`
- Se a primeira metade do arranjo (de `a[0]` até `a[n/2]`) e a segunda metade (de `a[n/2+1]` até `a[n-1]`) estiverem ordenadas, a função faz a intercalação dos dois e deixa o arranjo inteiro ordenado.

Utilize a função para fazer uma função que ordene um arranjo com o *merge sort*. ■

Exercício 16.9 Veja a função `particionar`:

- Ela recebe um arranjo e particiona os elementos entre `a[esq]` e `a[dir]`
- Após fazer o particionamento ela guarda as posições finais nos elementos `i` e `j`, que são passados por referência
- O particionamento faz com que os elementos menores fiquem entre `a[esq]` e `a[j]`. Os elementos maiores ficam entre `a[i]` e `a[dir]`.

Utilize esta função para fazer uma função que ordena um arranjo com o *quicksort*. ■

Exercício 16.10 Altere o código de modo que as duas ordenações sejam usadas e compare o tempo entre as duas. ■

Exercício 16.11 Coloque o código dentro de um `for`.

- Faça com que neste for, a variável contadora seja utilizada para testar os algoritmos em arranjos de vários tamanhos
- Altere o código de modo que ele imprima apenas o tamanho do arranjo, o tempo com uma ordenação, e o tempo com a outra
- Qual é a relação entre os algoritmos e o tamanho do arranjo?
- No geral, qual a relação entre os dois métodos de ordenação?

■

16.4 Comparando algoritmos de ordenação

Neste capítulo apresentamos dois algoritmos simples e dois algoritmos eficientes para ordenação de arranjos. Na Tabela 16.5 temos uma comparação do comportamento deste algoritmos. Algoritmos simples têm um custo médio $O(n^2)$ enquanto algoritmos eficientes têm um custo $O(n \log n)$. Existem, porém, vantagens e desvantagens em relação a estes algoritmos.

	Algoritmo			
	Simples		**Eficiente**	
	Seleção	**Inserção**	**Merge sort**	**Quicksort**
Comparações				
Caso médio	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Pior caso	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Movimentações				
Caso médio	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Pior caso	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Melhor caso	$O(n)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Memória				
Caso médio	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
Pior caso	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Melhor caso	$O(1)$	$O(1)$	$O(n)$	$O(\log n)$
Características				
Estabilidade	Não	Sim	Sim	Não
Adaptabilidade	Não	Sim	Não	Sim

Tabela 16.5: Comparação de custo em termos de número de comparações para algoritmos de ordenação. Os algoritmos simples têm um custo médio $O(n^2)$ enquanto algoritmos mais eficientes têm custo médio $O(n \log n)$.

Entre os métodos eficientes, veremos que o quicksort é ainda mais rápido na média do que o mergesort, apesar de terem mesma ordem de grandeza $O(n \log n)$.

Se os elementos já estiverem ordenados, a ordenação por inserção é sempre o método mais rápido. Este melhor caso da inserção, porém, é pouco provável na maioria das aplicações. Contudo, a ordenação por inserção é interessante para se adicionar alguns elementos a um arranjo já ordenado. Assim, estaremos próximos de seu melhor caso.

Apesar de ter a mesma ordem de grandeza $O(n^2)$ da ordenação por seleção, a ordenação por inserção é a mais lenta para arranjos em ordem decrescente. Mesmo assim, pode continuar melhor que a ordenação por seleção. Ainda apesar da mesma ordem de grandeza $O(n^2)$ no caso

médio, a ordenação por inserção tende a ser melhor que a ordenação por seleção no caso médio com arranjos aleatórios.

Nos casos gerais, a inserção é um método interessante apenas para arranjos bastante pequenos, com menos de 20 elementos.

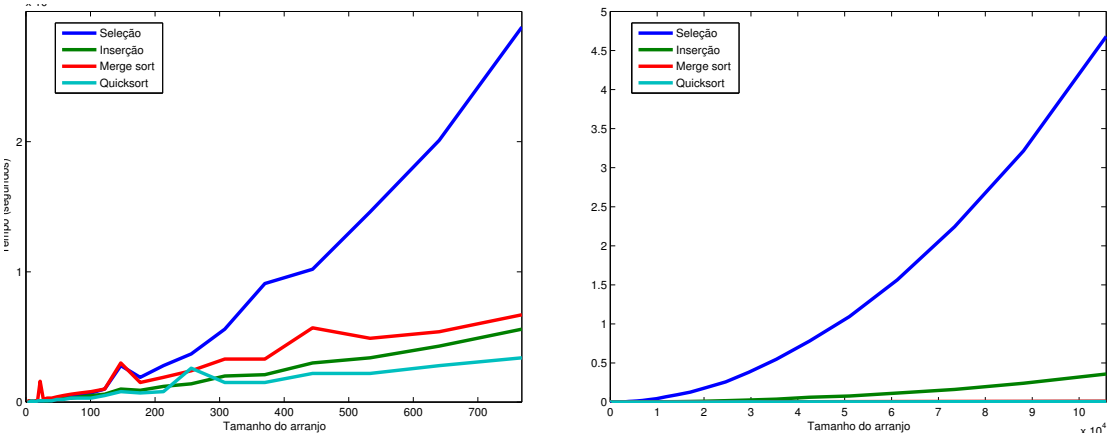
Se os registros são grandes, as movimentações se tornam caras e a ordenação por seleção se torna interessante por fazer apenas $O(n)$ movimentações, caso não haja tantos elementos. Porém, outra maneira de se evitar movimentos é usar ordenação indireta, ou seja, ordenamos ponteiros para elementos e só fazemos $O(n)$ movimentações no final. Esta estratégia, porém, utiliza uma memória extra $O(n)$ para armazenar estes ponteiros.

No caso médio, quicksort é a melhor opção para a maioria da situações. Apesar de no geral ser o melhor método entre os apresentados, ele não garante estabilidade. Suas chamadas recursivas também demandam um pouco de memória extra.

Apesar de haver um pior caso no quicksort, sua ocorrência é quase impossível para qualquer estratégia razoável de seleção de pivôs. Quando os arranjos já estão quase ordenados, usar o elemento do meio como pivô melhora muito o desempenho do algoritmo. Porém, nem sempre usar o pivô do meio é uma boa estratégia.

Os métodos de inserção e quicksort podem ser combinados. A inserção pode ordenar subarranjos pequenos do quicksort, o que costuma melhorar muito seu desempenho médio.

A Figura 16.17 apresenta o tempo gasto por cada um dos quatro algoritmos apresentados para ordenar arranjos de diferentes tamanhos.

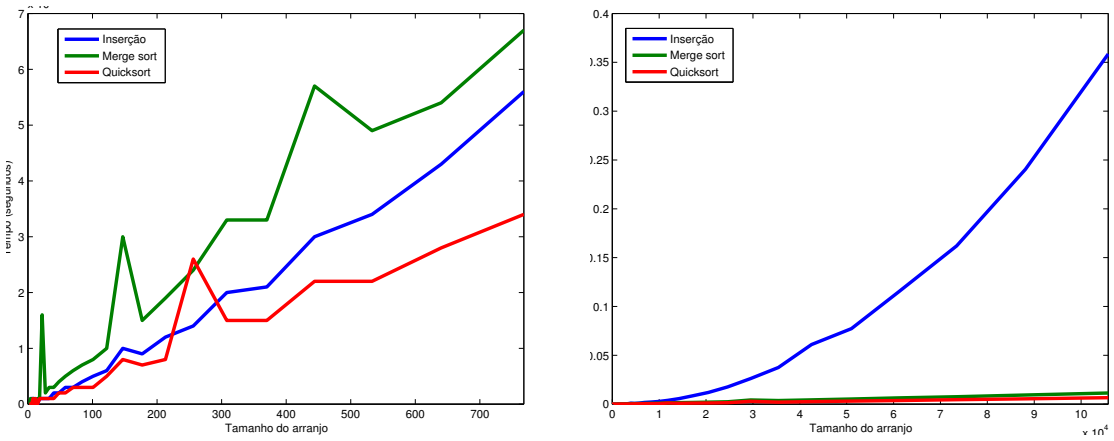


(a) Arranjos menores (b) Arranjos maiores

Figura 16.17: Tempo necessário pelos algoritmos simples e eficientes para se ordenar um arranjo com ordem inicial aleatória (caso médio).

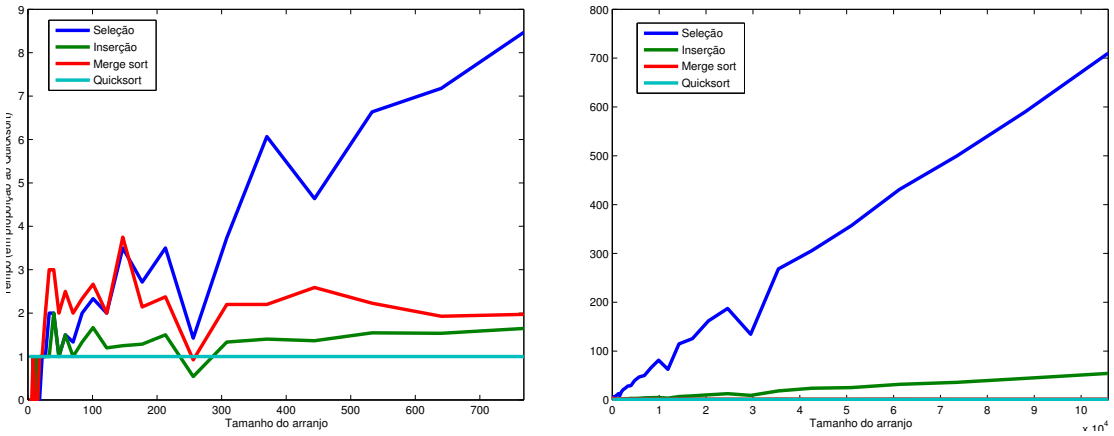
O tempo gasto pela ordenação por seleção é tão maior que é difícil distinguir a diferença de tempo entre os outros algoritmos para arranjos grandes. Assim, a Figura 16.18 apresenta o tempo gasto pelos métodos eficientes de ordenação em relação ao método de ordenação por inserção. Mesmo que não consideremos a ordenação por seleção, há uma grande diferença entre a ordenação por inserção e os métodos eficientes de ordenação. Para arranjos onde $n < 800$, porém, a ordenação por inserção pode ser mais eficiente que o merge sort. Para arranjos onde $n < 20$, o custo da ordenação por inserção pode até se confundir com o custo de um quicksort.

Como esperado, os métodos simples de ordenação são muito mais lentos. A Figura 16.22 apresenta a proporção de custo entre todos os algoritmos. O algoritmo de ordenação por seleção chega a ser até 700 vezes mais lento que algoritmos eficientes em arranjos grandes. Esta diferença se tornaria ainda maior a medida que aumentamos o tamanho dos arranjos.



(a) Arranjos menores (b) Arranjos maiores

Figura 16.18: Tempo necessário pelos algoritmos eficientes e pela ordenação por inserção para se ordenar um arranjo com ordem inicial aleatória (caso médio).



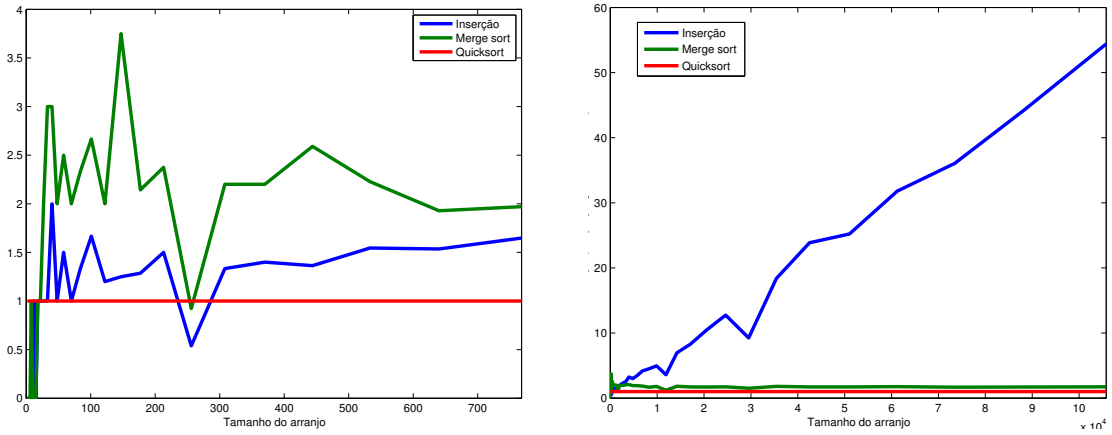
(a) Arranjos menores (b) Arranjos maiores

Figura 16.19: Proporção do tempo necessário para se ordenar um arranjo com ordem inicial aleatória (caso médio).

A Figura 16.20 apresenta a proporção de custo entre os algoritmos eficientes e a ordenação por inserção. Mesmo o algoritmo de ordenação por inserção chega a ser até 55 vezes mais lento que algoritmos eficientes em arranjos grandes. Quanto maior o valor de n, piores serão os métodos $O(n^2)$ (de complexidade quadrática) em relação aos métodos $O(n \log n)$ (de complexidade loglinear).

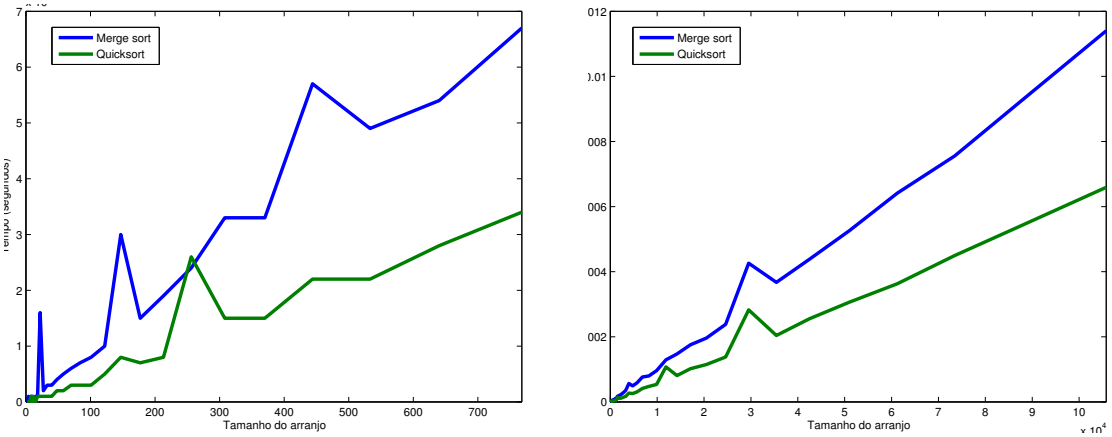
Como sabemos logicamente que os algoritmos $O(n^2)$ terão pior desempenho que os algoritmos $O(n \log n)$, a Figura 16.21 apresenta uma comparação de tempo entre os algoritmos merge sort e quicksort. Em comparação direta entre os métodos, o quicksort é usualmente mais eficiente que o merge sort para arranjos aleatórios.

A Figura 16.22 apresenta a proporção de tempo gasto entre os métodos eficientes de ordenação. Vemos como a proporção de eficiência entre os dois métodos se mantém constante para diferentes tamanhos arranjo, já que os dois métodos são $O(n \log n)$ e têm uma função de custo que cresce na mesma ordem.



(a) Arranjos menores (b) Arranjos maiores

Figura 16.20: Proporção de tempo necessário pelos algoritmos eficientes e pela ordenação por inserção para se ordenar um arranjo com ordem inicial aleatória (caso médio).



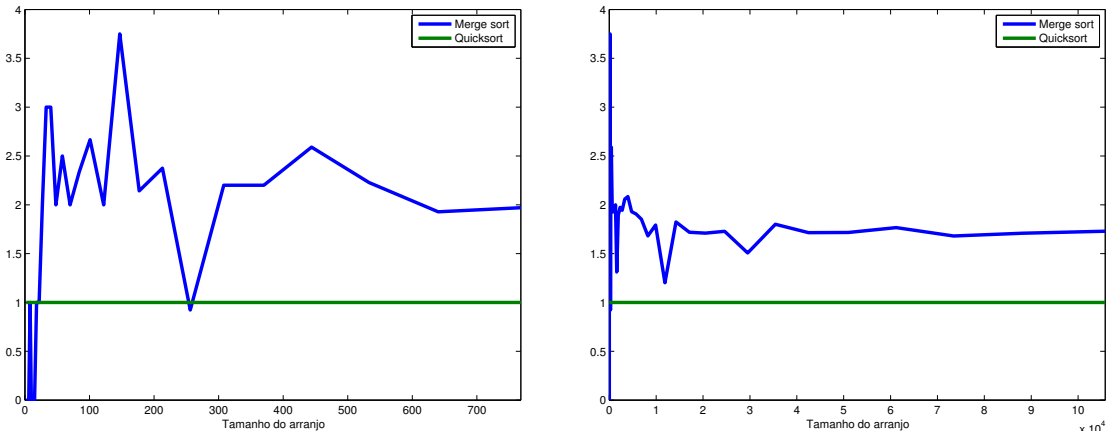
(a) Arranjos menores (b) Arranjos maiores

Figura 16.21: Tempo necessário para se ordenar um arranjo com ordem inicial aleatória (caso médio).

16.5 Conclusão

Os métodos de ordenação mostram como é possível obter variados tipos de vantagens e desvantagens entre algoritmos para a mesma classe de problemas, mesmo que este problema seja simples. Mesmo com o limite inferior de $O(n \log n)$, há vários fatores a se considerar em cada algoritmo como pior caso, caso médio, estabilidade, adaptabilidade, memória extra e operações de atribuição.

Apesar dessa introdução ao tópico de ordenação ter sido limitada, as comparações entre algoritmos mostram a importância de conceitos como: notação O, divisão e conquista, estruturas de dados, análise de melhor caso, pior caso e caso médio, e, por fim, conflito entre tempo e memória. Todas estas lições são fundamentais para quaisquer outros algoritmos.



(a) Arranjos menores (b) Arranjos maiores

Figura 16.22: Tempo necessário para se ordenar um arranjo com ordem inicial aleatória (caso médio).

16.6 Outros algoritmos de ordenação

Além dos métodos apresentados aqui, existem vários outros métodos de ordenação, cada um com suas vantagens e desvantagens. Listamos aqui alguns dos algoritmos mais famosos de ordenação:

Heapsort Melhora sobre ordenação por seleção com estruturas de dados mais eficientes. No algoritmo de ordenação por seleção, o menor elemento do arranjo é selecionado a cada passo. No heapsort, uma estrutura chamada *heap* é utilizada para que este menor elemento seja encontrado de maneira mais eficiente a cada passo.

Introsort Híbrido entre quicksort e Heapsort. O quicksort é utilizado até que o certo nível seja atingido na pilha de funções chamadas recursivamente. A partir deste ponto, o heapsort ordena os subarranjos. A combinação faz com que não exista mais o pior caso $O(n^2)$ do quicksort quando um pivô ruim é escolhido repetidas vezes. Este pior caso, quando se comparam os algoritmos, é uma das poucas desvantagens do quicksort. Por isto, tem sido um dos algoritmos mais utilizados atualmente para funções gerais de ordenação de arranjos.

In-place merge sort Variação do merge sort onde não há o gasto extra de memória $O(n)$. Isto é feito através de um processo mais complexo de intercalação de elementos. Isso, porém, leva o algoritmo a ter um custo computacional ainda mais alto. Um motivo para utilizar o merge sort mesmo com seu custo mais alto em relação a outros algoritmos eficientes é que ele tem a capacidade de fazer uma ordenação estável.

Bubble sort Famosos método de implementação simples. O bubble sort, ou ordenação por bolha, é muito apresentado em livros didáticos junto aos métodos de ordenação por seleção e inserção como métodos simples, porém pouco eficientes, de ordenação. O método simplesmente percorre os elementos várias vezes fazendo com que elementos subjacentes sejam trocados caso estejam em posição relativa incorreta. Assim como a ordenação por inserção, este método tem custo $O(n^2)$ no caso médio e pior caso e, no melhor caso, o método tem custo $O(n)$.

Shellsort Generalização da ordenação por inserção muito apresentada em livros didáticos. Subarranjos com elementos a uma distância grande entre si são ordenados primeiro. A distância entre os elementos ordenados é reduzida a cada passo até que estejamos ordenando os elementos com distância 1 entre si. A ordenação por inserção é então um shellsort onde elementos com distância 1 entre si já estão sendo ordenados no primeiro

passo. A complexidade média do shellsort depende da sequência de distâncias consideradas mas, para todas as sequências possíveis, ele tem um pior caso $O(n^2)$ e um melhor caso $O(n \log^2 n)$.

Todos os métodos apresentados neste capítulo são chamados métodos de ordenação por comparação. Isto porque os elementos são comparados e movimentados até que terminem em suas posições corretas. Os métodos mais eficientes por comparação têm custo $O(n \log n)$.

Além dos métodos por comparação, existem os métodos de ordenação por contagem. Nestes métodos contamos o número de ocorrências de cada elemento em um arranjo secundário e recolocamos os elementos no arranjo original. A **ordenação por contagem simples** nos possibilita ordenar um arranjo com custo $O(n+r)$, onde r é a faixa de valores possível para os elementos do arranjo.

Entre os métodos de ordenação de contagem, o **radix sort** utiliza a ordenação por contagem para cada casa dígito de um elemento. Com uma ordenação estável, cada dígito pode ser ordenado separadamente com um custo $O(n)$. Assim, esta é um estratégia de ordenação com custo $O(nk)$ no caso médio, onde k é o número de dígitos dos elementos. À medida que $k < \log n$, o radix sort se torna mais eficiente que os métodos de ordenação por comparação.

III Estruturas de Dados

17. Templates

São comuns situações em que operações muito parecidas podem ser executadas em diferentes tipos de dados. Isto é especialmente verdade entre os tipos de dados aritméticos. Por exemplo, o algoritmo para encontrar o máximo ou mínimo em um conjunto de dados do tipo `int` é muito similar ao algoritmo para os tipos de dado `double`, `short`, `float` ou `long`. Considere o código abaixo que encontra o maior elemento entre 3 números inteiros:

```
1  int maximo(int a, int b, int c){
2      int max = a;
3      if (b > max){
4          max = b;
5      }
6      if (c > max){
7          max = c;
8      }
9      return max;
10 }
```

Veja agora o código que faz o mesmo para dados do tipo `double`.

```
1  double maximo(double a, double b, double c){
2      double max = a;
3      if (b > max){
4          max = b;
5      }
6      if (c > max){
7          max = c;
8      }
9      return max;
10 }
```

Com exceção do tipo de dado, os códigos são idênticos. Não é difícil perceber que o mesmo código funcionaria para qualquer outro tipo de dado, como `float` ou até mesmo `char`. Precisaríamos apenas alterar o tipo. Já vimos neste curso vários algoritmos que poderiam ser generalizados para qualquer tipo de dado. Os algoritmos de ordenação, por exemplo, são os mesmos para qualquer tipo de dado.

Em C++, os templates nos permitem definir uma função que funciona para mais de um tipo de dado como no exemplo abaixo:

```
1  template <class T>
2  T maximo(T a, T b, T c){
3      T max = a;
4      if (b > max){
5          max = b;
6      }
7      if (c > max){
8          max = c;
9      }
10     return max;
11 }
```

Definir um template equivale a ter definido a função para qualquer tipo de dado possível. Por exemplo, se a função chamadora tem um comando `maximo(4,2,6)`, o compilador gera uma versão da função `maximo` que aceita dados do tipo `int`. Se a função chamadora tem um comando `maximo(4.4,2.4,6.2)`, o compilador gera uma versão da função `maximo` que aceita dados do tipo `double`. Isso é feito substituindo-se o tipo de dado do template chamado de `T` com um tipo de dado compatível com a função que precisamos.

Os templates, assim, representam uma coleção de definições para qualquer tipo de dados. Essas definições são geradas sob demanda pelo compilador de acordo com as chamadas de função que aparecem no código. Deste modo, eles são um recurso poderoso de reutilização de código em C++ pois, com eles, conseguimos fazer o que chamamos de *programação genérica*. Retornaremos a este tópico mais vezes neste livro.

17.1 Standard Template Library

A contribuição mais relevante e mais representativa de programação genérica em C++ é a *Standard Template Library* (STL) ou *Biblioteca Padrão de Templates*, em português. A STL é uma parte do padrão C++ aprovada em 1997/1998 e estende o núcleo do C++ fornecendo componentes gerais.

Como alguns exemplos, a STL fornece o tipo de dado `string`, diferentes estruturas para armazenamento de dados, classes para entrada/saída e algoritmos utilizados frequentemente por programadores. A Tabela 17.1 apresenta as três partes lógicas que compõem a STL.

Parte lógica	Descrição
Containers	Gerenciam coleções de objetos
Iteradores	Percorrem elementos das coleções de objetos
Algoritmos	Processam elementos da coleção

Tabela 17.1: Partes Lógicas da Standard Template Library

Para que os recursos sejam úteis a programadores de maneira genérica, todos os recursos da STL são baseados em templates. Neste Capítulo, estudaremos todas as partes lógicas da STL.

18. Containers Sequenciais

18.1 Containers

As estruturas de dados se referem a como armazenamos e organizamos dados de uma aplicação. Elas são representadas com a STL através de containers. Como a analogia sugere, dentro de um container, podemos guardar várias variáveis de qualquer tipo. Isto está representado na Figura 18.1.

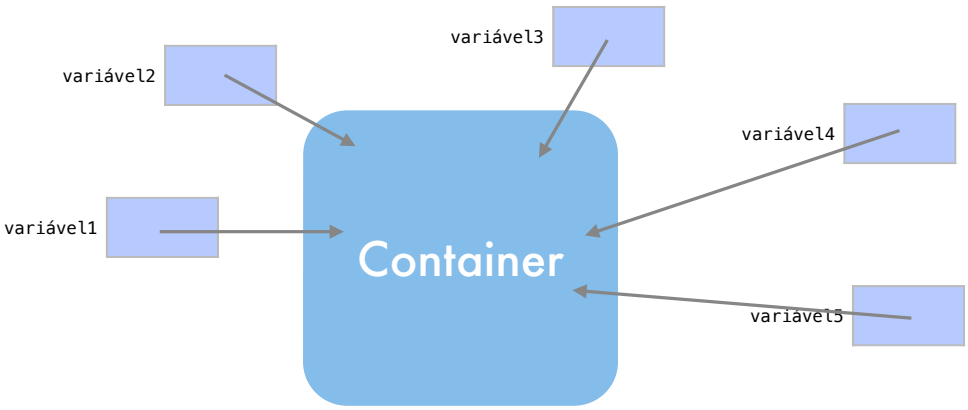


Figura 18.1: Os containers guardam dados utilizando diferentes estruturas de dados.

Porém, cada container tem suas próprias características pois utiliza internamente diferentes estruturas de dados para guardar estas variáveis. Ou seja, cada tipo de container possui uma estratégia diferente para organização interna de seus dados. Esta estratégia é controlada pela STL, de acordo com o tipo de containers.

Por isso, cada container possui vantagens e desvantagens em diferentes situações. A Tabela 18.1 apresenta os dois tipos básicos de containers.

Os containers são representados através de objetos que utilizam templates. Os templates, por sua vez, permitem que o container seja utilizado em qualquer tipo de dado. Estudaremos objetos

Containers	Representa	Característica
Sequenciais	Listas	Cada elemento tem uma posição específica
Associativos	Conjuntos	A posição interna de um elemento depende de seu valor

Tabela 18.1: Tipos Básicos de Container

em um capítulo futuro deste livro. A grosso modo, por ora, objetos são recursos similares aos tipos de dados definidos com `struct`, que aprendemos na Seção 10. Porém, além de ter suas próprias variáveis, os objetos possuem suas próprias funções.

Supondo um container chamado `c`, a Tabela 18.2 apresenta funções importantes que estão disponíveis em todos os containers. Algumas funções retornam dados lógicos, números inteiros ou iteradores. Os iteradores serão utilizados para acessar os elementos de um container, como veremos mais adiante. Outras funções apenas alteram a estrutura do container, sem fazer retorno algum.

Função	Retorna
`c.empty()`	Se o container está vazio
`c.size()`	Tamanho ou número de elementos
`c.begin()`	Iterador para primeiro elemento
`c.end()`	Iterador para fim do container
`<, >, >=, <=, ==, !=`	Operadores de comparação
`c.rbegin()`	Iterador reverso para início
`c.rend()`	Iterador reverso para o fim
Função	**Efeito**
`c.erase(i)`	Apaga o elemento `i`
`c.clear()`	Limpa o container
`c.swap(c2)`	Troca elementos com `c2`

Tabela 18.2: Funções-membro comuns aos containers. Algumas funções retornam valores e outras alteram o container.

18.2 Containers sequenciais

Os containers sequenciais são aqueles utilizados para representar sequências de elementos. Em uma sequência de elementos, cada elemento deve ter uma posição específica. Estudaremos os containers sequenciais `vector`, `deque` e `list`.

18.3 Vector

O `vector` é talvez o container de sequência mais utilizado. Esse container tem funções para acessar, remover ou inserir elementos no fim da sequência de elementos de maneira eficiente. Para utilizarmos este container, é necessária a inclusão do cabeçalho `<vector>` no início de nosso programa.

18.3.1 Utilização

Imagine a sequência de elementos abaixo.

Imagine que o vector com esta sequência se chama `c`. Novos elementos podem ser inseridos ao final de `c` com a função `push_back`.

```
1 c.push_back(5);
2 c.push_back(1);
3 c.push_back(9);
```

Elementos podem ser removidos do final de `c` com a função `pop_back`.

```
1 c.pop_back();
2 c.pop_back();
```

O primeiro e último elementos de `c` podem ser acessados com as funções `front` e `back`.

```
1 cout << c.front() << "␣" << c.back() << endl;
```


```
4 5
```

18.3.2 Como funciona

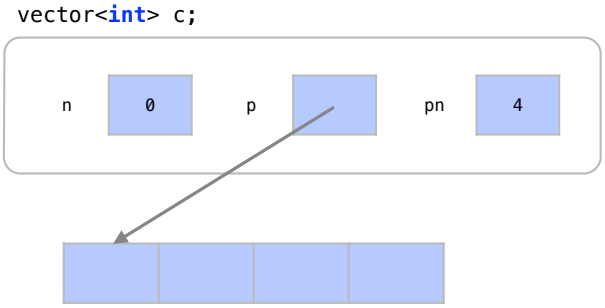
Apesar de todas as simplificações oferecidas pelos templates, os containers são complexos internamente. Todo container precisa ser representado internamente através de recursos que já conhecemos, como ponteiros e arranjos. Este recursos são organizados para formarem uma estrutura de dados que organiza os elementos.

A estrutura exata para representar um container depende do compilador mas, de acordo com o container, sabemos que há algumas estruturas básicas utilizadas por todos os compiladores. É esta estrutura comum que precisamos conhecer para entender as vantagens e desvantagens de cada container.

Com este comando, podemos criar um `vector` vazio.

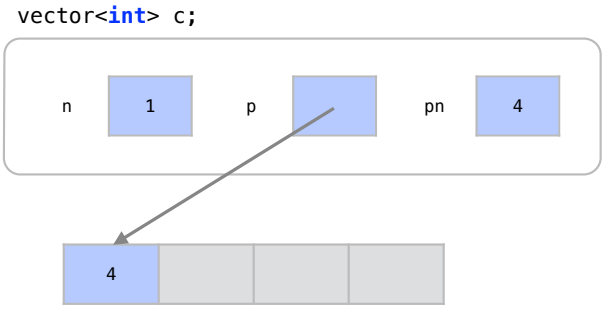
```
1 vector<int> c;
```

Internamente, o `vector` terá usualmente um ponteiro um dois números inteiros, chamados aqui de `n`, `p` e `pn`. Um número inteiro `n` guarda o número de elementos no container enquanto o ponteiro `pn` aponta para um arranjo de elementos na memória. Note como o arranjo não está no escopo do `vector`, pois ele é alocado dinamicamente. O segundo número inteiro `pn` guarda o tamanho deste arranjo.



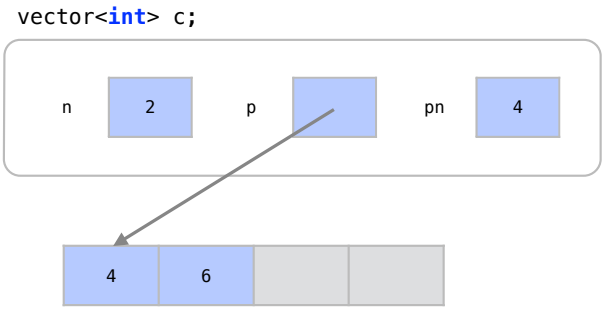
Quando inserimos um elemento no fim do container, este é inserido na posição `n` do arranjo e `n` é incrementado.

```
1 c.push_back(4);
```



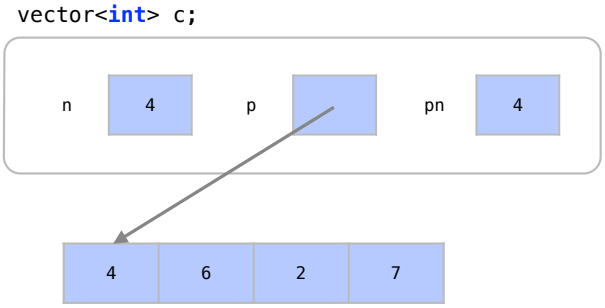
Como o arranjo já tem espaço para mais que `n` elementos, mais elementos podem ser inseridos sem necessidade de se alocar mais memória. Sempre que não precisarmos de mais espaço para um elemento, este pode ser inserido no arranjo com custo constante $O(1)$. As posições representadas em cinza não estão sendo utilizadas até o momento.

```
1 c.push_back(6);
```



Eventualmente, o número de elementos será igual ao tamanho do arranjo. Neste caso, não podemos colocar mais elementos sem alocar mais memória.

```
1 c.push_back(2);
2 c.push_back(7);
```

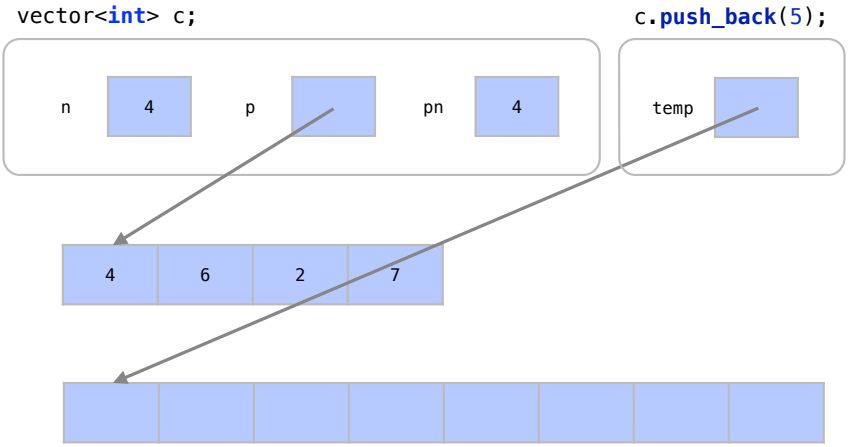


Queremos agora inserir o elemento 5 ao container.

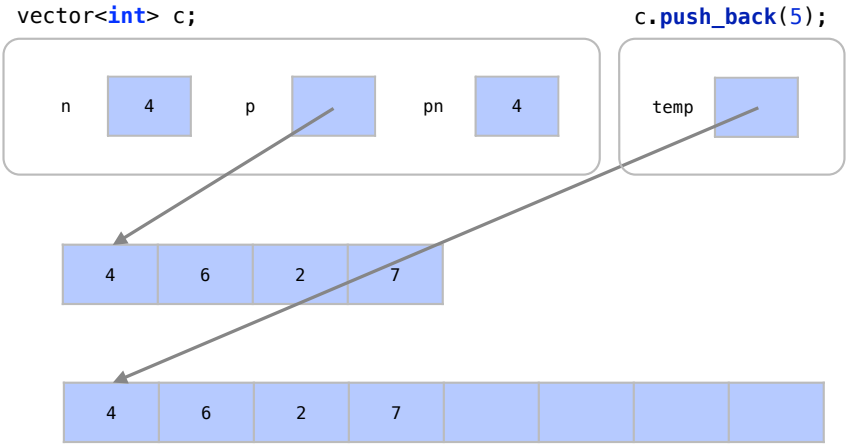
```
1  c.push_back(5);
```

Como não há mais espaço para elementos no arranjo, este procedimento ocorre em quatro passos.

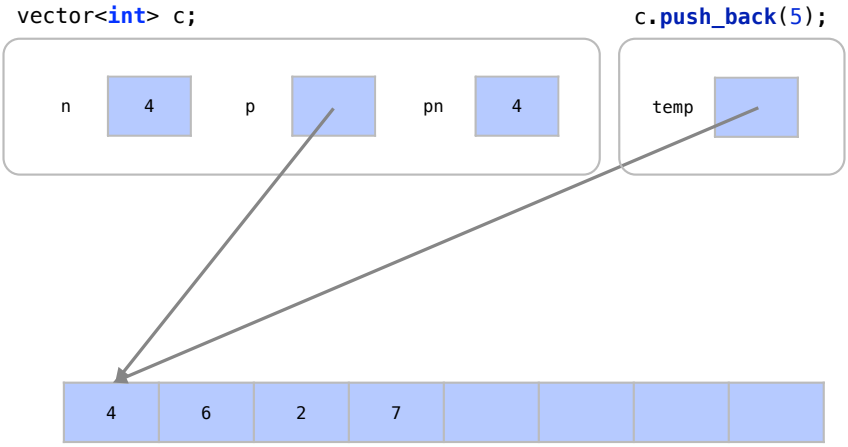
1) Alocamos espaço para um arranjo com o dobro de elementos e fazemos um ponteiro temporário apontar para ele.



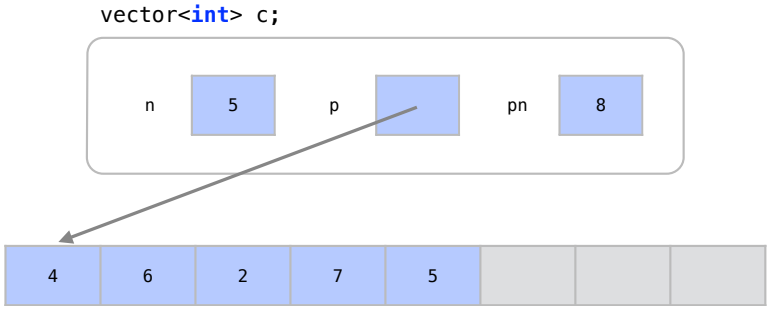
2) Copiamos todos os elementos do arranjo apontado por `p` para o arranjo apontado por `temp`. Naturalmente, para n elementos, este passo tem um custo $O(n)$.



3) Desalocamos a memória apontada por `p` e fazemos com que `p` aponte para o mesmo arranjo de `temp`.



4) Atualizamos o valor de `pn`, inserimos o valor `5` normalmente, com custo $O(1)$, e incrementamos `n`. O ponteiro `temp` deixa de existir por ter apenas escopo de função.

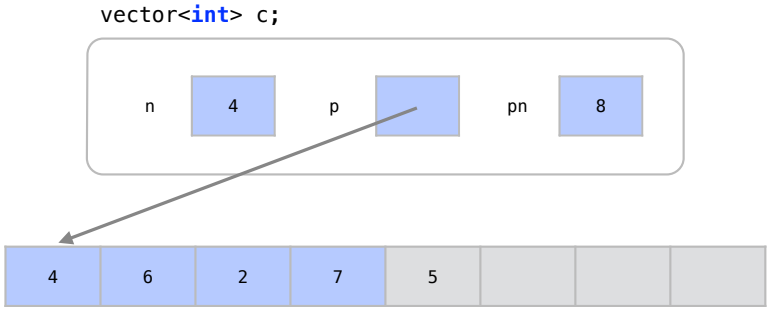


Se considerarmos todos os passos, toda a operação de inserção teve custo O(n) quando houve necessidade de se alocar mais memória. Este custo, porém, se amortizado entre várias inserções tem um custo médio de O(1). O custo médio ocorre pois só fazemos uma alocação de custo $O(n)$ a cada n inserções de custo $O(1)$. Por isto a alocação prévia de memória é tão importante.

Existe inclusive a função `capacity()` que retorna quantos elementos ainda podem ser colocados em um `vector` sem se alocar mais memória.

De maneira similar, a remoção de um elemento no fim da sequência é feita com um custo constante $O(1)$.

```
1 c.pop_back();
```



Com custo $O(1)$, a função para remover o último elemento do container é realizada apenas pela atualização do valor de `n`, o fazendo indicar que apenas os 4 primeiros elementos devem ser considerados.

18.4 Deque

O container `deque` é uma abreviação para *double ended queue* ("fila com duas pontas"). É um container indicado para sequências que crescem nas duas direções pois tem a capacidade de inserção rápida de elementos no início e no final da sequência. Para utilizarmos um `deque`, é necessário incluir o cabeçalho `<deque>` no início de nosso programa.

18.4.1 Utilização

Imagine a sequência de elementos abaixo.

Imagine que o `deque` que contém esta sequência se chama `c`. Da mesma maneira que fizemos com um `vector`, novos elementos podem ser inseridos ao final de um `deque` `c` com a função `push_back`.

```
1 c.push_back(5);
2 c.push_back(1);
3 c.push_back(9);
```

Diferentemente de um `vector`, elementos podem ser removidos do início de `c` com a função `pop_front`.

```
1 c.pop_front();
2 c.pop_front();
3 c.pop_front();
```

Novos elementos podem ser inseridos eficientemente em `deque` no início de `c` com a função `push_front`.

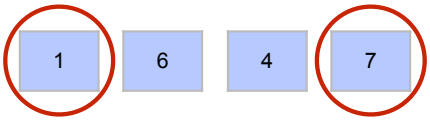
```
1 c.push_front(4);
2 c.push_front(6);
3 c.push_front(1);
```

Assim como em um `vector`, elementos podem ser removidos no fim do `deque` `c` com a função `pop_back`.

```
1 c.pop_back();
2 c.pop_back();
3 c.pop_back();
```

O primeiro e último elementos de `c` podem ser acessados com as funções `front` e `back`.

```
1 cout << c.front() << "␣" << c.back() << endl;
```




```
1 7
```

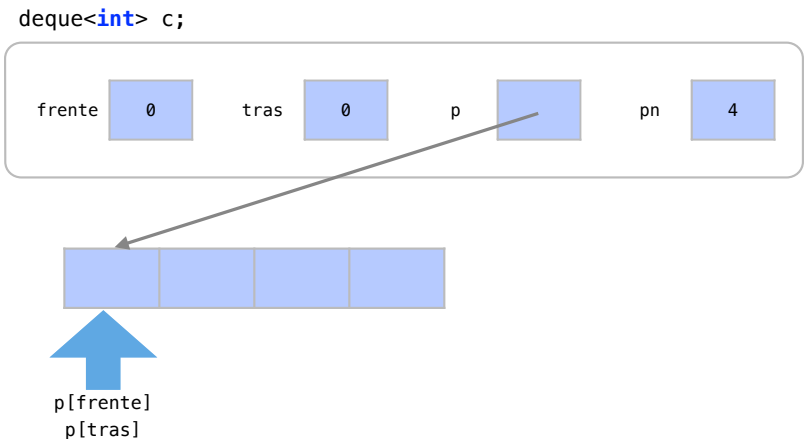
18.4.2 Como funciona

Um `deque` utiliza uma estrutura de dados um pouco mais complexa que um `vector`. Para que elementos do início sejam removidos eficientemente, um `deque` utiliza uma estrutura de arranjo diferente para não precisar deslocar os elementos.

Com este comando, podemos criar um `deque` vazio.

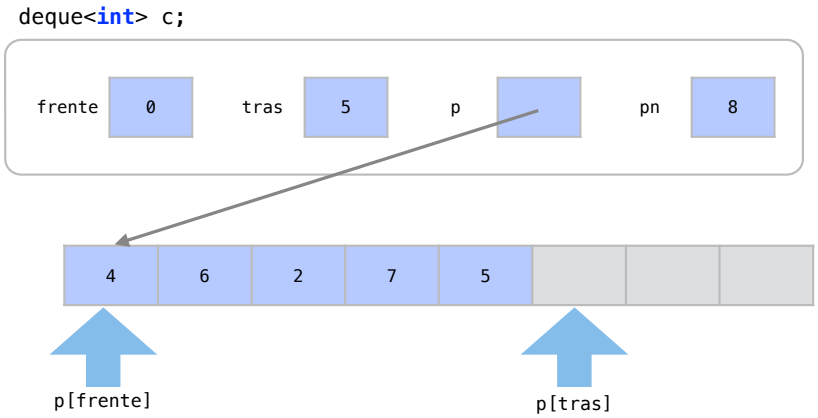
```
1 deque<int> c;
```

Internamente, o `deque` tem um ponteiro um três números inteiros. Diferentemente de `vector`, não temos mais o número `n` de elementos no container e sim dois elementos `frente` e `tras`, que indicam onde começa e termina o `deque`.



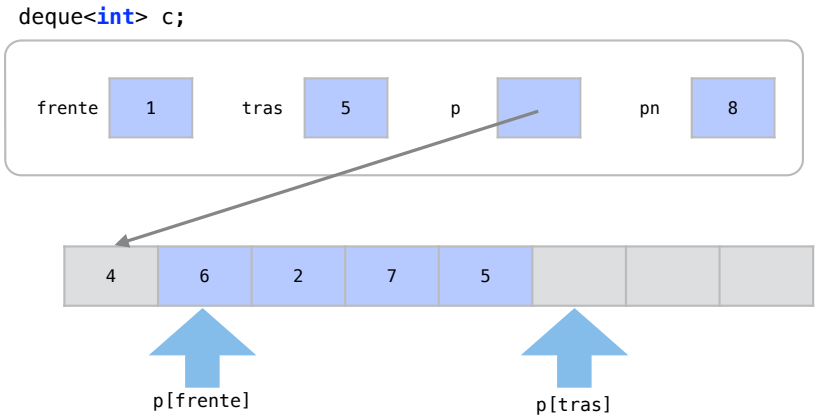
Assim, como no `vector`, a inserção de vários elementos causa o aumento de memória alocada no arranjo.

```
1 c.push_back(4);
2 c.push_back(6);
3 c.push_back(2);
4 c.push_back(7);
5 c.push_back(5);
```



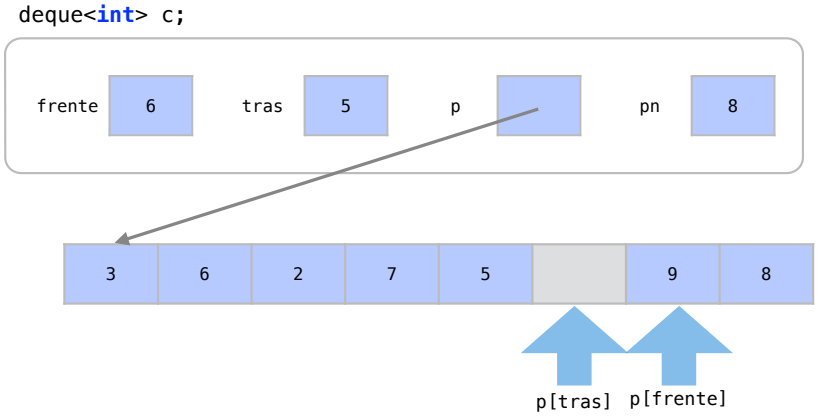
Quando um elemento do início é removido, isto é feito incrementando o valor de `frente`. Os valores de `frente` e `tras` indicam que devemos considerar apenas valores entre p[frente] e p[tras-1] como pertencentes ao container.

```
1 c.pop_front();
```



Quando muitos elementos são inseridos no início do arranjo, uma estrutura toroidal é utilizada para representar os elementos do container.

```
1 c.push_front(3);
2 c.push_front(8);
3 c.push_front(9);
```



Pode parecer estranho que `frente > tras`, mas isto indica que os elementos do container vão de `p[6]` até `p[7]` e depois continuam entre `p[0]` e `p[4]`, formando uma estrutura circular.

18.5 List

O container `list` representa listas duplamente encadeadas. Assim como o `deque`, ele consegue eficientemente inserir e remover elementos das primeiras posições com os mesmos comandos (`push_front`, `push_back`, `pop_front`, `pop_back`). Assim, do ponto de vista de utilização destas funções, ele é equivalente ao `deque`. Porém, sua estrutura de dados é muito diferente, pois não é baseada em arranjos. Para utilizamos o `list`, precisamos incluir o cabeçalho `<list>` no início de nossos programas.

18.5.1 Como funciona

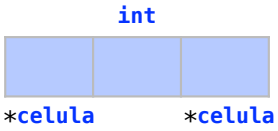
As listas são compostas de *células* que contém ponteiros e valores. Estas células são representadas com definições do tipo `struct`. Cada célula aponta para uma célula similar, de modo que se forme uma lista. Assim, quando um novo elemento é inserido, é alocado mais espaço na memória apenas para uma célula com este elemento.

Com este comando, podemos criar um `list` vazio chamado `c`.

```
1 list<int> c;
```

Internamente, o `list` tem usualmente dois ponteiros (chamados aqui de `frente` e `tras`) e um número inteiro para guardar o número de elementos no container (chamado aqui de `n`).

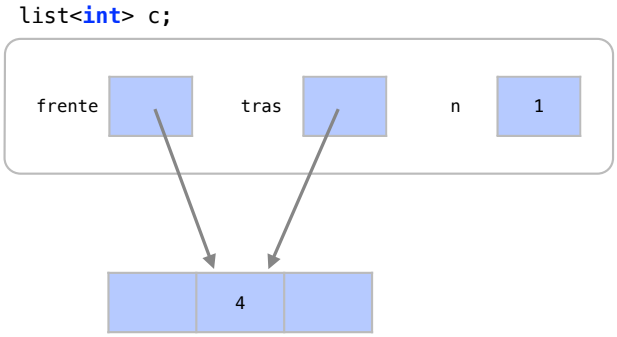
Quando inserimos um elemento no container, uma estrutura chamada célula é criada para este elemento. Cada célula contém 2 ponteiros e um elemento.



Os ponteiros apontam para outras células de modo que possamos formar uma lista de células.

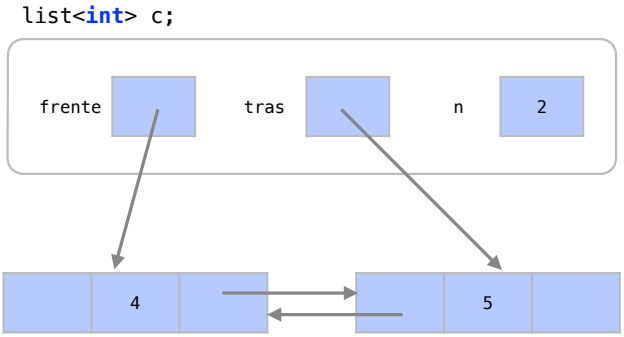
Quando utilizamos o comando para inserir um elemento, o elemento é inserido em uma célula cujos ponteiros não são utilizados por enquanto. Esta célula é criada em uma posição qualquer disponível na memória. Como esta é a única célula da lista, os ponteiros `frente` e `tras` apontam para ela.

```
1 c.push_back(4);
```



Ao se inserir mais um elemento, é criada para ele mais uma célula que entra na lista encadeada.

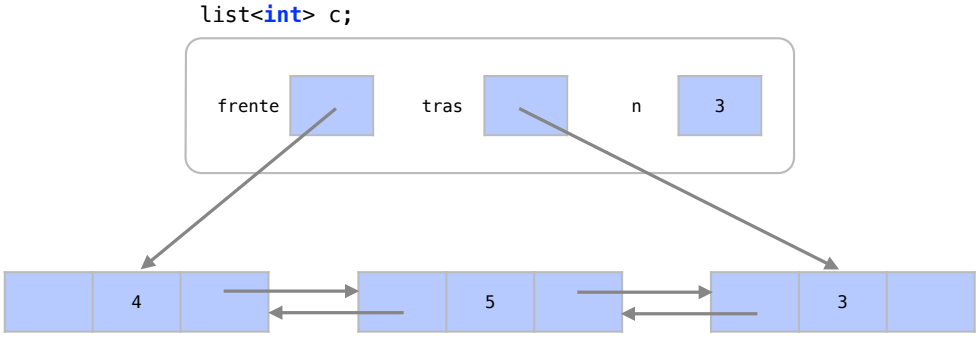
```
1 c.push_back(5);
```



A memória alocada para esta célula está em um local qualquer da memória. Um ponteiro da célula aponta para o último elemento da lista e o último elemento da lista aponta para a nova célula. O ponteiro `tras` passa a apontar para a nova célula, `n` é incrementado e o novo elemento está inserido.

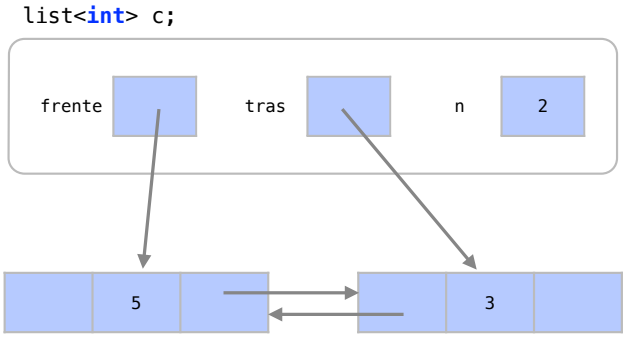
Podemos assim inserir vários elementos com tempo $O(1)$. Para cada nova inserção é alocada uma célula em uma posição qualquer da memória.

```
1 c.push_back(3);
```



Para remover um elemento do início ou do fim da sequência, um processo muito similar é feito.

```
1 c.pop_front();
```



O ponteiro `frente` aponta para o próximo elemento e o anterior é removido, com seus ponteiros.

19. Percorrendo containers

19.1 Subscritos

Assim como utilizamos os subscritos `[` e `]` para acessar posições de arranjos, podemos utilizar subscritos para acessar elementos dos containers de sequência. Porém, apenas os containers de sequência chamados de containers de **acesso aleatório** tem este recurso. Da mesma maneira que fazemos em arranjos, o subscrito `[i]` é utilizado para acessar o $(i+1)$-ésimo elemento do container.

19.1.1 Vector

Neste exemplo utilizaremos subscritos para acessar elementos de um `vector`. Este código utiliza subscritos para imprimir os valores do container `c`.

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      vector<int> c;
8      for (int i=1; i<6; i++) {
9          c.push_back(i);
10     }
11     for (int i=0; i<c.size(); i++) {
12         cout << c[i] << "␣";
13     }
14     cout << endl;
15 }
```

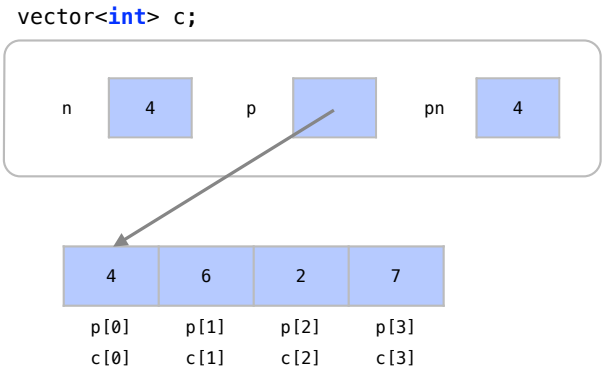
Note na linha 2 que incluimos o cabeçalho `vector`. Nas linhas 7 a 10, criamos um `vector` chamado `c` e inserimos em seu final os elementos 1, 2, 3, 4 e 5. Nas linhas 11 a 14, o subscrito

`c[i]` é utilizado para imprimir os elementos de `c`.

```
1 2 3 4 5
```

Como funciona

Internamente, o `i`-ésimo elemento do `vector` pode ser acessado ao se acessar o `i`-ésimo elemento do arranjo `p` que guarda seus elementos.



Assim, as posições de subscrito do arranjo apontado por `p` são as mesmas que devem ser retornadas pelos subscritos do container `c`.

19.1.2 Deque

Este código utiliza subscritos também para imprimir os valores no `deque` chamado `c`.

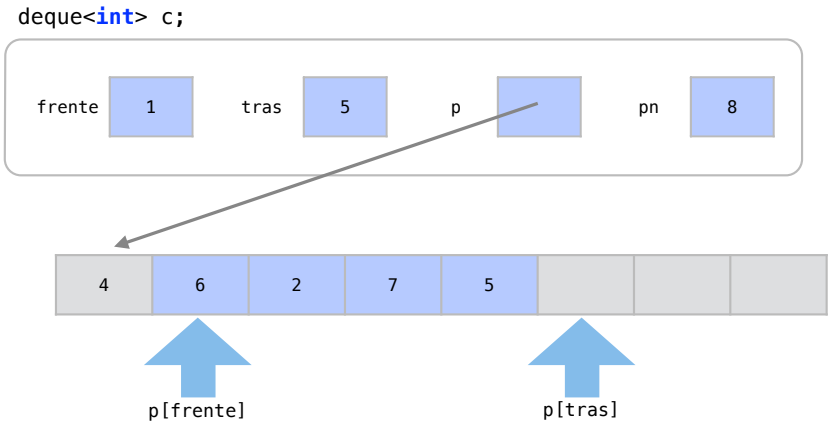
```
1  #include <iostream>
2  #include <deque>
3
4  using namespace std;
5
6  int main()
7  {
8      deque<float> c;
9      for (int i=1; i<6; i++) {
10         c.push_front(i*1.1);
11     }
12     for (int i=0; i<c.size(); i++) {
13         cout << c[i] << "␣";
14     }
15     cout << endl;
16 }
```

Note na linha 2 como incluímos o cabeçalho `<deque>`. O bloco de código nas linhas 8 a 11 cria um `deque` chamado `c` e insere em seu início os elementos `1.1`, `2.2`, `3.3`, `4.4` e `5.5`. No trecho de código das linhas 12 a 15, o subscrito `c[i]` é utilizado para imprimir os elementos de `c`.

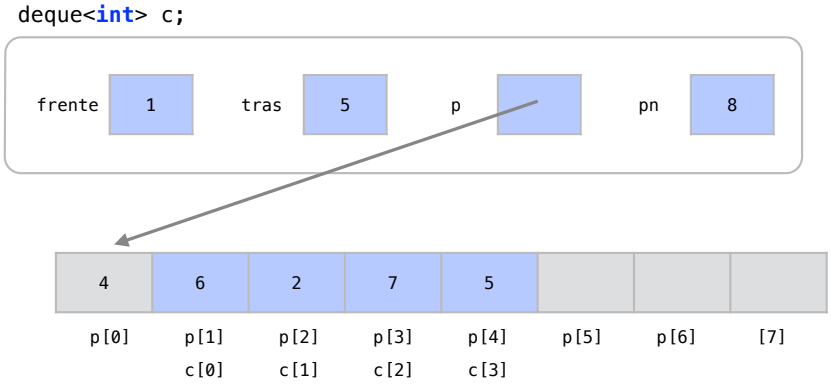
```
5.5 4.4 3.3 2.2 1.1
```

Como funciona

Internamente o `i`-ésimo elemento do deque pode ser acessado ao se acessar o `i`-ésimo elemento do arranjo `p` após o elemento `p[frente]`. Suponha o seguinte `deque`:

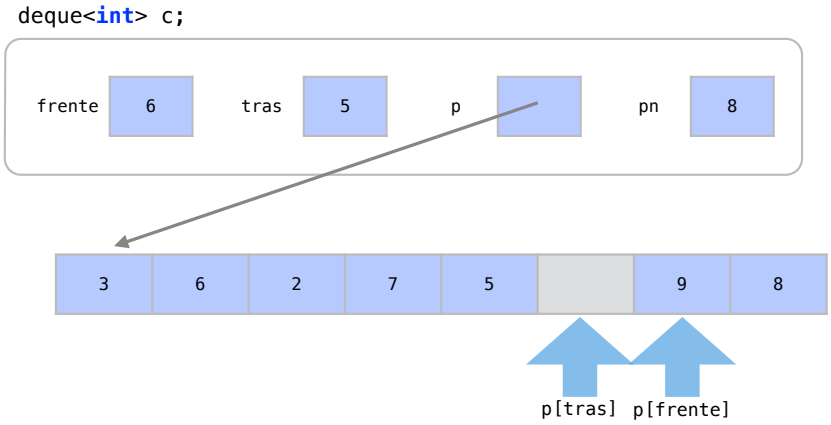


Podemos converter assim a localização dos elementos:

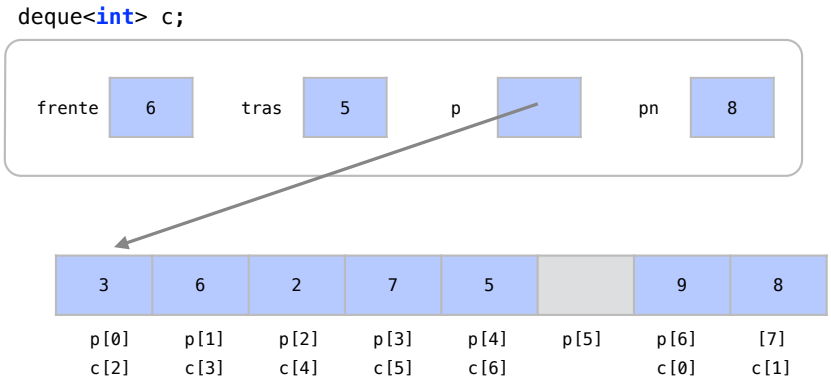


Deste modo, a localização das posições `frente` e `tras` devem ser consideradas ao se procurar o `i`-ésimo elemento do `deque c`. Neste caso, a posição do i-ésimo elemento do container pode ser encontrado na posição `p[frente+i]` do arranjo.

Quando ciclos ocorrem, uma operação de módulo no valor encontrado é necessária para achar a posição correta. Considere o exemplo abaixo onde ocorrem um ciclo:



Neste exemplo, veja os subscritos do arranjo e os subscritos do container neste caso.

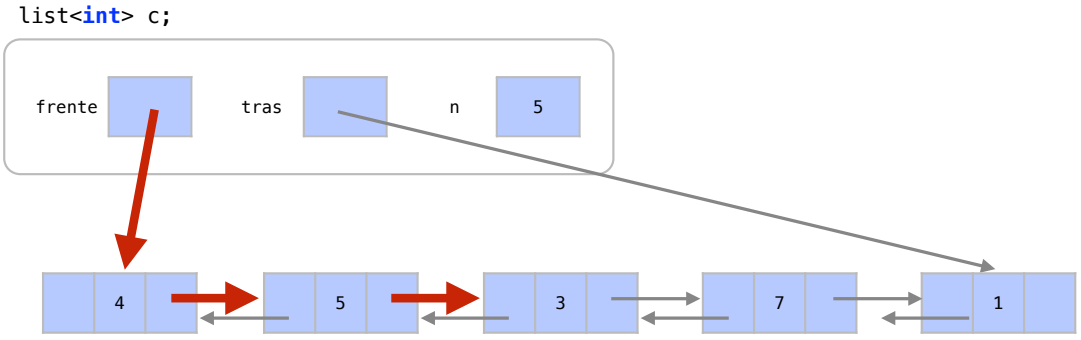


A posição do `i`-ésimo elemento do container pode ser encontrada na posição `p[(frente+i)%pn]` do arranjo apontado por `p`.

19.1.3 List

Já para o container `list`, não é possível encontrar diretamente o `i`-ésimo elemento da sequência, já que os elementos não são organizados em arranjos nesta estrutura. Apenas os containers `vector` e `deque` são de *acesso aleatório*, por serem baseadas em arranjos.

Vejamos um exemplo onde queremos acessar o terceiro elemento de um `list`.



Em um arranjo, sabemos que o `i`-ésimo elemento está `i` posições de memória a frente do primeiro elemento. Como os elementos do `list c` estão espalhados na memória com uma estrutura baseada em ponteiros, não existe uma operação que calcule diretamente a posição do `i`-ésimo elemento do container na memória. Com uma estrutura baseada em ponteiros, a única maneira de se encontrar o `i`-ésimo elemento de `c` é analisando no primeiro elemento onde está o segundo, no segundo onde está o terceiro, ..., no `(i-1)`-ésimo onde está o `i`-ésimo.

Ou seja, precisamos seguir os ponteiros i vezes para encontrar o elemento da posição i. Assim, achar o `i`-ésimo elemento a partir do primeiro tem um custo $O(i)$. Isto significa $O(n)$ no pior caso e caso médio e $O(1)$ no melhor caso, quando procuramos o primeiro ou o último elemento.

19.1.4 Análise

Como vimos, os subscritos são utilizados para acessar diretamente o `i`-ésimo elemento do container. Este é na verdade um recurso que encontra o `i`-ésimo elemento do container no arranjo

alocado para guardar os elementos. Isto é possível apenas para os containers `vector` e `deque`, que usam arranjos e por isso alocam os elementos em sequência na memória. Estes containers são chamados de containers de acesso aleatório.

Cabe aqui uma comparação entre conteiners sequenciais e arranjos. Em C++, a utilização de subscritos é muito útil pois permite até que utilizemos um container de sequência para substituir arranjos. Esta é uma prática comum, especialmente com containers do tipo `vector`. Com esta prática ganhamos várias vantagens em relação a arranjos:

- Containers já controlam seus próprios tamanhos
- Containers contém em si mesmos algoritmos eficientes para várias tarefas

19.2 Iteradores

Como vimos, a opção de se acessar elementos de um container através do operador de subscrito é restrita apenas a containers de acesso aleatório. Para conseguirmos acessar elementos de todos os tipos de container, precisamos de iteradores.

Os iteradores são objetos que caminham (iteram) sobre elementos de containers. Eles funcionam como um ponteiro especial para elementos de containers: enquanto um ponteiro representa uma posição na memória, um iterador representa uma posição em um container. São muito importantes pois permitem criar funções genéricas para qualquer tipo de container.

Supondo um iterador chamado `i`, a Tabela 19.1 apresenta funções comuns a iteradores.

Função	Retorna
`*i`	Retorna o elemento na posição do iterador
`++i`	Avança o iterador para o próximo elemento
`==`	Confere se dois iteradores apontam para mesma posição
`!=`	Confere se dois iteradores apontam para posições diferentes

Tabela 19.1: Funções-membro comuns aos iteradores.

Os iteradores podem ser gerados através de funções de um container, como apresentado na Figura 19.1. Suponha um container chamado `c`. O comando `c.begin()` retorna um iterador para o primeiro elemento deste container `c`. O comando `c.end()` retorna um iterador para uma posição após o último elemento do container `c`.

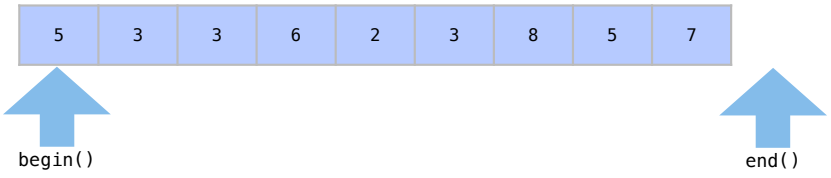


Figura 19.1: As funções `begin()` e `end()` geram iteradores a partir de containers.

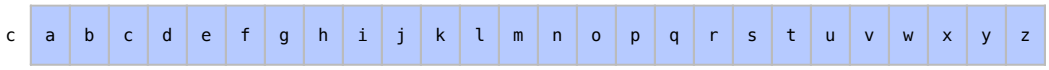
19.2.1 Exemplo

Neste exemplo usamos um iterador para percorrer os elementos de um `list`.

```
1 #include <iostream>
2 #include <list>
3
```

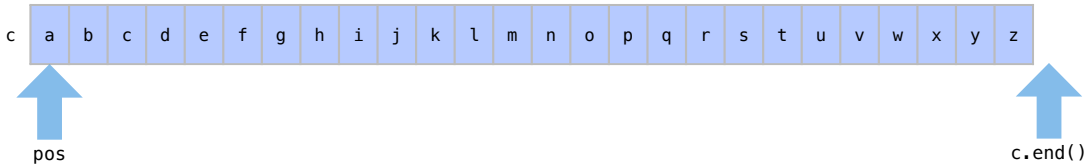
```
4  using namespace std;
5
6  int main(){
7      list<char> c;
8      for (char letra='a'; letra<='z'; letra++) {
9          c.push_back(letra);
10     }
11     list<char>::iterator pos;
12     for (pos = c.begin(); pos != c.end(); ++pos){
13         cout << *pos << "␣";
14     }
15     cout << endl;
16 }
```

Na linha 7, criamos um list que se chama `c` e guardará elementos do tipo `char`. Nas linhas 8 a 10, colocamos no container `c` todos os dados possíveis do tipo `char` entre `'a'` e `'z'`.



Criamos agora, na linha 11, um iterador chamado `pos`. Repare que quando declaramos o tipo de dado deste iterador, utilizamos `list<char>::iterator` para indicar que `pos` é um iterador para um `list` de dados do tipo `char`.

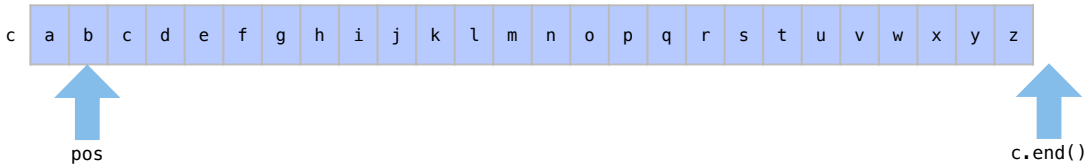
Na linha 12, quando dizemos que `pos = c.begin()`, inicializamos o `for` fazendo com que `pos` seja um iterador apontando para o primeiro elemento de `c`. A condição de parada `pos != c.end()` é que o iterador `pos` tenha chegado na posição `c.end()`, que é uma posição após o último elemento de `c`.



Retornamos então, na linha 13, o elemento na posição do iterador com `*pos`.

```
a
```

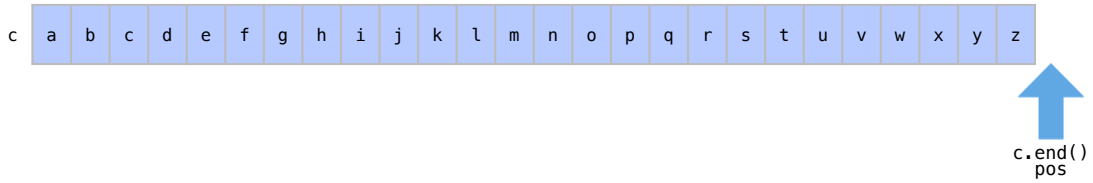
Independente do tipo de container, a condição de incremento `++pos` faz com que `pos` aponte para o próximo elemento do container.



O elemento na posição do iterador é impresso de novo na linha 13.

```
a b
```

O trecho completo de código das linhas 12 a 14 faz com que consigamos imprimir todos os elementos de `c`. Ao fim, `pos` finalmente chega à posição `c.end()`.




```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Repare que, com os iteradores, conseguimos acessar os elementos de um `list`, que não é um container de acesso aleatório.

19.2.2 Categorias de Iteradores

Cada container pode gerar um iterador de certa categoria. O container `list` contém iteradores bidirecionais pois a partir de um elemento só temos acesso ao próximo elemento ou o elemento anterior. Para o containers `vector` e `deque`, temos iteradores de acesso aleatório pois a partir de uma posição qualquer, basta uma operação aritmética para encontrar qualquer outro elemento.

A Tabela 19.2 apresenta funções disponíveis para iteradores de acordo com sua categoria.

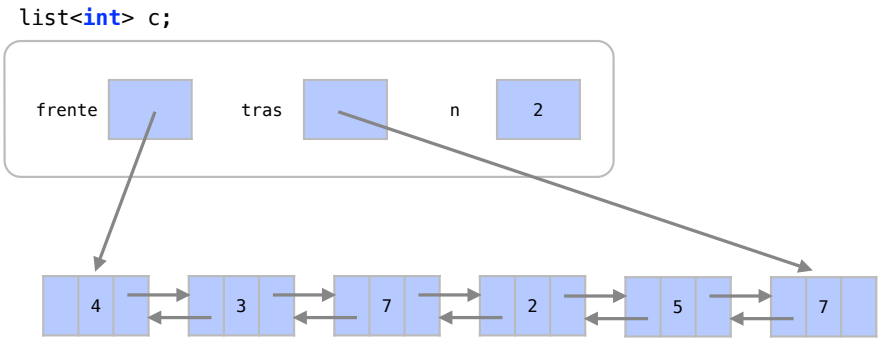
19.2.3 Funções com iteradores

Insert - List

Há ainda funções que pertencem aos containers porém dependem de iteradores como parâmetro. Uma função dos containers de sequência que precisa de um iterador é a função `insert`. Esta função insere um elemento em uma posição qualquer da sequência. Para tal, a função precisa de um elemento e de um iterador para a posição onde queremos colocá-lo.

Considere a estrutura interna de um container do tipo `list` chamado `c`.

```
1 list<int> c;
2 c.push_back(4);
3 c.push_back(3);
4 c.push_back(7);
5 c.push_back(2);
6 c.push_back(5);
7 c.push_back(7);
```

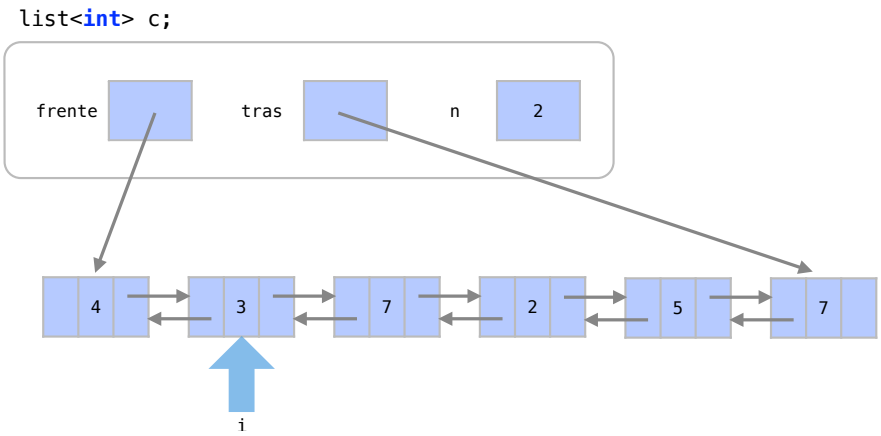


Função	Retorna
Todos iteradores	
`*i`	Desreferencia
`i = i2`	Atribuição
`i == i2`	Compara igualdade de posição
`i != i2`	Compara desigualdade de posição
Iteradores unidirecionais	
`++i`	Pré-incrementa
`i++`	Pós-incrementa
Iteradores bidirecionais	
`--i`	Pré-decrementa
`i--`	Pós-decrementa
Iteradores de acesso aleatório	
`i += n`	Incrementa `n` posições
`i -= n`	Decrementa `n` posições
`i + n`	Aritmética (soma) com iteradores
`i - n`	Aritmética (subtração) com iteradores
`i[i2]`	Desreferencia `i2` posições após `*p`
`i < i2`	Compara posição no container
`i > i2`	Compara posição no container
`i <= i2`	Compara posição no container
`i >= i2`	Compara posição no container

Tabela 19.2: Funções-membro de iteradores de acordo com suas categorias.

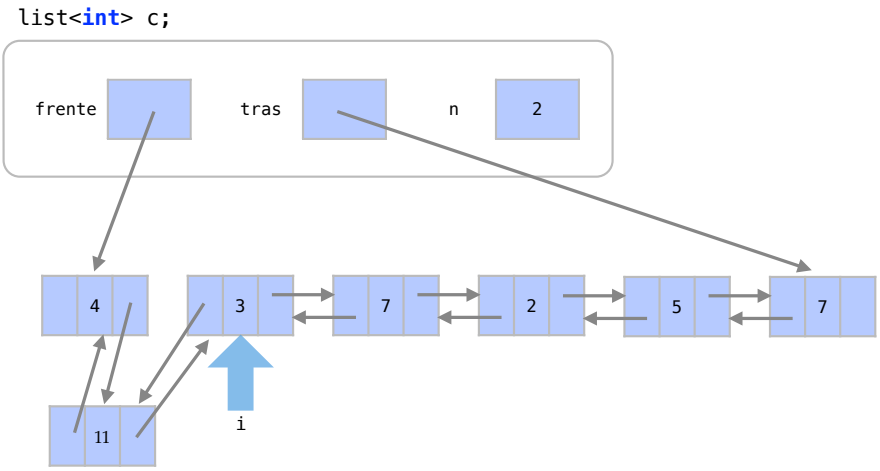
Considere também que geramos um iterador `i` apontando para a segunda posição de `c`.

```
1 list<int>::iterator i;
2 i = c.begin();
3 ++i;
```



Podemos chamar deste modo uma função para inserir um elemento `11` na posição `i` do container.

```
1 c.insert(i,11);
```



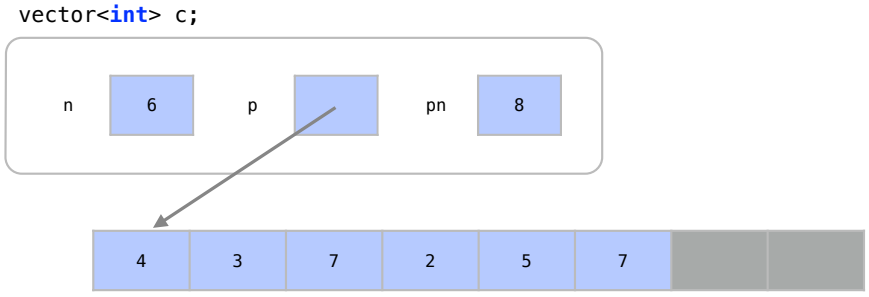
Será alocada na memória uma nova célula para este elemento `11`. A nova célula aponta para a célula anterior à da posição `i` e para a própria célula da posição `i`. Alteramos então os ponteiros do elemento `i` e do elemento anterior a `i`.

Supondo que já temos um iterador para a posição que queremos, o número de operações para esta inserção é constante $O(1)$. Esta é exatamente a maior vantagem do container `list`. Fazer com que o iterador chegue à posição `i`, porém, pode ter custo $O(n)$ pois um container do tipo `list` não tem acesso aleatório.

Insert - Vector

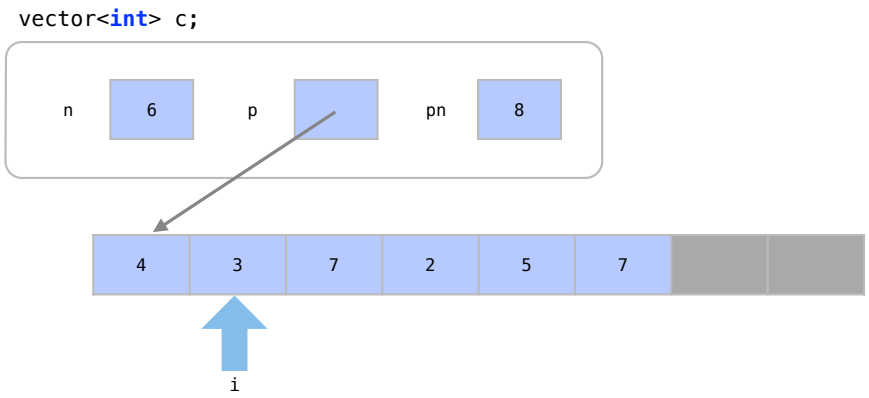
Vejamos agora a utilização da função `insert` em um `vector`. Considere agora a estrutura interna da mesma sequência sendo representada por um `vector`.

```
1 vector<int> c;
2 c.push_back(4);
3 c.push_back(3);
4 c.push_back(7);
5 c.push_back(2);
6 c.push_back(5);
7 c.push_back(7);
```



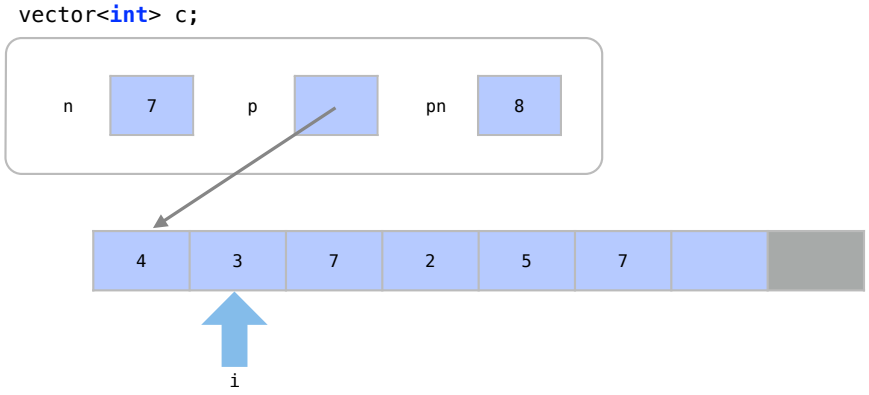
Considere também que geraremos um iterador `i` apontando para o segundo elemento da sequência.

```
1 vector<int>::iterator i;
2 i = c.begin();
3 ++i;
```

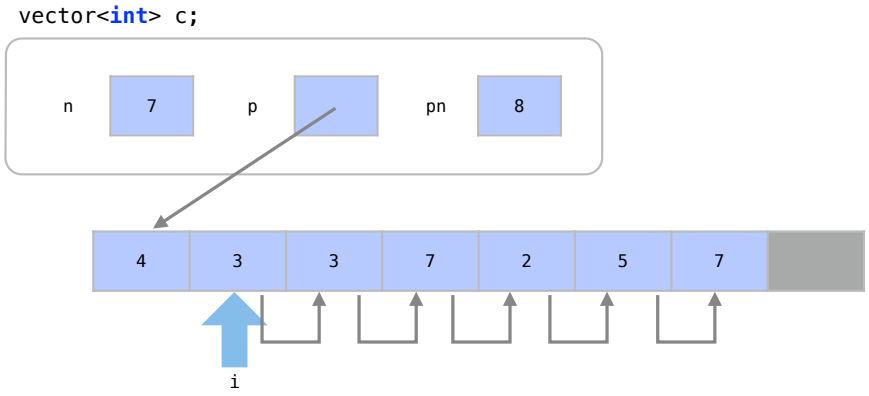


A função que insere o elemento `11` na posição `i` é mais complicada para um `vector`. Temos uma sequência de passos:

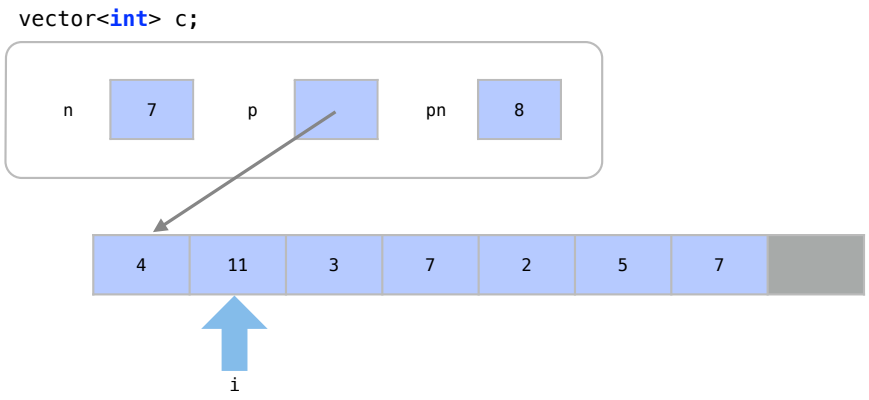
1) A variável `n` é incrementada, abrindo espaço para mais um elemento. Este processo tem um custo constante $O(1)$. Em algumas situações, pode ser que um novo arranjo precise ser alocado, levando a um custo $O(n)$, como vimos na Seção 18.3.2.



2) Todos os elementos entre `i` e o penúltimo elemento precisam ser deslocados para a próxima posição do arranjo. Esta operação tem um custo $O(n-i)$. Se `i` for o último elemento, temos um melhor caso $O(1)$. Se `i` for o primeiro elemento, temos um pior caso $O(n)$. Se consideramos que, no caso médio, `i` tem a mesma probabilidade de estar em qualquer posição, temos também um custo $O(n)$ no caso médio.



3) O elemento `11` pode ser inserido na posição `i`. Este passo tem apenas um custo $O(1)$.



Devido a todas as movimentações, esta operação completa de inserção no meio da sequência tem um custo $O(n)$.

Como vimos, a grande vantagem dos containers `vector` e `deque` vêm do fato de eles serem baseados em arranjos. Isso permite que os elementos fiquem em sequência na memória possibilitando assim acesso aleatório aos elementos.

Curiosamente, a grande desvantagem destas estruturas também se deve ao fato de utilizarem arranjos, pois todos os elementos precisam ser movidos para que um elemento seja inserido no meio da sequência.

Insert e Erase

De modo análogo à função `insert(i, elem)`, existe a função `erase(i)`, que remove do container o elemento da posição `i`.

A função `insert` pode também receber 2 iteradores em vez de um elemento para inserir vários elementos de uma vez. Neste caso, os elementos entre os dois iteradores em um segundo container são inseridos no container.

A função `erase` pode também receber 2 iteradores em vez de um para remover vários elementos de uma vez.

Função construtura

Com iteradores, um container pode até mesmo já ser construído com elementos de outro container de outro tipo. No exemplo abaixo, já criamos o container `c2` com uma cópia dos elementos do container `c1`. Dizemos que esta é a função que constrói `c2`.

```
1 vector<int> c2(c.begin(), c.end());
```

Um container também pode ser construído com os elementos de um arranjo, através de seus endereços.

```
1 vector<int> c(a, a+n);
```

Funções genéricas

Vimos no exemplo anterior a função `insert`, que depende de iteradores para definir em qual posição de um container será inserido o elemento. Outra utilidade de iteradores é possibilitar funções genéricas, que funcionem para qualquer tipo de container.

A função abaixo imprime todos os elementos de um container:

```
1 template <typename T>
2 void imprime (T const& x)
3 {
4     typename T::const_iterator pos;
```

```
5       typename T::const_iterator end(x.end());
6       for (pos=x.begin(); pos!=end; ++pos) {
7           cout << (*pos) << "␣";
8       }
9       cout << endl;
10  }
```

Como ela utiliza templates, ela pode ser utilizada com qualquer container.

Nesta função, na linha 1, temos no template que o tipo de container será referido como T.

Nas linhas 4 e 5, criamos um iterador `pos` que caminhará sobre os itens do container e um iterador `end` que é inicializado com a posição final do container chamado aqui de `x`.

Utilizamos entãoo iterador, nas linhas 6 a 8, como em qualquer outro código.

É importante lembrar que algumas operações não estão disponíveis para todos os tipos de iteradores. Apresentamos as categorias de iteradores na Seção 19.2.2. É preciso tomar cuidado ao tentar criar um código genérico para qualquer tipo de container. O código abaixo, por exemplo, tem capacidade de percorrer os elementos de qualquer tipo de container.

```
1  for (pos = c.begin(); pos != c.end(); ++pos){
2      cout << *pos << "␣";
3  }
```

Já o código abaixo não conseguirá percorrer elementos de qualquer tipo de container pois o operador < só está disponível para iteradores de acesso aleatório.

```
1  for (pos = c.begin(); pos < c.end(); ++pos){
2      cout << *pos << "␣";
3  }
```

20. Análise dos Containers Sequenciais

Vimos nas últimas Seções três tipos de containers sequenciais e como eles podem ser acessados com iteradores. Podemos agora fazer uma breve comparação entre estas diferentes maneiras de representar estruturas de dados.

20.1 Vector

Quando acaba a memória, um `vector` cria um novo arranjo dinamicamente e copia os elementos. Isto tem custo $O(n)$.

Em geral, tanto a inserção quanto a remoção de elementos no fim da sequência tem custo $O(1)$ em um `vector`. É preciso saber que há um custo maior $O(n)$ quando for necessário alocar memória para um novo arranjo. Porém, como esta alocação não é frequente, no caso médio, o custo de se inserir um elemento continua $O(1)$.

Por ter seus elementos organizados em uma arranjo (com elementos em sequência na memória), uma inserção ou remoção no meio da sequência tem custo $O(n)$ pois elementos precisam ser deslocados.

Por outro lado, uma consulta em qualquer posição de um `vector` tem custo $O(1)$ pois eles são containers de acesso aleatório. Esta é uma grande vantagem do `vector`.

Uma outra grande vantagem de um `vector` é o baixo gasto de memória auxiliar para gerenciar sua estrutura. Em um arranjo, não temos gasto extra algum com ponteiros.

Sua maior desvantagem é o custo $O(n)$ de se inserir um elemento no início ou no meio da sequência.

20.2 Deque

A estrutura do `deque` permite que elementos sejam inseridos com baixo custo $O(1)$ tanto no início quanto no fim do arranjo.

Além desta única diferença para o `vector`, por ser também baseado em arranjos, um `deque` tem os mesmos resultados assintóticos do vector para (i) inserção e remoção no meio da sequência $O(n)$, (ii) realocação de memória $O(n)$ e (iii) acesso aleatório a elementos $O(1)$.

Em relação ao `list`, o `deque` possui as mesmas vantagens e desvantagens do `vector`: (i) baixo custo de memória extra $O(1)$, (ii) acesso aleatório aos elementos $O(1)$ e (iii) alto custo para inserção ou remoção no meio da sequência $O(n)$.

Em relação ao `vector`, em termos assintóticos, ele oferece a possibilidade de inserção de elementos no início da sequência em tempo $O(1)$.

Em suma, além das vantagens assintóticas do `vector` relacionadas à capacidade de acesso aleatório, um `deque` apresenta a possibilidade de inserção ou remoção no início da sequência em tempo $O(1)$, o que levaria tempo $O(n)$ para um `vector`. Deste modo, do ponto de vista assintótico, pode parecer que não é vantagem utilizar um `vector` em situação alguma.

Porém, nas operações de acesso a elementos onde `deque` e `vector` gastam tempo constante $O(1)$, o tempo total gasto pelo `deque` é sempre maior do que um `vector`. Isto ocorre porque um `deque` precisa de operações aritméticas de soma e resto para encontrar seus elementos enquanto o `vector` pode procurar a posição diretamente no arranjo. Vimos isto na Seção 19.1, quando analisamos a utilização de subscritos em containers. Deste modo, a utilização do `deque` só é recomendada se o recurso de inserção e remoção eficientes no início da sequência for realmente muito utilizado e importante.

20.3 List

A maior vantagem dos containers do tipo `list` é a inserção ou remoção de elementos em qualquer posição com custo constante $O(1)$, desde que tenhamos um iterador para esta posição. Não existe realocação de arranjos nestas estruturas.

Um grande desvantagem é o alto gasto de memória extra $O(n)$ nestas estruturas pois para cada elemento são guardados dois ponteiros. Outra desvantagem é que este container não oferece acesso aleatório a seus elementos e gerar um iterador para um elemento pode ter custo $O(n)$.

20.4 Comparação

A Tabela 20.1 apresenta o custo das principais operações utilizadas nas estruturas de dados representadas pelos containers sequenciais.

	Container Sequencial		
	`vector`	`deque`	`list`
Inserção/Remoção			
Início	$O(n)$	$O(1)$	$O(1)$
Meio	$O(n)$	$O(n)$	$O(1)$
Fim	$O(1)$	$O(1)$	$O(1)$
Acesso			
Início	$O(1)$	$O(1)$	$O(1)$
Meio	$O(1)$	$O(1)$	$O(n)$
Fim	$O(1)$	$O(1)$	$O(1)$
Características			
Acesso Aleatório	Sim	Sim	Não

Tabela 20.1: Comparação de custo em termos de número de atribuições para estruturas de dados sequenciais. As estruturas de acesso aleatório tem mais facilidade de acesso a elementos no meio da sequência. As estruturas baseadas em ponteiro tem mais facilidade de inserção e remoção de elementos no meio da sequência.

Qualquer container tem a capacidade inserir, remover, ou acessar elementos no fim da sequência em tempo $O(1)$. Dado um iterador, apenas um `list` pode colocar elementos no meio de uma sequência em tempo $O(1)$. Apenas um `vector` não pode colocar elementos no início de uma sequência em tempo $O(1)$.

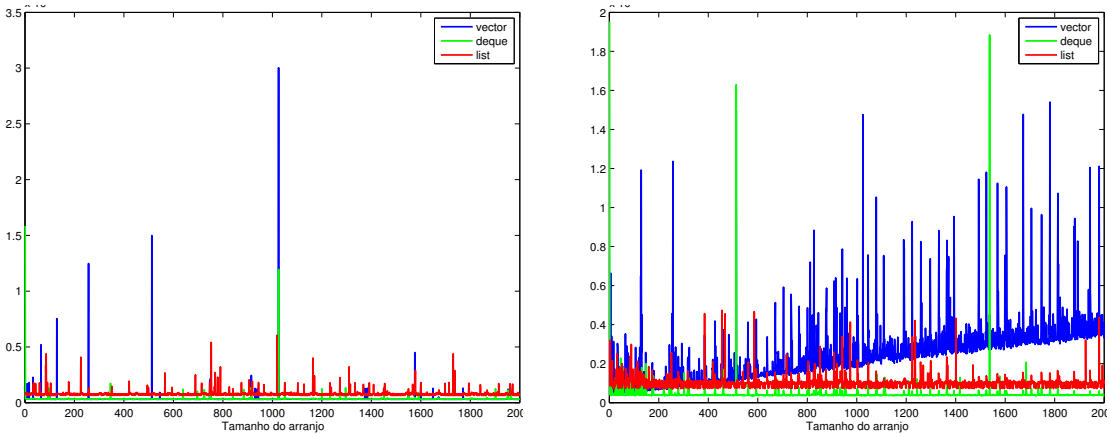
O `vector` possui então apenas capacidade de inserir e remover elementos no fim da sequência em tempo $O(1)$. Um `deque`, apesar de ser baseado em arranjos também, pode inserir ou remover de elementos no início de uma sequência em tempo $O(1)$. A grande vantagem do `list` é ter a capacidade de inserir ou remover elementos em qualquer posição de uma sequência em tempo $O(1)$.

Do ponto de vista de acesso, por outro lado, o `vector` é uma estrutura muito vantajosa. Em relação a custo de acesso, um `vector` ou um `deque` podem acessar qualquer posição da sequência em tempo $O(1)$. Apesar de ambos terem um custo constante de acesso, o custo de acesso em um `vector` será menor que em um `deque` pois ele precisa de operações aritméticas mais simples. Já um `list`, tem um gasto $O(n)$ para acessar elementos no meio da lista pois ele não tem acesso aleatório.

Assim, uma questão fundamental ao se escolher um container é saber se precisamos do recurso de acesso aleatório para elementos no meio da sequência. Isto faz com que os containers de acesso aleatório sejam ideais para aplicações pouco dinâmicas. Se vamos usar um `deque` ou um `vector`, precisamos lembrar que apesar de ser $O(1)$, o custo de acesso em um `vector` é menor que em um `deque`.

A Figura 20.1 apresenta o custo de inserção de um elemento em um container sequencial. Em experimentos reais, apesar da inserção no fim de um container ter sempre custo assintótico $O(1)$, podemos ver que os containers `vector` e `deque` apresentam alguns picos de custos bem maiores que a média para alguns tamanho de arranjo. Isso se deve à realocação de arranjos. Além disso, a inserção no fim de um `deque` costuma ser 23% mais lenta que um `vector`, enquanto a inserção no fim de um `list` costuma ser 416% mais lenta do que um `vector`.

Para inserção no início da sequência podemos ver como esta operação em um `vector` é maior para arranjos maiores, já o custo desta operação é $O(n)$ neste container. Porém, a inserção no início da sequência ainda é vantajosa em um `vector` do que em um `list` se temos arranjos com menos que 250 elementos. Apesar da complexidade constante para os outros containers, um `list` costuma ser 23% mais lento que um `deque`.



(a) Fim da sequência (b) Início da sequência

Figura 20.1: Custo de inserção em um container sequencial.

20.5 Exercícios

Exercício 20.1 Analise o código abaixo, que está incompleto.

```
#include <iostream>
 // bibliotecas com os containeres
#include <vector>
#include <deque>
#include <list>

using namespace std;

template <typename T>
void imprimeContainer (T const& coll);

int main()
{
    // cria um vector de int chamado dados
    vector<int> vetor;

    // inserindo dados no fim do vector, um a um
    vetor.push_back(2);
    vetor.push_back(4);
    vetor.push_back(6);
    vetor.push_back(9);
    vetor.push_back(5);
    vetor.push_back(7);
    vetor.push_back(3);
    vetor.push_back(8);
    cout << "Elementos no vector inicial (" << vetor.size()
        << "):" << endl;
    imprimeContainer(vetor);

        // Faça aqui um algoritmo de inserção
    vetor.resize(vetor.size()+1);

    // Imprimindo o vector novamente com o elemento
       inserido
    cout << "Elementos no vector com elemento na posicao
       inicial (" << vetor.size() << "):" << endl;
    imprimeContainer(vetor);

    // Conseguindo um iterador para o meio do vetor
    vector<int>::iterator i_vector = vetor.begin();
    i_vector+=vetor.size()/2;

    // Imprimindo o elemento do meio do vector
    cout << "O elemento que esta no meio do vetor é o " <<
       *i_vector << endl;

```

```
43      // Função insert coloca um elemento na posição indicada
            pelo iterador
44      vetor.insert(i_vector, 90);
45      cout << "Elementos␣no␣vector␣com␣90␣inserido␣no␣meio␣("
            << vetor.size() << "):" << endl;
46      imprimeContainer(vetor);
47
48      // A função resize aumenta o tamanho do vector
49      vetor.resize(vetor.size()+1);
50      cout << "Elementos␣no␣vector␣aumentado␣em␣1␣(" << vetor
           .size() << "):" << endl;
51      imprimeContainer(vetor);
52
53      // Faça aqui uma inserção no meio do vector
54
55      // Imprimindo o vector mais uma vez
56      cout << "Elementos␣no␣vector␣com␣elemento␣inserido␣("
           << vetor.size() << "):" << endl;
57      imprimeContainer(vetor);
58
59      // Operações similares serão feitas com um deque.
60      deque<int> fila;
61      // Copiando os elementos do vector para a fila
           duplamente encadeada
62      while (!vetor.empty()){
63          fila.push_front(vetor.back());
64          vetor.pop_back();
65      }
66
67      // Imprime a fila
68      cout << "Elementos␣na␣fila␣com␣duas␣pontas␣inicial␣("
           << fila.size() << "):" << endl;
69      imprimeContainer(fila);
70
71      // Inserir elementos com push_front
72
73      // Imprimindo a nova fila
74      cout << "Elementos␣na␣fila␣com␣duas␣pontas␣com␣
           elementos␣no␣início␣(" << fila.size() << "):" <<
           endl;
75      imprimeContainer(fila);
76
77      // Conseguindo um iterador para o meio do deque
78      deque<int>::iterator i_fila = fila.begin();
79      i_fila+=fila.size()/2;
80
81      // Imprimindo o elemento do meio do deque
82      cout << "O␣elemento␣que␣esta␣no␣meio␣da␣fila␣com␣duas␣
           pontas␣eh␣o␣" << *i_fila << endl;
```

```
83
84      // Função insert coloca um elemento após a posição
           indicada
85      fila.insert(i_fila, 91);
86      cout << "Elementos␣na␣fila␣com␣duas␣pontas␣com␣91␣
           inserido␣(" << fila.size() << "):" << endl;
87      imprimeContainer(fila);
88
89      fila.resize(fila.size()+1);
90      cout << "Elementos␣na␣fila␣com␣duas␣pontas␣aumentada␣("
            << fila.size() << "):" << endl;
91      imprimeContainer(fila);
92
93      // Fazer inserção no meio do deque
94
95      // Imprimindo a fila mais uma vez
96      cout << "Elementos␣na␣fila␣com␣duas␣pontas␣com␣elemento
           ␣inserido␣(" << fila.size() << "):" << endl;
97      imprimeContainer(fila);
98
99      // Operações similares serão feitas com uma lista
           duplamente encadeada
100     list<int> lista;
101
102     // Copiando os elementos da fila para a lista
           duplamente encadeada
103     while (!fila.empty()){
104         lista.push_back(fila.front());
105         fila.pop_front();
106     }
107
108     // Imprime a fila
109     cout << "Elementos␣na␣lista␣duplamente␣encadeada␣
           inicial␣(" << lista.size() << "):" << endl;
110     imprimeContainer(lista);
111
112     // Fazer inserção no inicio da lista
113
114     // Imprimindo a nova lista
115     cout << "Elementos␣na␣lista␣duplamente␣encadeada␣(" <<
           lista.size() << "):" << endl;
116     imprimeContainer(lista);
117
118     // Obter iterador para o meio da lista
119     list<int>::iterator i_lista = lista.begin(); // pegando
           a posição inicial
120
121     // Imprimindo o elemento do meio da lista
```

```
122     cout << "O elemento que esta no meio da lista
            duplamente encadeada eh o " << *i_lista << endl;
123
124     // Função insert coloca um elemento após a posição
            indicada
125     lista.insert(i_lista, 92);
126     cout << "Elementos na lista duplamente encadeada com o
            92 (" << lista.size() << "):" << endl;
127     imprimeContainer(lista);
128
129     // Limpa a lista
130     lista.clear();
131     cout << "Lista duplamente encadeada limpa (" << lista.
            size() << "):" << endl;
132     imprimeContainer(lista);
133
134     return 0;
135 }
136
137 template <typename T>
138 void imprimeContainer (T const& coll)
139 {
140     if (coll.empty()){
141         std::cout << "(Vazio)" << endl;
142         return;
143     }
144     typename T::const_iterator pos;  // iterator que vai
            percorrer coll
145     typename T::const_iterator end(coll.end());  //
            iterador que aponta para a posição final
146
147     for (pos=coll.begin(); pos!=end; ++pos) {
148         std::cout << *pos << ' ';
149     }
150     std::cout << std::endl;
151 }
```

■

Exercício 20.2 O container `vector` não tem uma função `push_front` que insere um elemento na primeira posição do `vector`. Faça na linha 29 um código para inserir um elemento qualquer na primeira posição do `vector` (sem usar a função `insert`).

Na linha 29, a função resize já está sendo utilizada para aumentar o tamanho do vetor em 1. A função resize aumenta o tamanho do vector (completando com elementos de valor 0) para caber mais elementos. Você precisará mover todos os elementos existentes para frente e abrir espaço para o novo elemento.

Qual a dificuldade de se inserir um elemento em um `vector`?

Observação: A função `resize` não necessariamente realoca o arranjo no `vector`. A *capacidade* dele se mantém constante, se possível. ■

Exercício 20.3 Faça na linha 53 um código para inserir um elemento no meio do `vector` sem utilizar a função `insert`. Para isto, você precisará mover elementos após o meio em uma posição. ■

Exercício 20.4 Na linha 71, insira mais alguns elementos no início do `deque` com a função `push_front`. Esta função não estava disponível para um `vector`. Porquê? Como ela é feita em um `deque`? ■

Exercício 20.5 Na linha 93, faça um código para inserir um elemento no meio do `deque`. Porém, desta vez, a função `insert` não deverá ser utilizada. ■

Exercício 20.6 O `deque` faz todas as operações que o `vector` é capaz de fazer com a mesma complexidade em notação O e ainda tem a capacidade de inserir elementos na primeira posição em tempo $O(1)$. Sendo assim, qual a vantagem da utilização de um `vector` em relação a um `deque`? ■

Exercício 20.7 Na linha 112, insira mais alguns elementos no início da lista com a função `push_front`. Esta função não estava disponível para um `vector` mas estava disponível para um `deque`. Como ela é ocorre em um `list`? ■

Exercício 20.8 Na linha 118, você deve conseguir um iterador para o elemento do meio da lista duplamente encadeada. Porém diferentemente deste trecho de código para `vector` e `deque`, a operação `+=list.size()/2` não pode ser utilizada. Isso ocorre pois o container `list` não é um container de acesso aleatório. Faça o código de maneira que funcione para `list`, utilizando o operador ++ repetidamente em um `for`. Qual sua dificuldade? Por que isso acontece? ■

Exercício 20.9 A função de insert deve ser utilizada para se inserir elementos no meio de listas. Não é possível fazer isto de maneira manual como fizemos para `vector` e `deque` pois não existe a operação `[]` em filas. Conhecendo como é implementada na verdade a operação insert de um `vector` e um `deque` (exercícios anteriores), qual a vantagem da operação `insert` para listas? ■

Exercício 20.10 Em todo o código, sempre que criamos um container novo, copiamos os elementos do container antigo através de um loop. Altere estes trechos de código de modo os contrutores dos novos containers sejam utilizados com os elementos do container antigo. Exemplo: `vector<int> c(c2.begin(), c2.end());`. ■

21. Containers Associativos e Conjuntos

Os containers associativos são utilizados para representar conjuntos. Em conjuntos, é importante saber se um elemento pertence ou não ao conjunto. Porém, não queremos saber da posição de um elemento no conjunto. Alterar a posição dos elementos internamente, não altera o conjunto sendo representado. Esta falta de controle na relação de ordem difere estes dos containers de sequência, como apresentado na Figura 21.1.

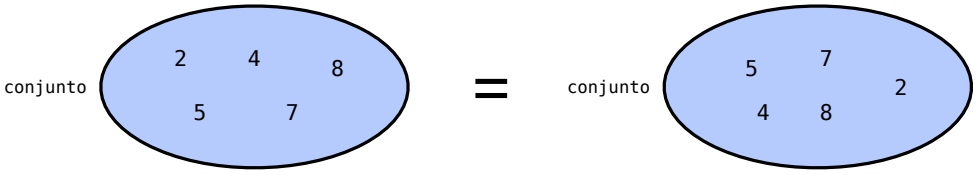


Figura 21.1: Os containers associativos são utilizados para representar conjuntos. Não existe uma relação de ordem entre os elementos. Se dois conjuntos têm os mesmos elementos, eles são o mesmo conjunto, independentemente da posição de cada elemento.

Como a posição dos elementos não é relevante, eles ficam organizados automaticamente. O critério de organização dos elementos é especificado em uma função de comparação utilizada pelo container.

Nestes containers, elementos são encontrados rapidamente pois a estrutura de dados sabe encontrá-los de acordo com sua chave. Busca-se elementos pela própria chave e não há uma posição específica de sequência para o elemento.

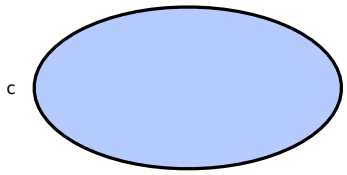
21.1 Containers Associativos Ordenados

21.1.1 Set

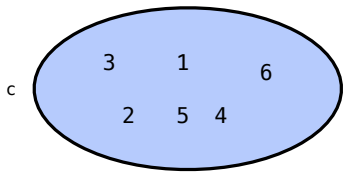
O container `set` é o container utilizado para representar conjuntos. Ao acessar os elementos de um `set` um a um, eles são ordenados de acordo com seu valor. Cada elemento pode ocorrer apenas uma vez em um conjunto representado por um `set`.

Considere que o conjunto de números inteiros abaixo está representado por um `set` chamado `c`.

```
1 set<int> c;
```



O conjunto criado está inicialmente vazio.
Elementos podem ser inseridos em um conjunto com a função `insert`.



Note que a função `insert` não precisou de um iterador indicando em qual posição do conjunto será inserido o elemento.

Como em qualquer container, podemos usar iteradores para acessar os elementos do container um a um.

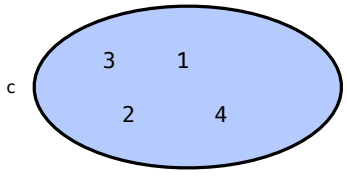
```
1 set<int>::iterator i;
2 for (i = c.begin(); i != c.end(); ++i){
3         std::cout << *i << "␣";
4 }
5 cout << endl;
```

Ao fazer isto, os elementos são impressos em ordem crescente. Isto não depende da ordem em que foram inseridos.

```
1 2 3 4 5 6
```

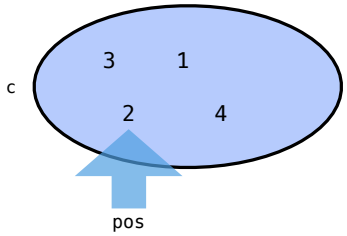
Já a função `erase` pode ser utilizada para apagar elementos de acordo com suas chaves.

```
1 c.erase(5);
2 c.erase(6);
```



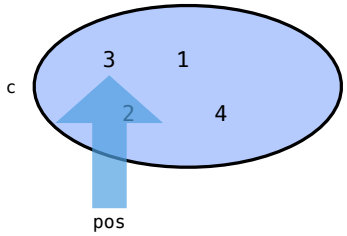
A função `find` procura um elemento e retorna um iterador para ele caso este seja encontrado.

```
1 set<int>::iterator pos;
2 pos = c.find(2);
```



O operador ++ leva o iterador para o próximo elemento do conjunto, considerando os elementos em ordem.

```
1 ++pos;
```

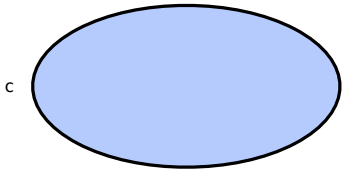


21.1.2 Multiset

O `multiset` é um container muito similar ao `set`. A diferença é que ele representa multiconjuntos, ou seja, conjuntos onde um elemento pode ocorrer mais de uma vez.

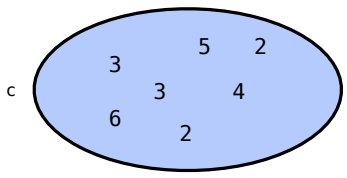
Suponha o seguinte `multiset` abaixo chamado `c`.

```
1 multiset<int> c;
```



Podemos agora inserir elementos replicados no conjunto.

```
1 c.insert(3);
2 c.insert(3);
3 c.insert(6);
4 c.insert(4);
5 c.insert(2);
6 c.insert(2);
7 c.insert(5);
```



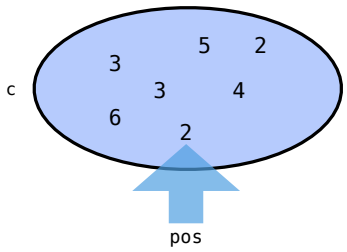
Ao iterar pelos elementos, estes ainda estão ordenados. As repetições de elementos ocorrem em sequência.

```
1 multiset<int>::iterator i;
2 for (i = c.begin(); i != c.end(); ++i){
3         cout << *i << "␣";
4 }
5 cout << endl;
```

```
2 2 3 3 4 5 6
```

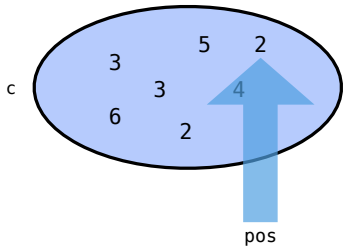
A função `find` procura a primeira ocorrência de um elemento e retorna um iterador para ele, caso este seja encontrado.

```
1 set<int>::iterator pos;
2 pos = c.find(2);
```



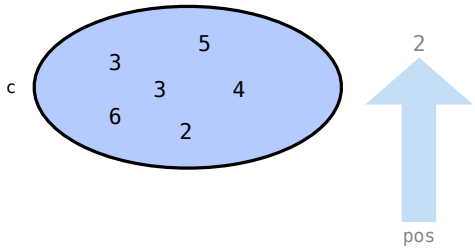
O operador ++ leva o iterador para o próximo elemento do conjunto ou para a próxima repetição do elemento, considerando os elementos em ordem.

```
1 ++pos;
```



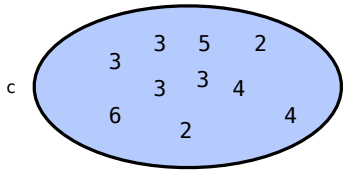
A função `erase` pode receber um iterador para retirar apenas o elemento apontado do conjunto.

```
1 c.erase(pos);
```



Além das funções usuais, a função `count` pode ser utilizada em um `multiset` para determinar quantas ocorrências de um elemento existe em um conjunto.

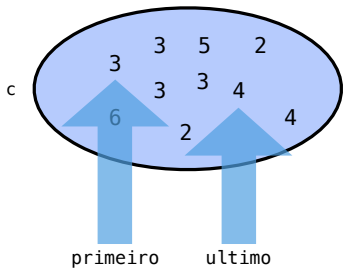
```
1 c.insert(3);
2 c.insert(3);
3 c.insert(2);
4 c.insert(4);
5 cout << c.count(3);
```




```
2
```

Há outras duas funções importantes em multiconjuntos. A função `lower_bound` retorna um iterador para a primeira ocorrência de um elemento no conjunto. A função `upper_bound` procura a última ocorrência de um elemento no conjunto e retorna um iterador para a próxima posição.

```
1 set<int>::iterator primeiro = c.lower_bound(3);
2 set<int>::iterator ultimo = c.upper_bound(3);
```

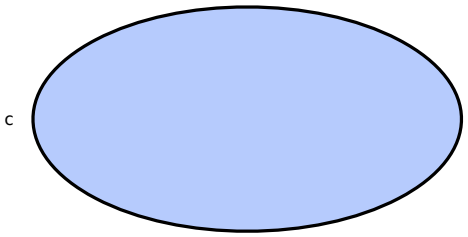


21.1.3 Map

O conteiner `map` é utilizado para representar um tipo de dados chamado *arranjos associativos*. Em um conjunto, temos várias chaves. Em um arranjo associativo, temos várias chaves e para cada chave temos um registro associado. A chave e o registro não precisam ser do mesmo tipo.

Considere o arranjo associativo abaixo representado por um `map` chamado `c`.

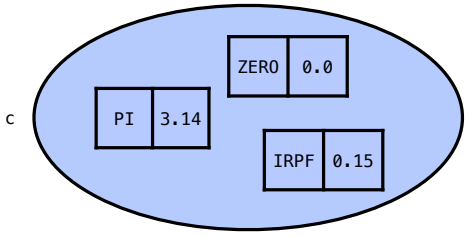
```
1 map<string,double> c;
```



O container terá chaves do tipo `string` e registros do tipo `double`.

Podemos inserir elementos facilmente neste conjunto de dados através do operador de subscrito. Colocamos a chave no subscrito e o registro à direita.

```
1 c["PI"] = 3.14;
2 c["ZERO"] = 0.0;
3 c["IRPF"] = 0.15;
```



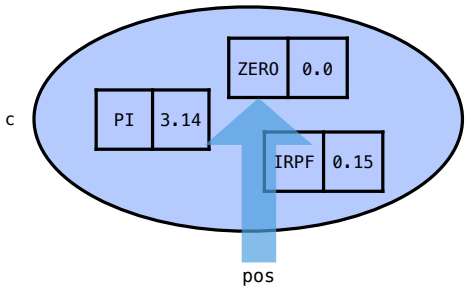
Esta estrutura é frequentemente chamada de *arranjo associativo* pois quando a utilizamos com o operador de subscrito é como se estivéssemos acessando um arranjo onde as posições são as chaves. Podemos, assim, de maneira simples retornar um valor.

```
1 cout << c["PI"] << endl;
```

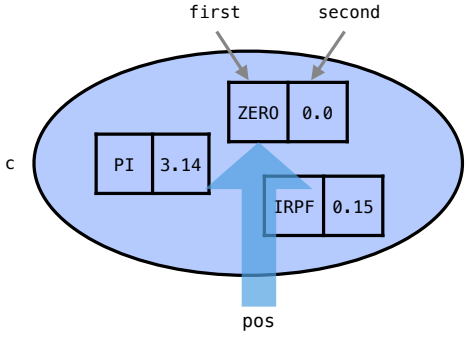
```
3.14
```

Assim como em outros containers, a função find é utilizada para encontrar um elemento pela chave. É retornado um iterador para este elemento.

```
1 map<string,double>::iterator pos;
2 pos = c.find("ZERO");
```



Ao desreferenciar um iterador de um `map`, é retornada uma estrutura com duas variáveis: `first` e `second`.



Podemos guardar toda a estrutura ou utilizar diretamente os campos `first` e `second`.

```
1 cout << pos->first << ":␣" << pos->second << endl;
2 // comando equivalente
3 cout << (*pos).first << ":␣" << (*pos).second << endl;
```

```
ZERO: 0
ZERO: 0
```

Assim como em outros containers, ao se iterar pelos elementos, estes ocorrem no conjunto ordenados pela chave.

```
1 for (pos = c.begin(); pos != c.end(); ++pos){
2     cout << pos->first << ":␣" << pos->second << endl;
3 }
```

```
IRPF: 0.15
PI: 3.14
ZERO: 0
```

21.1.4 Multimap

Assim como os containers `set` e `multiset`, o container `map` também contém um equivalente que aceita entradas repetidas de um elemento: o `multimap`.

Todas as funções específicas para `multiset`, como `lower_bound`, `upper_bound` e `count` podem ser utilizadas também no `multimap`.

Um recurso especificamente interessante em um `multimap` é que réplicas de uma chave podem ter registros diferentes, o que diferencia os elementos com chaves replicadas.

21.1.5 Como funcionam

Apesar dos tipos distintos de containers associativos, todos eles se baseiam usualmente em uma estratégia similar: Árvores Binárias de Pesquisa ou BST (*Binary Search Tree*).

As árvores binárias são estruturas eficientes para armazenar informação. Elas permitem o acesso direto a elementos de forma eficiente. As operações de inserção e remoção são também feitas com um baixo custo nestas estruturas.

Assim como as listas encadeadas, as árvores são baseadas em células que guardam elementos e ponteiros para outras células. Células à esquerda da árvore contêm elementos menores e células à direita contém elementos maiores.

A Figura 21.2 apresenta um exemplo de árvore binária de pesquisa com números inteiros. Veha que para qualquer célula da árvore, a célula à esquerda contém um elemento menor, como ocorre com os elementos 350 e 300. Veja que para qualquer célula da árvore, a célula à direita contém um elemento maior, como ocorre com os elementos 350 e 500.

O mesmo ocorre para subárvores à esquerda e à direita de uma célula, como apresentado na Figura 21.3. Podemos ver na Figura 21.3(a) que toda célula R tem uma subárvore E à esquerda e outra subárvore D à direita. Todos os elementos em E são menores que R. Todos os elementos em D são menores que R. Isto vale para qualquer célula, como podemos ver na Figura 21.3(b).

Em um container do C++, a árvore é representada através da alocação de memória para várias células representadas com `struct`. De modo similar às células utilizadas nos containers `list`, cada célula de uma árvore contém um elemento e dois ponteiros para outras células.

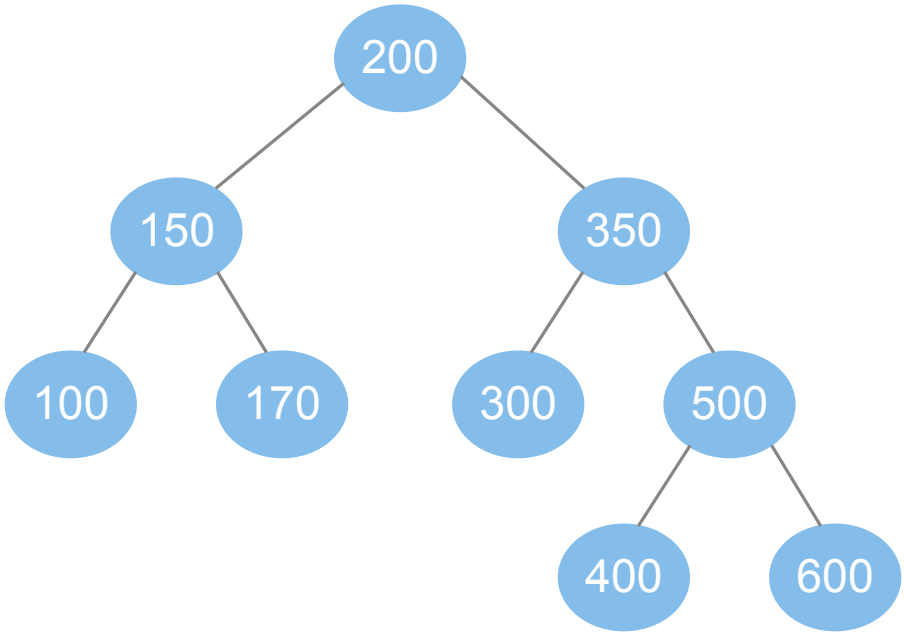
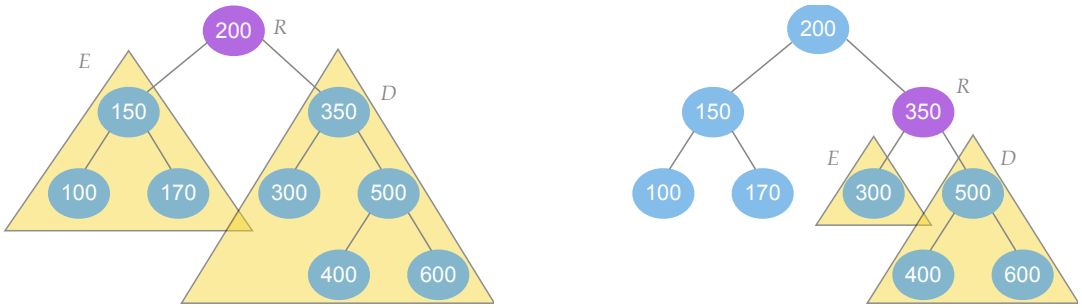


Figura 21.2: Exemplo de Árvore Binária de Pesquisa

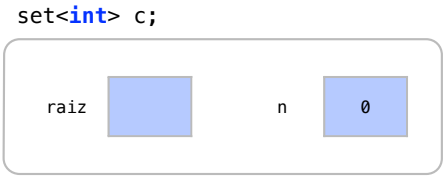


(a) Subávores 1 (b) Subávores 2

Figura 21.3: Toda célula *R* tem elementos menores em uma subárvore à esquerda *E* e elementos maiores em uma subárvore à direita *D*.

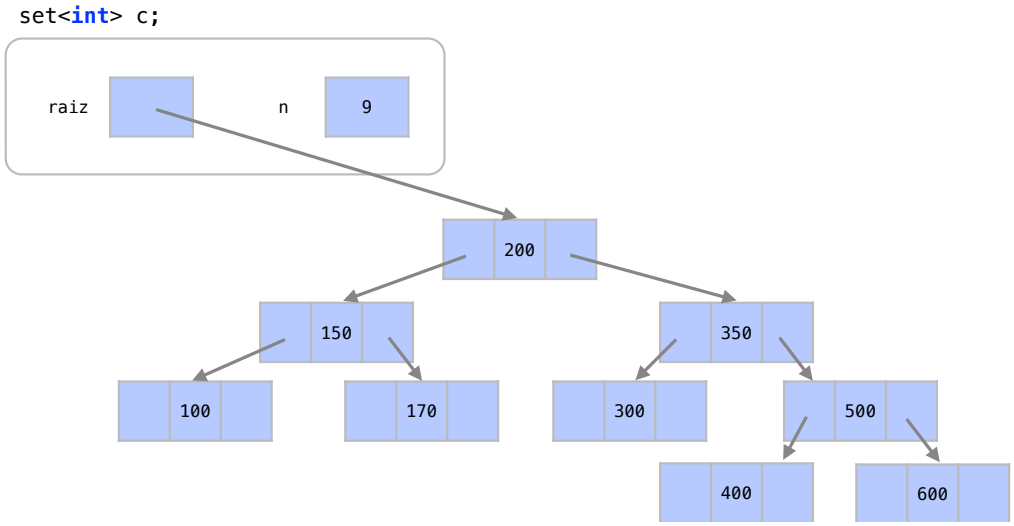
Um objeto do tipo `set` contém um ponteiro para a célula alocada como raiz da árvore (variavél chamada aqui de `raiz`) e uma variável chamada aqui de `n` para contar o número de elementos na estrutura.

```
1  set<int> c;
```



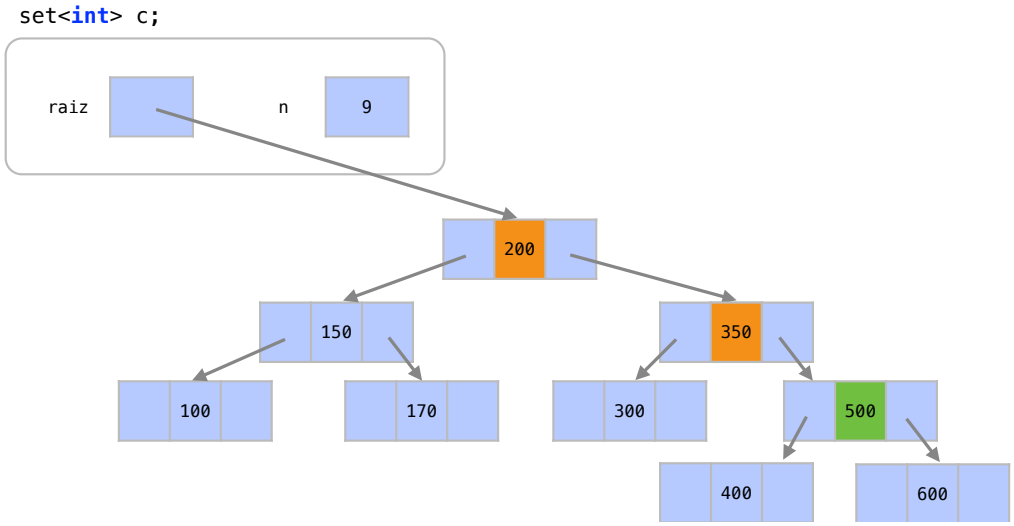
Quando os elementos são inseridos, os ponteiros das células apontam para os elementos raiz das árvores à esquerda e à direita.

```
1 c.insert(200);
2 c.insert(150);
3 c.insert(350);
4 c.insert(100);
5 c.insert(170);
6 c.insert(300);
7 c.insert(500);
8 c.insert(400);
9 c.insert(600);
```



Para procurar um elemento, basta ir à raiz da árvore e fazer uma comparação para saber em qual subárvore está o elemento. Ao descobrirmos em qual subárvore está o elemento, vamos a esta subárvore e repetimos o processo em sua raíz. Se a raíz da árvore é o elemento que procuramos, terminamos a busca. Se tal raíz não é encontrada, o elemento não está no conjunto.

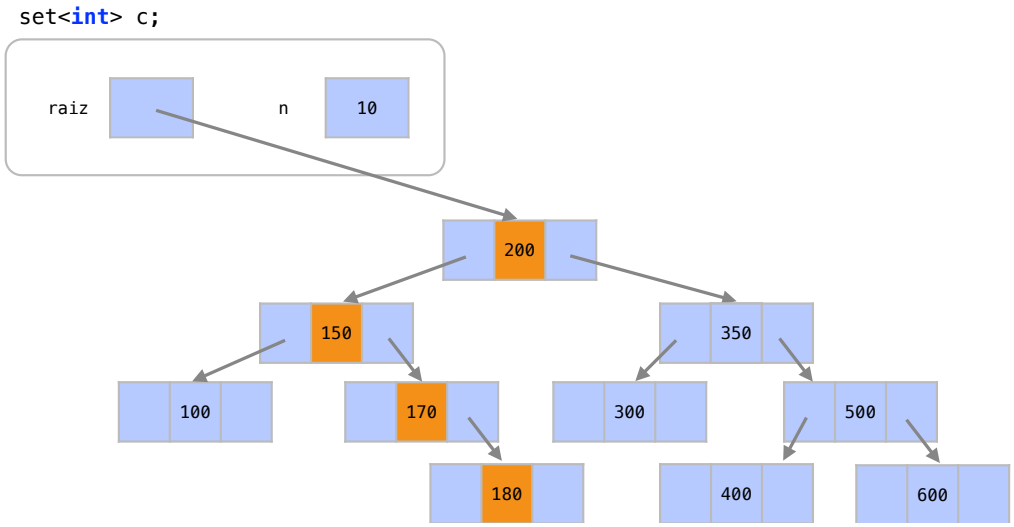
```
1 pos = c.find(500);
```



Ao procurar o elemento 500, vamos à raiz da árvore, que é 200. Assim, sabemos que o elemento, se existente, estará na subárvore à direita. O mesmo ocorre repetidamente até que encontramos o elemento 500.

Para inserir um elemento, um processo similar é utilizado. Suponha que queremos inserir o elemento 180.

```
1 c.insert(180);
```

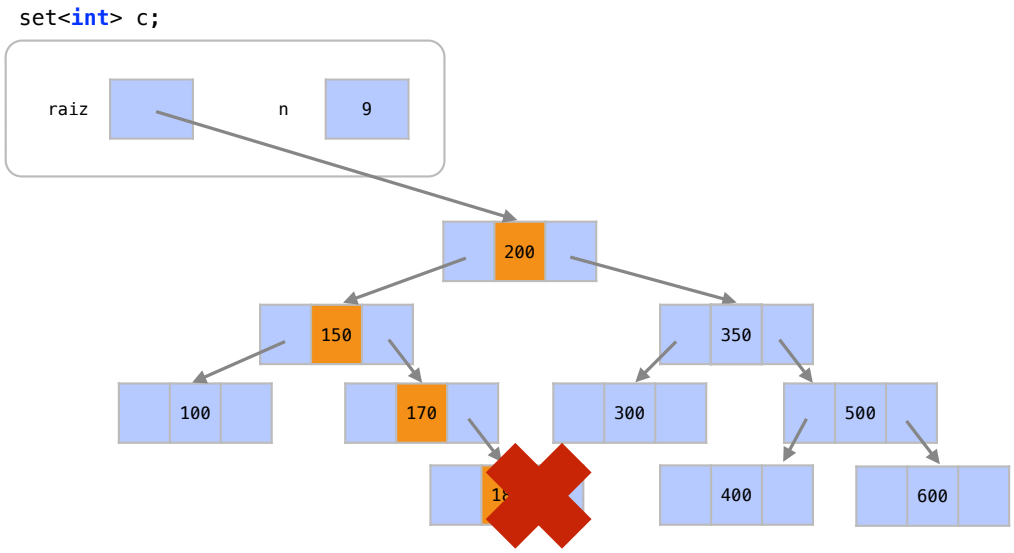


Percorremos as células da árvore como fazemos em uma pesquisa, até encontrar onde o elemento 180 está ou deveria estar. Basta criar uma célula para o elemento e inserí-lo nesta posição.

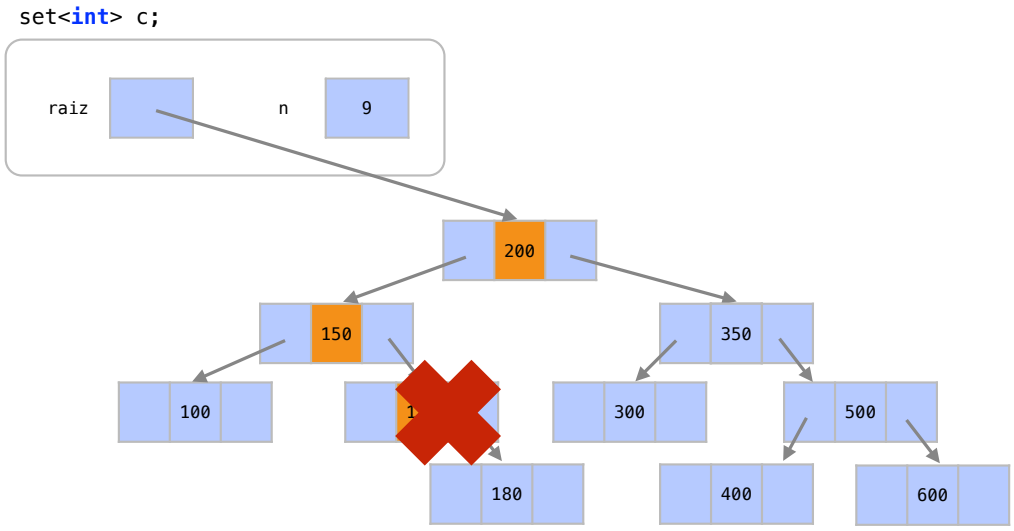
Para remover um elemento da árvore, temos um problema um pouco mais complicado, que pode ocorrer em 3 casos.

Caso 1) Para remover um nó folha, apenas procuramos por ele e o removemos.

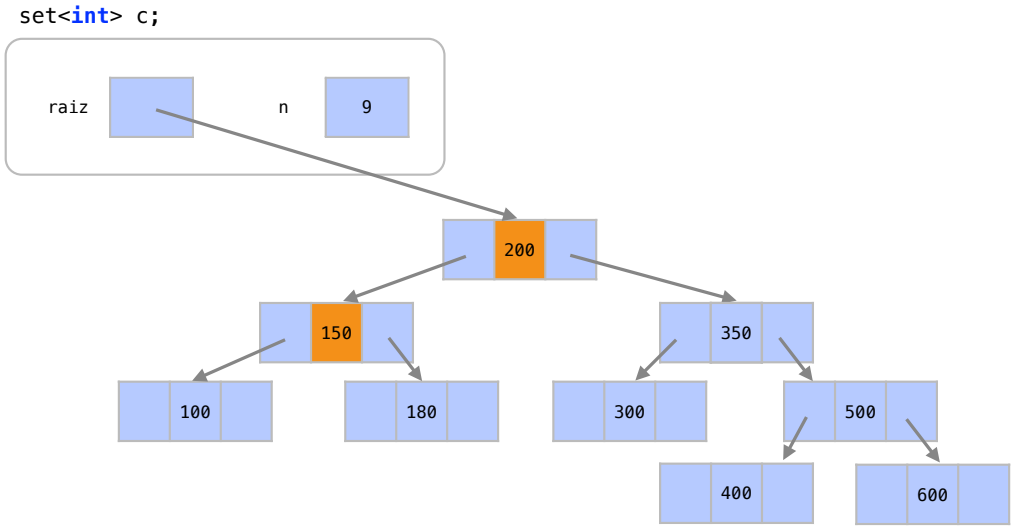
```
1 c.erase(180);
```



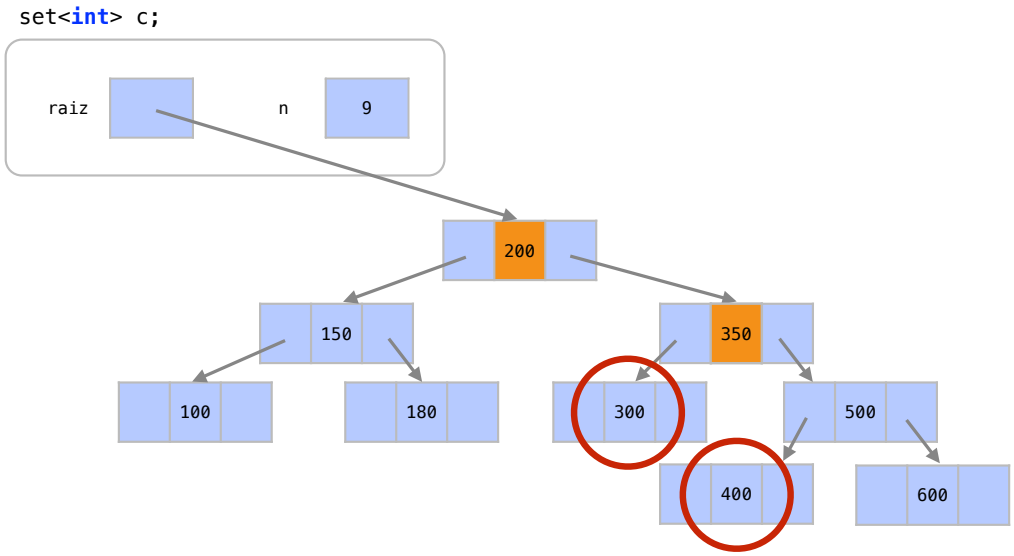
Caso 2) Para remover um nó que tem apenas um filho, temos um processo de dois passos.
Caso 2 - Passo 1 - Removemos o nó.



Caso 2 - *Passo 2* - Substituímos o nó por seu filho.

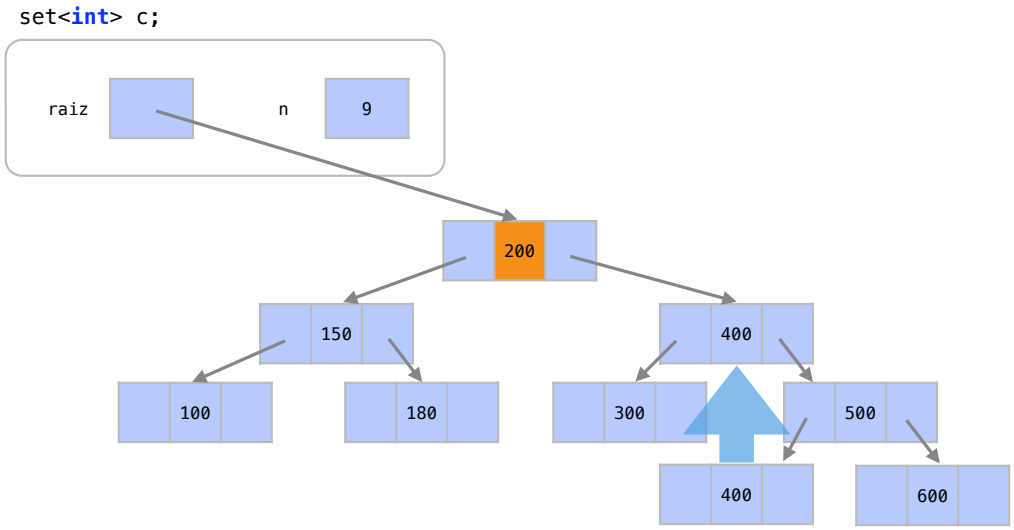


Caso 3) Para remover um nó com 2 filhos temos 3 passos. *Caso 3* - *Passo 1* - Em vez de remover o nó, procuramos seu sucessor ou predecessor de acordo com seu valor.

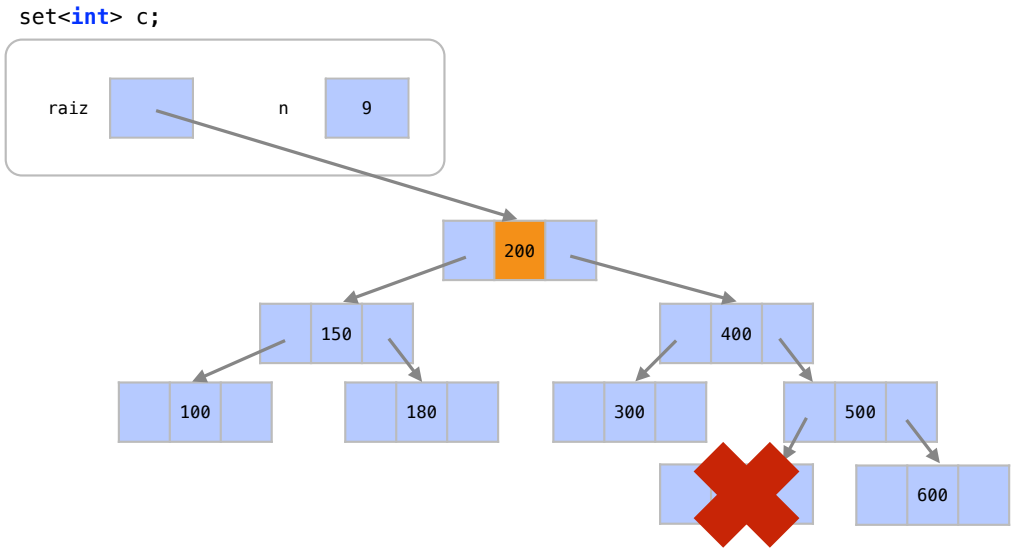


Neste caso, vamos utilizar o sucessor de 350, que é sempre o elemento mais à esquerda da árvore à direita. O predecessor, de modo análogo, seria o elemento mais à direita da árvore à esquerda.

Caso 3 - Passo 2 - O sucessor toma a posição do item que queremos remover.



Caso 3 - Passo 3 - Removemos então o sucessor marcado. Esta operação de remoção é recursiva e pode cair em qualquer um dos 3 casos novamente. A remoção do sucessor, neste caso, caiu no caso 1. O processo é assim encerrado.

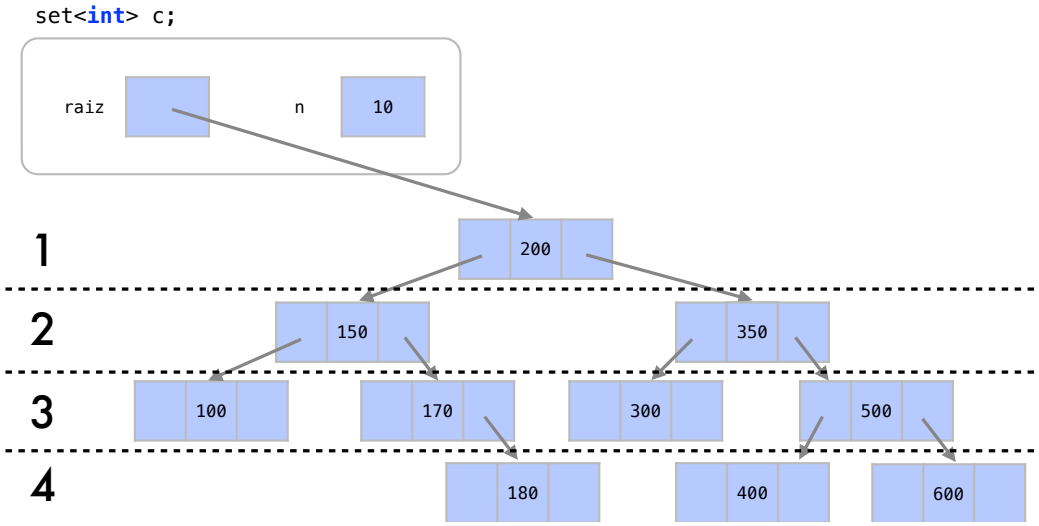


Balenceamento

Apesar da eficiência média das árvores apresentadas, a ordem de inserção dos elementos pode afetar o desempenho das árvores. Os elementos podem ser inseridos em uma ordem tal que a árvore final tenha muitos níveis e seja pouco eficiente.

Suponha a árvore gerada internamente pela inserção dos elementos na ordem apresentada:

```
1  c.insert(200);
2  c.insert(150);
3  c.insert(350);
4  c.insert(100);
5  c.insert(170);
6  c.insert(300);
7  c.insert(500);
8  c.insert(180);
9  c.insert(400);
10 c.insert(600);
```



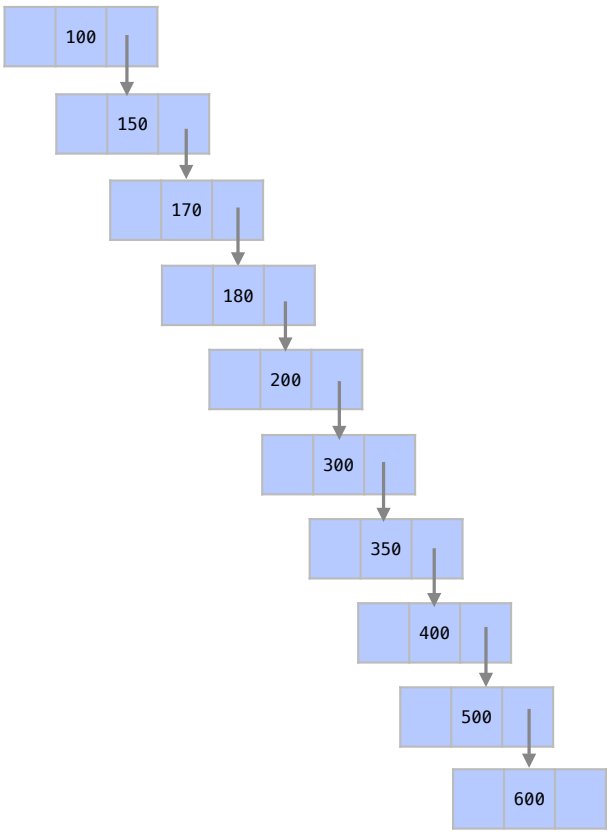
As linhas pontilhadas representam os níveis da árvore. Para encontrar um elemento nesta árvore, no máximo 4 comparações são feitas. Uma em cada nível. Esta é uma árvore com-

pletamente **balenceada**, onde o número de níveis é o mínimo possível para a quantidade de elementos.

O número de níveis em uma árvore balanceada com n elementos é $O(\log n)$ ($1 = \lg 2, 2 = \lg 4, 3 = \lg 8, 4 = \lg 16, \ldots$). Sendo o custo de se encontrar um elemento igual ao número de níveis, este custo também é $O(\log n)$.

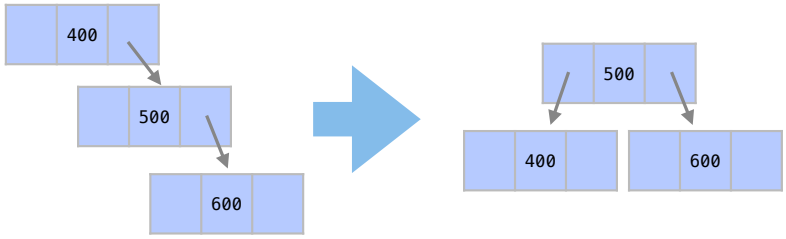
Agora suponha esta árvore gerada pela inserção dos mesmo elementos em ordem crescente:

```
1  c.insert(100);
2  c.insert(150);
3  c.insert(170);
4  c.insert(180);
5  c.insert(200);
6  c.insert(300);
7  c.insert(350);
8  c.insert(400);
9  c.insert(500);
10 c.insert(600);
```



Encontrar um elemento na árvore pode custar até $n = 10$ comparações. O número de níveis em uma árvore totalmente desbalanceada com n elementos é $O(n)$. Sendo o custo de se encontrar um elemento igual ao número de níveis, este custo também é $O(n)$. Perceba como em uma árvore desbalanda a estrutura de dados se parece com uma lista encadeada.

Para que este problema de desbalanceamento não ocorra, a STL prevê estratégias de balanceamento para que o custo das operações sobre conjunto seja sempre $O(\log n)$. Esta estratégias são sempre aplicadas quando inserimos um elemento a mais no conjunto. Estas estratégias usualmente envolvem rotações nos elementos inseridos.



Em C++, porém, estas estratégias variam muito de acordo com o compilador. Não faz parte do escopo deste livro discutir todas estas estratégias, que podem ser encontradas em qualquer bom livro didático sobre estruturas de dados.

21.1.6 Análise

Em uma árvore binária, cada comparação feita pelo algoritmo que busca um elemento divide o número de soluções possíveis pela metade, de modo similar a uma busca binária. Assim, o custo médio de inserção, remoção e pesquisa em todos os casos é $O(\log n)$.

Como a STL contém estratégias de balanceamento, o custo das operações continua $O(\log n)$ mesmo no pior caso. Na verdade, mesmo para árvores que não tem estratégias de balanceamento, o custo médio ainda é $O(\log n)$ para inserções feitas aleatoriamente. Seu pior caso, porém, é $O(n)$.

21.2 Containers Associativos Desordenados

Os containers ordenados que apresentamos até o momento foram `set`, `multiset`, `map` e `multimap`. Do ponto de vista de comandos, podemos criar os containers desordenados com os tipos de dados equivalentes `unordered_set`, `unordered_multiset`, `unordered_map` e `unordered_multimap`.

Também do ponto de vista de comandos, a utilização dos containers desordenados se dá de forma idêntica a seus pares ordenados. Porém, ao percorrer estes containers com um iterador, se perceberá que internamente os elementos não ficam estruturados em ordem. A quebra desta restrição de ordenação leva a um ganho em velocidade de busca.

Veja o seguinte exemplo:

```
1  #include <iostream>
2  #include <string>
3  // <unordered_map> em vez <map>
4  #include <unordered_map>
5
6  using namespace std;
7
8  int main(){
9      // unordered_map<string, int> em vez de map<string,int>
10     unordered_map<string, int> meses;
11     meses["janeiro"] = 31;
12     meses["fevereiro"] = 28;
13     meses["marco"] = 31;
14     meses["abril"] = 30;
15     meses["maio"] = 31;
16     meses["junho"] = 30;
17     meses["julho"] = 31;
```

```
18      meses["agosto"] = 31;
19      meses["setembro"] = 30;
20      meses["outubro"] = 31;
21      meses["novembro"] = 30;
22      meses["dezembro"] = 31;
23      cout << "Setembro␣␣->␣";
24      cout << meses["setembro"] << endl;
25      cout << "Abril␣␣␣␣␣->␣";
26      cout << meses["abril"] << endl;
27      cout << "Dezembro␣␣->␣";
28      cout << meses["dezembro"] << endl;
29      cout << "Fevereiro␣->␣";
30      cout << meses["fevereiro"] << endl;
31      return 0;
32  }
```

A única diferença entre este programa e um programa no qual utilizamos um `map`, é a declaração do tipo de dados na linha 9 e o cabeçalho correspondente na linha 11. Nos dois casos criamos um arranjo associativo que tem como chave o nome dos meses e como registro o número de dias neste mês.

Do ponto de vista de resultados, os dois programas (`map` e `unordered_map`) seriam equivalentes e mostrariam o número de dias nos meses indicados.

```
Setembro -> 30
Abril -> 30
Dezembro -> 31
Fevereiro -> 28
```

Internamente, porém, sabemos que os dados do `unordered_map` não estão ordenados.

Veja agora este exemplo onde os elementos do container são percorridos nas linhas 22 a 25. Veja como os elementos do `unordered_map` não estão em ordem quando percorridos.

```
1   #include <iostream>
2   #include <string>
3   #include <unordered_map>
4
5   using namespace std;
6
7   int main(){
8       unordered_map<string, int> meses;
9       meses["janeiro"] = 31;
10      meses["fevereiro"] = 28;
11      meses["marco"] = 31;
12      meses["abril"] = 30;
13      meses["maio"] = 31;
14      meses["junho"] = 30;
15      meses["julho"] = 31;
16      meses["agosto"] = 31;
17      meses["setembro"] = 30;
18      meses["outubro"] = 31;
```

```
19      meses["novembro"] = 30;
20      meses["dezembro"] = 31;
21      unordered_map<string, int>::iterator i;
22      for (i = meses.begin(); i!=meses.end(); ++i){
23          cout << i->first << ":␣";
24          cout << i->second << endl;
25      }
26      return 0;
27  }
```

Temos a seguinte saída para o programa.

```
dezembro:  31
novembro:  30
setembro:  30
outubro:  31
agosto:  31
junho:  30
maio:  31
fevereiro:  28
abril:  30
julho:  31
marco:  31
janeiro:  31
```

Se utilizarmos um container `map` no mesmo exemplos, os elementos do exemplo estariam em ordem em relação à chaves. Como a chave é do tipo `string`, eles estariam em ordem alfabética.

```
abril:  30
agosto:  31
dezembro:  31
fevereiro:  28
janeiro:  31
julho:  31
junho:  30
maio:  31
marco:  31
novembro:  30
outubro:  31
setembro:  30
```

Os elementos do `unordered_map` estão em ordem arbitrária quando percorremos o container, mas claramente isto não é um problema para o tipo de dado que estamos guardando. Ter os meses do ano em ordem alfabética é de pouca utilidade na maioria das aplicações. Por isto, podemos abrir mão da ordenação interna dos dados caso isto gere um ganho de tempo nas consultas.

21.2.1 Como funcionam

Assim como os containers ordenados, todos os containers desordenados são também baseados em estruturas de dados similares. Estes containers desordenados são baseados estruturas de

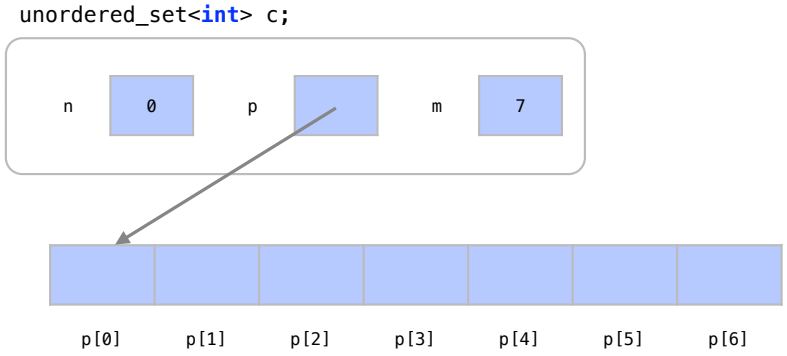
dados chamadas **tabelas *hash***. Estas se baseiam em estratégias de transformação de chave, o que é chamado usualmente de *hashing*.

As estruturas baseadas em transformação de chave utilizam um arranjo para guardar os elementos do conjunto. Quando uma chave é pesquisada, esta chave é transformada em um número através de uma operação de transformação de chave. O número retornado representa uma posição em um arranjo, onde o elemento está guardado.

Considere um `unordered_set` que representa um conjunto desordenado.

```
1 unordered_set<int> c;
```

Temos neste objeto uma variável `n` que guarda o número de elementos no container, um ponteiro chamado aqui de `p` para um arranjo alocado na memória, e o tamanho deste arranjo chamado aqui de `m`.

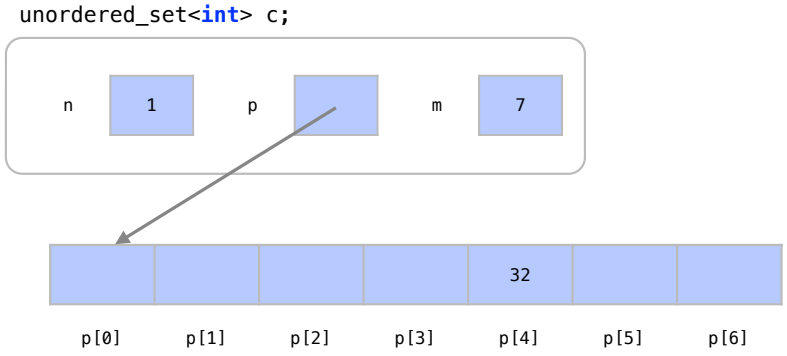


Este arranjo alocado na memória e apontado por `p` é chamado de **Tabela Hash**. É nele que os elementos serão inseridos. O tamanho inicial desta tabela depende da implementação existente no compilador. Vamos supor aqui que a tabela hash tenha espaço alocado `m` para 7 elementos.

Suponha agora que queremos inserir na tabela o elemento 32.

```
1 c.insert(32);
```

A função de transformação $h(x)$ deverá transformar esta chave 32 em uma posição do arranjo. A função de transformação $h(x)$ mais comum para números inteiros é $h(x) = x\%m$, onde m é o tamanho da tabela hash e x é o valor da chave. Em nosso exemplo, teríamos então que $h(32) = 32\%7 = 4$. Assim, a função de transformação indica que o elemento 32 deve ser inserido na posição `p[4]` da Tabela Hash.



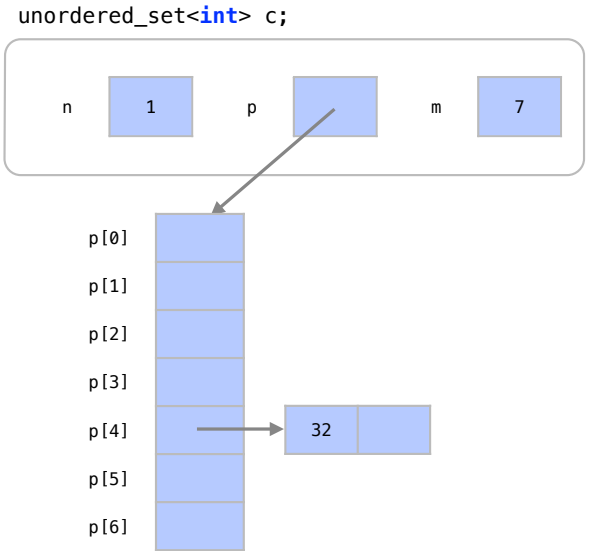
21.2.2 Tratamento de Colisões

Um problema óbvio com tabelas *hash* é que a transformação de chave pode querer colocar um outro elemento na mesma posição da tabela. Por exemplo, se tentássemos inserir a chave 46, teríamos que $h(46) = 46\%7 = 4$. A posição `p[4]`, porém, já está tomada pelo elemento 36.

A estratégia mais comum para tratamento de colisões é transformar este arranjo de elementos apontado por p em um arranjo de ponteiros para células de listas encadeadas de elementos.

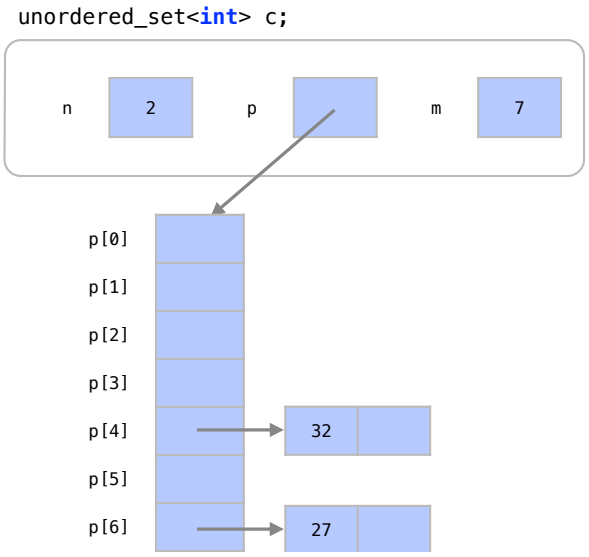
Por exemplo, ao se inserir o elemento 32, ele vai para a posição $h(32) = 4$ da tabela hash como um elemento em uma célula de uma lista encadeada.

```
1 c.insert(32);
```



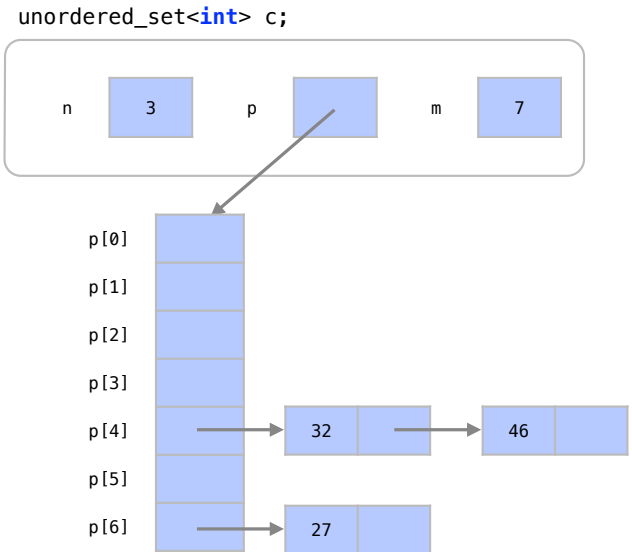
Cada elemento fica em uma célula que é composta de um elemento e um ponteiro para uma possível próxima célula. Por exemplo, ao se inserir o elemento 27, temos que $h(27) = 27\%7 = 6$.

```
1 c.insert(27);
```

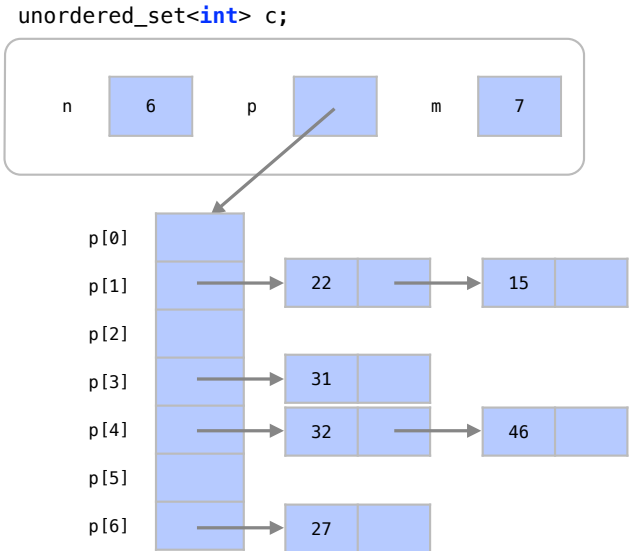


Em caso de colisão, o elemento é simplesmente colocado como último da lista.

```
1 c.insert(46);
```




```
1 c.insert(31);
2 c.insert(22);
3 c.insert(15);
```



21.2.3 Análise

A maior vantagem de tabelas hash é a sua eficiência em relação a custo médio para pesquisa, inserção e remoção. Para uma tabela onde os dados estão bem distribuídos o custo de qualquer operação na tabela é $O(1)$.

O STL já inclui estratégias para que os dados fiquem sempre bem distribuídos na tabela. Estas estratégias incluem funções de *hashing* que garantam uma boa distribuição dos dados e alocação de maiores tabelas caso a carga de elementos esteja muito alta.

Para a estratégia de tratamento colisão apresentada, no pior caso, se todos os elementos forem para a mesma posição, o custo de cada operação seria $O(n)$. Porém, além da probabilidade deste caso ser desprezível, um tratamento de colisão através de árvores em vez de listas pode levar este pior caso a $O(\log n)$, fazendo com que a tabela hash seja, na pior das hipóteses, tão eficiente quanto uma árvore.

Mais ainda, é possível garantir de várias formas uma boa distribuição dos dados na tabela. Todas as tabelas hash alocam espaço para tabelas maiores quando o número de elementos cresce. Usualmente uma nova tabela maior é criada quando o número de elementos é maior que 70% do tamanho da tabela. Esta proporção é chamada de **fator de carga**.

Quando o tamanho da tabela aumenta uma operação $O(n)$ copia os elementos para a nova tabela, causando um custo de realocação de memória. Para evitar isto, é comum manter duas tabelas hash para representar um conjunto de dados. Sempre que um elemento novo é inserido na tabela nova, um elemento da tabela antiga é transferido para a tabela nova com custo $O(1)$. Porém, nesta estratégia, a busca de um elemento, apesar de ser ainda $O(1)$, sempre precisará ser feita em duas tabelas.

A maior desvantagem das tabelas hash é que os dados ficam desordenados na tabela. Assim, para retornar todos os itens em ordem seria necessário colocar tudo para uma outra estrutura ordenada ou ordená-los em uma estrutura de sequência. Qualquer uma destas opções terá um custo $O(n \log n)$. Por isto, esta é uma estrutura a ser utilizada apenas quando os dados em ordem realmente não são necessários.

21.3 Análise dos Containers Associativos

O diagrama apresentado na Tabela 21.1 apresenta os principais fatores a se considerar ao escolher um container para armazenamento de dados.

A posição dos elementos em uma sequência é importante?				
Sim			Não	
Containers Sequenciais			Containers Associativos	
Onde serão feitas as inserções?			**Precisa manter os itens ordenados?**	
Fim	Meio	Início	Sim	Não
`vector`	`list`	`deque`	Associativo ordenado	Associativo desordenado

Tabela 21.1: Fatores a se considerar ao escolher uma categoria de container para representar uma estrutura de dados.

Discutimos na Seção 20 a diferença de desempenho entre os containers sequenciais. Se ignorarmos a ordem dos elementos em um container sequencial, é claro que poderíamos utilizar um container sequencial qualquer para representar conjuntos. Para inserir um elemento no conjunto, colocaríamos o elemento fim da sequência, em tempo $O(1)$. Para remover um elemento ou fazer um pesquisa no conjunto, percorríamos todo o arranjo em busca do elemento, com custo $O(n)$. Veja que qualquer container teria o mesmo custo para estas operações (inserir ao fim da sequência e percorrer os elementos). Por isto, é usual utilizamos um container `vector` para este fim.

Para retornar os dados em ordem, um algoritmo de ordenação deve ser aplicado à sequência. É possível também representar conjuntos com um container sequencial que seja mantido ordenado, o que garantiria uma busca $O(\log n)$ com uma busca binária. O custo de se inserir e posicionar um elemento na posição correta, porém, seria $O(n)$. Esta seria uma melhor solução em aplicações menos dinâmicas.

A Tabela 21.2 apresenta o custo assintótico destas operações de representação de conjuntos para diferentes containers, incluindo os sequenciais. Vemos como a representação de conjuntos com containers sequenciais é desvantajosa do ponto de vista assintótico. Entre os containers associativos, se não precisamos ter os elementos em ordem, o container desordenado é sempre mais vantajoso. Se precisamos percorrer os elementos em ordem, teríamos um custo $O(n \log n)$

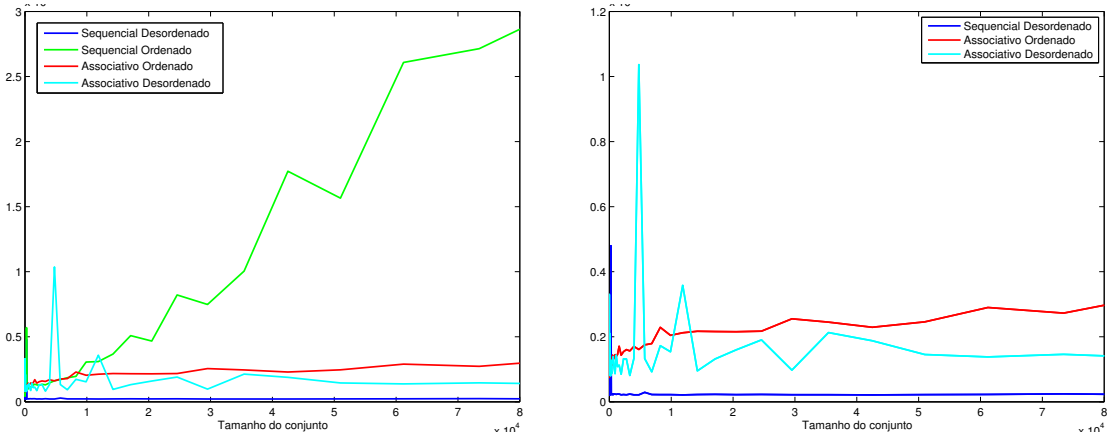
para ordenar os elementos. O custo desta ordenação pode ser ou não compensado pela operação que faremos nos elementos ao percorré-los.

	Container Sequencial		**Container Associativo**	
	Desordenado	Ordenado	Ordenado	Desordenado
Inserção	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
Remoção	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Pesquisa	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Percorrer em ordem	$O(n \log n)$	$O(n)$	$O(n)$	$O(n \log n)$

Tabela 21.2: Custo para representação de conjuntos através das estruturas de dados e containers mais comuns.

A Figura 21.4 apresenta o custo de inserção de um elemento em um container que representa conjuntos. Foram utilizados os containers `vector`, `set` e `unordered_set` para armazenar dados do tipo `int` neste experimento. O custo de uma inserção em um container sequencial ordenado cresce em proporção com o número de elementos, já que precisamos posicionar o elemento em sua posição correta da sequência. Mesmo com seu custo assintótico maior, para conjuntos com menos de 200 elementos o container sequencial ordenado é mais vantajoso que todos os containers associativos. Para conjuntos com menos de 5000 elementos o container sequencial ordenado é mais vantajoso que um container associativo ordenado.

Os containers desordenados sempre têm custo contante de inserção, sendo que um contaner sequencial desordenado tem sempre um custo mais baixo, já que precisa apenas inserir um elemento ao fim de um arranjo. Os containers associativos ordenados, baseados em árvores, tem um custo $O(\log n)$, que passa a ser discernível de um custo constante à medida conjuntos se tornam muito grandes. Se e apenas se a operação dominante da aplicação for inserções, pode ser vantajoso utilizar um container sequencial desordenado para representar conjuntos. No caso geral, como containers desordenados também têm custo constante de inserção, é provável que containers associativos continuem mais interessantes, devido ao custo de suas outras operações.



(a) Todos os containers

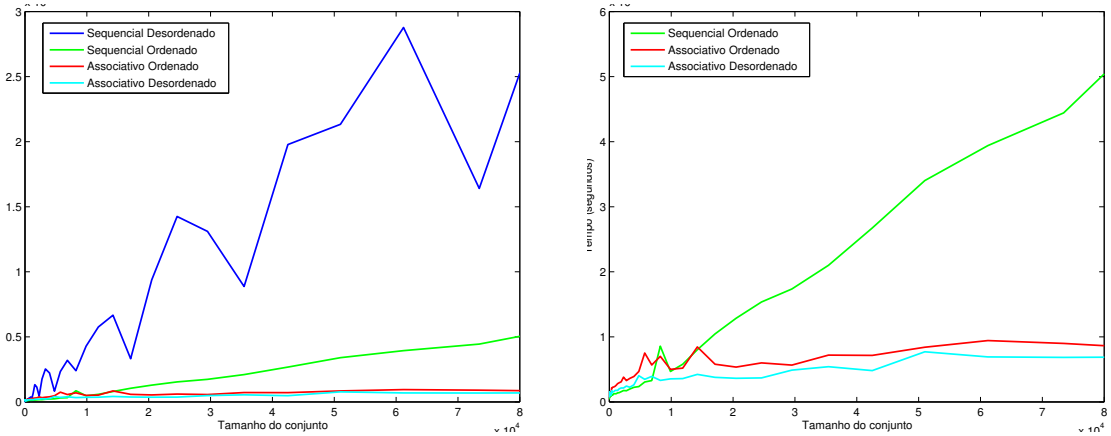
(b) Seleção de containers

Figura 21.4: Custo de inserção em um container que representa conjuntos.

A Figura 21.5 apresenta o custo de remoção de um elemento em um container que representa conjuntos. O custo de uma remoção em um container sequencial cresce em proporção com o número de elementos, já que precisamos reposicionar os elementos na sequência. Nestes containers, para toda operação de remoção, há uma operação de pesquisa. O container sequencial

desordenado é especialmente mais lento pois o custo de pesquisa do elemento a ser removido é $O(n)$. Em nossos experimentos, para conjunto pequenos, um container sequencial desordenado é mais vantajoso que um sequencial ordenado para menos de 250 elementos, que um associativo desordenado para menos de 320 elementos e que um associativo ordenado para menos de 600 elementos. Um container sequencial ordenado tem uma remoção mais rápida que um associativo desordenado para menos de 7000 elementos e que um associativo ordenado para menos de 7200 elementos.

Já os containers associativos têm um custo baixo de remoção muito mais baixo, sendo que a diferença entre containers associativos ordenados $O(\log n)$ e desordenados $O(1)$ se torna maior à medida que os conjuntos se tornam muito grandes. Containers associativos ordenados têm sempre uma eficiência menor que associativos desordenados para inserção de elementos.



(a) Todos os containers

(b) Seleção de containers

Figura 21.5: Custo de remoção em um container que representa conjuntos.

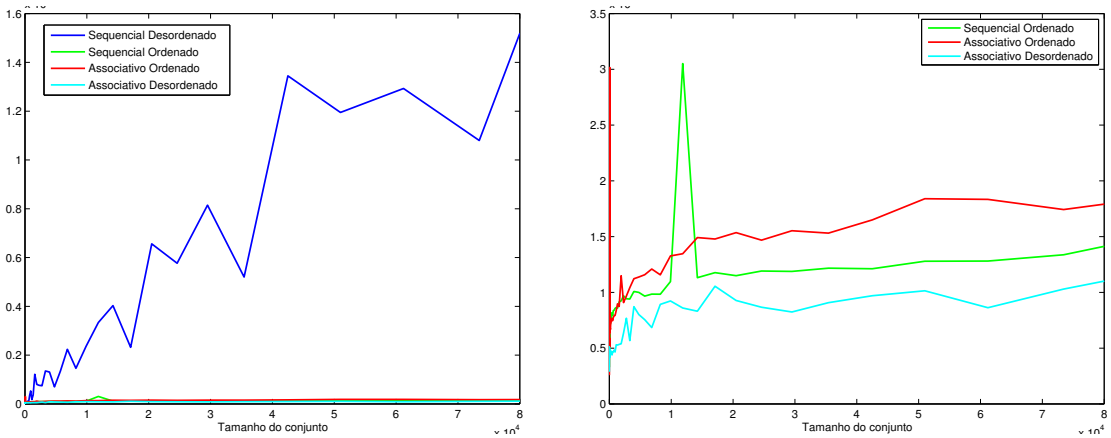
A Figura 21.6 apresenta o custo de pesquisa de um elemento em um container que representa conjuntos. Os containers sequenciais desordenados têm um custo alto $O(n)$ para este tipo de operação. Porém, em nossos experimentos, containers sequenciais desordenados podem ter consultas mais eficientes do que containers associativos desordenados para menos de 50 elementos, que containers associativos desordenados para menos de 200 elementos e que containers sequenciais ordenados para menos de 225 elementos. Para outros casos, containers associativos desordenados são sempre vantajosos para pesquisa. Nos casos onde há necessidade de ordenação, containers sequenciais ordenados são cerca de 50% mais rápidos do que containers associativos ordenados.

Entre os tipos de containers associativos ordenados e desordenados, como cada um destes dois utiliza a mesma estrutura com a mesma eficiência, nossa escolha depende de nossas necessidades em relação a características do conjunto a ser representado. A Tabela 21.3 apresenta as principais características dos containers associativos.

Em geral, os containers desordenados (os do tipo "`unordered`") têm operações menos custosas de inserção, remoção e pesquisa. Estas operações têm apenas custo $O(1)$. Mesmo assim, as operações dos containers ordenados, apesar de não tão eficientes quanto as dos containers desordenados, tem ainda um custo baixo: $O(\lg n)$.

Os containers desordenados, por outro lado, não têm os seus elementos estruturados internamente em ordem. No caso geral, seria necessária uma operação $O(n \log n)$ de ordenação dos elementos para que estes sejam acessados em ordem. Em um arranjo ordenado, os dados podem ser normalmente acessados em ordem com apenas um custo $O(n)$.

Em relação a outras características que podem influciar a decisão, os containers do tipo "`map`"



(a) Todos os containers

(b) Seleção de containers

Figura 21.6: Custo de pesquisa em um container que representa conjuntos.

Container	Custo operações	Itens em ordem	Registros	Itens repetidos
`set`	$O(\log n)$	Sim	Não	Não
`multiset`	$O(\log n)$	Sim	Não	Sim
`map`	$O(\log n)$	Sim	Sim	Não
`multimap`	$O(\log n)$	Sim	Sim	Sim
`unordered_set`	$O(1)$	Não	Não	Não
`unordered_multiset`	$O(1)$	Não	Não	Sim
`unordered_map`	$O(1)$	Não	Sim	Não
`unordered_multimap`	$O(1)$	Não	Sim	Sim

Tabela 21.3: Custo para representação de conjuntos através das estruturas de dados e containers mais comuns.

são os que têm registros associados às chaves e os do tipo "`multi`" são os que permitem registros duplicados.

21.3.1 Dicas de utilização de containers

É muito importante entender como os dados são organizador em um container. Mais ainda, é importante saber como é alocado espaço para elementos em um container. Os containers `vector`, `deque` e associativos desordenados precisam realocar arranjos sempre que o número de elementos cresce. Quando sabemos que já teremos vários elementos serão colocados nestes containers, ele contêm a função `c.reserve(n)` que permite já alocar espaço para um certo número `n` de elementos. Isto evita o custo de várias realocações de arranjo.

É também importante entender como os elementos são organizados dentro de um container. Entender como os dados são organizados no container garante que se entenda o número de operações envolvidas em cada função disponível para o container. Assim, entendendo a estrutura, não precisamos decorar o custo de cada operação

Use a função `c.empty()` para saber se um container está vazio. Esta função utiliza dados pré-processados para retornar uma resposta. Não use `c.size()==0` para saber se um container está vazio. Fazer isto envolve 2 operações: cálculo do número de elementos e comparação.

21.4 Exercícios

Exercício 21.1 Analise o código abaixo, que está incompleto. O código abaixo utiliza um container do tipo `set` para guardar um conjunto de siglas, que são do tipo `string`.

- Veja como o conjunto é criado com `set<string>`.
- Veja como siglas podem ser adicionadas ao conjunto com a função `insert`.
- Veja como a função `imprimeContainer` imprime todas as siglas de um `set`.
- Veja como a função `pesquisaSigla` utiliza a função `find` para pesquisar uma sigla.
- Veja como temos um laço com um menu dentro. As funções para o menu precisam ser implementadas.
- Veja que incluímos as bibliotecas `stdlib.h` e `time.h` neste código. Elas permitem geração de números aleatórios e medir tempo.
- Veja que criamos um arranjo dinamicamente com o tamanho pedido. (`new int[n]`)
- Veja que a função `rand()` gera números aleatórios que são inseridos no arranjo.
- Veja que temos algumas funções que não são utilizadas ainda.

```
1  #include <iostream> // biblioteca de entrada e saida
2  #include <string> // biblioteca para usar strings
3  // bibliotecas com os containeres
4  #include <set>
5  #include <map>
6  #include <unordered_map>
7  #include <algorithm>
8
9  using namespace std;
10
11 template <typename T> // função similar à do exercício
       anterior
12 void imprimeContainer (T const& x);
13
14 template <typename T> // Pesquisa uma sigla no container
15 void pesquisaSigla (T const& x);
16
17
18 int main()
19 {
20     // cria um set de strings chamado x
21     set<string> x;
22
23     // inserindo siglas no conjunto, um a um
24     x.insert("INSS"); // Instituto Nacional de Segurança
           Social
25     x.insert("CEP"); // Código de Endereçamento Postal
26     x.insert("FUNAI"); // Fundação Nacional do Índio
27     x.insert("IOF"); // Imposto sobre Operações de Crédito
28     x.insert("IR"); // Imposto de Renda
29     x.insert("EMBRATEL"); // Empresa Brasileira de
           Telecomunicações
30     x.insert("ONU"); // Organização das Nações Unidas
31     x.insert("SPC"); // Serviço de Proteção ao Crédito
```

```
32      x.insert("IBGE"); // Instituto Brasileiro de Geografia
           e Estatística
33
34      cout << "Elementos␣no␣conjunto␣inicial␣(" << x.size()
           << "):" << endl;
35      imprimeContainer(x);
36
37      // Menu com opções
38      string opcao;
39      do {
40          // Mostra o menu
41          cout << endl;
42          cout << "[1]␣Listar␣todas␣as␣siglas" << endl;
43          cout << "[2]␣Pesquisar␣uma␣sigla" << endl;
44          cout << "[3]␣Adicionar␣uma␣sigla" << endl;
45          cout << "[4]␣Remover␣uma␣sigla" << endl;
46          cout << "[5]␣Retornar␣o␣numero␣de␣siglas␣
               cadastradas" << endl;
47          cout << "[0]␣Sair␣do␣programa" << endl;
48          cout << "Digite␣uma␣opcao:␣" ;
49
50          // lê uma opção
51          cin >> opcao;
52
53          // executa uma função de acordo com a opção
54          if (opcao == "1"){
55              cout << "Listando␣todas␣as␣siglas" << endl;
56              imprimeContainer(x);
57          } else if (opcao == "2") {
58              cout << "Pesquisando␣uma␣sigla" << endl;
59              pesquisaSigla(x);
60          } else if (opcao == "3") {
61              cout << "Adicionando␣uma␣sigla" << endl;
62              cout << "Esta␣funcao␣ainda␣nao␣foi␣implementada
                   " << endl;
63          } else if (opcao == "4") {
64              cout << "Removendo␣uma␣sigla" << endl;
65              cout << "Esta␣funcao␣ainda␣nao␣foi␣implementada
                   " << endl;
66          } else if (opcao == "5") {
67              cout << "Retornando␣o␣número␣de␣siglas" << endl
                   ;
68              cout << "Esta␣funcao␣ainda␣nao␣foi␣implementada
                   " << endl;
69          } else if (opcao == "0"){
70              cout << "Saindo␣do␣programa␣" << endl;
71          } else {
72              cout << "Opcao␣invalida" << endl;
73          }
```

```
74      } while (opcao != "0");
75
76      return 0;
77  }
78
79  template <typename T>
80  void imprimeContainer (T const& x)
81  {
82      if (x.empty()){
83          std::cout << "(Vazio)" << endl;
84          return;
85      }
86      typename T::const_iterator pos;  // iterator que vai
            percorrer x
87      typename T::const_iterator end(x.end());  // iterador
            que aponta para a posição final
88      int i = 1;
89      for (pos=x.begin(); pos!=end; ++pos, ++i) {
90          std::cout << i << "-" << *pos << endl;
91      }
92      std::cout << std::endl;
93  }
94
95  template <typename T> // Pesquisa uma sigla no container
96  void pesquisaSigla (T const& x)
97  {
98      cout << "Digite␣o␣nome␣da␣sigla␣que␣deseja␣pesquisar:␣"
            << endl;
99      string pesquisa;
100     // Guarda o que o usuário digitou
101     cin >> pesquisa;
102     // Esta linha converte todas as letras para maiúsculas
103     std::transform(pesquisa.begin(), pesquisa.end(),
            pesquisa.begin(), ::toupper);
104     // cria-se um iterador
105     typename T::const_iterator i;
106     // procura o elemento e faz o iterador apontar para ele
107     i = x.find(pesquisa);
108     if (i!= x.end()){
109         cout << "A␣sigla␣" << pesquisa << "␣esta␣no␣
                conjunto␣de␣siglas" << endl;
110     } else {
111         cout << "A␣sigla␣nao␣foi␣encontrada" << endl;
112     }
113     return;
114 }
```

■

Exercício 21.2 Implemente as funções que ainda não foram implementadas. Use a função já implementada como referencia. ■

Exercício 21.3 Vamos agora transformar este programa em um dicionario de siglas, ou seja, para cada sigla pesquisada, teremos retornada sua definição.

- Para isto, troque a estrutura `set` por uma estrutura `map`. Os registros e as chaves desse `map` deverão ser do tipo `string`.
- A inserção de siglas no `map` é feita de maneira mais conveniente com o operador de subscrito `[]`.

■

Exercício 21.4 Com o dicionário de siglas implementado, troque a estrutura `map` por `unordered_map`.

- É necessário fazer mais alguma alteração para que o código funcione?
- Qual a desvantagem e qual a vantagem de se utilizar um `unordered_map`?

■

Exercício 21.5 A estrutura `unordered_map` não consegue retornar os itens em ordem crescente como o map. Porém, a estrutura `unordered_map` tem um ganho em eficiência para pesquisa e inserção. Se quisermos utilizar o `unordered_map` e mesmo assim quisermos retornar os itens ordenados, como podemos fazer isto manualmente? ■

Exercício 21.6 Neste cenário, onde tanto as funções de pesquisa em `unordered_map` e `map` retornam os itens ordenados, em quais casos seria vantagem utilizar `map` ou `unordered_map`? ■

Exercício 21.7 Implemente a alteração na função `imprimeContainer` para que ela imprima os itens de um `unordered_map` de maneira ordenada. Os elementos podem ser transferidos temporariamente para um `map` simples. ■

22. Adaptadores de Container

Os containers que vimos até agora são chamados de **containers de primeira classe**. Os adaptadores de container não são containers de primeira classe pois são apenas versões limitadas do containers de primeira classe para aplicações específicas.

Estas versões limitadas dos containers de sequência contêm apenas funções `push` e `pop`. As outras opções dos containers de sequência não estão disponíveis no adaptador de container. Estes containers também não podem ser percorridos com iteradores. Assim, a utilização de adaptadores de container é bem simples.

22.1 Stack

O container `stack` é utilizado para representar **pilhas** de elementos. Ele permite apenas a inserção ou remoção em uma extremidade do container de sequência. Este adaptador de container tem apenas as funções de `push` e `pop` para inserção e remoção. Estas funções, na verdade, apenas chamam as funções `push_back` e `pop_back` de um container de sequência original, que está sendo utilizado internamente para representar a pilha.

Um `stack` pode ser usado para implementar uma *pilha* com um `vector`, `list`, ou `deque`. Nos casos de omissão, um `deque` será utilizado por padrão. Neste adaptador de container, a função `top` obtém o elemento no topo da pilha, equivalente à função `back` dos containers de sequência.

22.1.1 Utilização

Para utilizar um `stack`, o cabeçalho `#include <stack>` precisa ser incluído em nosso programa. O tipo de dados `stack<int>` cria uma pilha de números inteiros. Para especificar que queremos a pilha representada com um `list`, utilizamos `stack<int, list<int> >`.

Considere a pilha de elementos abaixo.

```
1 stack<int> c;
2 c.push(7);
3 c.push(7);
```

```
4 c.push(2);
5 c.push(6);
```

Em uma pilha, só temos acesso ao elemento do topo de uma sequência. Neste caso, o elemento no topo da pilha é o 6.

A função push coloca um elemento no topo da pilha.

```
1 c.push(4);
```

A função pop retira um elemento do topo da pilha.

```
1 c.pop();
```

A função top retorna o valor do elemento no topo da pilha.

```
1 cout << c.top() << endl;
```

22.1.2 Como funciona

Internamente, um `stack` pode ser representado com um `vector`, `deque` ou `list`. Para um `vector` ou `deque`, seu funcionamento é muito similar.

Vimos como `vector` e `deque` são baseados em arranjos. Considere agora apenas o arranjo de um `vector` ou um `deque` representando uma pilha com um elemento.

```
1 stack<int> c;
2 c.push(2);
```

Quando inserimos mais um elemento no `stack`, os elementos são copiados para um arranjo com o dobro do tamanho para colocar o novo elemento.

```
1 c.push(3);
```

Para inserir mais um elemento, um arranjo com o dobro do tamanho precisa novamente ser alocado.

```
1 c.push(7);
```

Agora é possível inserir mais um elemento no topo da pilha sem aumentar o tamanho do arranjo.

```
1 c.push(1);
```

Para inserir o elemento 4, um arranjo maior precisa ser novamente alocado.

```
1 c.push(4);
```

Ao remover elementos do topo do `stack`, um arranjo menor não é necessariamente criado.

```
1 c.pop();
```


```
1 c.pop();
```


```
1 c.pop();
2 c.pop();
3 c.pop();
```

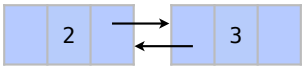
Como estudamos na Seção 18.2, ao se remover um elemento de um arranjo de um `vector` ou `deque`, os elementos não são fisicamente removidos do arranjo. No `vector`, apenas decrementamos a variável `n` que controla o número de elementos no container e no `deque`, decrementamos a variável `tras` que indica onde está o último elemento do container no arranjo. Isto representa um elemento sendo apagado do arranjo.

Já quando colocamos elementos em um `stack` representado por um `list`, apenas uma célula para este elemento é alocada.

```
1 stack<int,list<int>> c;
2 c.push(2);
```

Sempre que um elemento novo é inserido, o espaço exato para este elemento é alocado.

```
1 c.push(3);
```



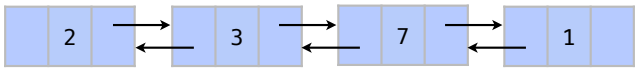

```
1 c.push(7);
```

Para cada elemento, porém, são alocados dois ponteiros, que acabam gastando memória extra $O(n)$.

```
1 c.push(1);
2 c.push(4);
```

Quando um elemento é removido, a memória gasta com este elemento é também desalocada. Assim, não temos o processo de realocação de memória $O(n)$ necessário para um `vector` ou `deque`. Todas as operações de inserção têm custo constante $O(1)$.

```
1 c.pop();
```




```
1 c.pop();
2 c.pop();
3 c.pop();
```

No fim do processo, se não há elementos, nenhuma memória está sendo gasta.

22.2 Queue

O adaptador `queue` é utilizado para representar **filas** de elementos. O que caracterizam filas é que elas permitem inserções (`push`) no fim da lista e retiradas do início (`pop`). Como faremos remoções no início da sequência, é melhor implementar filas com um `list` ou `deque`. Por padrão, o C++ utiliza um `deque`. Em um `queue`, as funções `front` e `back` dão acesso aos primeiro e último elementos.

22.2.1 Utilização

Para utilizar um `queue`, o cabeçalho `#include <queue>` precisa ser incluído. O tipo de dado `queue<int>` é utilizado para criar uma pilha de números inteiros. Para especificar que queremos a fila representada com um `list`, usamos `queue<int, list<int> >`.

A Figura 22.1 representa uma fila sendo representada por um `queue`. As funções para inserção e remoção de elementos têm a mesma sintaxe de um container `stack`. Porém, com a função `push`, novos elementos são sempre inseridos no fim da fila. Com a função `pop`, os elementos no início da fila são removidos primeiro.

Diferentemente de um `stack`, onde a função `top` acessava o elemento no topo da pilha, temos as funções `front` e `back` em uma fila representada por um `queue`. A função `back` retorna o elemento do fim da fila. A função `front` retorna o elemento do início da fila.

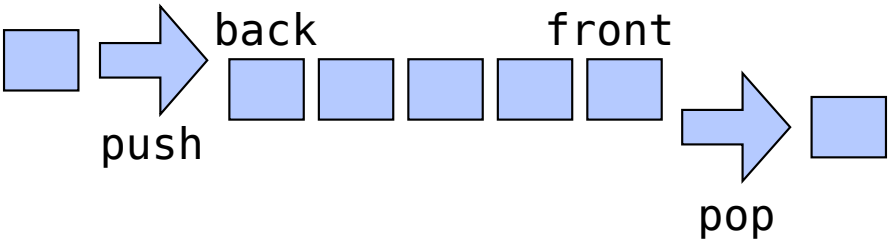


Figura 22.1: Exemplo de uma fila sendo representada por um `queue` e suas principais funções. Novos elementos são sempre inseridos no fim da fila. Os elementos no início da fila são removidos primeiro.

22.2.2 Como funciona

Como mencionamos, não se representa um `queue` com um `vector` pois este não tem uma função eficiente para remoção de seu primeiro elemento. Por exemplo, suponha o arranjo de um vector ou um deque representando uma pilha com um elemento.

```
1 c.push(2);
```

Todas as etapas iniciais de inserção funcionariam exatamente como em um `stack`. Quando inserimos mais um elemento no `stack`, os elementos são copiados para um arranjo com o dobro do tamanho para colocar o novo elemento.

```
1 c.push(3);
```

Para inserir mais um elemento, um arranjo com o dobro do tamanho precisa novamente ser alocado.

```
1 c.push(7);
```

Agora é possível inserir mais um elemento no topo da pilha sem aumentar o tamanho do arranjo.

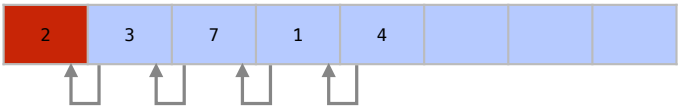
```
1 c.push(1);
```

Para inserir o elemento 4, um arranjo maior precisa ser novamente alocado.

```
1 c.push(4);
```

Agora, ao se remover um elemento, aquele elemento que está na frente da fila é removido. Como o `vector` não tem uma operação $O(1)$ para remover o primeiro elemento do container, todos os elementos do container precisam ser deslocados.

```
1 c.pop();
```



Este deslocamento resulta no primeiro elemento da fila removido. Porém, isto ocorre com um alto custo $O(n)$, o que torna o `vector` uma estrutura pouco apropriada para um `queue`.

```
1 c.pop();
```

Na remoção do próximo elemento, todos os elementos são deslocados novamente, com um custo $O(n)$.

```
1 c.pop();
```

A cada operação de remoção, um deslocamento $O(n)$ é executado.

```
1 c.pop();
```

Deste modo, o `vector` é uma estrutura muito ineficiente para `queue`.

```
1 c.pop();
```

Assim, utilizando um `vector` para representar o `queue`, sempre que inseríssemos um elemento no container teríamos um custo constante $O(1)$ e cada remoção teria um custo $O(n)$.

Para evitar este custo, é mais comum se representar um `queue` com um `deque`. Considere agora o arranjo de um `deque` que está representando um `queue`. Ao fim dos passos de inserção, teríamos um arranjo igual ao do `vector`.

```
1 queue<int> c;
2 c.push(2);
3 c.push(3);
4 c.push(7);
5 c.push(7);
6 c.push(1);
7 c.push(4);
```

Porém, em um `deque`, quando um elemento é removido da frente da fila, os elementos não são deslocados pois no `deque` apenas indicamos a posição na qual começa a fila.

```
1 c.pop();
```

Isto se repete para as outras remoções.

```
1 c.pop();
```


```
1 c.pop();
```

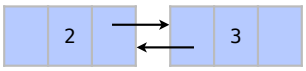

```
1 c.pop();
```

Assim, com um `deque` representando um `queue`, todas as operações são $O(1)$.

De outro modo, ao colocar elementos em um `stack` representado por um `list`, apenas uma célula para este elemento é alocada.

```
1 queue<int,list<int>> c;
2 c.push(2);
```


```
1 c.push(3);
```

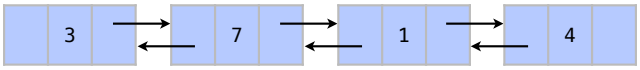


Para cada elemento, porém, são alocados dois ponteiros, que gastam memória extra $O(n)$.

```
1 c.push(7);
2 c.push(1);
3 c.push(4);
```

Quando um elemento é removido, ele é removido diretamente do início do `list` e a memória gasta com este elemento é desalocada.

```
1 c.pop();
```




```
1 c.pop();
2 c.pop();
3 c.pop();
```

Assim como em um `deque`, todas as operação são também $O(1)$.

22.3 Priority queue

Os adaptadores de container `priority_queue` são utilizados para representar **filas de prioridades**. Estes tipos de fila permitem inserir elementos de maneira que o maior elemento do container é removido a cada chamada de `pop`. Ele pode ser implementado com um `deque` ou `vector` (padrão).

22.3.1 Utilização

Assim como em outros adaptadores temos as funções `push` e `pop` para inserção e remoção. A função `push` insere um elemento na fila de prioridade. A função `pop` retira o maior elemento (chamado de elemento de maior prioridade). A função `top` consulta o elemento de maior prioridade.

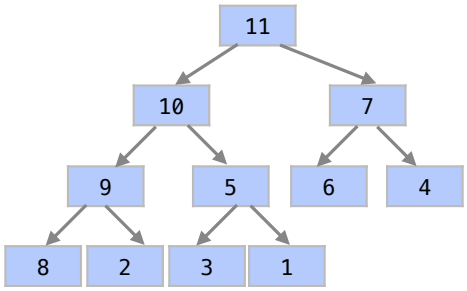
Para utilizar um `priority_queue`, o cabeçalho `#include <queue>` precisa ser incluído. O tipo de dados `priority_queue<int>` cria uma fila de prioridade com números inteiros.

22.3.2 Como funciona

As filas de prioridade ficam sempre organizadas em forma de arranjo. Assim fica organizado o arranjo de um `vector` que representa um `queue`.

Apesar de não estar ordenado, o primeiro elemento deste arranjo é sempre o maior elemento do container, no caso, o `11`.

Este arranjo, na verdade, representa uma árvore chamada **heap**. O heap é representado pelo arranjo de modo que os filhos de um elemento na posição `i` estejam nas posições `2i+1` e `2i+2`. Ou seja, apesar dos elementos estarem no arranjo, eles representam a árvore abaixo.

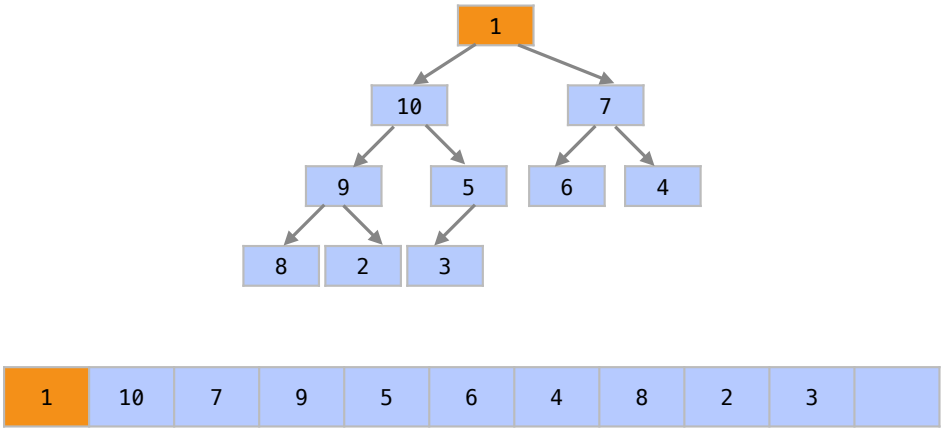


Por exemplo, os filhos do elemento `11` da posição `0` do arranjo são `10` e `7`, das posições `1` e `2` do arranjo. Os filhos do elemento `7`, da posição `2`, são `6` e `4`, das posições `5` e `6`.

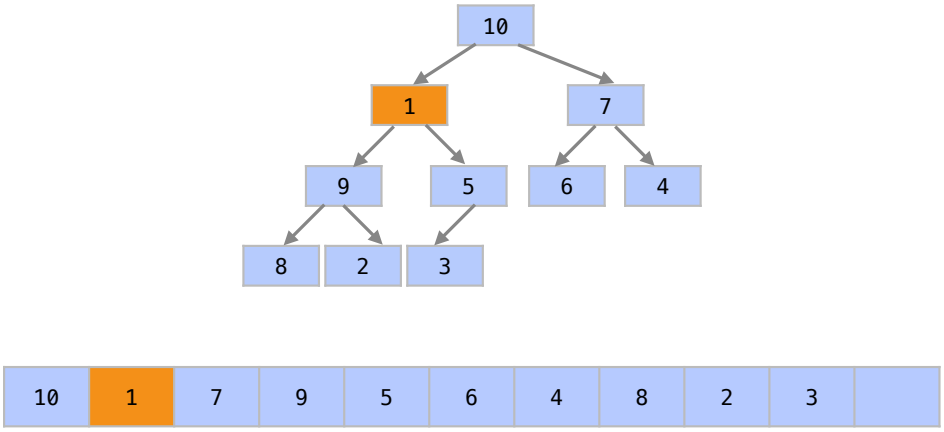
Uma propriedade importante destas árvores **heap** é que, apesar dos elementos não estarem ordenados, os filhos de um elemento sempre são menores que o pai. Esta propriedade permite que, caso o elemento de maior prioridade seja removido, o segundo maior elemento consegue tomar seu lugar em tempo $O(\log n)$. Vejamos este processo.

```
1 c.pop();
```

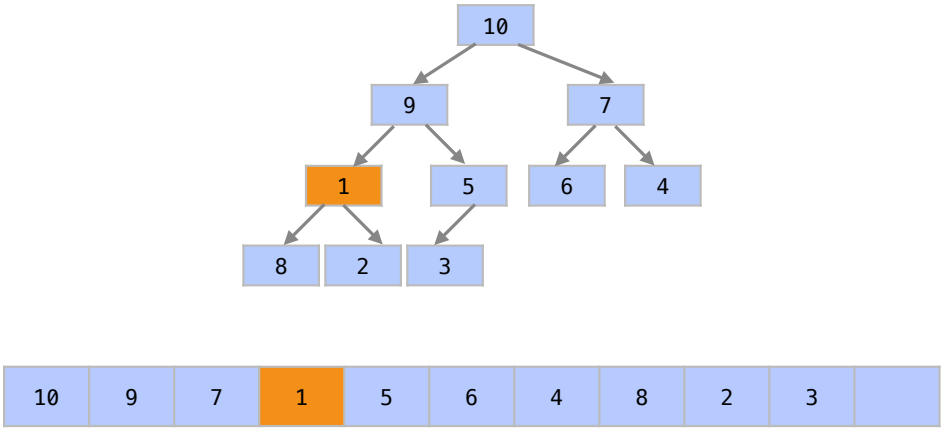
Para rearranjar o heap após remover o elemento do topo, o último elemento do heap toma o lugar do elemento removido. Veja este processo no heap e no arranjo que o representa.



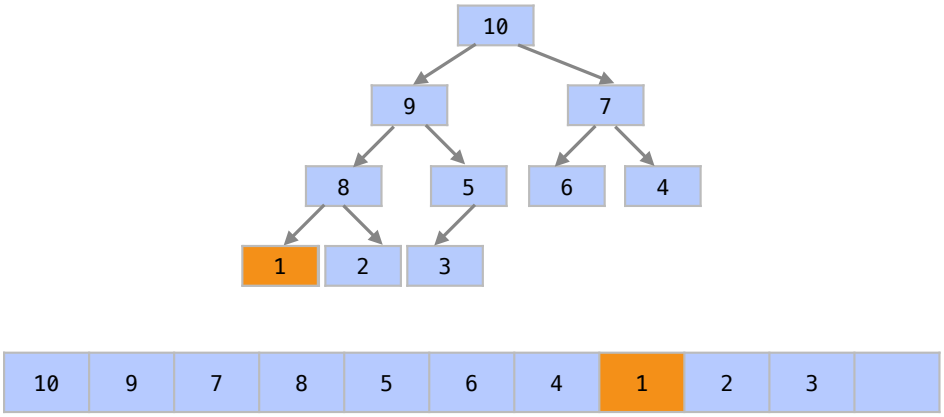
O elemento que tomou o lugar do maior elemento é comparado com seus filhos e troca de posição com maior deles.



Novamente, o elemento é comparado com seus filhos e troca de posição com maior deles.



O processo se repete até que o elemento esteja em uma posição que respeita a condição de ser maior que seus filhos.



Como o elemento pode trocar de posição uma vez para cada nível da árvore e uma árvore balanceada tem $O(\log n)$ níveis, o processo de remoção em uma fila de prioridade tem custo $O(\log n)$.

23. Algoritmos da STL

Parte fundamental da STL são os seus algoritmos. Estes algoritmos podem ser utilizados para executar operações comuns a dados guardados em containers da STL como ordenação, busca, cópia, somatório, ou contagem.

Além de nos poupar tempo de programação, estes algoritmos são usualmente implementações muito eficientes dos algoritmos que conhecemos. Assim, antes de implementar uma operação básica, é sempre bom conferir se a STL já disponibiliza este algoritmo. O cabeçalho `<algorithm>` inclui algoritmos comuns para containers. O cabeçalho `<numeric>` inclui algoritmos comuns para dados numéricos.

23.1 Algoritmos modificadores da STL

A Tabela 23.1 apresenta uma lista de alguns algoritmos modificadores e algoritmos para tarefas importantes da STL. É importante consultar a lista completa de funções na versão mais recente da linguagem para consultar se há funções que implementam algum algoritmo que seja necessário.

Um exemplo de função muito utilizada no STL é o `sort`. A função `sort` ordena uma sequência de elementos, como no exemplo abaixo:

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,12,45,26,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 26 80 53 33
6  // Ordenar elementos nas posições entre begin() e begin()+4
7  sort(v.begin(), v.begin()+4);
8  // (12 32 45 71) 26 80 53 33
9  // Ordenar elementos nas posições entre begin() e end()
10 sort(v.begin(), v.end());
11 // 12 26 32 33 45 53 71 80
12 // ...
```

Função	Descrição
Modificadores de sequência	
`copy`	Copia elementos
`fill`	Preenche elementos com um valor
`iter_swap`	Troca valores de objetos de dois iteradores
`random_shuffle`	Embaralha os elementos
`remove`	Remove valor da sequência
`replace`	Substitui valor na sequência
`reverse`	Reverte sequência
`rotate`	Rotaciona elementos da sequência para esquerda
`swap`	Troca o valor de dois objetos
`transform`	Transforma sequência com uma função
`unique`	Remove duplicatas em sequência
Partições	
`partition`	Particiona sequência em duas
Ordenação	
`sort`	Ordena elementos
`stable_sort`	Ordena elementos com algoritmo estável
`partial_sort`	Ordena primeiros elementos
`is_sorted`	Confere se elementos estão ordenados
Intercalação	
`merge`	Intercala duas sequências ordenadas
Heap	
`make_heap`	Cria um heap da sequência
`push_heap`	Coloca elemento em um heap
`pop_heap`	Retira elemento de um heap
`is_heap`	Testa se sequência é um heap
Permutações	
`next_permutation`	Transforma sequência em sua próxima permutação
`next_permutation`	Transforma sequência em sua permutação anterior

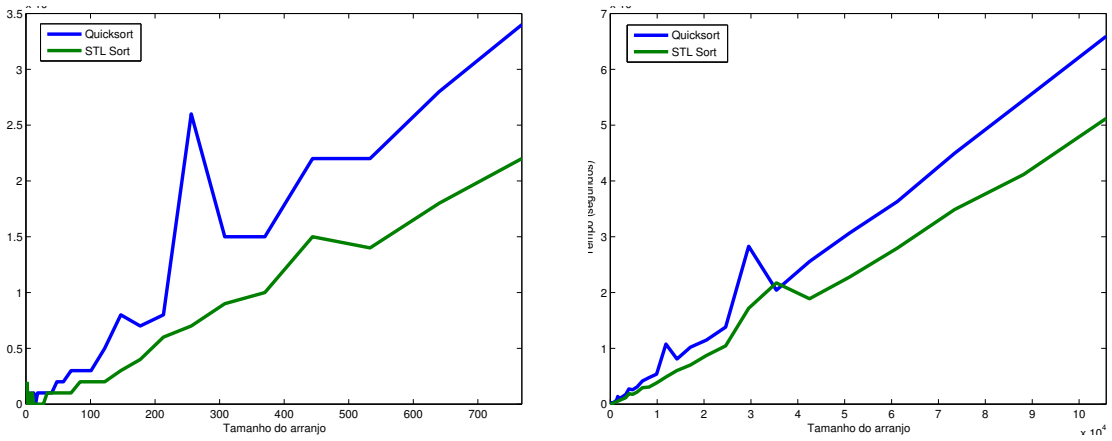
Tabela 23.1: Lista de algoritmos modificadores e algoritmos para tarefas importantes da STL.

Neste exemplo, as sequências nos comentários representam o efeito da aplicação da função. Nas linhas 3 e 4, criamos uma sequência em um arranjo e a copiamos para um `vector` chamado `v`. A função recebe sempre dois iteradores que representam a sequência. Na linha 7, ordenamos a sequência formada pelos 4 primeiros elementos, entre os iteradores `v.begin()` e `v.begin() + 4`. Na linha 10, ordenamos a sequência com todos os elementos, entre os iteradores `v.begin()` e `v.end()`.

Apesar de ter o mesmo comportamento assintótico $O(n \log n)$, o `sort` do STL faz uma ordenação que é ainda mais eficiente do que um quicksort. A Figura 23.1 compara o tempo gasto por estes algoritmos.

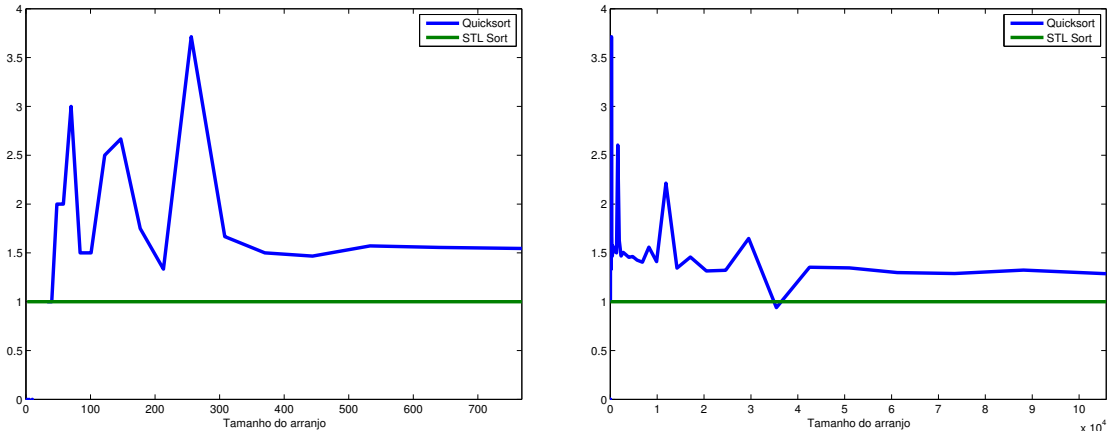
Já a Figura 23.2 apresenta a proporção de tempo entre os dois algoritmos. O quicksort é quase 4 vezes mais lento em arranjos pequenos e quase 50% mais lento em arranjos maiores.

O algoritmo `fill` coloca o valor passado como parâmetro em toda uma sequência de elementos.



(a) Arranjos menores (b) Arranjos maiores

Figura 23.1: Tempo necessário pelos algoritmos do STL e por uma implementação eficiente do quicksort para se ordenar um arranjo com ordem inicial aleatória.



(a) Arranjos menores (b) Arranjos maiores

Figura 23.2: Proporção de tempo necessário pelos algoritmos do STL e por uma implementação eficiente do quicksort para se ordenar um arranjo com ordem inicial aleatória.

```
1  #include <algorithm>
2  // ...
3  vector< char > letras(10);
4  fill(letras.begin(),letras.end(),'B');
5  // B B B B B B B B B B
6  imprimeContainer(letras);
7  // Colocar A nos 5 elementos a partir de begin()
8  fill_n(letras.begin(),5,'A');
9  // A A A A A B B B B B
10 imprimeContainer(letras);
11 // ...
```

Na linha 4 do exemplo, a letra `B` é colocada em todas as posições da sequência, entre os elementos apontados pelos iteradores `letras.begin()` e `letras.end()`. Na linha 8, a função `fill_n` coloca a letra `A` nas 5 posições a partir do primeiro elemento `letras.begin()`.

O algoritmo `remove` remove elementos que tenham o valor passado como parâmetro em toda uma sequência de elementos.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,33,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 33 45 33 80 53 33
6  // Remover as ocorrências de 33 entre begin() e end()
7  remove(v.begin(), v.end(), 33);
8  // 32 71 45 80 53
9  imprimeContainer(v);
10 // ...
```

Na linha 7 do exemplo, todos os elementos com valor 33 são removidos da sequência. Considera-se toda a sequência, entre os iteradores `v.begin()` e `v.end()`.

O algoritmo `replace` substitui o valor de todos os elementos que tenham o valor passado como parâmetro em toda uma sequência de elementos.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,33,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 33 45 33 80 53 33
6  // Substitui ocorrências de 33 por 100
7  replace(v.begin(), v.end(), 33, 100);
8  // 32 71 100 45 100 80 53 100
9  imprimeContainer(v);
10 // ...
```

Na linha 7 do exemplo, todos os elementos com valor 33 são substituídos por elementos de valor 100. Considera-se toda a sequência, entre os iteradores `v.begin()` e `v.end()`.

O algoritmo `random_shuffle` embaralha uma sequência de elementos.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {1, 2, 3, 4, 5, 6, 7, 8};
4  vector<int> v(nums, nums+8);
5  // 1 2 3 4 5 6 7 8
6  // Embaralha elementos entre begin() e end()
7  random_shuffle(v.begin(), v.end());
8  // 6 4 8 2 3 5 1 7
9  imprimeContainer(v);
10 // ...
```

Na linha 7 do exemplo, todos os elementos da sequência são embaralhados, entre os iteradores `v.begin()` e `v.end()`.

23.2 Algoritmos não modificadores da STL

A Tabela 23.2 apresenta uma lista de alguns algoritmos não modificadores da STL. É importante consultar a lista completa de funções na versão mais recente da linguagem para consultar se há funções que implementam algum algoritmo que seja necessário.

O algoritmo `equal` testa se duas sequências de elementos são iguais.

Função	Descrição
Não modificadores de sequência	
`count`	Conta elementos com certo valor
`count_if`	Conta elementos com certa condição
`equal`	Testa se duas sequências são iguais
`find`	Procura elemento com busca sequencial
`find_if`	Procura elemento que respeita condição
`search`	Procura subsequência
Min / Max	
`min`	Retorna o menor de 2 elementos
`max`	Retorna o maior de 2 elementos
`minmax`	Retorna o menor e o maior entre 2 elementos
`min_element`	Retorna o menor de uma sequência
`max_element`	Retorna o maior de uma sequência
`minmax_element`	Retorna o menor e o maior de uma sequência
Busca binária	
`lower_bound`	Busca primeira ocorrência de elemento
`upper_bound`	Busca última ocorrência de elemento
`binary_search`	Testa se elemento existe

Tabela 23.2: Lista de algoritmos não modificadores para tarefas importantes da STL.

```
1  #include <algorithm>
2  // ...
3  vector<int> v1(5,10); // 10 10 10 10 10
4  vector<int> v2(5,10); // 10 10 10 10 10
5  // compara duas sequências de valores
6  bool resultado = equal(v1.begin(), v1.end(), v2.begin());
7  if (resultado){
8      cout << "Os␣vetores␣têm␣os␣mesmos␣valores" << endl;
9  } else {
10     cout << "Os␣vetores␣são␣diferentes" << endl;
11 }
12 // ...
```

Na linha 6 do exemplo, a sequência entre os iteradores `v1.begin()` e `v2.end()` é comparada com a sequência que inicia no iterador `v2.begin()`. O resultado é testado na linha 7 e se imprime uma mensagem.

O algoritmo `find` procura um elemento em uma sequência.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,12,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 33 80 53 33
6  vector<int>::iterator r;
7  // Procura elemento 12 e retorna iterador para ele
8  r = find(v.begin(), v.end(), 12);
9  // Se o elemento foi encontrado
```

```
10 if (r != v.end()){
11     // Elemento: 12
12     cout << "Elemento:" << *r << endl;
13     // Posição: 2
14     cout << "Posição:" << r-v.begin() << endl;
15 }
16 // ...
```

Na linha 8 do exemplo, procuramos o elemento 12 na sequência formada entre os iteradores `v.begin()` e `v.end()`. O resultado é um iterador que aponta para o elemento procurado. Se o elemento não for encontrado, um iterador para `v.end()` é retornado. Na linha 10 testamos se o elemento foi realmente encontrado. Caso sim, o elemento é impresso nas linhas 12 e 14.

O algoritmo `binary_search` retorna se um elemento está presente em uma sequência.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,12,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 33 80 53 33
6  vector<int>::iterator r;
7  // Ordena os elementos entre begin() e end()
8  sort(v.begin(),v.end());
9  // 12 32 33 33 45 53 71 80
10 // Se o elemento foi encontrado
11 if (binary_search(v.begin(), v.end(), 12)){
12     cout << "Elemento␣encontrado" << endl;
13 }
14 // ...
```

Na linha 8 do exemplo, os elementos da sequência são ordenados para permitir uma busca binária. Na linha 11, procuramos o elemento 12 na sequência formada entre os iteradores `v.begin()` e `v.end()`. O valor retornado pela função é um `bool` que já é utilizado para imprimir uma mensagem. Para se encontrar um iterador para o elemento, as funções `lower_bound` e `upper_bound` devem ser utilizadas.

O algoritmo `count` retorna quantos elementos com um dados valor existem na sequência.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,33,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 33 45 33 80 53 33
6  // Conta quantas ocorrências de 33 entre begin() e end()
7  int r = count(v.begin(), v.end(), 33);
8  cout << r << "␣números␣33" << endl;
9  // 3 números 33
10 // ...
```

Na linha 7 do exemplo, contamos quantos elementos da sequência formada entre os iteradores `v.begin()` e `v.end()` são iguais a 33.

O algoritmo `min_element` retorna um iterador para o menor elemento de uma sequência.

```
1  #include <algorithm>
2  // ...
```

```
3  int nums[] = {32,71,12,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 33 80 53 33
6  // Procura o menor elemento entre begin() e end()
7  int r = *min_element(v.begin(),v.end());
8  cout << "Menor elemento:" << r << endl;
9  // Menor elemento: 12
10 // ...
```

Na linha 7 do exemplo, a função retorna um iterador para o menor elemento da sequência. Utilizamos o operador `*` para atribuirmos o valor apontado por este iterador na variável `r`.

De modo análogo, o algoritmo `max_element` retorna um iterador para o maior elemento de uma sequência.

```
1  #include <algorithm>
2  // ...
3  int nums[] = {32,71,12,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 33 80 53 33
6  // Procura o maior elemento entre begin() e end()
7  int r = *max_element(v.begin(),v.end());
8  cout << "Maior elemento:" << r << endl;
9  // Maior elemento: 80
10 // ...
```

23.3 Algoritmos numéricos da STL

A Tabela 23.3 apresenta uma lista de alguns algoritmos numéricos da STL. Estes algoritmos são utilizados para dados aritméticos.

Função	Descrição
Numéricas	
`accumulate`	Somatório de elementos
`inner_product`	Produto interno
`partial_sum`	Somas parciais
`adjacent_difference`	Diferença adjacente
`iota`	Atribui sequência crescente de números

Tabela 23.3: Lista de algoritmos não modificadores para tarefas importantes da STL.

Como exemplo de algoritmo numérico, a função `accumulate` faz um somatório de toda uma sequência de valores.

```
1  #include <numeric>
2  // ...
3  int nums[] = {32,71,12,45,33,80,53,33};
4  vector<int> v(nums, nums+8);
5  // 32 71 12 45 33 80 53 33
6  // Soma todos os elementos do vector iniciando somatório com 0
7  int r = accumulate(v.begin(), v.end(), 0);
8  cout << "Somatório:" << r << endl;
```

```
9  // Somatório: 359
10 // ...
```

Na linha 7 do exemplo, fazemos um somatório com todos os elementos da sequência entre os iteradores `v1.begin()` e `v2.end()`. Este somatório e iniciado em `0`. O resultado, atribuído a `r`, é impresso na linha 8.

23.4 Exercícios

Exercício 23.1 Veja o código abaixo. Ao completá-lo codigo abaixo, vamos implementar o jogo de cartas 21. Neste jogo, há uma pilha com todas as cartas do baralho (`stack`). Cada carta sera representada com um numero de 1 a 13 (1 = A, 11=J, 12=Q, 13=K). O naipe não será representado pois não é relevante.

A cada passo, o jogador pode tirar uma carta da pilha e colocar em seu conjunto de cartas (em outro container) ou parar de jogar. O somatório dos valores das cartas em sua mão é sua pontuação mas o jogador perde o jogo se o somatório ultrapassar 21.

Analise o código abaixo, que está incompleto.

```
1  #include <iostream>
2  #include <string>
3  // bibliotecas com os containeres
4  #include <vector>
5  #include <stack>
6  #include <algorithm>
7
8  using namespace std;
9
10 void imprimeCarta(int const& x);
11
12 template <typename T> // função que imprime container
13 void imprimeBaralho (T const& x);
14
15 int main()
16 {
17     // cria um container vetor com as cartas
18     vector<int> x;
19     // Aloca espaço para 13*4 elementos
20     x.reserve(13*4);
21     int i;
22
23     // Coloca 4 vezes cada numero de 1 a 13 no baralho
24     for (i=1; i<13; i++){
25         x.push_back(i);
26         x.push_back(i);
27         x.push_back(i);
28         x.push_back(i);
29     }
30
31     cout << "Cartas␣do␣baralho:" << endl;
32     imprimeBaralho(x);
```

```
33
34      // Embaralha e imprime
35      cout << "Embaralhando:" << endl;
36      random_shuffle ( x.begin(), x.end() );
37      imprimeBaralho(x);
38
39      // Cria uma pilha vazia
40      stack<int> pilha;
41
42      // Faça a pilha receber as cartas do baralho aqui
43      // Use a funcao push para colocar algo na pilha
44
45      vector<int> cartas;
46
47      // Menu com opções
48      string opcao;
49      do {
50          cout << "Suas␣cartas␣são:" << endl;
51          imprimeBaralho(cartas);
52
53          // Mostra o menu
54          cout << endl;
55          cout << "[1]␣Puxar␣mais␣uma␣carta␣da␣pilha" << endl;
56          cout << "[0]␣Sair␣do␣jogo" << endl;
57          cout << "Digite␣uma␣opção:␣" ;
58
59          // Lê uma opção
60          cin >> opcao;
61
62          // executa uma função de acordo com a opção
63          if (opcao == "1"){
64              cout << "Puxando␣mais␣uma␣carta" << endl;
65              // Implemente este trecho
66              cout << "Código␣não␣implementado" << endl;
67          } else if (opcao == "0"){
68              cout << "Saindo␣do␣programa␣" << endl;
69          } else {
70              cout << "Opcao␣invalida" << endl;
71          }
72      } while (opcao != "0");
73
74      cout << "Fim␣de␣jogo" << endl;
75      // Implemente este trecho
76      cout << "Código␣não␣implementado" << endl;
77
78      return 0;
79  }
80
81  void imprimeCarta(int const&x){
```

```
82      if (x==1){
83          cout << "A␣";
84      } else  if (x<11){
85          cout << x << "␣";
86      } else if (x==11) {
87          cout << "J␣";
88      } else if (x==12) {
89          cout << "Q␣";
90      } else if (x== 13) {
91          cout << "K␣";
92      }
93
94  }
95
96  template <typename T>
97  void imprimeBaralho (T const& x)
98  {
99      if (x.empty()){
100         std::cout << "(Vazio)" << endl;
101         return;
102     }
103     // iterator que vai percorrer x
104     typename T::const_iterator pos;
105     // iterador que aponta para a posição final
106     typename T::const_iterator end(x.end());
107     for (pos=x.begin(); pos!=end; ++pos) {
108         imprimeCarta(*pos);
109     }
110     cout << endl;
111 }
```

■

Exercício 23.2 Logo após a criação de uma pilha vazia, faça com que ela receba todas as cartas do baralho em x. ■

Exercício 23.3 O `vector cartas` receberá os elementos que o jogador retirar da pilha. Implemente a retirada de uma carta da pilha e sua insercao no vector cartas quando o jogador pedir. ■

Exercício 23.4 Quando o jogador não quiser mais retirar cartas, ele utilizará a `opcao == 0` e sairá do `for`. Quando isto ocorrer, implemente o trecho de código que mostrará a pontuação final do jogador. Se a soma ultrapassar 21, deverá ser emitida uma mensagem que ele perdeu o jogo. Utilize a função `accummulate`. ■

Exercício 23.5 Qual a vantagem da utilização de uma pilha `stack` em vez de se representar a pilha de cartas com um container de sequência? ■

Exercício 23.6 Qual a vantagem da utilizacao do header `<algorithm>`? ■

Exercício 23.7 No jogo real de 21, as cartas K, J, Q e 10 (representadas por `10`, `11`, `12`, `13`) valem 10. As cartas A (representadas aqui por `1`) podem valer 1 ou 11. Faça esta alteração no programa na hora do resultado final. O jogador receberá a maior pontuação possível para seu conjunto de cartas. Dica: Utilize a função `replace` para alterar as cartas K, J e Q, que devem valer 10. Considere que um A vale 1 e conte o número de As (função `count`). Troque estes As que valem 1 por As iguais a 10 enquanto tenhamos menos que 21. ■

IV Programação Orientada a Objetos

24. Classes e Instâncias de Objetos

24.1 Introdução

Temos estudado até o momento o que chamamos de **programação estruturada**. Em programação estruturada, o fluxo no qual ocorrem os comandos tem a maior importância. O foco está na ordem das ações. É assim que temos trabalhado até agora.

A Figura 24.1 apresenta uma representação em diagrama de programas baseados em programação estruturada. Nestes programas, seguimos um fluxo de ações, que pode ser totalmente sequencial ou alterado por desvios e laços.

Já na **programação orientada a objetos** (POO), a nossa preocupação é com os objetos que têm atributos e ações. Dados os objetos, nos preocupamos em determinar como eles se comportam e não em um fluxo geral de ações. Queremos saber como é cada objeto e o que faz cada objeto.

A Figura 24.2 apresenta uma representação em diagrama de objetos utilizados em um programa para representar animais. Nestes programas, as características dos objetos e relações entre objetos são mais importantes do que um fluxo de comandos.

No exemplo apresentado, temos 3 objetos: Animal, Cachorro e Ovelha. Os objetos Cachorro e Ovelha são apenas tipos de objeto do tipo Animal. Cachorro e Ovelha têm suas características e comportamentos específicos.

24.2 Classes e Instâncias de Objetos

Para criar objetos em um programa, devemos definir primeiro novas **classes** de objetos. As classes são utilizadas para definir como são objetos de um certo tipo.

A relação entre classes (que têm a palavra-chave `class`) e um `struct` (como os que estudamos na Seção 10) é bem direta. Assim podemos criar um `struct` que representa um retângulo.

```
1 struct Retangulo {
2     int largura, altura;
3 };
```

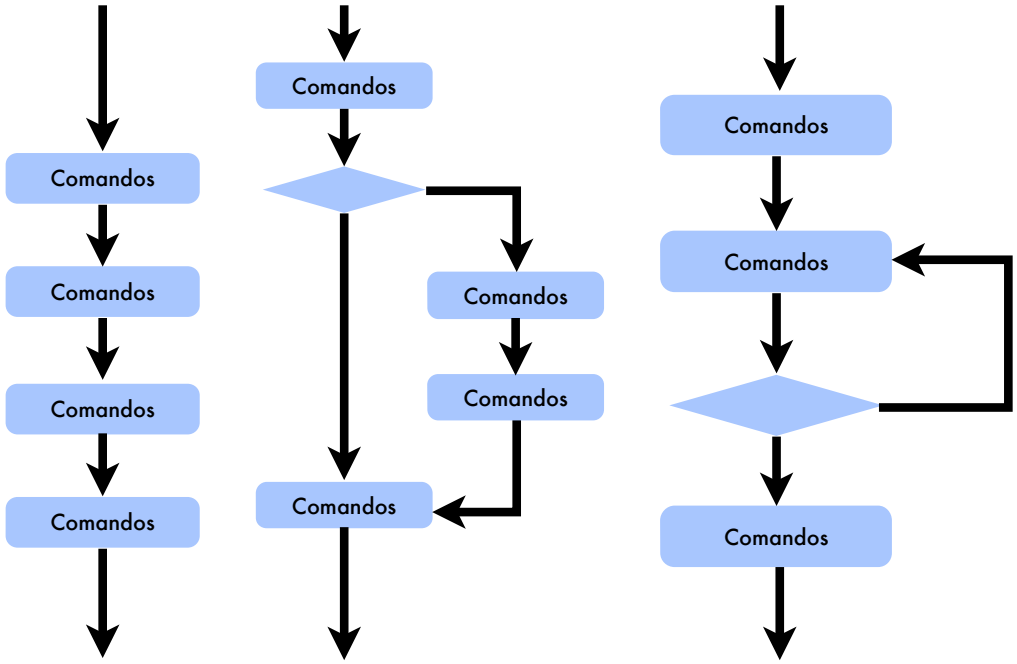


Figura 24.1: Representação em diagrama de programação estruturada.

Cada retângulo tem dois atributos: `largura` e `altura`. Com a definição deste `struct` poderíamos criar dados do tipo `Retangulo`. Através da instrução `class`, poderíamos definir o mesmo tipo de dado da seguinte maneira:

```
1 class Retangulo {
2   public:
3     int largura, altura;
4 };
```

Para criar um novo tipo de dados definindo uma `class`, devemos dizer que os membros da classe são públicos para que eles sejam acessíveis. Isto foi feito na linha 2 com a palavra-chave `public`. Sem a palavra chave `public`, os membros da classe seriam considerados privados e não seriam acessíveis. Isto é um conceito importante em POO. Em um `struct`, todos os membros são considerados públicos por padrão.

Nos objetos, além de variáveis, podemos ter funções como membros de uma classe. As funções representam comportamentos de uma classe de objetos. Podemos ter alguns comportamentos comuns a retângulos:

```
1 class Retangulo {
2     int largura, altura;
3   public:
4     void set_valores (int,int);
5     int area() {return largura*altura;}
6 };
```

Nas linhas 4 e 5, definimos 2 funções públicas relativas a retângulos. Definimos comportamentos relativos a retângulos. Na linha 2, definimos atributos de um retângulo. Note que os atributos na linha 2 não são públicos pois são definidos antes da palavra-chave `public`. Podemos ter em um objeto membros públicos e privados ao mesmo tempo. Neste exemplo, temos membros públicos e privados: comportamentos públicos e atributos privados.

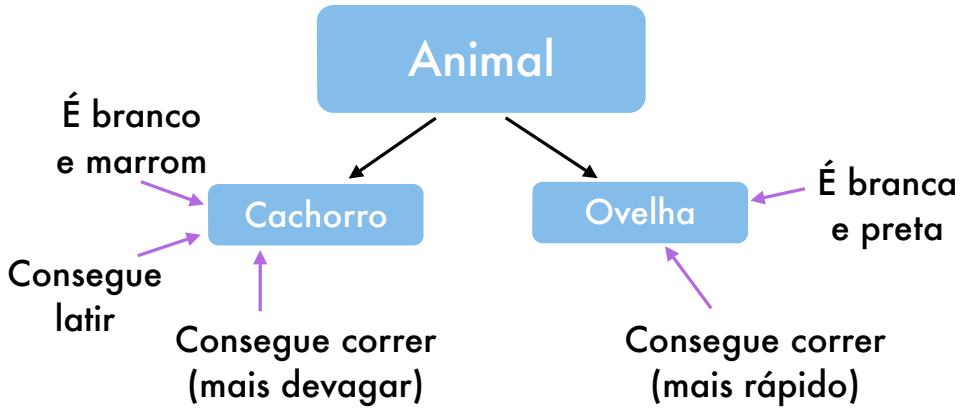


Figura 24.2: Representação em diagrama de programação estruturada.

Na verdade, é comum deixar explícito quais são os membros privados. Para isto usamos a palavra-chave `private`.

```
1 class Retangulo {
2   public:
3     void set_valores (int,int);
4     int area() {return largura*altura;}
5   private:
6     int largura, altura;
7 };
```

Com isto, deixamos os membros privados no fim da definição, o que é interessante pois as pessoas estão normalmente mais interessadas nos membros que elas podem acessar em um objeto.

As funções do próprio objeto, é claro, são as únicas que têm acesso aos membros privados deste objeto. Como os atributos do retângulo não estão acessíveis, podemos utilizar a função pública `set_valores` para determinar os valores de `largura` e `altura`. Isto é uma prática muito comum pois podemos usar a função para validar os valores inseridos para `largura` e `altura`. Podemos, por exemplo, verificar se são valores maiores que zero.

Note também que, na linha 3, apenas o cabeçalho da função está definido no objeto. A implementação da função deve vir depois.

Na linha 4, temos a definição da função `area`. Esta função utiliza os atributos conhecidos para retornar a área do retângulo. Como esta é uma função simples, sua implementação já consta no objeto.

Note com a função `area`, como a orientação a objetos pode ser útil para agregar nos retângulos as funções que são associadas apenas a retângulos.

Incluímos no código em seguida também a definição da função `set_valores`. A sintaxe `Retangulo::set_valores` é utilizada para especificar fora da definição da classe que a função sendo implementada pertence à classe `Retangulo`.

```
1 class Retangulo {
2   public:
3     void set_valores(int,int);
```

```
4       int area () {return (largura*altura);}
5     private:
6       int largura, altura;
7   };
8
9   void Retangulo::set_valores (int x, int y) {
10    largura = x;
11    altura = y;
12  }
```

24.3 Instanciando objetos

Neste exemplo completo, definimos e criamos um objeto `Retangulo` e imprimimos sua área.

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Retangulo {
6     public:
7       void set_valores(int,int);
8       int area () {return (largura*altura);}
9     private:
10      int largura, altura;
11  };
12
13  void Retangulo::set_valores (int x, int y) {
14    largura = x;
15    altura = y;
16  }
17
18  int main(){
19      Retangulo r;
20      r.set_valores(3,4);
21      cout << "A␣área␣de␣r␣é␣"  << r.area() << endl;
22      return 0;
23  }
```

Entre as linhas 5 e 16 temos a definição completa da classe `Retangulo`. Como podem existir vários retângulos diferentes, definimos aqui uma classe de objetos.

Na linha 19, criamos um retângulo `r` que ainda não tem seus atributos inicializados. Dizemos que o objeto `r` é uma **instância** da classe de objetos `Retangulo`.

Na linha 20, a função `set_valores` do retângulo é utilizada para inicializar sua `largura` com 3 e `altura` com 4.

Na linha 21, a própria função `area` do retângulo é utilizada para imprimir a área.

```
A área de r é 12
```

Note que função `area` não recebe parâmetros pois tudo que ela precisa é dos próprios atributos internos do retângulo `r`.

Vejamos agora um exemplo com dois retângulos a e b.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      void set_valores(int,int);
8      int area () {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
12
13 void Retangulo::set_valores (int x, int y) {
14   largura = x;
15   altura = y;
16 }
17
18 int main(){
19     Retangulo a,b;
20     a.set_valores(3,4);
21     b.set_valores(4,5);
22     cout << "A área de a é " << a.area() << endl;
23     cout << "A área de b é " << b.area() << endl;
24     return 0;
25 }
```

Os retângulos são definidos na linha 19 e recebem valores nas linhas 20 e 21. Nas linhas 22 e 23, as respectivas áreas são impressas.

```
A área de a é 12
A área de b é 20
```

24.4 Ponteiros para objetos

Assim como fizemos em um `struct`, podemos também criar ponteiros e alocar memória para um objeto.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      void set_valores(int,int);
8      int area () {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
```

```
12
13 void Retangulo::set_valores (int x, int y) {
14   largura = x;
15   altura = y;
16 }
17
18 int main(){
19     Retangulo a;
20     Retangulo *b = new Retangulo();
21     a.set_valores(3,4);
22     b->set_valores(4,5); // ou (*b).set_valores(4,5);
23     cout << "A área de a é " << a.area() << endl;
24     cout << "A área de *b é " << b->area() << endl;
25     delete b;
26     b = nullptr;
27     return 0;
28 }
```

Na linha 20, criamos um ponteiro para `Retangulo` chamado `b`. Alocamos dinamicamente um `Retangulo` na memória e fazemos com que `b` aponte para ele.

Assim como se pode fazer em um `struct`, podemos utilizar o operador de seta `->` em objetos para acessar um membro de um objeto apontado. Isto é feito na linha 22. Veja como o operador de seta da linha 22 equivale a retornar o valor apontado por `b` com o operador `*` e depois acessar um membro.

Na linha 24, utilizamos novamente o operador de seta para imprimir a área do `Retangulo` apontado por `b`.

```
A área de a é 12
A área de *b é 20
```

24.5 Arranjos de objetos

Como qualquer outro tipo de dado, podemos também criar arranjos de objetos.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      void set_valores(int,int);
8      int area () {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
12
13 void Retangulo::set_valores (int x, int y) {
14   largura = x;
15   altura = y;
16 }
```

```
17
18 int main(){
19     Retangulo a[2];
20     a[0].set_valores(3,4);
21     a[1].set_valores(4,5);
22     cout << "A␣área␣de␣a[0]␣é␣" << a[0].area() << endl;
23     cout << "A␣área␣de␣a[1]␣é␣" << a[1].area() << endl;
24     return 0;
25 }
```

Na função deste exemplo, trabalhamos com `a[0]` e `a[1]`, que guardam dois retângulos diferentes.

```
A área de a[0] é 12
A área de a[1] é 20
```

Podemos também alocar arranjos de objetos dinamicamente.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      void set_valores(int,int);
8      int area () {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
12
13 void Retangulo::set_valores (int x, int y) {
14   largura = x;
15   altura = y;
16 }
17
18 int main(){
19     Retangulo *a = new Retangulo[2];
20     a[0].set_valores(3,4);
21     a[1].set_valores(4,5);
22     cout << "A␣área␣de␣a[0]␣é␣" << a[0].area() << endl;
23     cout << "A␣área␣de␣a[1]␣é␣" << a[1].area() << endl;
24     return 0;
25 }
```

Neste exemplo, temos o ponteiro `a` que apontando para o endereço onde começa um arranjo de dois retângulos na memória. Para todos os outros efeitos, o operador de subscrito `[]` é utilizado para acessar os elementos do arranjo.

```
A área de a[0] é 12
A área de a[1] é 20
```

24.6 POO em sistemas complexos

Como vimos, a POO serve para modelar sistemas mais complexos. Já que o mundo é rodeado de objetos, o conceito de objeto é útil para modelagem de coisas reais. Na POO, devemos modelar quais são os atributos (variáveis) e quais são os comportamentos (funções) de um objeto.

25. Definindo classes

Há várias práticas comuns e recursos para definição de objetos. Nesta seção veremos algumas delas.

25.1 Contrutores

Quando um objeto é criado, uma função construtora é chamada. Chamamos esta função de **construtor**. Esta função constrói o objeto dando valores iniciais a seus atributos. Para definir o construtor, criamos uma função com o mesmo nome da classe.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo(int,int);
8      int area () {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
12
13 Retangulo::Retangulo (int a, int b) {
14   largura = a;
15   altura = b;
16 }
17
18 int main () {
19   Retangulo ra (3,4);
20   Retangulo rb (5,6);
```

```
21    cout << "A área de ra é " << ra.area() << endl;
22    cout << "A área de rb é " << rb.area() << endl;
23    return 0;
24  }
```

Ne linha 7 do exemplo, nosso objeto tem agora uma função construtora que recebe dois números inteiros. Na implementação da função construtora, nas linhas 13 a 16, guardamos o primeiro parâmetro recebido como `largura` e o segundo como `altura`.

Na função principal, nas linhas 19 e 20, ao criar um objetos do tipo `Retangulo`, os parâmetros para o construtor já devem ser passados.

Nas linhas 21 e 22, como os objetos `ra` e `rb` já foram construídos com os valores de `altura` e `largura`, já podemos imprimir a área sem utilização de uma função como `set_valores`.

```
A área de ra é 12
A área de rb é 30
```

25.2 Sobrecarga de construtores

Podemos também *sobrecarregar* construtores. Assim temos diferentes versões que dependem dos parâmetros.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo ();
8      Retangulo (int,int);
9      int area () {return (largura*altura);}
10   private:
11     int largura, altura;
12 };
13
14 Retangulo::Retangulo () {
15   largura = 5;
16   altura = 5;
17 }
18
19 Retangulo::Retangulo (int a, int b) {
20   largura = a;
21   altura = b;
22 }
23
24 int main () {
25   Retangulo ra(3,4);
26   Retangulo rb;
27   cout << "Área de ra: " << ra.area() << endl;
28   cout << "Área de rb: " << rb.area() << endl;
29   return 0;
```

```
30 }
```

No primeiro construtor, implementado nas linhas 14 a 17, se nenhum parâmetro é passado, o `Retangulo` é criado com `altura` e `largura` iguais a `5`. No segundo construtor, implementado nas linhas 19 a 22, se os parâmetros são passados, eles são atribuídos aos membros.

Criamos então neste exemplo um retângulo com cada construtor. Na linha 25, utilizamos o construtor que recebe dois parâmetros e, na linha 26, utilizamos o construtor que não recebe parâmetros.

```
A área de ra:  12
A área de rb:  25
```

Podemos criar quantas sobrecargas quisermos. Precisamos apenas que cada sobrecarga dependa de uma combinação diferente de parâmetros.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo ();
8      Retangulo (int);
9      Retangulo (int,int);
10     int area () {return (largura*altura);}
11   private:
12     int largura, altura;
13 };
14
15 Retangulo::Retangulo () {
16   largura = 5;
17   altura = 5;
18 }
19
20 Retangulo::Retangulo (int a) {
21   largura = a;
22   altura = a;
23 }
24
25 Retangulo::Retangulo (int a, int b) {
26   largura = a;
27   altura = b;
28 }
29
30 int main () {
31   Retangulo ra(3);
32   Retangulo rb = 4;
33   cout << "Área de ra:" << ra.area() << endl;
34   cout << "Área de rb:" << rb.area() << endl;
35   return 0;
36 }
```

Neste exemplo, se apenas um parâmetro é passado, ele é atribuído tanto à `altura` quanto à `largura`, como definido nas linhas 20 a 23. Nas linhas 31 e 32 utilizamos apenas este construtor para definir novos retângulos. Como vimos, o parâmetros ou os parâmetros devem ser passados para o objeto em sua criação, como fizemos na linha 31.

Construtores com apenas um parâmetro são interessantes pois podem ser utilizados para atribuir um tipo de dado a outro. Neste exemplo, atribuímos um `int` em `Retangulo` na linha 32. A conversão é feita pelo construtor de apenas um parâmetro.

```
A área de ra:  9
A área de rb:  16
```

25.2.1 Construtor de Cópia

Outro tipo comum de construtor é o **construtor de cópia**.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo (int,int);
8      Retangulo (const Retangulo &);
9      int area () {return (largura*altura);}
10   private:
11     int largura, altura;
12 };
13
14 Retangulo::Retangulo (int a, int b) {
15   largura = a;
16   altura = b;
17 }
18
19 Retangulo::Retangulo (const Retangulo &direita) {
20   largura = direita.largura;
21   altura = direita.altura;
22 }
23
24 int main () {
25   Retangulo ra(3,4);
26   Retangulo rb = ra;
27   cout << "Área de ra:" << ra.area() << endl;
28   cout << "Área de rb:" << rb.area() << endl;
29   return 0;
30 }
```

Este construtor recebe outro elemento do mesmo tipo e faz uma cópia, como definido nas linhas 19 a 22 e utilizado na linha 26.

```
A área de ra:  12
A área de rb:  12
```

25.3 Sobrecarga de operadores

Existe a possibilidade de se somar, dividir ou comparar dois objetos. A sobrecarga de operadores define o que fazer quando são chamados os operadores para uma classe sendo definida.

Neste exemplo, criamos uma função nos permitirá somar dois retângulos.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo(int,int);
8      Retangulo operator+ (const Retangulo &);
9      int area () {return (largura*altura);}
10   private:
11     int largura, altura;
12 };
13
14 Retangulo::Retangulo (int a, int b) {
15   largura = a;
16   altura = b;
17 }
18
19 Retangulo Retangulo::operator+ (const Retangulo& direita) {
20   Retangulo temp(largura + direita.largura, altura + direita.altura);
21   return temp;
22 }
23
24 int main () {
25   Retangulo ra(3,4);
26   Retangulo rb(4,5);
27   Retangulo rc = ra + rb;
28   cout << "Área␣de␣rc:␣" << rc.area() << endl;
29   return 0;
30 }
```

Dado um retângulo, o operador de soma deve receber um segundo retângulo e retornar um terceiro. Para definir a função de soma, utilizaremos `operator+` como nome da função. Esta definição está na linha 8 do exemplo.

Esta operação de soma receberá como parâmetro um outro `Retangulo` ao qual a instância da classe será somada. O tipo de dado de retorno é também `Retangulo`, pois o resultado da soma será um outro retângulo.

Na linha 19 iniciamos a implementação da função. Na linha 20, um novo `Retangulo` chamado `temp` é construído com a soma das larguras e alturas do própria retângulo e do retângulo recebido como parâmetro. Na linha 21, este retângulo `temp` é retornado como resultado da soma de dois retângulos.

Na função principal, nas linnhas 25 a 27, usamos estes recursos para criar um retângulo `rc` como a soma de `ra` e `rb`.

```
Área de rc:  63
```

Neste segundo exemplo, usamos o recurso de sobrecarga de operadores para podermos comparar dois retângulos de acordo com suas áreas.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Retangulo {
6    public:
7      Retangulo(int,int);
8      bool operator < (Retangulo &dir) {return area() < dir.area();}
9      int area () {return (largura*altura);}
10   private:
11     int largura, altura;
12 };
13
14 Retangulo::Retangulo (int a, int b) {
15   largura = a;
16   altura = b;
17 }
18
19 int main () {
20   Retangulo ra(3,4);
21   Retangulo rb(5,3);
22   if (ra < rb){
23       cout << "A área de ra é menor que a de rb" << endl;
24   } else {
25       cout << "A área de rb é menor que a de ra" << endl;
26   }
27   return 0;
28 }
```

O `operador<` é definido na linha 8 do exemplo e retorna um `bool` indicando se a área do retângulo é maior que a área do retângulo recebido como parâmetro.

Na linha 22 da função principal, o `operador<` retorna um `bool` de acordo com a área dos retângulos `ra` e `rb`. O resultado é impresso em um `cout` nas linhas 23 ou 25.

```
A área de ra é menor que a de rb
```

25.4 Setters e Getters

Os *setters* e *getters* são as funções que atribuem ou retornam atributos de um objeto. Os atributos do objeto usualmente ficam privados enquanto as funções fazem o papel de acesso. Esta prática deixa o objeto menos propenso a erros do que se deixássemos os atributos como públicos. No exemplo seguinte, temos várias funções deste tipo.

```
1  class Retangulo {
2    public:
3      Retangulo(int,int);
4      void setAltura(int x);
5      void setLargura(int x);
6      int getAltura() {return altura;}
7      int getLargura() {return largura;}
8      int area() {return (largura*altura);}
9    private:
10     int largura, altura;
11 };
12
13 Retangulo::Retangulo(int a, int b) {
14     setAltura(a);
15     setLargura(b);
16 }
17
18 void Retangulo::setAltura(int x) {
19     if (x >=0){
20         altura = x;
21     } else {
22         altura = 0;
23     }
24 }
25
26 void Retangulo::setLargura(int x) {
27     if (x >=0){
28         largura = x;
29     } else {
30         largura = 0;
31     }
32 }
```

Na nova classe temos 2 setters e 2 getters que servem para enviar e receber valores relativos às variáveis. Estas funções estão definidas nas linhas 4 a 7. Os *getters*, definidos nas linhas 6 e 7, simplesmente retornam os valores de `altura` e `largura`.

As funções *setters* que dão valores aos atributos são definidas nas linhas 18 a 32, conferem se os valores são maiores do que zero pois a altura e largura de um retângulo não podem ser valores negativos.

Já a função construtora, definida nas linhas 13 a 16, que define os valores iniciais de `altura` e `largura`, utiliza os próprios *setters* para garantir que os valores serão válidos.

Considere agora uma função `main` que utiliza este `Retangulo`:

```
1 int main () {
2   Retangulo ra(3,2);
3   cout << "A␣área␣de␣ra␣é␣" << ra.area() << endl;
4   ra.setAltura(4);
5   ra.setLargura(7);
6   cout << "A␣área␣de␣ra␣é␣" << ra.area() << endl;
7   ra.setAltura(3);
8   ra.setLargura(-5); // tentando passar um valor inválido
```

```
9     cout << "A␣área␣de␣ra␣é␣" << ra.area() << endl;
10    return 0;
11  }
```

Na função principal, nas linha 8, tentamos atribuir valores inválidos para um retângulo mas a função não nos permite. Apenas os valores válidos são atribuídos.

A utilização de *getters* e *setters* tem várias vantagens:

- Se for necessário alterar várias variáveis no objeto, o usuário da classe não precisa fazer isto diretamente.
- Os dados enviados pelo usuário podem ser sempre validados.
- O código do usuário continua válido se for alterado como o *setter* funcionará. Isso por envolver uma troca de estrutura de dados interna, por exemplo.
- O modo como o objeto está sendo representado fica oculto.

25.5 Destrutores

Além dos construtores, podemos definir quais comandos serão executados quando um objeto for destruído. Pode parecer estranho alterar valores do objetos ou fazer ações quando não precisaremos mais dos objetos, mas isto é especialmente importante para evitar vazamento de memória quando destruímos objetos que haviam alocado espaço na memória.

Suponha uma classe utilizada para representar listas de números inteiros em um arranjo. Como não sabemos qual será o tamanho da lista, utilizaremos alocação dinâmica de memória.

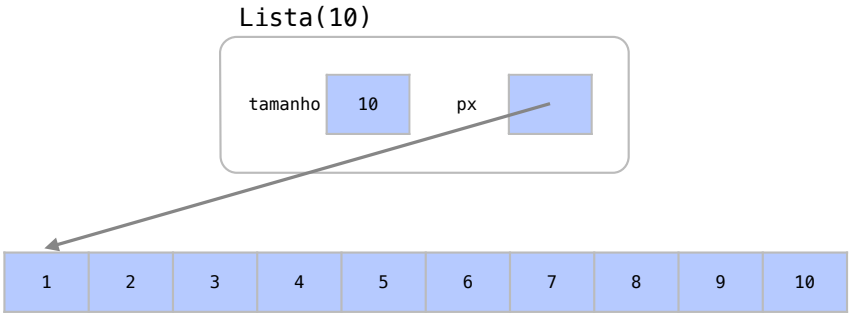
```
1  class Lista {
2    public:
3      Lista(int);
4      void setElemento(int pos, int x) {px[pos-1] = x;}
5      int getElemento(int pos) {return px[pos-1];}
6    private:
7      int *px;
8      int tamanho;
9  };
10
11 Lista::Lista(int n){
12      tamanho = n;
13      px = new int[n];
14      for (int i = 0; i < n; i++){
15          px[i] = i+1;
16      }
17 }
```

Entre os atributos da lista, definidos nas linhas 7 e 8, um ponteiro `px` apontará para o arranjo alocado na memória e um outro membro guardará o `tamanho` do arranjo alocado, que é também o tamanho da lista.

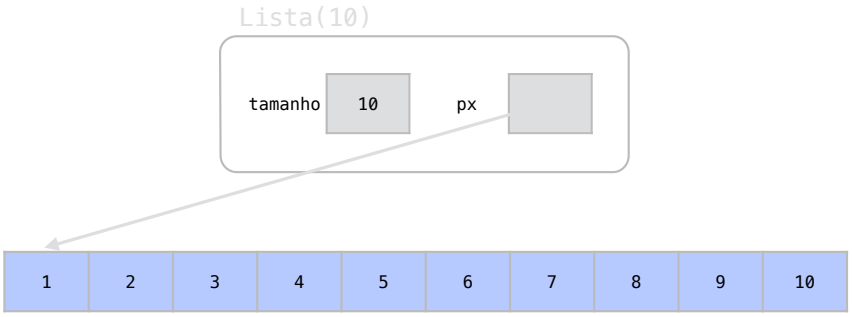
O construtor declarado na linha 3 recebe o tamanho da lista de elementos. Em sua implementação, nas linhas 11 a 17, o construtor aloca um arranjo do tamanho pedido nas linhas 12 e 13, e dá valores de 1 a n para os elementos da lista, nas linhas 14 a 16.

Os *setters* e *getters* das linhas 4 e 5 retornam o elemento em uma posição da lista. A função considera que as posições na lista são de 1 até `n`.

Veja agora o que acontece em relação aos atributos do objeto quando ele é construído. Apenas o ponteiro `px` e `tamanho` são atributos da lista. O arranjo, apesar de ser usado para representar a lista, está alocado na memória, fora do escopo do objeto.



Quando esta lista é destruída, apenas o ponteiro `px` e `tamanho` são destruídos. O arranjo continua existindo, o que causa um *vazamento de memória*.

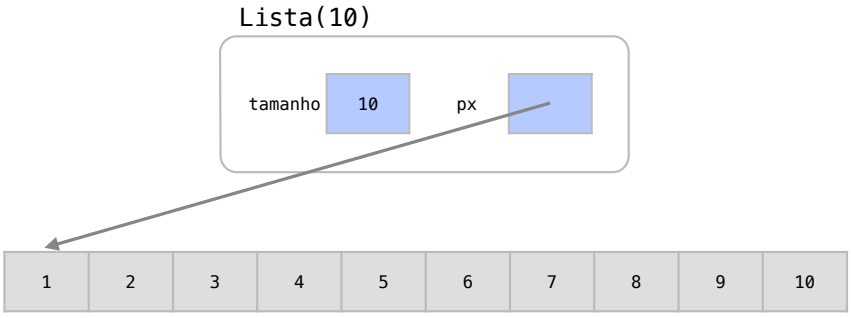


Para que isto não ocorra, precisamos definir na função destrutora que o arranjo será desalocado também sempre que o objeto for destruído. Nesta nova definição da classe, temos uma função destrutura na linha 4.

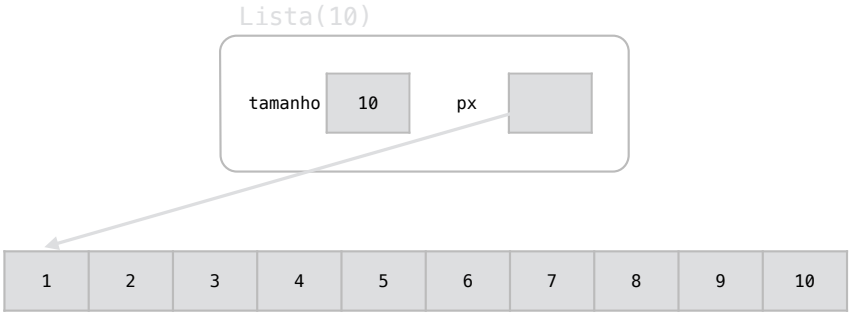
```
1  class Lista {
2    public:
3      Lista(int);
4      ~Lista();
5      void setElemento(int pos, int x) {px[pos-1] = x;}
6      int getElemento(int pos) {return px[pos-1];}
7    private:
8      int *px;
9      int tamanho;
10 };
11
12 Lista::Lista(int n){
13      tamanho = n;
14      px = new int[n];
15      for (int i = 0; i < n; i++){
16          px[i] = i+1;
17      }
18 }
19
20 Lista::~Lista(){
21      delete[] px;
22 }
```

A função destrutura tem o mesmo nome da classe precedido de um `~`. A função destrutora não recebe parâmetros e é executada sempre que um objeto será destruído.

Na implementação do destrutor, feita nas linhas 20 a 22, podemos dizer ao objeto para desalocar a memória apontada por px antes de excluir os atributos da lista.



Apenas após a execução do destrutor os atributos da lista são desalocados.



26. Relações entre objetos

Estudaremos nas próximas seções várias maneiras como objetos podem se relacionar.

26.1 Composição de objetos

É muito útil em sistemas complexos a relação na qual objetos podem conter outros objetos como membros. Por exemplo:

- Um computador que tem um teclado
- Um carro que tem 4 rodas
- Uma universidade contém vários departamentos
- Um departamento contém vários professores

Com este tipo de definição, podemos criar objetos mais complexos que os objetos originais através de composição.

Considere uma classe que define objetos do tipo `Professor`.

```
1  class Professor {
2  public:
3      Professor(string s, int x) {nome = s; cpf = x;};
4      void setNome(string s) {nome = s;}
5      void setCpf(int x) {cpf = x;}
6      string getNome() {return nome;}
7      int getCpf() {return cpf;}
8  private:
9      string nome;
10     int cpf;
11 };
```

Nos membros privados da classe, como definidos nas linhas 9 e 10, cada `Professor` tem como atributos seu `nome` e `cpf`. Os *setters* e *getters* definidos nas linhas 4 a 7 podem alterar estes dados.

Podemos agora criar uma classe `Departamento` que tem vários professores.

```
1  class Departamento {
2    public:
3      Departamento(string s) {nome = s;};
4      void contrataProfessor(Professor x) {docentes.push_back(x);}
5      int getTamanho() {return docentes.size();};
6      void imprime();
7    private:
8      vector<Professor> docentes;
9      string nome;
10 };
11
12 void Departamento::imprime(){
13      cout << nome << "\tProfessores" << endl;
14      vector<Professor>::iterator i;
15      for (i = docentes.begin(); i != docentes.end(); ++i){
16          cout << i->getCpf() << "\t" << i->getNome() << endl;
17      }
18
19 }
```

O departamento tem um conjunto de professores. O `vector` da linha 8 guardará o conjunto de professores neste departamento. Através da função da linha 4, contratamos mais professores com um `push_back` neste `vector`. Na linha 5, o "tamanho" do departamento pode ser obtido pelo próprio tamanho do conjunto de professores.

Criamos também uma função que mostra todos os professores de um departamento. A função imprime está definida nas linhas 12 a 19 e percorre todo o conjunto de professores com um iterador.

Com as definições destas classes, podemos criar na função principal um departamento, contratar professores e imprimir os dados deste departamento.

```
1 int main () {
2      Professor x("Joao", 123);
3      Departamento dept("DECOM");
4      dept.contrataProfessor(x);
5      Professor y("Jose", 345);
6      dept.contrataProfessor(y);
7      dept.imprime();
8      return 0;
9 }
```

Assim como criamos um Departamento com vários professores, poderíamos ter criado também uma classe `Universidade`, com vários departamentos, e assim por diante. Podemos perceber como composição de objetos é uma ferramenta útil para modelar sistemas complexos.

26.1.1 Destrutores de objetos membros

Quando um objeto é destruído, todos os seus itens membros são destruídos automaticamente. Além disso, quando o objeto é destruído, todos os objetos de sua propriedade também são destruídos. Assim, não é necessário se preocupar com a função destrutora de objetos membros.

26.1.2 Construtores de objetos membros

Os objetos membros de outro objetos também precisam ser construídos. A construção de todos os membros de uma classe ocorre antes da execução do próprio construtor desta classe. Porém, é possível definir como qualquer atributo será construído.

Considere a seguinte classe utilizada para representar uma roda.

```
1  class Roda {
2  public:
3      Roda();
4      Roda(double);
5      void setRaio(double x) {raio = x;}
6      double getRaio() {return raio;}
7  private:
8      double raio;
9  };
10
11 Roda::Roda(){
12     raio = 5;
13 }
14
15 Roda::Roda(double x){
16     raio = x;
17 }
```

Cada roda terá um atributo `raio` que guarda o valor de seu raio, como definido na linha .

Temos para a classe dois contrutores. O primeiro construtor, declarado na linha 3, é o construtor padrão, que em sua definição, nas linhas 11 a 13, dá valor `5` ao `raio`. O segundo construtor, declarado na linha 4 e definido nas linhas 15 a 17, guarda o valor do `raio` que é recebido como parâmetro.

Temos ainda os *getters* e *setters* para um raio nas linhas 5 e 6.

Considere agora uma segunda classe que representa motocicletas.

```
1  class Motocicleta {
2    public:
3      Motocicleta(double, double);
4      void setDianteira(Roda x) {dianteira = x;}
5      void setTraseira(Roda x) {traseira = x;}
6      Roda getDianteira() {return dianteira;}
7      Roda getTraseira() {return traseira;}
8    private:
9      Roda dianteira;
10     Roda traseira;
11 };
12
13 Motocicleta::Motocicleta(double r1, double r2){
14     dianteira.setRaio(r1);
15     traseira.setRaio(r2);
16 }
```

Cada motocicleta tem duas rodas, representadas pelos atributos das linhas 9 e 10. Não temos um construtor padrão para a `Motocicleta`. Temos apenas um construtor que recebe 2 parâmetros, declarado na linha 3.

Nas linhas 13 a 16, na definição do construtor da motocicleta, associa-se os valores passados como parâmetro às rodas da motocicleta. Para isto, a função `setRaio` é utilizada.

Note que antes mesmo do primeiro comando do construtor ser executado na linha 14, os objetos que representam as rodas já foram construídos. Só por isto podemos utilizar a função `setRaio`. Os objetos já estão construídos com seus construtores padrão, que dão valor `5` ao raio de qualquer roda. Assim, precisamos apenas utilizar a função de `setRaio` para definir o raio das rodas que *já existem* na linha 14.

Deste modo, o construtor da motocicleta realiza 2 passos:

1. Antes mesmo do bloco de comandos se iniciar, cada Roda é construída com o construtor padrão (`raio` recebe `5`)
2. O raio de cada roda é alterado

Veja como uma `Roda`, além do construtor padrão, tem um construtor que recebe um valor `double`. Seria muito bom se pudéssemos já definir como uma roda da motocicleta deve ser construída. Assim, poderíamos utilizar o construtor que já recebe o raio da roda e economizar operações.

Podemos fazer isto entre a passagem dos parâmetros para o construtor e início do bloco de comandos.

```
1 Motocicleta::Motocicleta(double r1, double r2):
2 dianteira(r1), traseira(r2){
3     // Não precisamos de comando algum aqui
4 }
```

Deste modo, o segundo construtor para a `Roda` já é chamado e não precisamos alterar o raio da roda após sua criação pois elas já são criadas com os valores corretos.

De maneira similar, o `double` que guarda o raio de uma roda já pode ser criado com seu valor correto.

```
1 Roda::Roda():
2 raio(5){
3     // Não precisamos de comando algum aqui
4 }
5
6 Roda::Roda(double x):
7 raio(x){
8     // Não precisamos de comando algum aqui
9 }
```

Esta, porém, não é uma operação que economizaria tantos recursos.

Note que outra vantagem de se definir como os objetos membros serão criados é que não precisamos de um construtor padrão. A `Motocicleta`, por exemplo, não precisa mais do construtor padrão definido pela função `Raio()` para funcionar.

Inicializar os membros de um objeto antes de começar a função chama os construtores dos membros. Esta é uma boa prática pois evita utilizar de maneira desnecessária o construtor padrão para estes itens, o que pode ser um desperdício grande de recursos. Além disto, alguns objetos podem não ter construtores padrão e esta é a única maneira de inicializá-los.

26.2 Herança

Quando uma classe *herda* de outra classe, ele pega para si todas as funções e membros desta classe. Chamamos esta classe superior de **superclasse**.

A Figura 26.1 apresenta um exemplo de relação de herança. Os retângulos e triângulos são todos polígonos. Por isto, eles devem necessariamente ter todas os atributos e comportamentos de polígonos.

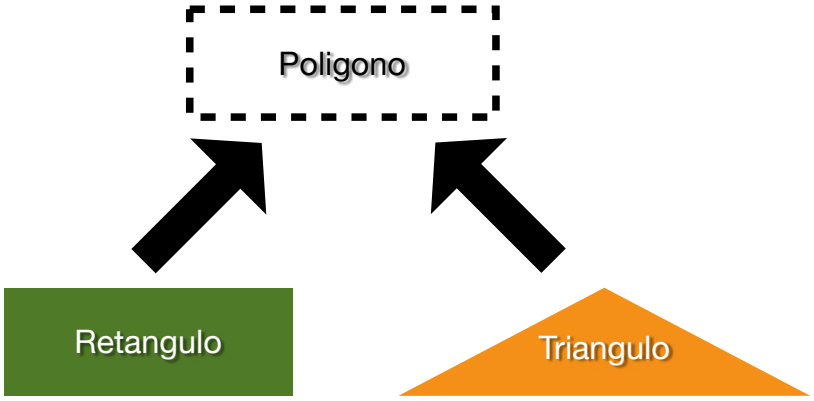


Figura 26.1: Representação em diagrama de programação estruturada.

Note que triângulos e retângulos **não contém** polígonos. Eles **são** polígonos. Não podemos confudir relações de composição com relações de herança. As motocicletas **contém** rodas. As motocicletas **não são** rodas. Os retângulos **são** polígonos. Os triângulos **são** polígonos.

Na relação de herança apresentada, retângulos e triângulos têm os mesmos comportamentos e atributos de polígonos. Polígonos, porém, não necesserariamente têm todos os comportamentos e atributos de retângulos e triângulos.

Dizemos que o `Poligono` é uma **superclasse** e que o as classes `Retangulo` e `Triangulo` herdam da classe `Poligono`.

Suponha de modo mais concreto uma classe utilizada para representar polígonos.

```
1 class Poligono {
2 public:
3     void set_valores (int a, int b){
4         largura=a;
5         altura=b;
6     }
7 protected:
8     int largura, altura;
9 };
```

Nesta classe, vamos poder representar polígonos que tenham largura e altura. Veja que temos uma nova palavra-chave: `protected`. Os membros protegidos de uma classe não são acessíveis por quem cria objetos desta classe, assim como os membros privados. Porém, membros protegidos são acessíveis por objetos que herdam desta classe.

Nas linhas 3 a 6, temos também uma função pública que define os valores de `largura` e `altura`. Qualquer objeto que herdar de Poligono, terá também este recurso.

Veja agora esta classe utilizada para representar retângulos.

```
1 class Retangulo: public Poligono {
2 public:
3     int area (){
4         return largura * altura;
```

```
5     }
6 };
```

Na linha 1, o trecho : `public Poligono` faz com que a classe `Retangulo` herde de `Poligono`.

Além das funções definidas em `Poligono`, a classe `Retangulo` tem uma função que utiliza seus valores de `altura` e `largura` para cálculo de sua área. Assim, apesar de todo `Retangulo` ser um `Poligono`, ele também pode ter seus atributos e comportamentos específicos.

Deste modo, a definição acima de `Retangulo` é equivalente à seguinte definição:

```
1  class Retangulo {
2  public:
3     void set_valores (int a, int b){
4          largura=a;
5          altura=b;
6      }
7      int area (){
8          return largura * altura;
9      }
10 private:
11     int largura, altura;
12 };
```

Pela diferença entre as definições equivalentes, note como a herança facilita o reaproveitamento de código.

De volta a nosso exemplo, podemos definir outras classes de polígonos que tenham `altura` e `largura`. Definimos então uma nova classe `Triangulo` que também herda os atributos e comportamentos de `Poligono`.

```
1  class Triangulo: public Poligono {
2  public:
3      int area (){
4          return (largura * altura)/2;
5      }
6  };
```

Note que cada classe que herda de `Poligono` pode ter também seus atributos e comportamentos específicos. Neste caso, o cálculo da área é diferente para um triângulo. Note mais uma vez como isto facilita o reaproveitamento de código.

Através da herança, conseguimos definir apenas com uma implementação de `set_valores` os valores de qualquer objeto que herde de `Poligono`.

```
1  int main () {
2    Retangulo rect;
3    Triangulo trgl;
4    rect.set_valores(4,5);
5    trgl.set_valores(4,5);
6    cout << rect.area() << endl;
7    cout << trgl.area() << endl;
8    return 0;
9  }
```

Além da atribuição de valores feita nas linhas 4 e 5, conseguimos também utilizar diretamente as funções específicas destes objetos, como fizemos nas linhas 6 e 7.

```
20
10
```

26.2.1 Polimorfismo

Outra possibilidade interessante relacionada à herança entre objetos é polimorfismo. Um retângulo, por exemplo, por ser também um polígono, pode ser tratado então simplesmente como `Retangulo` ou como `Poligono`. Em outras palavras, um objeto pode assumir mais de uma forma, dependendo de sua relação de herança.

Veja este exemplo de como um retângulo ou um triângulo podem ser tratados simplesmente como polígonos.

```
1  int main () {
2    Retangulo rect;
3    Triangulo trgl;
4    Poligono * ppoli1 = &rect; // Retangulo é um Poligono
5    Poligono * ppoli2 = &trgl; // Triangulo é um Poligono
6    ppoli1->set_valores(4,5);
7    ppoli2->set_valores(4,5);
8    cout << rect.area() << endl;
9    cout << trgl.area() << endl;
10   return 0;
11 }
```

O exemplo cria um `Retangulo` e um `Triangulo` nas linhas 2 e 3. Em seguida, nas linhas 4 e 5, são criados dois ponteiros para polígonos. Um aponta para o retângulo e outro para o triângulo. Não há problema nisto pois um retângulo e um triângulo também são polígonos! Eles podem assumir esta forma.

Utilizamos nas linhas 6 e 7 os ponteiros então para dar valores aos polígonos. Isto também é possível pois `set_valores` é uma função comum a polígonos.

Em seguida, nas linhas 8 e 9, acessamos as funções de área que pertencem aos objetos.

```
20
10
```

Note que o seguinte código geraria erro.

```
1  int main () {
2    Retangulo rect;
3    Triangulo trgl;
4    Poligono * ppoli1 = &rect;
5    Poligono * ppoli2 = &trgl;
6    ppoli1->set_valores(4,5);
7    ppoli2->set_valores(4,5);
8    cout << ppoli1->area() << endl;
9    cout << ppoli1->area() << endl;
10   return 0;
11 }
```

Nas linhas 8 e 9, a função `area` não é uma função comum aos polígonos definida em `Poligono` e sim funções específicas das classes `Retangulo` e `Triangulo`.

27. Recursos de Objetos

27.1 Templates de Objetos

Como já vimos na Seção 17, templates são recursos poderosos de programação genérica pois generalizam conceitos para quaisquer tipos de dados. Templates são muito úteis para criar funções que funcionam com vários tipos variáveis. Usamos também no Capítulo III vários objetos que representam containers. Através de templates, estes containers conseguem guardar qualquer tipo de dado.

Veja esta classe para representar um par de números inteiros.

```
1  class par {
2  public:
3      par (int a, int b){
4          primeiro=a;
5          segundo=b;
6      }
7      int getPrimeiro() {return primeiro;}
8      int getSegundo() {return segundo;}
9      int getMaior();
10     int getMenor();
11 private:
12     int primeiro;
13     int segundo;
14 };
15
16 int par::getMaior(){
17   if (primeiro > segundo){
18       return primeiro;
19   } else {
20       return segundo;
21   }
```

```
22 }
23
24 int par::getMenor(){
25   if (primeiro < segundo){
26       return primeiro;
27   } else {
28       return segundo;
29   }
30 }
```

Os únicos atributos desta classe são os dois números inteiros, declarados nas linhas 12 e 13. Entre os recursos da classe podemos não só retornar o primeiro e segundo elementos mas também o maior e o menor elementos, pelas funções declaradas nas linhas 7 a 10. Por serem maiores, as funções `getMaior` e `getMenor` estão definidas fora da classe, nas linhas 16 a 30.

Veja o resultado da execução deste programa.

```
1 int main () {
2     par objeto(6, 12);
3     cout << "Par␣de␣objetos" << endl;
4     cout << "Primeiro␣=␣" << objeto.getPrimeiro() << endl;
5     cout << "Segundo␣=␣" << objeto.getSegundo() << endl;
6     cout << "Maior␣=␣" << objeto.getMaior() << endl;
7     cout << "Menor␣=␣" << objeto.getMenor() << endl;
8     return 0;
9 }
```

```
Par de objetos
Primeiro = 6
Segundo = 12
Maior = 12
Menor = 6
```

Parece claro que um objeto que guarda um par de qualquer tipo de dados não é muito diferente de um objeto que guarda um par de dados do tipo `int`. Para isto, com a seguinte conversão, conseguimos representar a mesma classe com templates.

```
1 template <class T>
2 class par {
3 public:
4     par (T a, T b){
5         primeiro=a;
6         segundo=b;
7     }
8     T getPrimeiro() {return primeiro;}
9     T getSegundo() {return segundo;}
10    T getMaior();
11    T getMenor();
12 private:
13    T primeiro;
14    T segundo;
```

```
15 };
16
17 template <class T>
18 T par<T>::getMaior(){
19   if (primeiro > segundo){
20       return primeiro;
21   } else {
22       return segundo;
23   }
24 }
25
26 template <class T>
27 T par<T>::getMenor(){
28   if (primeiro < segundo){
29       return primeiro;
30   } else {
31       return segundo;
32   }
33 }
```

Na linha 1, iniciamos a definição da classe avisando que `T` representará um template. Em toda a definição da classe, até a linha 15, substituímos então todas as ocorrências de `int` pelo tipo de dado `T`, representando um template.

As implementações das funções do objeto também precisam ser feitas com templates. Por isto os templates `T` são definidos novamente nas linhas 17 e 26, antes do início da implementação. Pelas definições das linhas 18 e 27, vamos que estas funções agora retornam dados do tipo `T` e pertencem ao objeto `par<T>`.

Veja agora o resultado da execução deste programa.

```
1 int main () {
2     par<int> objeto(6, 12);
3     cout << "Par␣de␣objetos" << endl;
4     cout << "Primeiro␣=␣" << objeto.getPrimeiro() << endl;
5     cout << "Segundo␣=␣" << objeto.getSegundo() << endl;
6     cout << "Maior␣=␣" << objeto.getMaior() << endl;
7     cout << "Menor␣=␣" << objeto.getMenor() << endl;
8     return 0;
9 }
```

```
Par de objetos
Primeiro = 6
Segundo = 12
Maior = 12
Menor = 6
```

Obtivemos o mesmo resultado utilizando templates. A única alteração necessária na função principal foi definir com qual tipo de dado queremos utilizar o objeto. Isto foi feito com `Par<int>`. Indicamos assim qual tipo de dado será utilizado no lugar do template. Isto é feito da mesma maneira que utilizamos containers, que são objetos que utilizam templates para representar estruturas de dados.

Agora, é claro, podemos utilizar nosso objeto com qualquer tipo de dado ou objeto.

```
1 int main () {
2     par<char> objeto('g', 't');
3     cout << "Par de objetos" << endl;
4     cout << "Primeiro = " << objeto.getPrimeiro() << endl;
5     cout << "Segundo = " << objeto.getSegundo() << endl;
6     cout << "Maior = " << objeto.getMaior() << endl;
7     cout << "Menor = " << objeto.getMenor() << endl;
8     return 0;
9 }
```

```
Par de objetos
Primeiro = g
Segundo = t
Maior = t
Menor = g
```

```
1 int main () {
2     par<double> objeto(6.7, 3.2);
3     cout << "Par de objetos" << endl;
4     cout << "Primeiro = " << objeto.getPrimeiro() << endl;
5     cout << "Segundo = " << objeto.getSegundo() << endl;
6     cout << "Maior = " << objeto.getMaior() << endl;
7     cout << "Menor = " << objeto.getMenor() << endl;
8     return 0;
9 }
```

```
Par de objetos
Primeiro = 6.7
Segundo = 3.2
Maior = 6.7
Menor = 3.2
```

27.2 Operador de Conversão

Um problema que pode ocorrer com os novos objetos que criamos é que podem não existir funções que funcionem com ele, afinal, estamos criando um novo tipo de dado. Em alguns casos, o operador de conversão pode ser útil para transformar um objeto em um outro tipo de dado já conhecido.

Veja este exemplo onde temos uma classe `Roda`.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Roda{
6     public:
```

```
7           Roda(double x) {raio = x;}
8           void setRaio(double x) {raio = x;};
9           double getRaio() {return raio;}
10      private:
11          double raio;
12  };
13
14  int main () {
15      Roda x(7.2);
16      cout << "Roda de raio " << x.getRaio() << endl;
17      return 0;
18  }
```

Utilizamos esta classe na função principal para criarmos uma `Roda` e imprimirmos seu raio.

```
Roda de raio 7.2
```

Para imprimir o raio, utilizamos a função `getRaio`, que retorna um `double`. A impressão só é possível porque o objeto `cout` reconhece variáveis do tipo `double`. Já o seguinte comando levaria a um erro:

```
1  cout << "Roda de raio " << x << endl;
```

Isto porque o objeto `cout` não sabe trabalhar com variáveis do tipo `Roda`. Assim, neste próximo exemplo, criamos um operador de conversão.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Roda{
6      public:
7          Roda(double x) {raio = x;}
8          void setRaio(double x) {raio = x;};
9          double getRaio() {return raio;}
10         operator double(){return raio;}
11     private:
12         double raio;
13 };
14
15 int main () {
16     Roda x(7.2);
17     cout << "Roda de raio " << x.getRaio() << endl;
18     cout << "Roda de raio " << x << endl;
19     return 0;
20 }
```

Este operador `double()`, na linha 10, define como converter uma `Roda` em um `double`, retornando o valor de seu raio.

Na linha 18, o objeto `cout` agora funciona com uma `Roda`, já que existe um operador que converte uma roda para um tipo de dados conhecido por ele.

```
Roda de raio 7.2
Roda de raio 7.2
```

27.3 Interface e Implementação

Em projeto grandes, ao definir uma classe, é comum termos dois componentes:

- Uma **interface**, que inclui o cabeçalho das funções, a definição dos atributos e uma implementação das funções simples
- Uma **implementação**, onde temos a definição das funções declaradas nos cabeçalhos

É uma prática vantajosa separar estes dois componentes. Para isto, usualmente os objetos são separados em arquivos `.h` e `.cpp`.

Vejamos novamente um exemplo de classe de objetos.

```
1  // Cabeçalho / Interface
2  class Par {
3  public:
4      Par (int a, int b){
5          primeiro=a;
6          segundo=b;
7      }
8      int getPrimeiro() {return primeiro;}
9      int getSegundo() {return segundo;}
10     int getMaior();
11     int getMenor();
12 private:
13     int primeiro;
14     int segundo;
15 };
16
17 // Código-Fonte / Implementação
18 int Par::getMaior(){
19   if (primeiro > segundo){
20       return primeiro;
21   } else {
22       return segundo;
23   }
24 }
25
26 int Par::getMenor(){
27   if (primeiro < segundo){
28       return primeiro;
29   } else {
30       return segundo;
31   }
32 }
```

Nas linhas 1 a 16, temos a definição dos cabeçalhos, onde chamamos de interface. Nas linhas 17 a 32, temos o código-fonte que implementa as funções. É comum separar estas duas coisas em dois arquivos separados com o nome da função. Neste caso, a interface ficaria em um arquivo `Par.h` e a implementação ficaria em um arquivo `Par.cpp`.

No arquivo `Par.h` temos os recursos da classe. Este é o arquivo que será consultado por um usuário da classe.

```
1  // Arquivo Par.h
2  #ifndef PAR_H
3  #define PAR_H
4
5  class Par
6  {
7      public:
8          Par(int, int);
9          int getPrimeiro() {return primeiro;}
10         int getSegundo() {return segundo;}
11         int getMaior();
12         int getMenor();
13     private:
14         int primeiro;
15         int segundo;
16 };
17
18 #endif
```

Note que todos os recursos estão nas linhas 5 a 16. Note que acrescentamos os comandos `#ifndef PAR_H` na linha 2, `#define PAR_H` na linha 3 e `#endif` na linha 18. Estes comandos garantem que a classe `Par` só seja definida se não estiver definida ainda. Isso acontece se tentarmos incluir a classe mais de uma vez no código.

O arquivo `Par.cpp` com o código-fonte pode ser visto abaixo:

```
1  // Arquivo Par.cpp
2  #include "Par.h"
3
4  Par::Par(int a, int b){
5      primeiro = a;
6      segundo = b;
7  }
8
9  int Par::getMaior(){
10     if (primeiro > segundo){
11         return primeiro;
12     } else {
13         return segundo;
14     }
15 }
16
17 int Par::getMenor(){
18     if (primeiro < segundo){
19         return primeiro;
20     } else {
21         return segundo;
22     }
23 }
```

Incluímos o arquivo `Par.h` na linha 2. Isto equivale a replicar todo o código de `Par.h` no início de `Par.cpp`. Note que as inclusões de nossos arquivos são feitas com aspas (`"Par.h"`) e não com chaves (`<Par.h>`).

Para utilizar nossa classe, o usuário precisa somente do arquivo `Par.h` para conhecer seus recursos.

Note como o arquivo `Par.h` deve ser incluído em nosso arquivo `main.cpp` e como isto torna nosso código mais organizado.

```
1  // Arquivo main.cpp
2  #include <iostream>
3  #include "Par.h"
4
5  using namespace std;
6
7  int main(){
8      Par x(2,3);
9      cout << "Maior:␣" << x.getMaior() << endl;
10     cout << "Menor:␣" << x.getMenor() << endl;
11     return 0;
12 }
```

Temos agora todos os recursos da classe Par em nosso arquivo principal.

```
Maior:  3
Menor:  2
```

A separação da interface e implementação é muito útil por vários motivos. Pessoas diferentes podem trabalhar em arquivos diferentes. A implementação pode ficar escondida do usuário da classe. Se os cabeçalhos forem mantidos, as implementações podem ser alteradas sem causar problemas ao usuário da classe. Por fim, o código fica mais organizado e modularizado.

27.4 Exercícios

Exercício 27.1 Analise o código abaixo, que está incompleto. Ela define o objeto `Aluno` para ser usado em um sistema de cadastro de uma academia.

- A classe já define como imprimir um aluno, atribuir dados, retornar dados e comparar alunos.
- A função `apresentaMenu` entra em um laço que permite trabalhar com um conjunto de alunos.
- Utilizando a classe de alunos, o menu apresenta funções para cadastrar e remover alunos.
- A intenção deste exercício é fazer um sistema de matrícula de alunos.
- Os alunos são cadastrados em um map que usa o número de matrícula como chave para organizar os dados.

```
1  #include <iostream>
2  #include <string>
3  #include <map>
4
5  using namespace std;
6
```

```
7  class Aluno {
8  public:
9      void imprime(){
10         cout << matricula << "␣-␣" << nome << endl;
11         return;
12     }
13
14     // Funções modificadoras
15     void setValores(string x, int y){
16         nome = x;
17         matricula = y;
18     }
19
20     void setNome(string x){
21         nome = x;
22         return;
23     }
24
25     void setMatricula(int x){
26         matricula = x;
27         return;
28     }
29
30     // Funções não-modificadoras
31     string getNome(){
32         return nome;
33     }
34
35     int getMatricula(){
36         return matricula;
37     }
38
39     bool operator<( const Aluno x )const {
40         return (this->matricula < x.matricula);
41     }
42
43 private:
44     int matricula;
45     string nome;
46 };
47
48 void apresentaMenu(map<int,Aluno> &cadastro);
49 void imprimeCadastro(map<int,Aluno> &x);
50 void insereAluno(map<int,Aluno> &x);
51 void pesquisaAluno(map<int,Aluno> &x);
52
53 int main(){
54     map<int,Aluno> cadastro; // chave é a matricula (int) e
          registro é o próprio aluno
```

```
55      cout << "Bem␣vindo␣ao␣sistema␣de␣cadastro␣de␣alunos␣da␣
            academia" << endl;
56      string opcao;
57      apresentaMenu(cadastro);
58      cout << "\nObrigado␣por␣utilizar␣nosso␣sistema␣de␣
            cadastro␣BCC702" << endl;
59      return 0;
60  }
61
62  void apresentaMenu(map<int,Aluno> &cadastro){
63      string opcao;
64      do {
65          cout << "\n****␣Sistema␣de␣cadastro␣de␣alunos␣****"
                << endl;
66          cout << "[1]␣Imprimir␣lista␣com␣todos␣os␣alunos" <<
                endl;
67          cout << "[2]␣Inserir␣um␣novo␣aluno" << endl;
68          cout << "[3]␣Consultar␣um␣aluno" << endl;
69          cout << "[0]␣Sair␣do␣programa" << endl;
70          cout << "Digite␣uma␣opcao:␣";
71          cin >> opcao;
72          if (opcao == "1"){
73              imprimeCadastro(cadastro);
74          } else if (opcao == "2") {
75              insereAluno(cadastro);
76          } else if (opcao == "3") {
77              pesquisaAluno(cadastro);
78          }
79      } while (opcao != "0");
80      return;
81  }
82
83  void imprimeCadastro(map<int,Aluno> &x){
84      map<int,Aluno>::iterator i;
85      for (i=x.begin(); i!=x.end(); ++i){
86          i->second.imprime();
87      }
88      return;
89  }
90
91  void insereAluno(map<int,Aluno> &x){
92      Aluno temp;
93      string tempnome;
94      int tempmatricula;
95      cout << "Digite␣o␣nome␣do␣aluno:␣";
96      cin.ignore(100,'\n'); // getline difere de cin pois lê
            toda a linha
97      getline (cin,tempnome);
98      temp.setNome(tempnome);
```

```
99      cout << "Digite␣o␣numero␣de␣matricula:␣";
100     cin >> tempmatricula;
101     temp.setMatricula(tempmatricula);
102     x[tempmatricula] = temp;
103     return;
104 }
105
106 void pesquisaAluno(map<int,Aluno> &x){
107     cout << "Digite␣o␣numero␣de␣matricula:␣";
108     int numero;
109     cin >> numero;
110     map<int,Aluno>::iterator i;
111     i = x.find(numero);
112     if (i == x.end()){
113         cout << "Nao␣existe␣aluno␣com␣o␣numero␣de␣matricula
                ␣" << numero << endl;
114     } else {
115         i->second.imprime();
116     }
117 }
```

■

Exercício 27.2 Imagine que cada aluno pode agora ser cadastrado em vários cursos da academia. O **conjunto** (`set`) de atividades (musculacao, danca, spinning, etc...) no qual um aluno pode estar cadastrado pode então ser representado com um `set<int>`, onde o inteiro representa o código da atividade (por exemplo,1 para musculacao, 2 para danca, etc...).

- Crie esta variável privada `set<int>` para cada aluno.
- Crie funções na classe `Aluno` que matriculem ou desmatriculem o aluno de uma atividade
- Crie uma função na classe `Aluno` para imprimir as atividades em quais atividades ele está matriculado
- Crie uma função na classe `Aluno` que retorne em quantas atividades ele está matriculado
- Insira uma opção no menu para matricular o `Aluno` em uma atividade
- Insira uma opção no menu para desmatricular o `Aluno` de uma atividade
- Altere a função de listar todos os alunos para que ela mostre em quantas atividades cada `Aluno` está matriculado
- Altere a função de pesquisa do menu para que ela mostre também em quais atividades um `Aluno` está matriculado

■

Exercício 27.3 Os construtores são funções que são chamadas sempre que um `Aluno` é criado.

- Suponha que todo Aluno cadastrado ganha uma inscrição grátis na aula musculação, que tem o código `1`. Faça um construtor que insira automaticamente esta aula de musculação para todo aluno que é criado no cadastro.
- É possível criar um contrutor que também já receba valores iniciais. Crie um construtor que já receba o nome e número de matrícula do aluno para não precisarmos de `setNome()` e `setMatricula()` na função `insereAluno()`.

Exercício 27.4 As funções que se iniciam com `set` e `get` são utilizadas por padrão para que o usuário utilize a classe de maneira segura sem ter acesso a variaveis privadas diretamente.

- Altere a função que imprime as atividades de um aluno para que em vez dos códigos das atividades sejam exibidos os nomes das atividades. Crie nomes de atividades e para cada número deve ser impresso um nome.
- Altere as funções que inscrevem os alunos em atividades para que apenas números de atividades válidas sejam cadastradas. Caso não seja escolhida uma atividade válida, uma mensagem de erro deve ser exibida.
- Suponha agora que o número de matricula deve ter no minimo 4 digitos (`numero > 999`). Altere as funções que definem números de matrícula para que apenas numeros válidos sejam cadastrados. Caso não seja escolhido um número válido, uma mensagem de erro deve ser exibida.

V Práticas de Programação em C++

28. Práticas Comuns de Programação em C++

Neste capítulo discutiremos brevemente práticas importantes de programação que não podem ser discutidas em uma introdução didática aos tópicos de programação. Discutiremos recursos do C++ que formam práticas de programação distintas para programadores especializados em C++.

28.1 Ponteiros inteligentes

Ponteiros inteligentes são uma classe baseada em templates que imita o comportamento de ponteiros, chamados aqui de ponteiros crus. Esta imitação ocorre através da sobrecarga de operadores.

Ponteiros que apontam para alguma memória alocada dinamicamente com a instrução `new` precisam podem gerar vazamento de memória caso o programador não desaloque a memória ao fim com a instrução `delete`.

Os ponteiros inteligentes, em C++, desalocam automaticamente a memória apontada aos destruir o objeto que simula um ponteiro, facilitando assim a prática de programação. O seguinte comando cria um ponteiro inteligente para números inteiros:

```
1 shared_ptr<int> p;
```

O tipo de dado `shared_ptr` é utilizado para ponteiros inteligentes compartilhados: o tipo mais comum de ponteiros inteligentes. Seu construtor pode receber como parâmetro um objeto sendo alocado na memória ou mesmo o endereço.

```
1 #include <memory>
2 // ...
3 shared_ptr<int> p(new int);
4 *p = 10;
5 shared_ptr<int> p2;
6 p2 = p1;
7 p1 = make_shared<int> (20);
```

A função `unique()` retorna se um ponteiro é o único que aponta para aquele endereço. Os dados apontados por um ponteiro só são desalocados quando mais nenhum ponteiro aponta para o endereço.

Além do ponteiro inteligente `shared_ptr`, existem os ponteiros inteligentes `unique_ptr`, que não permitem que mais de um ponteiro aponte para o mesmo endereço, e `weak_ptr`, que são utilizados para guardar ponteiros crus.

28.2 Dedução do tipo de dado

Em várias situações, o C++ pode deduzir pelo o código o tipo de dado necessário para uma variável. Sempre que é possível inferir o tipo de dado, a instrução `auto` pode ser utilizada em vez se escrever o tipo da variável. As duas linhas de código abaixo têm o mesmo efeito para criar variáveis do tipo `int`.

```
1 int x = 10;
2 auto y = 10;
```

É claro que este é um caso onde não existe um ganho no uso deste recurso. O comando `auto` é especialmente útil quando trabalhamos com templates e iteradores:

```
1 vector<int> vec;
2 auto itr = vec.iterator();
```

No exemplo, utilizamos `auto` para definir o tipo de dado do iterador. Sem a função `auto`, teríamos que definir o tipo correto do iterador `vector<int>::iterator`.

A função `decltype` é complementar a `auto`. Ela consegue determinar o tipo de uma variável em relação ao tipo de outra. No exemplo seguinte, conseguimos criar uma variável y que tem o mesmo tipo de dados de `x`, mesmo sem precisarmos saber o tipo de `x` neste ponto do código.

```
1 decltype(x) y = x;
```

28.3 Objetos de função

Um objeto de função é um objeto que criamos apenas para guardar uma função útil como parâmetro para outra função. Estes objetos podem ser tratados como se fossem funções.

No exemplo abaixo, chamamos a função `sort` para ordenar uma sequência de elementos.

```
1 int main()
2 {
3     vector<int> items v{ 4, 3, 1, 2 };
4     sort(v.begin(), v.end());
5     return 0;
6 }
```

Perceba que na chamada da função `sort`, está implicito que o critério de comparação que define a ordem entre dois elementos é o operador <. De modo explícito, se quisermos utilizar qualquer outra critério de comparação, a função `sort` precisa receber um terceiro parâmetro que é uma função indicando este critério.

```
1 bool comparaInt(int a, int b)
2 {
3   return a < b;
4 }
5
```

```
6  int main()
7  {
8      vector<int> items v{ 4, 3, 1, 2 };
9      sort(v.begin(), v.end(), comparaInt);
10     return 0;
11 }
```

Neste exemplo, a função `comparaInt` diz se um elemento é maior que o outro e este critério é utilizado para ordenar a sequência. Assim, se quisermos ordenar a sequência em ordem decrescente podemos utilizar uma função que retorna o contrário de `a < b`.

```
1  bool comparaInt(int a, int b)
2  {
3    return a > b;
4  }
5
6  int main()
7  {
8      vector<int> items v{ 4, 3, 1, 2 };
9      sort(v.begin(), v.end(), comparaInt);
10     return 0;
11 }
```

Em C++, um objeto com uma função definida pelo operador `()` pode ser utilizada no lugar da função passada como argumento. Este é um objeto de função:

```
1  class comparaInt {
2    public:
3    bool operator()(const int &a, const int &b) const {
4      return a < b;
5    }
6  };
7
8  int main()
9  {
10     vector<int> items v{ 4, 3, 1, 2 };
11     sort(v.begin(), v.end(), comparaInt);
12     return 0;
13 }
```

O objeto de função tem uma função `operator()` que determina como se comparar dois números inteiros. A função `sort` pode utilizar então este objeto para ordenar a sequência de elementos. Os objetos de função podem ter seus próprios atributos e comportamentos, estendendo a utilidade de funções auxiliares.

28.4 Funções anônimas

Além destes recursos para passagem de funções como parâmetros, o C++ permite a utilização de funções anônimas, ou funções lambda. Uma função lambda é uma função que pode ser escrita no meio do próprio código para ser passada para outra função ou até mesmo guardada em uma variável. Com as funções anônimas, as criação de funções pequenas se torna muito mais fácil. O código abaixo apresenta um exemplo de função lambda:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     auto funcao = [] () { cout << "Olá Mundo!"; };
8     funcao(); // chamando a função guardada na variável
9 }
```

No exemplo acima, o `[]` indica o início de uma função lambda. Em seguida temos a lista vazia de argumentos `()`. Não é necessário definir o tipo de valor retornado por funções lambda pois isto pode ser deduzido pelo compilador. A função é guardada em uma variável chamada `funcao` e não é executada na linha 7. Na linha 8, a função guardada na variável `funcao` é executada.

Assim, lambdas são abordagens muito mais fáceis para enviar funções auxiliares em algoritmos da STL, como no exemplo abaixo:

```
1 int main()
2 {
3     vector<int> items v{ 4, 3, 1, 2 };
4     sort(v.begin(), v.end(), [] (int a, int b){return a < b;});
5     return 0;
6 }
```

Neste exemplo, uma função lambda passada para `sort` determina como devem ser comparados dois números inteiros. Para ordenar os elementos em ordem decrescente poderíamos apenas utilizar:

```
1 sort(v.begin(), v.end(), [] (int a, int b){return a > b;});
```

Além da praticidade, as funções lambda também podem capturar elementos não passados como parâmetro. A Tabela 28.1 apresenta estas formas de se capturar elementos.

Captura	Descrição
`[]`	Não captura variáveis externas
`[&]`	Captura qualquer variável externa por referência
`[=]`	Captura qualquer variável externa fazendo uma cópia
`[=,&v1]`	Captura tudo por valor e a variável `v1` por referência
`[v1]`	Captura `v1` fazendo uma cópia
`[&v1]`	Captura `v1` por referência

Tabela 28.1: Opções para captura de variáveis em funções lambda.

Este recurso permite o acesso a dados externos às funções auxiliares. No exemplo a seguir, utilizaremos capturas para remover elementos menores que um certo valor na sequência.

```
1 int main()
2 {
3     vector<int> c { 1,2,3,4,5,6,7 };
4     int x = 5;
5     remove_if(c.begin(), c.end(), [x](int n) { return n < x; } );
6 }
```

A função `remove_if`, neste exemplo remove todos os elementos que são menores que `x`. A função recebe o critério de remoção de elementos através de uma função lambda que recebe números inteiros como parâmetro mas é definida em termos de um valor `x`, que é externo.

Por fim, uma função lambda pode ser guardada em uma variável do tipo `function`.

```
1 function<int(int)> funcao = [](int i) { return i+10; };
2 cout << "Resultado: " << funcao(6) << '\n';
```

28.5 For baseado em intervalo

O `for` baseado em intervalos é uma sintaxe para escrever laços sobre elementos de maneira conveniente. O código seguinte imprime todos os elementos de uma sequência.

```
1 vector<int> c { 1,2,3,4,5,6,7 };
2 for (int i : c )
3 {
4     cout << i;
5 }
```

Este código imprime o conteúdo de um `vector`, com a variável `i` tendo a cada passo o valor de cada elemento.

Para tipos de dados mais complicados, a instrução `auto` pode ser utilizada para facilitar o uso da estrutura.

```
1 map<int, string> calendario;
2 for ( auto entrada : calendario ) {
3     cout  << entrada.first << " - " << entrada.second << endl;
4 }
```

Para modificar os dados na sequência ou evitar copiar dados por valor, a sintaxe permite acessar os dados por referência. O código abaixo incrementa todos os valores da sequência.

```
1 vector<int> c { 1,2,3,4,5,6,7 };
2 for (int &i : c ) {
3     i++;
4 }
```

28.6 Tuplas

Tuplas são objetos utilizados para guardar uma combinação de elementos de diferentes tipos. Uma maneira de se retornar mais de um tipo de dado de uma função em C++ é através da utilização de tuplas.

Neste exemplo criaremos tuplas e acessaremos seus elementos.

```
1 #include <iostream>
2 #include <tuple>
3
4 int main ()
5 {
6   // criamos tupla com letra c e número 15
7   tuple<int,char> t('c',15);
8   // criamos tupla com função make_tuple
9   auto t2 = make_tuple (6.3, "texto", 2, 'o');
```

```
10
11    // acessamos elemento na posição 2 da tupla t2
12    get<2>(t2) = 125;
13
14    // desempacotamos elementos da tupla em variáveis
15    int x;
16    char y;
17    tie (y, x) = t;
18    // ignorando algumas das variáveis
19    tie (ignore, ignore, x, y) = t2;
20    // com a função get
21    y = std::get<3>(bar);
22
23    return 0;
24 }
```

A tupla pode ser definida explicitamente ou com o comando `auto`, que infere o tipo de dados pelo retorna da função `make_tuple`, que cria tuplas.

A função `tie` serve para "desempacotar" os elementos da tupla enquanto a função `get` retorna um dos elementos da tupla.

28.7 Programação com iteradores

Vimos que os algoritmos da STL são todos baseados em iteradores, pois isso torna os códigos generalizáveis. Quando fazemos códigos genéricos em C++, é uma boa prática fazer estes códigos também baseados em iteradores. Estes códigos são muito mais genéricos pois podem ser utilizados com qualquer tipo de container ou arranjo. Eles podem retornar dados para qualquer tipo de container. Além disso, eles podem ser utilizados em qualquer subsequência dentro de um container ou arranjo. Eles também são usualmente mais eficientes.

São duas as regras para fazer um código baseado em containers que seja generalizável e eficiente: 1) nunca passar containers para uma função (passar iteradores) e 2) nunca retornar containers de uma função (utilizar iteradores). Em vez de passar um container para uma função, passe iteradores para o primeiro e último elementos. Em vez de retornar um container, passe um iterador onde os resultados possam ser guardados.

Veja uma função que recebe um container e calcula o produto de seus valores.

```
1  template <class Container>
2  double produto( const Container & c )
3  {
4    Container::iterator i = c.begin();
5    double p = 1;
6    while ( i != c.end() ) {
7        p *= *i++;
8    }
9    return p;
10 }
```

Apesar desta função utilizar templates e servir para qualquer container, ela não é uma função genérica o suficiente. Ela não funcionaria para arranjos e para subsequências em um container. A definição seguinte, que utiliza iteradores, é mais clara e geral:

```
1  template <class Iterador >
```

```
2 double product( Iterador inicio, Iterador fim )
3 {
4   double p = 1;
5   while ( inicio != fim ){
6     p *= *inicio++;
7   }
8   return p;
9 }
```

Esta função agora funciona bem com arranjos, containers e subsequências.

Vamos supor agora uma função que precise retornar vários resultados. Isso é comum em funções que copiam elementos de um container para outro, por exemplo. Neste caso, não é uma boa solução utilizar iteradores como parâmetros e retornar um container. Uma abordagem mais geral é receber um iterador onde os resultados serão guardados. Veja por exemplo como pode ser definida uma função que copia elementos de um container para outro.

```
1 template<class Entrada, class Saida>
2 Saida copia(Entrada inicio, Entrada fim, Saida resultado) {
3   while (inicio!=fim) {
4     *resultado = *inicio;
5     ++resultado; ++inicio;
6   }
7   return resultado;
8 }
```

A função define dois templates: para iteradores de entrada e saída. São necessário dois templates para garantir que o tipo de iterador onde possam ser guardados os resultados possa ser diferente do tipo de iterador com os dados de entrada. A função guarda os elementos da sequência de entrada na saída, representada por resultado. A função ainda retorna um iterador que aponta para uma posição após a posição onde foi copiado o último elemento.

Supondo dois containers `c1` e `c2`, os elementos de `c1` poderiam ser copiados para `c2` com a seguinte chamada:

```
1 auto iter = copia(c1.begin(), c1.end(), c2.begin());
```

Um problema possível é não haver espaço para os elementos de `c1` em `c2`. Uma maneira simples de resolver este problema é utilizando o recurso `back_inserter`.

```
1 auto iter = copia(c1.begin(), c1.end(), back_inserter(c2));
```

O `back_inserter` retorna um iterador especial para um container, chamado iterador de inserção. Este iterador sempre aloca espaço ao fim do container quando necessário pela função.

Com a utilização de iteradores com templates, fizemos nossas funções muito mais úteis e generalizáveis do que antes. A Tabela 28.2 apresenta algumas funções úteis para programação com iteradores.

28.8 Operador de cópia e movimento

Sempre que vamos trocar valores entre duas variáveis, precisamos de uma variável temporária, como no exemplo seguinte, onde trocamos os valores das variáveis `x` e `y`.

```
1 // Criando as variáveis
2 int x = 4;
3 int y = 3;
```

Função	Descrição
`advance`	Avança o iterador um certo número de posições
`distance`	Distância entre 2 iteradores
`prev`	Retorna iterador para elemento anterior
`next`	Retorna iterador para o próximo elemento
`back_inserter`	Constrói iterador de inserção no fim da sequência
`front_inserter`	Constrói iterador de inserção no início da sequência
`inserter`	Constrói iterador de inserção em um posição determinada
`make_move_iterator`	Constrói iterador para movimentar elementos

Tabela 28.2: Funções importantes para programação com iteradores.

```
4 // Trocando os valores
5 int temp;
6 temp = x;
7 x = y;
8 y = temp;
```

Isto ocorre pois sempre que atribuímos um valor a uma variável, seu valor antigo é perdido. Com alguns objetos, porém, isto pode gerar um alto custo em termos de desempenho. Suponha estarmos fazendo uma troca de conteúdo entre dois objetos `vector<int>`.

```
1 // Trocando os valores
2 vector<int> temp;
3 temp = x;
4 x = y;
5 y = temp;
```

Neste exemplo, se há n elementos no container `x`, teremos que fazer uma cópia de todos os elementos de `x` em `temp`, o que teria custo $O(n)$, tanto em relação a tempo quanto a memória. Ao fim do processo, `temp` é descartada.

Neste caso, sabemos que tivemos um desperdício de recurso pois um `vector` não é um tipo fundamental de dados. Ele depende de ponteiro que, internamente, aponta para um arranjo alocado na memória com os elementos. Poderiamos "mover" todo o conteúdo de `x` para `temp` apenas fazendo com que o ponteiro de `temp` aponte para o arranjo que é apontado por `x`. Isso faz com que `temp` represente a sequência `x` em tempo $O(1)$. Mover os elementos para `temp` em vez de fazer uma cópia não gera também problema algum à lógica do programa, já que `temp` é apenas uma variável temporária.

A semântica de movimento permite que não precisemos fazer cópias desnecessárias para trabalhar com objetos temporários.

```
1 #include <utility>
2 // ...
3 // Trocando os valores
4 vector<int> temp = move(x);
5 x = move(y);
6 y = move(temp);
```

Um construtor de movimento pode ser criado de maneira similar a como é criado um construtor de cópia para um objeto. Supondo um objeto `A`, este seria o protótipo de seu construtor de movimento.

```
1 // Construtor de cópia
2 A::A(const A& outro);
3 // Construtor de movimento
4 A::A(A&& outro);
```

Note que para containers, existe também a função `swap`, que troca os elementos entre os containers.

```
1 // Trocando os valores
2 x.swap(y);
```

28.9 Threads

Atualmente, são comuns programas que executam várias tarefas ao mesmo tempo. Para isto, precisamos do recurso de *threads*. Uma *thread* de execução define instruções que podem ser executadas ao mesmo tempo pelo computador. Uma *thread* pode ser criada em C++ com o objeto do tipo `thread`.

```
1  // biblioteca de threads
2  #include <thread>
3  // ...
4  void funcao(int x)
5  {
6      cout << x * x << endl;
7  }
8  // ...
9  int main()
10 {
11     // começa a executar uma função
12     thread t1(funcao,4);
13     // comaça a executar outra função
14     thread t2(funcao,10);
15
16     // sincroniza as threads
17     // pausa até que primeira thread termine
18     t1.join();
19     // pausa até que segunda thread termine
20     t2.join();
21
22     return 0;
23 }
```

No exemplo, temos uma `funcao` que imprime um valor elevado ao quadrado. Nas linhas 11 a 14, criamos as threads `t1` e `t2`, que executarão esta função com diferentes parâmetros (`4` e `10`). Cada thread executa a função paralelamente. Nas linhas 16 a 20, pedimos ao programa para que espere que as funções sejam executadas nas threads.

Para problemas maiores, é possível criar arranjos ou containers com threads de maneira análoga. Porém, é preciso prestar atenção pois o tempo de criação de uma thread pode não compensar o ganho com a execução de problemas muito pequenos em paralelo.

28.10 Medindo tempo

O cabeçalho `chrono` é utilizado para trabalhar com três tipos de conceitos: durações, pontos no tempo e relógios. É comum utilizar durações para medir quanto tempo foi necessário para a execução de um programa.

```
1  // ...
2  #include <chrono>
3  // ...
4  using namespace std::chrono;
5  steady_clock::time_point t1 = steady_clock::now();
6  executa_funcao_qualquer();
7  steady_clock::time_point t2 = steady_clock::now();
8  duration<double> d = duration_cast<duration<double>>(t2-t1);
9  std::cout << d.count() << "␣segundos" << std::endl;
10 // ...
```

Na linha 2 deste exemplo, incluímos a biblioteca `chrono`, necessária para se medir tempo. Na linha 4, indicamos que vamos utilizar recursos do espaço de nomes `chrono`, que por sua vez está dentro do espaço de nomes `std`.

Na biblioteca `chrono`, temos 3 tipos de relógios, como os apresentados na Tabela 28.3. Todos os relógios dão acesso a pontos no tempo. Na linha 5 do código, utilizamos o `steady_clock` para calcular intervalos. Os recursos do relógio `steady_clock` estão no espaço de nomes `steady_clock`, que está dentro do espaço de nomes `chrono`.

Relógio	Descrição
`steady_clock`	Utilizado para calcular intervalos
`system_clock`	Acessa o relógio geral do sistema
`high_resolution_clock`	Relógio de maior precisão possível

Tabela 28.3: Relógios da biblioteca `chrono`.

Criamos nesta linha 4 uma variável `t1` do tipo `time_point`. O tipo de dado `time_point` está definido no espaço de nomes do relógio e é utilizado para guardar pontos no tempo. A função `now()`, também pertencente ao espaço de nomes do relógio, retorna um ponto no tempo representando o momento atual.

Na linha 6, executamos uma função qualquer, e na linha 7 fazemos um procedimento similar ao da linha 5 para guardar o ponto no tempo `t2` após a execução da função.

Na linha 8, calculamos a duração entre os dois pontos no tempo. Dados do tipo `duration` guardam a duração entre dois pontos no tempo. Estes dados guardam a diferença entre dois pontos no tempo e a proporção entre estes pontos no tempo e uma unidade de medida de tempo, como segundos, minutos ou horas.

O `duration<double>` indica que dados do tipo `double` são utilizados internamente para guardar a duração. A classe `duration` pode receber um segundo parâmetro no template indicando a proporção entre pontos no tempo e unidades de tempo. Por exemplo, `duration<double, seconds>` faria com que os valores da duração fossem retornados em segundos. Outras opções estão na Tabela 28.4. Por padrão, a duração será retornada em segundos.

Ao lado direito da expressão da linha 8, a função `duration_cast` transforma um tipo de duração em outro. A operação `t2 - t1` retorna uma duração que é representada por um tipo de dados correspondente ao relógio. A função `duration_cast` utiliza o template

Tipo	Descrição
`hours`	Horas
`minutes`	Minutos
`seconds`	Segundos
`milliseconds`	Milisegundos
`microseconds`	Microsegundos
`nanoseconds`	Nanosegundos

Tabela 28.4: Tipos de proporção entre pontos no tempo e unidades de medida de tempo.

`duration<double>` para transformar a duração recebida como parâmetro para uma duração do tipo `duration<double>`.

Por fim, na linha 9, a função `d.count()` retorna a duração guardada em `d` expressa em segundos, como definida no segundo parâmetro de seu template.

28.11 Números aleatórios

Para geração de números aleatório, utilizamos a biblioteca `random`. A biblioteca contém geradores de números aleatórios e distribuições.

```
1 // ...
2 #include <random>
3 // ...
4 default_random_engine gerador;
5 uniform_real_distribution<double> distribuicao(0.0,1.0);
6 cout << distribuicao(gerador) << endl;
7 // ...
```

Na linha 4 do exemplo de código, criamos um gerador de números aleatórios do tipo `default_random_engine`. Este tipo de dado representa o gerador padrão de números aleatório.

Na linha 5 criamos uma distribuição de números reais entre `0.0` e `1.0`. As distribuições utilizam os geradores para gerar números naquela distribuição. Para a geração de números reais com distribuição uniforme utilizamos a distribuição `uniform_real_distribution`. O `double` no template indica que os números gerados serão representados com dados do tipo `double`.

Na linha 6, a distribuição criada utiliza o gerador para imprimir um número aleatório entre `0.0` e `1.0`.

Na linha 5, vários tipos de distribuição poderiam ser utilizados. A Tabela 28.5 apresenta as distribuições disponíveis.

Distribuição	Descrição
Uniforme	
`uniform_int_distribution`	Distribuição inteira uniforme
`uniform_real_distribution`	Distribuição real uniforme
Tentativas de Bernoulli	
`bernoulli_distribution`	Distribuição de Bernoulli
`binomial_distribution`	Distribuição binomial
`geometric_distribution`	Distribuição geométrica
`negative_binomial_distribution`	Distribuição negativa binomial
Baseada em taxa de acerto	
`poisson_distribution`	Distribuição de Poisson
`exponential_distribution`	Distribuição exponencial
`gamma_distribution`	Distribuição gamma
`weibull_distribution`	Distribuição de Weibull
`extreme_value_distribution`	Distribuição de valores extremos
Normal	
`normal_distribution`	Distribuição normal
`lognormal_distribution`	Distribuição lognormal
`chi_squared_distribution`	Distribuição chi-quadrado
`cauchy_distribution`	Distribuição de Cauchy
`fisher_f_distribution`	Distribuição de Fisher
`student_t_distribution`	Distribuição t de Student
Por partes	
`discrete_distribution`	Distribuição discreta
`piecewise_constant_distribution`	Distribuição contante
`piecewise_linear_distribution`	Distribuição linear

Tabela 28.5: Distribuições disponíveis para números aleatórios.

29. Algoritmos Eficientes

Nos capítulos anteriores deste livro, apresentamos versões didáticas de algoritmos importantes para busca e ordenação. As versões apresentadas destes algoritmos, naturalmente, não são as versões mais eficientes por ser tratarem de versões escolhidas principalmente por serem mais didáticas. Este capítulo apresenta uma lista de algoritmos importantes em C++ com suas implementações eficientes. Estes são implementações que podem ser utilizadas em aplicações práticas dos métodos descritos neste livro. São também os métodos utilizados para geração dos gráficos de comparação dos algoritmos ao longo deste livro.

29.1 Busca em arranjo

O STL oferece implementações para os algoritmos de busca em arranjo. Em aplicações práticas, estes são utilizados por programadores em C++.

29.1.1 Busca sequencial

Uma busca sequencial pode ser feita com a função `find`.

```
1 // a função recebe 2 iteradores (ou endereços) e o elemento procurado
2 p = find(v.begin(), v.end(), 30);
3 // se p não aponta para v.end() o elemento foi encontrado
4 if (p != v.end())
5     std::cout << "Elemento␣encontrado:␣" << *p << endl;
6 else
7     std::cout << "Elemento␣não␣encontrado" << endl;
```

29.1.2 Busca binária

Uma busca binária pode ser feita com as funções `binary_search`, `upper_bound` e `lower_bound`.

```
1 // a função recebe 2 iteradores (ou endereços) e o elemento procurado
2 resultado = binary_search(v.begin(), v.end(), 30);
```

```
3 // resultado diz se o elemento estava na sequência
4 if (resultado)
5     std::cout << "Elemento␣encontrado" << endl;
6 else
7     std::cout << "Elemento␣não␣encontrado" << endl;
```

A função `binary_search` não retorna iteradores. Para iteradores, precisamos das funções `upper_bound` e `lower_bound`.

```
1 // a função recebe 2 iteradores (ou endereços) e o elemento procurado
2 p1 = lower_bound (v.begin(), v.end(), 20);
3 p2 = upper_bound (v.begin(), v.end(), 20);
4 cout << "primeiro␣elemento␣20␣na␣posição␣" << (p1 - v.begin()) << endl;
5 cout << "último␣elemento␣20␣após␣posição␣" << (p2 - v.begin()) << endl;
```

29.2 Ordenação

A ordenação de elementos pode ser feita com a função `sort`. No caso geral, é sempre mais conveniente utilizar a função `sort` do STL. Esta é usualmente uma implementação do algoritmo introsort, que é eficiente e tem custo $O(n \log n)$ para todos os casos.

```
1 // a função ordena a sequência entre 2 iteradores (ou endereços)
2 sort(v.begin(), v.end());
3 for (int i = 0; i < v.size(); ++i){
4     std::cout << v[i] << '␣';
5 }
6 cout << endl;
```

Note que a função `sort` aceita o envio de um predicado para a função, caso seja necessário utilizar um critério de ordenação diferente do operador <.

```
1 // a função ordena a sequência entre 2 iteradores (ou endereços)
2 sort(v.begin(), v.end(), [](int a, int b){return b < a;});
3 for (int i = 0; i < v.size(); ++i){
4     std::cout << v[i] << '␣';
5 }
6 cout << endl;
```

Nas próximas seções temos implementações de outros algoritmos importantes de ordenação, apresentados neste livro. Todas as funções tem a mesma sintaxe da função `sort`, recebendo apenas dois iteradores genéricos.

29.2.1 Ordenação por seleção

A ordenação por seleção pode ser implementada eficientemente da seguinte maneira.

```
1 template <typename Iterador, typename Predicado>
2 void selection_sort(Iterador inicio, Iterador fim, Predicado comp) {
3   for (auto i = inicio; i != fim; ++i) {
4     iter_swap(i,min_element(i, fim, comp));
5   }
6 }
```

A função depende de um template que define o iterador e outro template que define o predicado. O template do iterador garante que a função funcionará para qualquer tipo de iterador

ou mesmo endereço na memória. A função recebe um iterador que marca o início da sequência e outro que marca uma posição após o fim da sequência. O template do predicado é para uma função genérica que será usada para indicar quando um elemento é maior que outro. Caso seja necessário utilizar a função com o predicado padrãom que é o operador <, a seguinte função pode ser chamada:

```
1 template <typename Iterador>
2 void selection_sort(Iterador inicio, Iterador fim) {
3   typedef typename iterator_traits<Iterador>::value_type Tipo;
4   insertion_sort(inicio, fim, less<Tipo>());
5 }
```

Na linha 3 descobrimos o tipo do iterador e o chamamos de `Tipo`. Na linha 4, chamamos a versão original da função com o operador < para este tipo de dados.

O `for` do algoritmo percorre todos os elementos com o iterador `i`. Para cada elemento fazemos uma troca com a função `iter_swap`. A troca é feita entre o elemento `i` e o menor elemento entre `i` e `fim`. A função `min_element` do STL é utilizada para encontrar este menor elemento.

29.2.2 Ordenação por inserção

A ordenação por inserção pode ser implementada eficientemente da seguinte maneira.

```
1  template <typename Iterador, typename Predicado>
2  void insertion_sort(Iterador inicio, Iterador fim, Predicado comp) {
3    for (auto i = inicio; i != fim; ++i) {
4      rotate(upper_bound(inicio, i, *i, comp), i, i + 1);
5    }
6  }
7
8  template <typename Iterador>
9  void insertion_sort(Iterador inicio, Iterador fim) {
10   typedef typename iterator_traits<Iterador>::value_type Tipo;
11   insertion_sort(inicio, fim, less<Tipo>());
12 }
```

O templates e sobrecargas da função são novamente definidos para considerar um predicado qualquer na função.

Na primeira definição, um iterador `i` percorre todos os elementos. Para cada elemento, achamos sua posição correta no subarranjo ordenado entre `inicio` e `i`. Para isto, utilizamos uma busca binária com a função `upper_bound`. Isto faz com que a busca pela posição correta ocorre muito mais rapidamente. O elemento `i` é colocado em sua posição correta com a função `rotate`.

29.2.3 Merge sort

O merge sort pode ser implementado eficientemente da seguinte maneira.

```
1 template<typename Iterador, typename Predicado>
2 void mergesort(Iterador inicio, Iterador fim, Predicado comp) {
3   if (fim - inicio > 1)
4   {
5     Iterador meio = inicio + (fim - inicio) / 2;
6     mergesort(inicio, meio, comp);
7     mergesort(meio, fim, comp);
```

```
8       inplace_merge(inicio, meio, fim, comp);
9     }
10  }
11
12  template<typename Iterador>
13  void mergesort(Iterador inicio, Iterador fim) {
14    typedef typename iterator_traits<Iterador>::value_type Tipo;
15    mergesort(inicio, fim, less<Tipo>());
16  }
```

Nesta função, sempre que a sequência for maior que 1, temos um arranjo (ou sequência qualquer) potencialmente desordenado e entramos em um bloco de comandos que ordena recursivamente 2 subarranjo. Ordenamos o subarranjo que vai do início ao meio do arranjo e depois ordenamos o subarranjo que vai do meio ao fim do arranjo. A função `inplace_merge` intercala os dois subarranjos e um arranjo completo ordenado.

29.2.4 Quicksort

Para uma implementação eficiente do quicksort, precisamos de uma função auxiliar. Primeiramente, a seguinte função retorna o item mediano entre 3 elementos. Esta é uma função auxiliar importante para se encontrar um pivô eficiente.

```
1   template<typename T>
2   T median(T t1, T t2, T t3)
3   {
4     if (t1 < t2)
5     {
6       if (t2 < t3)
7         return t2;
8       else if (t1 < t3)
9         return t3;
10      else
11        return t1;
12    }
13    else
14    {
15      if (t1 < t3)
16        return t1;
17      else if (t2 < t3)
18        return t3;
19      else
20        return t2;
21    }
22  }
```

A função retorna o elemento mediano entre os elementos `t1`, `t2` e `t3`. Podemos então definir a função de ordenação como a seguir:

```
1   template<typename Iterador, typename Predicado>
2   void quicksort(Iterador inicio, Iterador fim, Predicado comp) {
3     if (fim - inicio > 1) {
4       typedef typename iterator_traits<Iterador>::value_type Tipo;
5       Iterador meio = inicio + (fim - inicio)/2;
```

```
6       Tipo pivo = mediana(*inicio, *meio, *(fim-1));
7       Iterador divisao = partition(inicio, fim,
8           [&pivo,&comp](Tipo x){return comp(x,pivo);});
9       quicksort(inicio, divisao-1, comp);
10      quicksort(divisao, fim, comp);
11    }
12  }
13
14  template<typename Iterador>
15   void quicksort(Iterador inicio, Iterador fim)
16  {
17    typedef typename iterator_traits<Iterador>::value_type Tipo;
18    quicksort(inicio, fim, less<Tipo>());
19  }
```

Função `quicksort` é executada recursivamente sempre que a sequência tiver mais de 1 elemento. O iterador `meio` aponta para o elemento do meio da sequência. A mediana entre o primeiro elemento, o último elemento e o elemento do meio é utilizada como pivô no processo de partição.

A partição é feita pela função `partition`, que recebe como terceiro parâmetro o critério para elementos que deve estar à esquerda da partição. No caso, este critério é ser menor que o pivô (utilizando o critério de comparação definido). Para isto, `pivo` e `comp` são enviados para a função lambda.

A partição divide a sequência em duas subsequências, divididas pelo iterador `divisao`. Assim, precisamos apenas chamar a função quicksort recursivamente para cada subsequência.

Índice

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V