C++
e Orientação a Objetos

PET Computação
Fábio Beltrão, Felipe Chies, Lucas Zawacki, Marcos Cavinato e Matheus Proença

2ª Edição, 18 de Agosto de 2009

1ª Edição por
Arthur Ribacki, Gabriel Portal, Leonardo Chatain e Rosália Galiazzi

Sumário

1 Introdução ao C++ **1**
- 1.1 História do C++ . . . 1
- 1.2 Primeiros Programas . . . 2
- 1.3 Namespaces . . . 4
- 1.4 Variáveis Locais e Globais . . . 4
- 1.5 Tipos de Dados . . . 5
- 1.6 Modificadores . . . 5
- 1.7 Estruturas de Controle . . . 6
- 1.8 Funções . . . 6
  - 1.8.1 Funções inline . . . 8
  - 1.8.2 Parâmetros Default . . . 8

2 Introdução a Orientação a Objetos **13**
- 2.1 Uma classe Carro . . . 13
- 2.2 Classes . . . 14
  - 2.2.1 Encapsulamento . . . 14
  - 2.2.2 Membros Públicos e Privados . . . 16
  - 2.2.3 Construtores e Destrutores . . . 17
  - 2.2.4 Divisão de Arquivos . . . 18
- 2.3 Composição . . . 21
- 2.4 Objetos Constantes . . . 21
- 2.5 Funções Membro Constantes . . . 22
- 2.6 Ponteiro this . . . 23

3 Sobrecarga **25**
- 3.1 Quando usar? . . . 25
- 3.2 Modificador friend . . . 25
- 3.3 Fundamentos de Sobrecarga de Operadores . . . 26
- 3.4 Funções Friend . . . 28
  - 3.4.1 Sintaxe . . . 29
- 3.5 Sobrecarga de Operadores Unários . . . 29
- 3.6 Sobrecarga de Operadores Binários . . . 30
- 3.7 Sobrecarga dos Operadores de Streams . . . 30

3.7.1 Exemplo dos operadores de streams (« e ») 31
3.8 Conversão entre Tipos 32
3.9 Sobrecarregando ++ e - - 32

4 Herança **33**
4.1 Classe base e Classes Derivadas: 33
4.2 Membros Protected 35
4.2.1 Sintaxe 35
4.2.2 Exemplos de Herança 35
4.3 Coerção de ponteiros entre classes bases e classes derivadas . 37
4.4 Sobrescrevendo membros da classe base em uma classe derivada 39
4.4.1 Output 41
4.5 Tipos de Herança 41
4.6 Herança Múltipla 42
4.6.1 Output 42

5 Funções Virtuais e Polimorfismo **45**
5.1 Funções Virtuais 45
5.1.1 Teste de Aprendizado 47
5.2 Ponteiros Polimorfos como Parâmetros 47
5.2.1 Teste de Aprendizado 47
5.3 Classes Base Abstratas e Classes Concretas 49
5.3.1 Teste de Aprendizado 49

6 Templates e Exceções **51**
6.1 Templates para Funções 51
6.1.1 Sintaxe 51
6.1.2 Exemplo 52
6.2 Template de Classe 52
6.2.1 Sintaxe 52
6.2.2 Exemplo 53
6.3 Introdução a Exceções 53
6.4 Try, Throw e Catch 53
6.4.1 Exemplo 55
6.4.2 Exercício 57

7 Processamento de Arquivos **59**
7.0.3 Arquivos em C++ 59
7.0.4 Arquivo Seqüencial 60
7.0.5 Modos de Abertura 60
7.1 Arquivo Aleatório 62

8 Standard Template Library 63
8.1 Conhecendo a STL 63
8.2 Contêiners 64
8.3 Funções Membro Comuns 64
8.4 Funções Membro Específicas 65
8.4.1 Arquivos de Cabeçalho 65
8.5 Vector 66
8.5.1 Teste de Aprendizado 66
8.6 Como lidar com capacidade ilimitada, ou como o Vector não é mágico 67
8.6.1 Observação 69
8.7 List 69
8.8 Outros Containers 70
8.9 Iteradores 71
8.10 Algoritmos 74
8.10.1 Usando o Algoritmo Sort para qualquer Ordenação . . 75
8.11 Usando a STL com Classes Próprias 75
8.11.1 Teste de Aprendizado 78
8.12 Últimas Considerações 78

Capítulo 1

Introdução ao C++

Antes de começarmos o estudo da linguagem, é necessário que se entenda por que é tão importante aprender C++. Existe, atualmente, uma demanda muito grande por software, que deve ser construído rápida, correta e economicamente. A orientação a objetos surge com o objetivo de ajudar a suprir essa demanda. Fortemente baseada no princípio de reutilização de componentes e estreitamente ligada à engenharia de software, permite uma redução do tempo de implementação bastante significativa. Além disso, através de uma organização modular e independente, melhora a manutenabilidade e aumenta a confiabilidade dos programas.

1.1 História do C++

A linguagem de programação C foi desenvolvida pela AT&T com o propósito de escrever um sistema operativo para a série de computadores PDP-11 que acabaria por ser o sistema operativo UNIX. O C foi desenvolvido com o principal objetivo de ser eficiente. Bjarne Stroustrup, também da AT&T, desenvolveu o C++ para acrescentar construções orientadas a objetos na linguagem C. C++ pode ser visto como uma linguagem procedimental com alguns construtores adicionais.

Começando pelo C, alguns construtores para programação orientada a objetos e para melhorar a sintaxe procedimental foram acrescentados. Como já dito anteriormente, um programa bem escrito em C++ irá refletir elementos tanto do estilo de programação orientada a objetos como programação procedimental clássica. E isto porque o C++ é uma linguagem extensível, ou seja, podemos definir novos tipos de tal maneira que eles ajam do mesmo modo que tipos pré-definidos que já fazem parte da linguagem padrão.

1.2 Primeiros Programas

Iniciaremos o estudo com um pequeno exemplo, que demonstrará algumas diferenças sintáticas de C++ em relação à linguagem C. O programa apenas escreve "Hello World!" no console, como todo bom primeiro programa em uma linguagem de programação. A seguir, cada parte do programa será vista em detalhe.

Código 1.1: Hello World

```
// Primeiro programa C++

#include <iostream>

int main()
{
  std::cout << "Hello World" << std::endl; //Escreve no terminal
  return 0;
}
```

Pode se observar que a primeira linha é precedida por //. Essa é a sintaxe do comentário de uma linha do C++. O compilador deve ignorar tudo aquilo que estiver à direita de duas barras invertidas, mesmo que não no começo, como se pode ver algumas linhas depois. Vale observar que os comentários do C (iniciados por /* e terminados por */) t ambém são utilizados em C++, principalmente para fazer comentários de várias linhas. A diretiva *#include* é conhecida para os programadores de C, bastando observar que a biblioteca não é a mesma e que, para bibliotecas padrão, não é incluído o '.h' do arquivo header.

As funções *std::cout* e *std::endl*, substituem, respectivamente, a função printf e o caractere '\n' do C. Só para relembrar, isso significa que *std::cout* escreverá na tela, enquanto *std::endl* adicionará uma nova linha. Uma das grandes vantagens desse método é a segurança de tipos, ou seja, o fato de que a função "sabe" qual o tipo do dado que está sendo passado a ela, e responde de acordo. Por exemplo, o código 1.2 imprime a frase "Meu nome é Lucas, tenho 18 anos.".

A entrada em C++ é tão simples quanto a saída, trocando apenas o nome da variável para *std::cin* e a direção dos operadores, para ». O tipo da variável e o significado dos operadores « e » serão discutidos em capítulos posteriores, por enquanto basta aprender seu funcionamento. O programa 1.3, que lê duas variáveis da tela e imprime sua soma, é trivial e tem por objetivo fixar a sintaxe, de modo que não será detalhado.

Código 1.2: Exemplo

```
// Impressao com cout

#include <iostream>

int main()
{
        char nome[] = "Lucas";
        int idade = 18;

        std::cout << "Meu nome eh " << nome << ", tenho " <<
            idade << " anos." << std::endl;

}
```

Código 1.3: Entrada

```
//Mesmo que o exemplo anterior, mas usando namespaces

#include <iostream>

using namespace std;

//Para nao abrir todo namespace poderiamos usar

/*
using std::cin;
using std::cout;
using std::endl;
*/

int main()
{
        char nome[30];
        int idade;

        cin >> nome;
        cin >> idade;

        cout << "Meu nome eh " << nome << ", tenho " << idade <<
             " anos." << endl;
}
```

1.3 Namespaces

Nos exemplos anteriores, podemos perceber a prefixação de três funções (*cin*, *cout* e *endl*) por *std::*. Isto não é uma coincidência, ou uma excentricidade do autor da linguagem. O fato, é que as três funções pertencem a um mesmo namespace (ambiente de nomes), no caso o ambiente de nomes padrão do C++, *std*. Caso desejemos, podemos omitir o *std* na frente de cada função, contanto que o compilador seja previamente avisado. Isso é feito declarando-se **using namespace** std; antes da função main do programa.

1.4 Variáveis Locais e Globais

Em C++, todo bloco de comandos deve estar entre chaves ("{ }"). Segundo as regras de hierarquia de escopo, qualquer variável declarada dentro de um bloco é visível apenas dentro do mesmo (inclua-se qualquer sub-bloco), sendo destruída assim que o bloco é finalizado. Essas são chamadas variáveis locais, em contraponto às variáveis declaradas fora de qualquer bloco, chamadas variáveis globais.

Foi introduzido na linguagem um novo operador: o operador de escopo, representado por *::*. Este operador é utilizado para se acessar uma variável de escopo superior cujo nome seja o mesmo de uma variável local (neste caso, a variável de escopo superior estaria inacessível). Exemplificando: caso se tenha uma variável com o nome *x* num escopo global e declare-se uma variável *x* num escopo local, teoricamente a variável *x* global ficaria inacessível no respectivo bloco. Contudo, referenciando-se *::x* é possível acessar o *x* global.

Código 1.4: Escopos

```
#include <iostream>

using namespace std;

// demonstrando os diferentes escopos
int i = 12;

int main()
{
        int i;
        //uso do operador ::
        for (i=0 ; i < ::i; i++)
        {
                cout << i << endl;
        }

}
```

1.5 Tipos de Dados

Em C++ as variáveis podem ser de cinco tipos primários distintos:

int: valores inteiros, variando entre 16 e 32 bits dependendo do compilador;

char: caracteres, variando entre -128 e 127, de acordo com a tabela ASCII;

float: valores em ponto flutuante (reais);

double: valores em ponto flutuante, com precisão dupla;

bool: valor false (0) ou true (1).

A declaração dos tipos é feita de modo bastante similar a C, diferindo apenas para ponteiros, onde é usado o comando *new*, em vez de *malloc*, e *delete*, em vez de *free*. Podemos ver a sintaxe desses comandos no código 1.5.

Código 1.5: Comando new e delete

```
#include <iostream>

using namespace std;
int main()
{

        // aloca o vetor
        int *vetor;

        //descomente para ter uma surpresa!
        //vetor[0] = 3;

        vetor = new int[10];
        vetor[0] = 3; //ok

        cout << vetor[0] << endl;

        // deleta o vetor
        delete(vetor);

}
```

1.6 Modificadores

Modificadores são utilizados para especificar determinados tipos de tratamento às variáveis. Os dois tipos mais comuns são const e static. Const declara uma variável (ou função, como veremos posteriormente) como sendo constante,

de modo que o compilador previne erros do programador (caso seja feita uma tentativa errônea de alteração do valor). Além disso, o compilador faz algumas otimizações quando sabe que o valor de uma variável não será alterado. A palavra-chave static é usada principalmente dentro de funções (também é usada em classes, mas isso será discutido no momento adequado). Ela serve pra dizer que o valor da variável não deve ser descartado no final na execução, como seria normal a uma variável local, mas guardado para uso posterior. O código 1.6 demonstra isso.

Código 1.6: Modificadores

```
// Demonstrando o modificador static

#include <iostream>

void incrementa()
{
  static int c = 0;
  c++;
  std::cout << c << std::endl;

}

int main()
{
  incrementa();
  incrementa();

  return 0;
}
```

1.7 Estruturas de Controle

As estruturas em C++ seguem o padrão da linguagem C, devendo-se atentar apenas para a declaração de variáveis locais dentro do bloco, que não serão enxergadas por outras partes do programa. A sintaxe apresentada nos códigos 1.7 a 1.10 já deve ser conhecida e é fornecida apenas para fins de revisão.

Algumas observações, porém, são válidas, no que diferem do C. Por exemplo, a declaração de uma variável dentro da inicialização do for é bastante usual entre programadores de C++.

1.8 Funções

Funções têm um papel bastante importante em C++, pois possibilitam a reutilização de código e segmentação dos programas, o que os torna muito mais

Código 1.7: do .. while

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

int main()
{
  int a = 0, b = 0;
  cin >> a >> b;

  cout << "Testando o while" << endl;

  // do ... while
  while(a < b)
  {
        cout << a << endl;
        a += 1;
  }

  cout << "Agora o do/while" << endl;
  do
  {
        cout << a << endl;
        a -= 1;
  }while(a > b);

}
```

Código 1.8: for

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

int main()
{
  int a = 0;
  cin >> a;
  cout << "Testando o for" << endl;

  for (int i=0; i < a; i++)
        cout << i << endl;

}
```

Código 1.9: if .. else

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

int main()
{
  int a = 0, b = 0;
  cin >> a >> b;
  cout << "Testando if" << endl;

  if (a > b)
  {
        cout << a;
  }
  else
  {
        cout << b;
  }

  cout << endl;

}
```

fáceis de serem compreendidos e depurados. É válido lembrar que existem dois modos de passar um parâmetro para uma função: por valor e por referência. Por valor, é passada apenas uma cópia da variável para a função de modo que a variável original não será alterada. Por referência, é passado um ponteiro para a variável, de modo que se o valor dela for alterado dentro da função, essa mudança será refletida fora também. A sintaxe usada e o efeito dos dois tipos de passagem de parâmetro são demonstrados no código 1.11.

1.8.1 Funções inline

Chamadas de funções envolvem um overhead durante a execução. Para reduzir este overhead, a linguagem dá suporte a 'funções inline'. Dessa forma, utilizando o qualificador inline antes do tipo de retorno de uma função aconselhamos o compilador a gerar cópias do código da função para evitar chamar a função. Contudo, este artifício deve ser utilizado apenas com funções pequenas e freqüentemente usadas. Além disso, o uso de funções inline pode diminuir o tempo de execução, mas deve aumentar o tamanho do programa.

1.8.2 Parâmetros Default

Normalmente, é preciso passar todos os valores de parâmetros numa chamada de função. Contudo, é possível determinar valores default para parâmetros de uma função; assim, pode-se chamar a função com menos argumentos. Faz-se

Código 1.10: switch .. case

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

int main()
{
  int a = 0;
  cin >> a;
  cout << "Testando o switch" << endl;

  switch(a)
  {
        case 1:
         cout << "um";
         break;
        case 2:
         cout << "dois";
         break;
        case 3:
         cout << "tr s";
         break;

        default:
          cout << "Nenhum dos tres";

        }

        cout << endl;

}
```

Código 1.11: Funções

```
#include <iostream>

void incrementaNaoAltera(int x)
{
  x += 1;
  std::cout << "Dentro da funcao: " << x << std::endl;
}

void incrementaAltera(int *x)
{
  *x += 1;
  std::cout << "Dentro da funcao: " << *x << std::endl;
}

int main()
{
  int valor = 0;

  std::cout << valor << std::endl;
  incrementaNaoAltera(valor);
  std::cout << valor << std::endl;

  incrementaAltera(&valor);

  std::cout << valor << std::endl;
}
```

Código 1.12: Funções Inline

```
#include <iostream>
#define CUBO(x) x*x*x //MACRO

using namespace std;

inline int cubo (int x) { return x*x*x; } //INLINE

int main()
{
        cout << cubo(5) << endl;// 125
        cout << CUBO(5) << endl;// 125

        cout << cubo(2+3) << endl; // 125
        cout << CUBO(2+3) << endl; // 17
        return 0;
}
```

isso fazendo uma atribuição para cada parâmetro default dentro da declaração da função. Contudo, para o compilador não se confundir, sempre que um parâmetro é omitido todos parâmetros à esquerda devem ter um valor default também e deverão ser omitidos.

Código 1.13: Parâmetros Default

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

int cubo(int x = 3) { return x*x*x; }

int main()
{
  cout << cubo(5) << endl;
  cout << cubo() << endl;
}
```

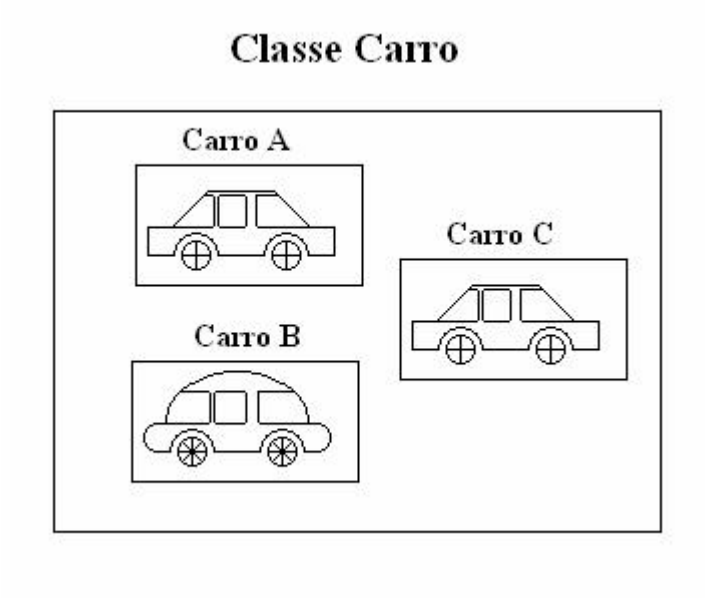
Capítulo 2

Introdução a Orientação a Objetos

Nessa seção, serão introduzidos os alicerces da programação orientada a objetos, um dos principais recursos exclusivos de C++ em relação a C. O total entendimento dos conceitos a seguir é imprescindível para a continuação do curso. Para entender a Orientação a Objetos, é preciso inicialmente que se entenda a que nos referimos quando falamos de objeto. O conceito de objeto na programação é bastante similar ao usual. Um objeto possui características e realiza determinadas ações.

2.1 Uma classe Carro

Um carro é um exemplo de objeto. Quando falamos de um objeto carro, devemos lembrar que um objeto é único. Um objeto carro é um carro específico. Existem, porém, objetos com características semelhantes, seguindo o exemplo anterior, dois carros. Ainda que não sejam iguais, possuem atributos em comum e realizam, a grosso modo, as mesmas funções. Descrever as características e ações de cada objeto carro separadamente é extremamente redundante. Existe, então, o conceito de classe, uma generalização de objetos de modo a possibilitar seu agrupamento. Criamos a classe Carro, e definimos os dois carros como objetos dessa classe. A figura abaixo facilita o entendimento.



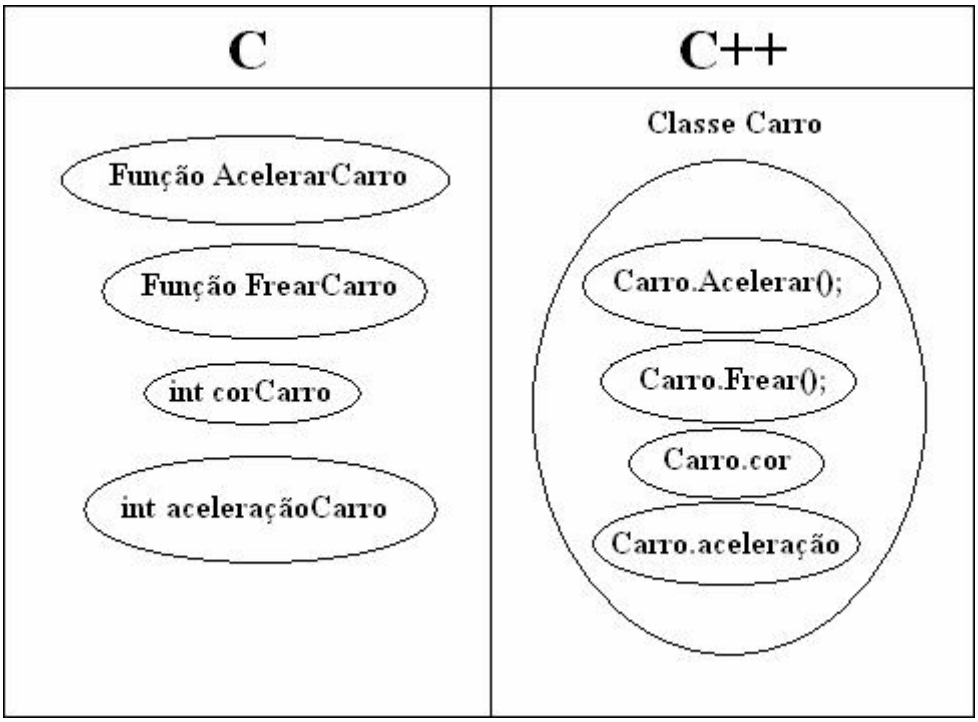
Simplificando, podemos definir uma classe como um tipo de dados, porém que possui funções em sua definição, além de atributos. Vendo dessa forma, entenderíamos um objeto como sendo uma variável desse tipo.

Uma observação importante é a de que dois objetos iguais não são o mesmo objeto, como enfatiza a figura. Esse erro conceitual, bastante freqüente no início do aprendizado, pode ser facilmente eliminado quando fazemos a analogia classe/objeto e tipo/variável. Duas variáveis inteiras de valor 5 obviamente não são a mesma variável, ainda que sejam do mesmo tipo e de mesmo valor.

2.2 Classes

2.2.1 Encapsulamento

O encapsulamento se constitui do englobamento de funções e dados dentro de classes. Esse é o conceito mais básico e abrangente da Orientação a Objeto, e é apenas uma formalização daquilo que entendemos como programação através desse paradigma. Abaixo, podemos ver uma comparação entre a programação baseada em procedimentos e a orientada a objetos. Como podemos ver, a classe Carro engloba todos os dados e funções que se referem ao tipo de dados Carro.



Os ganhos da Orientação a Objeto em relação a organização, ficam claros quando se imagina uma função main contendo dois objetos da classe carro em comparação a variáveis como corCarro1 e corCarro2, tipoCarro1 e tipoCarro2, aceleraçãoCarro1 e aceleraçãoCarro2.

Uma vez entendida a idéia, é necessário que se veja a sintaxe da implementação de classes no C++. Para isso, utilizaremos nosso exemplo da classe Carro, apresentado no código 2.1.

Código 2.1: Classe Carro

```
class Carro
{
  public:
  Carro();
  ~Carro();

  void acelerar();
  void frear();

  private:
  int cor;
  int aceleracao;
};
```

Como podemos ver, existem pequenos detalhes na implementação através da linguagem, mas basicamente se trata do que foi conceitualmente discutido.

Uma classe englobando atributos e funções a ela referentes. As pequenas peculiaridades da prática serão vistas a seguir.

2.2.2 Membros Públicos e Privados

Pode-se observar no exemplo que existem duas palavras chaves precedendo determinado grupo de dados ou funções, public e private (existe também um tipo de dados protected, que será tratado posteriormente). Dados públicos (public) são aqueles que podem ser acessados de qualquer lugar do programa. Dados privados (private), pelo contrário, são reconhecidos apenas por funções membro da classe.

Para acessar dados privados, é usual que se implementem funções set e get. Funções get retornam o valor do objeto, e funções set setam um novo valor para ele. A motivação para o uso de dados privados pode parecer obscura em um primeiro momento ("não seria mais fácil permitir o acesso?"), mas é bastante útil permitir a manipulação de dados apenas através de funções da classe. Pode ser necessária uma validação de dados no momento da atribuição, como mostra o código 2.2.

Código 2.2: Validação de entrada

```
#include "carro.h"

void Carro::setCor(int novaCor)
{
  if((novaCor >= 0)&&(novaCor <= 255))
  {
        cor = novaCor;
  }
  else
  {
         std::cout << "Cor Invalida!" << std::endl;
  }
}
```

No exemplo acima, são aceitas apenas 256 cores diferentes. Ainda que isso seja informado ao usuário, não se pode garantir que ele realmente siga essa recomendação. Torna-se então clara a utilidade dos membros privados (e das funções set, como são chamadas geralmente as funções de atribuição). Caso cor fosse um membro público, a verificação teria de ser feita pelo programador fora da classe, ou se permitiria que fossem atribuídos dados inválidos à variável.

A possibilidade de realizar a validação dos dados internamente, torna a classe auto-contida, tornando-a mais robusta e confiável, e contribuindo para a organização e reutilização do código.

2.2.3 Construtores e Destrutores

Além da divisão dos dados entre públicos e privados, outro detalhe estranho chama diretamente nossa atenção quando olhamos pela primeira vez a implementação de uma classe. Existem duas funções com o nome da classe, e sem qualquer tipo. No nosso exemplo, Carro() e Carro(). A primeira função, Carro(), é o construtor da classe.

Construtores são os responsáveis pela alocação de memória e inicialização de dados, sendo sempre chamados automaticamente na declaração um novo objeto. O uso de construtores elimina o problema de ter que manualmente inicializar os dados sempre que declarados, o que pode ser esquecido e levar a erros.

Código 2.3: Construtor

```
#include "carro.h"

Carro::Carro()
{
  cor = 0;
}
```

Além do construtor padrão, ou seja, sem qualquer parâmetro, podem existir construtores que recebam, por exemplo, um parâmetro cor. Deve ser permitido, porém, que o usuário declare objetos da classe Carro sem fornecer parâmetros, de modo que deve ser declarado um construtor padrão, ou, pelo menos, fornecidos valores padrão para os parâmetros. Uma explicação mais aprofundada sobre o uso de duas funções com mesmo nome (mesmo construtores), será vista ao decorrer do curso, quando forem tratadas as funções sobrecarregadas.

Código 2.4: Construtor com parâmetros

```
#include "carro.h"

Carro::Carro(int novaCor)
{
  cor = novaCor;
}
```

Para atribuir um valor padrão ao parâmetro novaCor, basta atribuir a ele o valor desejado já dentro da declaração.

A outra função, Carro(), é o destrutor da classe, a qual é chamada automaticamente sempre que um objeto deixa de existir (provavelmente porque o programa saiu de seu escopo). Destrutores podem também ser chamados pelo programador quando não for mais necessário, visto que liberam a memória alocada para o objeto. Na maioria dos exemplos dessa apostila, um destrutor

Código 2.5: Construtor com parâmetros default

```
#include "carro.h"

Carro::Carro(int novaCor = 255)
{
  cor = novaCor;
}
```

vazio será suficiente.

Código 2.6: Destrutor

```
#include "carro.h"

Carro::~Carro()
{

}
```

É válido observar que os destrutores não seguem a mesma regra que os construtores, não sendo possível que recebam parâmetros.

2.2.4 Divisão de Arquivos

Antes de começar a implementar classes em C++, é preciso que se observe um detalhe importante das boas práticas de programação. Quando se cria uma classe, não é raro pretender que ela seja utilizada por outras pessoas posteriormente. Não é necessário compartilhar com cada usuário como foi feita a implementação da classe. Para "esconder" o código, separamos a definição e a implementação, utilizando, respectivamente, arquivos header (.h) e source (.cpp).

Nos arquivos header, coloca-se idealmente apenas aquilo que define como uma classe deve ser utilizada, basicamente seus atributos e os cabeçalhos de suas funções, sem qualquer referência a como as funções são implementadas. Diz-se idealmente, pois nem sempre é recomendado, por questões de desempenho, que se faça isso. A disputa organização/desempenho, porém, não possui resposta correta, devendo cada um fazer seu próprio julgamento sobre o que é mais importante. As bases para tal discussão serão dadas mais tarde.

Um exemplo de definição de classe pode ser visto no código 2.2.4.

Observe que no início do arquivo existem um #ifndef e o nome da classe. Para explicar a necessidade dessa linha, será usado um exemplo: Suponhamos que existam duas classes de um mesmo programa, por exemplo, Imagem e Botão, e as duas classes possuam um objeto da classe Vetor2D para indicar sua posição na tela. As duas classes devem então, incluir o arquivo da classe

```
#ifndef __VETOR2D_H
#define __VETOR2D_H

class Vetor2D {
  public:
  Vetor2D();
  ~Vetor2D();

  int getX();
  int getY();

  void setX(int newX);
  void setY(int newY);

  private:
  int x, y;
}

#endif
```

Vetor2D. O problema é que uma classe não pode ser incluída mais de uma vez, já que, se isso acontece, uma variável é redeclarada, e o compilador acusa um erro de "Nome-davariável redefined". Para que isso não ocorra, criamos uma flag que testa se o arquivo já foi incluído.

Código 2.7: Diretiva ifndef

```
#ifndef __VETOR2D_H // se a variavel nao esta definida
#define __VETOR2D_H // define a variavel class Vetor2d
/*
  declara a classe
*/
#endif // termina o if
```

Dessa forma, o código de declaração só é executado uma vez (a próxima vez, o símbolo __VETOR2D_H já estará declarado). Este é um detalhe bastante importante, sendo impossível levar adiante projetos grandes sem a utilização dessas flags. Nos arquivos source, deve ser colocado aquilo que o usuário da classe não deve ver, ou seja, a implementação das funções membro. Não esqueça de incluir o arquivo header, para que o arquivo possa "enxergar" as definições. É fundamental adicionar Nome-da-classe:: antes dos nomes de função, para que o compilador saiba que se trata de uma função da respectiva classe. A implementação da classe acima segue em 2.8:

Código 2.8: Vetor 2D

```
#include "vetor2D.h"

Vetor2D::Vetor2D()
{
}

Vetor2D::~Vetor2D()
{
}

int Vetor2D::getX()
{
  return x;
}

void Vetor2D::setX(int newX)
{
  x = newX;
}

int Vetor2D::getY()
{
  return y;
}

void Vetor2D::setY(int newY)
{
  x = newY;
}
```

2.3 Composição

Por composição entende-se objetos de classes fazendo parte de outras classes. Este é um princípio básico da engenharia de software, onde módulos menores fazem parte de módulos maiores do projeto. E faz bastante sentido também, por exemplo, a classe Carro ter um componente que é o motor (um objeto da classe Motor) e um vetor de quatro rodas (quatro objetos da classe Roda). Para tomar esse tipo de decisão, costuma-se fazer a pergunta "tem um?". Então: "Um carro tem um motor?", "Um carro tem quatro rodas?". Para definir a composição, basta definir o objeto dentro dos membros da classe da mesma forma como se faz com variáveis primitivas.

2.4 Objetos Constantes

Às vezes, é interessante informarmos ao compilador que um objeto é constante. Isso porque o compilador consegue realizar algumas otimizações significativas quando usamos constantes. Além disso, caso seja feita uma tentativa de modificar uma constante, o compilador avisa e gera um erro sintático, muito mais facilmente corrigível que um erro de lógica.

Para tornar um objeto constante, basta adicionar antes dele a palavra-chave const como mostrado no código 2.9. Existem vários usos para essa possibilidade, sendo o mais simples deles a passagem de parâmetros constantes em funções, de modo que a função não possa alterálos em seu interior.

Código 2.9: Argumentos Constantes

```
#include "carro.h"

void Carro::setCor(const int novaCor)
{
  cor = novaCor;
}
```

Dessa forma, o compilador acusa possíveis inconsistências do código. Como qualquer programador sabe, encontrar um erro cuja linha é acusada pelo compilador é infinitamente mais simples do que debugar um código com erros de lógicas ou execução. Outro uso dos objetos constantes é quando desejamos passar o valor de um objeto grande para uma função. Nesse caso, passar apenas uma referência é uma opção inegavelmente mais adequada, para que o programa não precise fazer uma cópia do objeto. Devemos lembrar, entretanto, que quando alteramos um objeto que foi passado por referência a uma função, essa alteração será aplicada ao objeto também em seu escopo original. Desse modo, é bastante aconselhável que se avise ao compilador que não permita a alteração do objeto. Além disso, a utilização da palavra const contribui para a clareza do código.

2.5 Funções Membro Constantes

Funções membro constantes são funções que não alteram nenhum atributo do objeto em seu interior. Uma função get, por exemplo, deve apenas retornar um atributo, de modo que não lhe deve ser possibilitado alterar o objeto. Impedir em tempo de compilação que isso seja feito é considerado uma boa prática de programação. Outro motivo para declarar uma função como sendo constante é que apenas uma função constante pode ser chamada por um objeto constante.

Por exemplo, suponhamos que foi passado um objeto Carro constante para uma função. A essa função apenas será permitido usar a função getCor() se essa função for constante, ainda que ela não altere o objeto. Para declarar uma função constante, basta adicionar a palavra-chave const após a lista de parâmetros que essa função recebe. Como exemplificado na listagem 2.10.

Código 2.10: Métodos Const

```
#include "carro.h"

int Carro::getAceleracao() const
{
  return aceleracao;
}

void ultrapassarCarro(const Carro &carro_a_ser_ultrapassado)
{
  carro_a_ser_ultrapassado.getAceleracao();
}
```

Quando começamos a trabalhar com o modificador const é natural que haja um pouco de confusão, pois, por exemplo, as seguintes declarações são igualmente válidas [2.11]:

Código 2.11: Tipos de declaração Const

```
const Fred* p;
Fred* const p;
const Fred* const p;
```

Primeiro considere que Fred nesses exemplos é um nome de qualquer tipo de dados (classe ou tipo primitivo). Porém, antes que você que você se assuste com esses exemplos, note que a melhor maneira de ler esse tipo de declaração é indo da direita para a esquerda. Então no exemplo anterior teremos:

const Fred* p: *p* é um ponteiro para um *Fred* constante, logo não podemos usar *p* para mudar o valor do *Fred* apontado;

Fred* const p: *p* é um ponteiro constante para um *Fred*, logo podemos mudar o valor do *Fred*, mas não podemos mudar para qual *Fred p* aponta;

const Fred* const p: *p* é um ponteiro constante para um *Fred* constante, o que significa que não podemos mudar o *Fred* e nem para que *Fred* este ponteiro aponta.

2.6 Ponteiro this

Às vezes, desejamos fazer referência ao próprio objeto, dentro da classe. Para essas situações, existe um ponteiro "this" que aponta para o objeto da classe. Mais tarde, por exemplo, veremos como implementar para uma classe um operador "+=". Nesse caso, é interessante retornar o objeto, acrescido de algum atributo. Utilizemos a função MaisIgual para simular a semântica de um operador += entre vetores. A sintaxe seria como mostrado no código 2.12.

Código 2.12: Operador +

```
Vetor2D maisIgual(Vetor2D vetor_a_ser_somado)
{
  x += vetor_a_ser_somado.getX();
  y += vetor_a_ser_somado.getY();
  return *this;
}
```

Capítulo 3

Sobrecarga

Sobrecarga, pode ser encarada como uma prática utilizada na programação, que visa associar a um mesmo nome (ou símbolo) mais de um comportamento. Para entender o conceito vamos começar com um tipo simples de sobrecarga, a sobrecarga de funções. Entende-se por isso, definir várias funções com o mesmo nome. Em C++ esta prática é possível, desde que o conjunto de parâmetros de todas seja diferente. O compilador consegue definir qual função chamar de acordo com a lista de parâmetros da função chamada. Neste capítulo serão abordadas técnicas e possibilidades de sobrecarga na linguagem C++.

3.1 Quando usar?

A prática da sobrecerga é comum e aconselhada para funções que fazem coisas parecidas, ou seja, possuem uma mesma semântica, mas agem sobre um conjunto diferente de dados. Vale notar que abusar deste tipo de notação pode tornar nosso código de difícil leitura. Além disso é preciso tomar muito cuidado com funções sobrecarregadas e funções com parâmetros default, pois é bem fácil de se deparar com uma ambigüidade misturando estas facilidades da linguagem. O compilador não tem condição, nessas situações, de decidir que função chamar, como exemplificado no código 3.1.

3.2 Modificador friend

Antes de iniciarmos o assunto do capítulo propriamente dito, será necessário abordar um pequeno assunto da linguagem C++: funções e classes friend. Uma função friend é definida fora do escopo de uma classe, não sendo uma função membro . Ainda assim, esta função tem permissão para acessar membros private e protected da classe (e os membros public, naturalmente).

Usar funções friend melhora o desempenho, entretanto, alguns estudiosos consideram que funções friend violam a ocultação de dados da orientação a

Código 3.1: Sobrecarga

```
#include <iostream>

using namespace std;

int quad( int x )
{
  return x*x;
}

float quad( float x )
{
  return x*x;
}

int main()
{
  cout << quad( 2 ) << endl;
  cout << quad( 2.5 ) << endl;
  return 0;
}
```

objetos. Para declarar uma função como friend de outra classe deve-se preceder o protótipo da função na definição da classe com a palavra-chave friend.

Uma classe friend funciona da mesma forma que uma função friend, isto é, tem acesso tanto a membros private como protected da classe da qual é friend. Para declarar uma classe como friend de outra, deve-se colocar na declaração da segunda friend.

Exemplos da sintaxe de friend podem ser vistos nos códigos 3.2, 3.3.

3.3 Fundamentos de Sobrecarga de Operadores

Sobrecarga de Operadores é um recurso bastante poderoso da linguagem C++, que, por sua vez, não existe em outras linguagens como Java. Na verdade, sua inexistência em outras linguagens se deve ao fato de que tudo que pode ser feito com este recurso pode ser realizado também de outras maneiras, porém, não de maneira tão clara e elegante.

Basicamente, a sobrecarga de operadores permite ao programador defina o comportamento dos operadores básicos da linguagem, como o operador de adição (+) ou de atribuição (=), para que trabalhem de maneira correta com os tipos de dados criados pelo usuário. Isto é, será possível usar tais operadores com suas próprias classes, ao invés de apenas com os tipos de dados primitivos. Imagine, por exemplo, uma classe Vetor (um segmento de reta com origem em um ponto do espaço e destino em outro ponto do espaço). Esta classe foi criada para auxiliá-lo nas suas tarefas de cálculo. Contudo, como foi uma classe

Código 3.2: Função Friend

```
class X
{
  public:

  X() {}
  ~X() {}
  friend void f(X algo);

  private:

  int a,b,c;
};

void f(X algo)
{
  algo.a + algo.b + algo.c;
}

int main()
{
  X alguem;
  f(alguem); //okay
}
```

Código 3.3: Classe Friend

```
class ClasseN { friend class ClasseM; ... };

class ClasseM { ... };
```

criada pelo usuário, o compilador não saberia como proceder diante da seguinte expressão: vetor1 = vetor2 + vetor3;

Visto que os operadores de adição e atribuição estão definidos apenas para os tipos primitivos de dados. Para resolver este pequeno problema, precisamos utilizar o recurso de sobrecarga de operadores disponibilizado pelo C++. A própria linguagem utiliza este recurso internamente. O operador de adição, por exemplo, trabalha diferentemente com ints, floats e doubles, entretanto, utiliza-se o mesmo operador para realizar tais operações aritméticas.

Os seguintes operadores da linguagem C++ podem ser sobrecarregados:

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[ ]	( )	new	delete
new[ ]	delete[ ]						

Os seguintes operadores da linguagem C++ NÃO podem ser sobrecarregados:

.	.*	::	?:	sizeof

Cabe lembrar que o uso excessivo de sobrecarga de operadores pode tornar o código bastante difícil de ler. Logo, este recurso deve ser utilizado de maneira muito bem pensada e, de preferência, com algum significado similar ao significado original do operador. Classes matemáticas utilizam muito bem este recurso, como, por exemplo, uma classe de números complexos.

3.4 Funções Friend

A sobrecarga de operadores pode ser realizada de duas formas distintas. A primeira delas é declarar a função de sobrecarga de operador como uma função membro de uma classe. Esta maneira é a que deve ser sempre priorizada, contudo ela tem uma certa limitação: o operando a esquerda do operador deve ser obrigatoriamente da classe que contém a sobrecarga.

Esta limitação não costuma ser um grande problema, porém existem situações em que será necessário contornar este problema. Como forma alternativa, a segunda forma de sobrecarregar um operador é declarando a função de sobrecarga como uma função friend da classe relacionada. Quando sobrecarregamos os operadores de inserção em streams e extração de streams (« e »), por exemplo, o operando da classe ficará à direita, de modo que será necessário utilizar esse artifício.

3.4.1 Sintaxe

A sintaxe para realizar a sobrecarga de um operador é bastante parecida à de uma função comum, contudo o nome da função deve iniciar com a palavra reservada *operator* seguido do operador a ser sobrecarregado. Como mostrado no código 3.4.

Código 3.4: Sobrecarga de Operadores

```
bool operator! () const;

const Vetor2D operator+ (int escalar);
const Vetor2D operator* (int escalar);
```

3.5 Sobrecarga de Operadores Unários

A sobrecarga de operadores unários pode ser realizada de duas formas diferentes, como foi visto anteriormente. Se escolhermos que a função de sobrecarga deve ser uma função membro da classe, então será uma função sem parâmetros, visto que o único operando será o próprio objeto da classe que realiza a chamada do operador, como podemos ver no exemplo 3.5.

Código 3.5: Operadores Unários

```
//classe de numeros complexos
class Complexo
{
  public:

  bool operator!();
  /* verifica se os termos reais e imaginarios do numero
      complexo sao 0 */

};
```

Entretanto, ainda podemos escolher que a função de sobrecarga seja uma função friend da classe em questão. Nesse caso, a função deverá ter um parâmetro, que será o operando modificado pelo operador.

Fica claro, então, o porquê da função necessitar ser friend da classe: esta precisará acessar membros private da classe, e caso isto fosse realizado usando funções de encapsulamento (setter e getter) acarretaria em certa perda de desempenho, ou mesmo seria impossível se nem todos os valores necessários para computar a função tivessem funções setter e getter correspondentes.

3.6 Sobrecarga de Operadores Binários

A sobrecarga de operadores binários é feita de maneira bastante semelhante à sobrecarga de operadores unários. Cabe lembrar que o C++ não sobrecarrega operadores ternários, o único operador ternário (:?) não está na lista de operadores que podem ser sobrecarregados. Considerando o primeiro caso, em que a função de sobrecarga seja uma função membro da classe em questão, necessitamos de apenas um parâmetro, visto que o outro parâmetro já é o próprio objeto da classe em questão.

Fica claro, assim, a limitação citada anteriormente: o operando da esquerda é obrigatoriamente do mesmo tipo da própria classe em questão e o operando da direita pode ser definido como qualquer tipo no parâmetro da função.

Código 3.6: Operadores Binários

```
class Complexo
{
  public:
  Complexo operator+= ( Complexo &c);
  Complexo operator+ ( Complexo  &c);
  Complexo operator+ ( float f);
  //...
};

Complexo& Complexo::operator+= ( Complexo &c) {
  real += c.real;
  imaginario += c.imaginario;
  return *this;
}

Complexo Complexo::operator+ ( Complexo &c) {
  return Complexo (real + c.real, imaginario + c.imaginario);
}

Complexo Complexo::operator+ ( float f) {
  return Complexo(real + f, imaginario);
}
```

Já quando escolhemos que o operador binário será sobrecarregado através de uma função friend da classe, podemos determinar o tipo de dado tanto do operador da esquerda quanto do operador da direita através da declaração dos dois parâmetros da função.

3.7 Sobrecarga dos Operadores de Streams

Para sobrecarregar os operadores de manipulação de streams, é necessário utilizar a sobrecarga através de funções friend, e não como funções membro da classe. Isso se deve ao fato de que, numa expressão deste tipo, o membro

Código 3.7: Operadores Binários Friend

```
class Complexo
{
  friend Complexo operator+= ( Complexo &, Complexo & );
  //...
};
```

da classe ficará sempre no lado direito do operador. E deve-se lembrar que só se podem utilizar funções membro de uma classe para sobrecarga quando o operando a esquerda do operador for um membro desta classe.

3.7.1 Exemplo dos operadores de streams (« e »)

```
cin >> x; cout << y;
```

Fora isso, a sobrecarga dos operadores de streams é semelhante às funções de sobrecarga vistas anteriormente. O código 3.8 exemplificando como funciona este recurso. Esta seria a implementação das funções para o input e output de números complexos. Cabe lembrar:

Código 3.8: Operadores de Streams

```
ostream &operator<< (ostream &output, Complexo &c)
{
  output << "(" << c.getReal() << ", " << c.getImaginaria() << "
      )";
  return output;
}

istream &operator>> (istream &input, Complexo &c)
{
  input.ignore();
  // salta
  input >> c.real;
  input.ignore(2);
  input >> c.imaginaria;
  input.ignore();
  return input;
}
```

Agora, declare estas funções como friend da sua classe Complexo. Insira o código das funções no arquivo '.cpp' e teste o funcionamento delas em um programa teste.

3.8 Conversão entre Tipos

Freqüentemente, é necessário converter dados de um tipo para o outro em programas e isto é realizado utilizando operadores de cast. Contudo, o compilador só sabe realizar essas operações entre os tipos primitivos de dados. Então, caso deseje realizar um cast de um classe criada pelo usuário será preciso implementar a função que faz isso. O que pode ser visto no código 3.9:

Código 3.9: Operador de Conversão

```
// tipo de dados criado pelo usuario
string s;
// cast de um tipo criado pelo usu rio para um tipo primitivo
char *x = (char*)s;

//para o caso de strings, recomendamos a seguinte notacao
char *y = s.c_str();
```

Através deste artíficio, é possível realizar uma conversão de uma classe para outra classe ou de uma classe para um tipo primitivo. O protótipo de função para um operador de cast é o seguinte:

3.9 Sobrecarregando ++ e - -

Os operadores de incremento e decremento também podem ser sobrecarregados, contudo estes têm certa peculiaridade devido ao fato de que podem ser pré/pós – incremento/decremento. Na verdade, o único cuidado que deve ser tomado é o de diferenciar as funções para que o compilador saiba quando deve chamar cada função.

A versão de prefixo, isto é o pré-incremento e o pré-decremento, são implementadas da mesma forma que as funções de sobrecarga que vimos até agora, sem nenhuma mudança. A diferenciação vem na outra versão, a de pós-incremento e pósdecremento, pois agora é preciso diferenciá-la da anterior. Para isso, foi adotada uma convenção em que este tipo de função deve ter um parâmetro fantasma inteiro, com o valor de 0.

Código 3.10: Sobrecarga do ++

```
Complexo operator++();
// funcao de pre-incremento
Complexo operator++(int x);
// funcao de pos-incremento
```

Atenção: Estas diferenças garantem apenas a chamada da função correta. O programador devará garantir o fato da função ser pré-incrementada ou pós-incrementada.

Capítulo 4

Herança

Nesta aula iremos estudar um dos recursos mais importantes da programação orientada a objetos: a herança. Ela nos permite reutilização de software de qualidade reconhecida, redução da quantidade de código repetido e nos auxilia na manutenção do software.

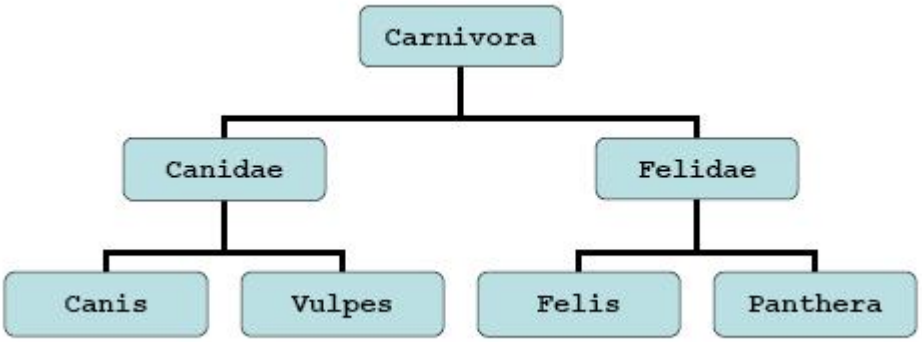
No decorrer da aula mostraremos exemplos de cada uma destas utilizações. Antes de começarmos a codificar a herança em C++ vamos entender seu significado no mundo dos objetos. Segundo o site www.direitonet.com.br, herança: "É o patrimônio deixado pela pessoa falecida aos seus herdeiros, ou seja, é a totalidade dos direitos e obrigações de uma pessoa no momento em que vem a falecer."

No mundo dos objetos, herança possui um sentido muito parecido com o exposto acima com duas diferenças: que não é necessário que uma classe "morra" para que outra possa herdar seus atributos e funções membros, nem a classe que é herdada "perde" seus atributos e funções membros para a classe que a herdou; é feita uma "cópia" deles. Vamos começar definindo melhor os conceitos apresentados acima:

4.1 Classe base e Classes Derivadas:

Classe base e Classes Derivadas: Uma relação entre objetos já vista foi a "tem um", onde um objeto continha como membros objetos de uma classe distinta da sua. Por exemplo, um carro tem um passageiro. A herança introduz um relacionamento do tipo "é um", onde um objeto de uma classe "é um" objeto de outra classe também. Podemos dizer que um retângulo é um quadrilátero (assim como um losango e um paralelogramo também o são). Assim, dizemos que a classe Retangulo herda da classe Quadrilatero. Dizemos então que Quadrilatero é a classe base enquanto Retangulo é a classe derivada.

Um dos exemplos mais clássicos de herança é a classificação taxonômica dos seres vivos. Mostramos a seguir um exemplo que mostra bem a estrutura de árvore apresentada pela herança:



Vemos que a classe base para todas é a classe Carnivora, que poderia apresentar uma função membro 'void comerCarne();' (pode-ser argumentar que nem todos carnívoros comerão carne da mesma maneira. Mais tarde veremos que isso pode ser resolvido através da sobrescrita dessas funções). Seguindo, vemos que a classe Canidae é base para Canis e Vulpes, e que Felidae é base para Felis e Panthera.

Tendo todas essas classes definidas é facil criarmos várias classes novas: podemos ter um Cachorro, um Lobo e um Coiote sem termos que escrever muito código, pois herdamos as características comuns da classe Canis. Também, podemos ter um Leao, um Tigre e uma Pantera como classes derivadas de Panthera, sem nos preocupar com suas características mais comuns (pois estas já estão escritas). Ou seja, o que fizemos foi juntar as partes comuns as classes que queremos definir.

Esse é um exemplo da redução de código repetido. Temos tudo pronto. Agora, no entanto, descobrimos que existe um algortimo de alimentação mais eficiente, e queremos implementá-lo em nossos animais. Se não estivéssemos usando herança, seria necessário escrever a nova função em cada uma das classes (quanto mais escrevermos, maiores as chances de erros). Mas, graças a herança, modificamos a função em nossa classe base e todos animais estarão comendo dessa maneira (excetuando as classes onde sobrescrevemos a função).

Esse é um exemplo da facilidade de manutenção de software que a herança proporciona. Vamos supor que o objetivo dessas classes era o de implementar uma simulação de zoológico. O software é terminado, testado e passa-se a usá-lo.

Alguns bugs vão aparecendo e sendo corrigidos e depois de um tempo o software se torna razoavelmente estável. Após alguns meses, chegam novos animais, um Leopardo por exemplo. Já temos nossa classe Panthera com a qualidade reconhecida. Ao criarmos a classe Leopardo, será fácil e teremos que nos concentrar em corrigir erros de um código muito menor (apenas a parte específica do Leopardo). Esse é um exemplo da reutilização do código.

4.2 Membros Protected

Até agora utilizamos duas formas de controle de acesso a membros: public e private. Como sabemos, membros public podem ser acessados por todas as funções no programa, enquanto membros private podem ser acessados apenas por funções membros da classe e suas friends. O acesso protected confere uma proteção intermediária entre os acessos public e private.

Membros protected podem ser acessados por funções membros e funções membros de classes derivadas, assim como as friends da classe base (cabe ressaltar que membros friends não são herdados pelas classes derivadas e devem ser redeclarados se desejamos que a função seja friend da nova classe).

4.2.1 Sintaxe

Representamos a herança em C++ utilizando os dois-pontos (:) após o nome da classe derivada, a que queremos que herde as características, seguida por um dos três tipos de herança que temos em C++ (public, private e protected), seguido do nome da classe base. Para declararmos que uma classe Derivada tem como base a classe Base, utilizando herança public, utilizamos o seguinte código:

```
class Derivada : public Base { ... };
```

O tipo de herança mais utilizado é o public. Com herança public, os membros public e protected da classe base são herdados como membros public ou protected na classe derivada, respectivamente. Membros private da classe base não podem ser acessados na classe derivada. No entanto, podemos utilizar funções public ou protected da classe base para acessar seus membros private (como as funções sets e gets).

4.2.2 Exemplos de Herança

Agora vamos apresentar um exemplo de implementação de herança. Iremos criar a classe Canis e a classe Cachorro. A classe Canis será base para Cachorro. As classes podem ser vistas nos códigos, 4.1, 4.2, 4.3

Na implementação da classe derivada Cachorro, vemos que seu construtor chama o construtor da classe base Canis, através de um inicializador de membros. Sempre devemos antes inicializar a parte da classe Base. Se não chamarmos explicitamente o construtor através de um inicializador de membros, o construtor 'default' será chamado (não havendo construtor 'default', será acusado um erro de sintaxe).

Os construtores são sempre chamados seguindo a ordem da herança: da classe base mais alta na árvore até a classe mais baixa, não importando a ordem em que são escritos no código. Destrutores são chamados na ordem inversa.

Código 4.1: canis.h

```
// Header da classe Canis
// Canis.h

#ifndef CANIS_H
#define CANIS_H
class Canis
{
  public:

  Canis(const char *a_name);
  void speak();

  protected:
  char *nome;
};
#endif
```

Código 4.2: cachorro.h

```
// Header da classe Cachorro;
#ifndef CACHORRO_H
#define CACHORRO_H
#include "Canis.h"

class Cachorro : public Canis
// Cachorro herda membros public e
// protected de Canis;
{
  public: Cachorro(const char *a_name, const char *a_owner);
  protected: char *owner;
};

#endif // Implementacao da classe Cachorro.
```

Código 4.3: cachorro.cpp

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

using std::cout;
using std::endl;

//Header da classe;
Cachorro::Cachorro(const char *a_name, const char *a_owner) :
    Canis(a_name)
//Inicializacao da parte Canis do Cachorro.
{
  owner = new char[ strlen(a_owner) + 1];
  strcpy(owner,a_owner);
}
```

4.3 Coerção de ponteiros entre classes bases e classes derivadas

Uma das utilidades que a herança nos proporciona é a de sempre podermos tratar nosso objeto como um objeto da classe Base. Supondo nossa simulação de zoológico, nada seria preciso fazer para tratarmos o novo Leopardo se já tratássemos todos Felinos da mesma maneira. Ou seja, quem trata dos animais enxerga todos como objetos da classe Felidae, assim sendo, basta convertermos nosso Leopardo para Felidae que a função 'void tratarFelinos(Felidae *listaFelinos,int quantidade);' já resolve o problema.

Mas, como fazer isso? A conversão pode ser feita em dois sentidos:

1. Um ponteiro para uma classe base sempre pode referenciar automaticamente um objeto de uma classe derivada. Estaremos tratando assim esse objeto como se ele fosse da classe base. Funções e variáveis específicas da classe derivada não poderão mais ser acessadas por esse ponteiro.

2. Um ponteiro para uma classe base pode ser convertido para um ponteiro de uma classe derivada. No entanto, isso pode ser perigoso se feito sem cuidado. Podemos começar a tratar um Lobo como um Cachorro, e tentarmos levar ele passear, como visto no código 4.4...

O problema é que o compilador confia, e deixa nós compilarmos e até tentarmos executar esse código! Pior ainda, se não referenciarmos nenhuma variável que o lobo não tenha, "funcionará". Se mais tarde referenciarmos alguma, vamos ter uma bela dor de cabeça tentando descobrir o erro. Sugestão: Isso não se faz!

Também aparece no trecho de código como podemos converter um ponteiro de classe derivada para um ponteiro de classe base: basta fazer uma atribuição que a conversão é automática! Podemos, no entanto, realizar essas conversões

Código 4.4: Coerção

```
#include "canis.h"
#include "cachorro.h"
#include "lobo.h"

int main()
{
  Cachorro dogObj("Rex","Arthur");
  Lobo wolfObj("Lobo Mau");

  Canis *canisPtr;
  Cachorro *dogPtr;
  Lobo *wolfPtr;

  canisPtr = &wolfObj;

  // Ok! Estamos tratando o lobo
  // como um Canis :D
  dogPtr = static_cast<Cachorro *>(&canisPtr);

  // Temos um Canis...
  //Pode ser um cachorro... sera?
  dogPtr->levar_passear();
}
```

com segurança. Novas técnicas foram introduzidas em versões mais recentes, entre elas: conversão dinâmica e descoberta de tipo em tempo real. Iremos dar uma breve introdução à coerção dinâmica de ponteiros e como podemos utilizá-la para transitar entre ponteiros de classe base para classe derivada.

Seu funcionamento é o seguinte: tenta-se realizar a coerção. Se bem sucedida, seu retorno é o ponteiro coertido, caso contrário o retorno é o ponteiro null (0). Assim sendo, podemos substituir o código antigo pelo em 4.5:

Código 4.5: Coerção Dinâmica de Tipos

```
// dynamic_cast<Derivada *>(Ponteiro_Classe_Base)

if (dogPtr = dynamic_cast<Cachorro *>(canisPtr))
{
  dogPtr -> levar_passear();
}
```

Da mesma forma, nossa função aparentemente genérica que alimenta todos nossos Felinos internamente pode tratá-los individualmente, como exemplificado no código 4.6:

Como vemos, o método consiste em encadear as tentativas de coerções e dentro dos ifs utilizar os ponteiros convertidos com segurança.

Código 4.6: Mais Coerção Dinâmica de Tipos

```
void tratarFelinos(Felidae *listaFelinos,int quantidade)
{
  Felis *FelisPtr;
  Panthera *PantheraPtr;

  for (i = 0 ; i < quantidade ; i++)
  {
     if (FelisPtr = dynamic_cast<Gato *>(listaFelinos[i]))
     {
              //Trata os Felinos atraves de FelisPtr.
          }
          else if (Panthera = dynamic_cast<Gato *>(listaFelinos[
              i]))
          {
              // Trata os Pantheras atraves de PantheraPtr.
          }
          else
          {
              // Trata-se os outras casos.
          }
  }
}
```

4.4 Sobrescrevendo membros da classe base em uma classe derivada

Ao sobrescrever uma função de uma classe base estamos redefinindo a função. Quando a função é mencionada na classe derivada, a versão nova é automaticamente selecionada. Pode-se selecionar a versão da classe base utilizando-se o operador de resolução de escopo (::). Sobrescrevemos uma função ao declararmos na classe derivada uma função com mesma assinatura que na classe base. A sobrescrita de funções é um recurso muito importante para o próximo tópico que é Polimorfismo.

Quando introduzirmos os conceitos de funções virtuais e polimorfismo é que veremos uma utilidade realmente fascinante da sobrescrita de funções. A função void tratarFelinos(...) poderia dispensar o uso da coerção de ponteiros dinâmica se cada vez que usássemos FelisPtr->comer(); fosse chamada a última função sobrescrita, no lugar da declarada por Felis. Isso é possível, mas é assunto para a próxima aula. O exemplo a seguir mostra a nova definição da classe cachorro e sua implementação com a sobrescrita da função void speak();.

Após, é mostrada um exemplo de sua utilização com a chamada da função por um ponteiro de Canis (será chamada a primeira versão). Os códigos podem ser encontrados em 4.7, 4.8 e 4.9.

Código 4.7: Header da classe cachorro

```
/* Cachorro com a funcao speak() */

// Header da classe Cachorro;
#ifndef CACHORRO_H
#define CACHORRO_H
#include "Canis.h"

class Cachorro : public Canis
// Cachorro herda membros public e protected de Canis;

{
  public: Cachorro(const char *a_name, const char *a_owner);
  void speak(); // Sobrescrita de 'void speak()' de Canis;
  void levar_passear();

  protected:
  char *owner;
};
#endif // Implementacao da classe Cachorro.
```

Código 4.8: Cachorro com função speak

```
//{ #includes necessarios ... suprimidos }

Cachorro::Cachorro(const char *a_name, const char *a_owner)
// Inicializa a parte Canis do Cachorro. : Canis(a_name)
{
  owner = new char[ strlen(a_owner) + 1];
  strcpy(owner,a_owner);
}

void Cachorro::speak()
// Sobrescrevendo a funcao 'void speak()';
{
  cout << "Metodo da classe base" << endl;
  Canis::speak();
  cout << endl << "Au-au-au!!!" << endl;
}

void Cachorro::levar_passear()
{
  cout << name << " , vem! Sou eu, " << owner << ". Vamos
      passear?" << endl;
}
```

Código 4.9: Demonstração da função speak

```
int main()
{
  Canis canisObj("Canis01");
  // Cria e inicializa um Canis
  Cachorro dogObj("Rex","Eu");
  // Cria e inicializa um Cachorro
  Canis *canisPtr = 0;
  canisObj.speak();
  dogObj.speak();
  canisPtr = &dogObj;
  canisPtr->speak();

  return 0;
} // Ponteiro para um Canis;
```

4.4.1 Output

```
Canis01 diz: Grrr...
Rex diz: Grrr... Au-au-au!!!
Rex diz: Grrr...
```

4.5 Tipos de Herança

Até o momento todos exemplos utilizaram o tipo de herança public, o que permite relações do tipo "é um" (todo Cachorro "é um" Canis também). A linguagem C++ permite outros dois tipos de herança: protected e private. Esses tipos são mais recentes e menos usados, principalmente por não permitirem relacionamentos do tipo "é um". Heranças do tipo protected são usadas muitas vezes como uma forma de composição.

Definimos o tipo de herança desejada especificando seu tipo antes do nome da classe base (após os dois-pontos). A diferença entre os tipos de herança está na forma como os membros herdados da classe base são vistos na classe derivada.

Na herança public os dados com escopo protected continuam a ser protected na classe base e os dados com escopo public continuam a ser public na classe base. Na herança protected, os dados com escopo protected e public passam a ser protected. Na herança private, os dados protected e public passam a ser private. Os dados private da classe base nunca podem ser acessados diretamente pelas classes derivadas (isso quebraria o encapsulamento das classes). A tabela a seguir resume o que foi dito:

Especificador de Acesso	Tipo de Herança		
	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	oculto	oculto	oculto

4.6 Herança Múltipla

A herança múltipla é uma extensão da herança simples. Nela, a classe derivada herda as funções e dados de diversas classes bases ao mesmo tempo. É caracterizada quando uma determinada classe C "é uma" classe B e "é uma" classe A ao mesmo tempo. Dominar a herança simples pode ser conseguido com o que foi apresentado e um pouco de prática. Heranças simples são muito mais fáceis de serem observadas e utilizadas em projetos de software do que a herança múltipla.

Esta última é um recurso muito poderoso de C++, possibilitando formas interessantes de reutilização de software, mas é muito mais sujeita a erros e pode causar diversos problemas de ambigüidade. Vamos apenas mostrar um exemplo de herança múltipla, por se tratar de um tópico mais avançado. Nunca se deve utilizar a herança múltipla quando a herança simples for suficiente. No exemplo abaixo, criamos as classes Homem e Lobo para criarmos então a classe derivada Lobisomem através da herança múltipla. Lobo é derivada de Canis (para treinarmos um pouco mais herança simples), enquanto Homem é uma classe completamente nova. Observa-se que ambas as classes possuem as funções speak() (mas que acaba sendo sobrescrita após), e que ambas possuem o atributo "name". Isso é um exemplo de ambigüidade. Ao tentar criar a função char *getNome() return name; foi gerado o seguinte erro de compilação:

1>c:\lobisomem.cpp : error C2385: ambiguous access of 'name' 1> could be the 'name' in base 'Homem' 1> or could be the 'name' in base 'Canis'

O que mostra os perigos da herança múltipla. No entanto, na sobrecarga de temos um exemplo de um uso interessante da herança múltipla, onde através do ponteiro this fomos capazes de chamar as funções speak(); de suas duas classes bases. O código pode ser visto em, 4.10, 4.11, 4.12.

Como pode ser observado, para criarmos uma herança múltipla basta colocarmos uma lista de classes base separadas por vírgulas após os dois pontos depois do nome da classe derivada. No código 4.13 podemos ver o teste e os seus resultados:

4.6.1 Output

```
Lobisomem de St. Bonnet diz: Grrr...
Gilles Garnier diz: Olá pessoal! Vou devorar vocês!
```

Código 4.10: Classe Homem

```
// Header da classe Homem

#ifndef H_HOMEM
#define H_HOMEM

class Homem
{
  public: Homem(const char *a_name);
  void speak();

  protected:
  char *name;
};

#endif
```

Código 4.11: Classe Lobo

```
// Header da classe Lobo
#ifndef H_LOBO
#define H_LOBO

#include "Canis.h"
class Lobo : public Canis
// Lobo herda membros public e protected de Canis;
{
  public: Lobo(const char *a_name);

};

#endif

//.......

// Implementacao da classe Lobo.

Lobo::Lobo(const char *a_name) : Canis(a_name)

// Inicializacao da parte Canis de Lobo.
//{ Nenhum inicializa   o extra. }
```

Código 4.12: Classe Lobisomem

```
// Implementacao da classe Lobisomem
// Ela herda caracteristicas dos humanos e dos lobos,
// Tambem acaba possuindo dois nomes! (ambiguidade)

Lobisomem::Lobisomem(const char *human_name, const char *
    wolf_name) :Homem(human_name),Lobo(wolf_name) { }

void Lobisomem::speak()
{
  Lobo *loboptr = static_cast<Lobo *>(this);
  Homem *homemptr = static_cast<Homem *>(this);
  loboptr->speak();
  cout << endl ;
  homemptr->speak();
  cout << endl;
  cout << "Vou devorar voc s!\n";
}
```

Código 4.13: Tudo funcionando

```
int main ()
{
  Lobisomem lobis("Gilles Garnier","Lobisomem de St. Bonnet");
  lobis.speak();
  return 0;
}
```

Capítulo 5

Funções Virtuais e Polimorfismo

O polimorfismo é a capacidade de alguns ponteiros apontarem para diferentes tipos de dados. Essa idéia não é trivial e está muito conectada ao conceito de herança entre classes, de modo que se faz necessário um entendimento profundo dos capítulos anteriores e uma dedicação especial ao estudo deste.

5.1 Funções Virtuais

Suponha a seguinte hierarquia de classes:

```
Classe Base: FormaGeométrica
Classes Derivadas: Quadrado, Triângulo, Círculo
```

Agora suponha que devemos possuir uma lista de formas, e desenhá-las de maneira genérica (sem selecioná-las por tipo) na tela. Obviamente, a função que desenha um círculo, não é igual a que desenha um Triângulo ou um Quadrado. Dessa forma, a existência de uma função desenha() na classe FormaGeométrica não serviria para todos os objetos. Porém, a função que desenha a lista deve ser genérica, de modo que não podemos selecionar separadamente a função desenha() em cada uma das classes.

E agora? O polimorfismo tem como objetivo exatamente a resolução desse problema. Percebemos, é claro, que poderíamos solucionar essa situação utilizando switchs, não fosse a restrição "genericamente" citada acima. Porém, existe uma solução muito mais elegante e limpa. Ponteiros polimorfos possuem a característica de, sendo da classe base, poderem apontar para uma função da classe base e escolher dinamicamente a função adequada na classe derivada a ser executada. Os trechos de código: 5.1, 5.2, 5.3 ajudam na compreensão.

Se testarmos o programa acima, veremos que o ponteiro "sabe" para que tipo de objeto está apontando. Como o polimorfismo é implementado é um

Código 5.1: Classe FormaGeometrica

```
#include <iostream>

class FormaGeometrica
{
  public:
  FormaGeometrica();
  ~FormaGeometrica();
  virtual void desenha() const;
  int getX() const;
  int getY() const;
  int getZ() const;
  void setX(const int z3);
  void setY(const int z2);
  void setZ(const int z1);

  private:
  int x, y, z;
};
```

Código 5.2: Classe Triangulo

```
#include "formaGeometrica.h"

class Triangulo : public FormaGeometrica

{
  public: Triangulo();
  ~Triangulo();
  virtual void desenha() const;

  private:
  int lado1, lado2, lado3;
};
```

Código 5.3: Classe Quadrado

```
#include "formaGeometrica.h"

class Quadrado : public FormaGeometrica
{
  public: Quadrado();
  ~Quadrado();
  virtual void desenha() const;
  private: int lado;
};
```

assunto complexo e não pertence ao escopo deste curso, de modo que veremos apenas o que deve ser feito para que ele aconteça. Podemos observar que na classe base a função desenha() foi declarada como virtual.

As declarações seguintes (nas classes derivadas) são apenas uma questão de clareza, visto que uma vez declarada virtual, uma função permanece virtual por toda a hierarquia de classes. Essa declaração é o que garante que o ponteiro escolherá dinamicamente qual a função adequada. É válido observar que essa escolha dinâmica é exclusividade de ponteiros. Se a função for chamada através de um objeto, a vinculação será estática, e a função será a pertencente à classe do objeto em questão.

5.1.1 Teste de Aprendizado

Construa três classes: uma classe base Empregado e duas classes derivadas Contratado e Comissionado. As classes devem possuir a função imprimeSalario, que no caso do Contratado, deve ser um valor fixo, e no caso do Comissionado deve ser um valor relativo à quantidade de produtos vendidos.

Implemente um programa que escreva o salário de cada comissionado a partir de um ponteiro polimorfo da classe base. Obs: Não perca tempo com o cálculo do salário, imprima um valor constante (Ex: "O contratado recebe 600 reais"/ "O comissionado recebe 25% do lucro gerado"). O objetivo do exercício é testar o entendimento do polimorfismo, não da matemática.

Obs2: Não declare as funções como virtual no .cpp, apenas no .h.

5.2 Ponteiros Polimorfos como Parâmetros

Às vezes, desejamos criar uma função genérica para tratar inúmeras classes de uma mesma hierarquia. Um caso em que essa técnica é útil, por exemplo, é caso desejemos derivar nossa classe de uma classe cujo código não nos é acessível. Isso é possível usando um parâmetro polimorfo. Como exemplo, utilizemos as classes já declaradas, FormaGeometrica, Triangulo e Quadrado.

Caso desejemos, por exemplo, criar uma função Move, para os dois tipos de forma geométrica, é necessário passar como parâmetro algo genérico, ou seja, um ponteiro para a classe base. Como descrito no código 5.4.

5.2.1 Teste de Aprendizado

1. Essa não seria a melhor maneira de resolver o problema, é apenas um modo de reforçar o conceito. Qual seria a melhor maneira?

2. Reescreva o programa de modo a utilizar a melhor solução para o problema, discutida no exercício 1.

Código 5.4: Exemplo de Polimorfismo

```
#include <iostream>

#include "Polimorfismo.h"

// Funcao Move

void Move(FormaGeometrica *forma, int x, int y, int z)
{
  forma->setX(x);
  forma->setY(y);
  forma->setZ(z);
}

// Programa principal

int main()
{
  Quadrado quad;
  Triangulo tri;
  FormaGeometrica *form;

  form = &quad;
  Move(form, 10, -10, 0);
  form = &tri;
  Move(form, 20, 20, 20);
  std::cout << "x do quadrado: " << quad.getX() << std::endl;
  std::cout << "y do quadrado: " << quad.getY() << std::endl;
  std::cout << "z do quadrado: " << quad.getZ() << std::endl;
  std::cout << "x do triangulo: " << tri.getX() << std::endl;
  std::cout << "y do triangulo: " << tri.getY() << std::endl;
  std::cout << "z do triangulo: " << tri.getZ() << std::endl;
  return 0;
}
```

5.3 Classes Base Abstratas e Classes Concretas

Quando pensamos em uma classe como um tipo de dados, imediatamente se tem a idéia de que variáveis daquele tipo serão criadas. E isso, obviamente, é o que deveria acontecer, pois uma classe sem objetos não tem utilidade nenhuma, não é? Não é. Tomemos como exemplo a classe FormaGeometrica, citada anteriormente.

Existe realmente um objeto FormaGeometrica? Ou cada objeto de FormaGeometrica se encaixaria dentro de uma classe derivada mais específica, mais concreta? Classes como a FormaGeometrica estão implementadas na linguagem C++, e são conhecidas como classes abstratas.

Classes abstratas não constituem um tipo, de forma que não podem ser declarados objetos dessas classes. Podem, entretanto, ser declarados ponteiros para essas classes, o que praticamente explicita seu uso extensivo para polimorfismo, como agrupador das classes derivadas. Para informar ao compilador que uma classe é abstrata, basta que se declare uma de suas funções como sendo uma função virtual pura. Para isso, adiciona-se um inicializador "= 0" na declaração da função. Note a sintaxe no exemplo de código 5.5

Código 5.5: Classe Abstrata

```
class FormaGeometrica
{
  public: FormaGeometrica();
  ~FormaGeometrica();
  virtual void desenha() const = 0;
};
```

5.3.1 Teste de Aprendizado

1. Qual a vantagem de usar uma classe abstrata?

2. Modifique a classe Empregado gerada anteriormente para ser uma classe abstrata.

Capítulo 6

Templates e Exceções

Templates é um dos recursos mais poderosos da linguagem C++. E será este o foco do estudo nesta parte do capítulo. O conceito consiste, basicamente, em escrever um modelo de código a partir do qual o compilador será capaz de produzir uma série de código automaticamente. Existem dois tipos de templates diferentes: de função e de classe.

6.1 Templates para Funções

Através de templates de função, é possível especificar, com apenas um segmento de código, uma série de funções correlacionadas, que na verdade são sobrecarregadas. Na verdade, o template é um molde para funções, contudo não se especifica um tipo para a mesma (o tipo é deixado genericamente). Quando ela é chamada, o compilador analisa o tipo chamado e escreve pelo usuário a função como deveria ser com o tipo especificado.

Parece mágica, mas é tão simples quanto parece. Imagine uma função que soma dois inteiros. Facilmente implementável, com certeza. Agora, é preciso somar dois floats: sem dúvida uma função semelhante. Assim, Somar dois doubles não deve apresentar maiores dificuldades. E somar dois vetores (tipo criado pelo usuário)...

Todas essas são tarefas semelhantes, que exigem a escrita de código duplicado. Utilizando templates é possível escrever uma única função que some o que você quiser. Deve ser observado que, para tipos criados pelo usuário, os operadores utilizados devem ter sido corretamente sobrecarregados.

6.1.1 Sintaxe

A sintaxe para uso de funções template é trivial. Comece pela palavra-chave template seguido da lista de tipos definidos pelo usuário entre os sinais de menor e maior. No código 6.1, a declaração normal da função, referindo ao tipo genérico especificado quando necessário.

6.1.2 Exemplo

Código 6.1: Template de Funções

```
#include <iostream>
using namespace std;
template <class T>

T somaDois(T x, T y)
{
  return (x + y);
}

int main()
{
  cout << somaDois(3, 5) << endl;
  cout << somaDois(3.555, 5.333) << endl;
  cout << somaDois('a', 'b') << endl;
  return 0;
}
```

Exercício:

1. Escreva um template de função que ordene quaisquer tipos. Teste com os tipos primitivos int e float.

2. Agora crie uma classe 'Funcionario' bem simples que tenha um atributo 'int idade'. E teste sua função template para essa classe ordenando os funcionários por idade (para funcionar é provável que necessite sobrecarregar alguns operadores).

6.2 Template de Classe

Um template de classe é extremamente semelhante a um template de função, contudo, agora as classes que são genéricas. Este recurso é muito bem utilizado em classes contêiner, como uma pilha (LIFO – Last In First Out) ou uma fila (FIFO – First In First Out). Dessa forma, é possível especificar uma classe contêiner genérica que pode vir a armazenar inteiros, strings, ou mesmo funcionários.

6.2.1 Sintaxe

A sintaxe é bem semelhante à vista anteriormente, a palavra-chave template seguida dos tipos genéricos entre os sinais de menor e maior vem antes da declaração da classe. Observe o código 6.2.

6.2.2 Exemplo

6.3 Introdução a Exceções

Agora introduziremos os fundamentos do tratamento de exceções. Como o nome diz, exceções são situações fora do comum nos programas. Em geral erros que o programa não consegue resolver, terminando então a execução. Por exemplo, uma divisão por zero é um erro que fatalmente irá derrubar o programa. Agora, imagine que você se depara com uma divisão, cujo denominador eventualmente possa vir a ser zero.

Ao invés de deixar esta possibilidade de erro em aberto, seria muito melhor detectar o erro e tratá-lo adequadamente ou, ao menos, indicá-lo para que outros resolvam. Pois é disso que se trata basicamente o capítulo: detectar exceções e tomar uma providência em relação a elas. Deve ficar claro que o tratamento de exceções é específico para quando o erro não pode ser tratado localmente e deve ser reportado. Caso contrário, o erro deve ser tratado localmente sem a necessidade de um tratamento de exceções.

Portanto, ao final do estudo deve ser entendido que o tratamento de exceções de C++ foi projetado para hierarquizar o tratamento de erros e tratar casos em que a função que descobre um erro está impossibilitada de tratá-lo (ou não deseja fazê-lo para não misturar código de tratamento de erros dentro da sua lógica). Contudo no curso apresentaremos erros que poderiam ser tratados localmente por motivos didáticos.

6.4 Try, Throw e Catch

Os três comandos básicos para realizar o tratamento de exceções são try, throw e catch. throw – este é o comando que lança uma exceção. Imagine estar desenvolvendo uma parte de um sistema, e percebe a possibilidade de um erro, mas não pode tratá-lo neste ponto. Ao invés de ignorar, o correto é, ao menos, avisar um possível usuário quando der um erro. Para isso que serve o comando throw, lançar um erro para que seja tratado em outro ponto, mesmo sem garantia de que este erro será efetivamente tratado em outro lugar.

try este é o comando que inicia um bloco que pode gerar erro. Todo comando contido neste bloco é verificado antes da execução para assegurar que um erro não cause o término do programa. Caso exista algum comando neste bloco que lança uma exceção através do comando throw (visto acima), isto será detectado para, em seguida, poder ser tratado, o que neste caso significa passar o fluxo de execução para o bloco do comando catch.

catch este é o comando que realmente trata exceções. Caso algum erro tenha sido encontrado no bloco try correspondente, será procurado um bloco catch subseqüente que possa tratar a exceção encontrada. É dentro desse bloco que serão tomadas as devidas providências em relação ao erro.

Código 6.2: Template de Classe

```
template < class T > class Camaleao
{
  public:

  Camaleao();
  Camaleao(T novaVariavel);
  ~Camaleao();
  T getVariavel();
  void setVariavel(T novaVariavel);

  private:
  T variavel;

};

template <class T>
Camaleao<T>::Camaleao() {}

template <class T>
Camaleao<T>::Camaleao(T novaVariavel) { variavel = novaVariavel;
    }

template <class T>
Camaleao<T>::~Camaleao() { }

template <class T>
T Camaleao<T>::getVariavel() { return variavel; }

template <class T>
void Camaleao<T>::setVariavel(T novaVariavel) {
  variavel = novaVariavel;
}

//trecho de camaleao.cpp

#include "Camaleao.h"
#include <iostream>

using namespace std;

int main()
{
  Camaleao<int> meuInteiro(7);
  cout << "meu inteiro: " << meuInteiro.getVariavel() << endl;
  return 0;
}
```

Deve ficar claro que o tratamento de exceções não faz milagre. É bastante difícil solucionar problemas da forma desejada através deste mecanismo. Na verdade, ele é muito útil para terminar o programa 'elegantemente' quando ocorre uma emergência. É bastante desagradável para o usuário ver o programa cair, ou fazer coisas que ele realmente não deveria estar fazendo.

Então, se utiliza bastante tratamento de exceções para avisar o que aconteceu ("Olha, estou fechando porque ocorreu um estouro de memória... desculpe!"), e em seguida realizar alguns procedimentos antes de terminar o programa, como, por exemplo, liberar a memória alocada. Sem dúvida alguma, é bem melhor fazer isto, em vez de deixar o programa executando de modo imprevisível.

6.4.1 Exemplo

Os códigos 6.3 e 6.4 exemplificam o uso.

Código 6.3: Exceções

```
#include <iostream>
using namespace std;
// classe para o tratamento de uma excecao de divisao por zero

class DivideByZeroException
{
  public:
  DivideByZeroException() : message(" voce tentou dividir por
      zero ") { }
  const char* what() const
  { return message; }
  private:
  const char* message;
}; // funcao que lanca a excecao

double quotient( int numerador, int denominador )
{
  // momento em que a  e x c e  eh lancada!!!
  if (denominador == 0) throw DivideByZeroException();
  return (double)numerador / denominador;
}
```

Note que o comando throw chama o construtor da classe DivideByZeroException criando, dessa forma, um objeto dessa classe. Este objeto será lançado pelo comando throw, e quando o erro for detectado pelo bloco try o comando catch capturará este objeto e irá tratar adequadamente a exceção. O comando throw pode lançar objetos, ou mesmo tipos primitivos. E será este tipo que definirá qual bloco catch tratará da exceção em questão.

Código 6.4: Exceções funcionando

```
#include "excecoes.cpp"

// programa teste para tratar uma excecao
int main()
{
  int x, y;
  double resultado;
  cout << "Digite dois numeros para serem divididos (CTRL + Z
      para acabar): ";
  while (cin >> x >> y)
  {
    try
    {
         resultado = quotient(x, y);
         cout << "O quociente eh " << resultado << endl;
    }
    catch (DivideByZeroException ex)
    {
      cout << "Ocorreu uma excecao: " << ex.what();
      cout << endl;
    } cout << "Digite dois numeros para serem divididos (CTRL +
        Z para acabar): "; }
        return 0;
}
```

```
catch (int x)
{ ... }

catch (double y)
{ ... }
```

Estes são blocos sintaticamente corretos, contudo, costuma-se escolher objetos, pois estes podem transportar alguma informação útil para o tratamento da exceção. Caso se deseje capturar todas as exceções, de qualquer tipo, utiliza-se reticências entre os parênteses que sucedem o catch:

```
catch(...) {}
```

Se nenhum bloco catch for capaz de capturar uma determinada exceção, o programa é abortado. Caso contrário, ele continua com a primeira instrução após o último bloco catch (logo, qualquer parte do bloco try que não tenha sido executada antes do lançamento da exceção é ignorada).

Cabe ainda ressaltar que os efeitos colaterais do polimorfismo valem aqui também. Caso o tipo de alguma exceção seja uma classe mãe, qualquer classe derivada desta será capturada pelo respectivo bloco catch da classe mãe.

6.4.2 Exercício

1) Crie uma situação semelhante ao exemplo em que ocorra um erro por estouro (overflow). Utilize variáveis que estouram mais comumente, como short int ou byte.

Capítulo 7

Processamento de Arquivos

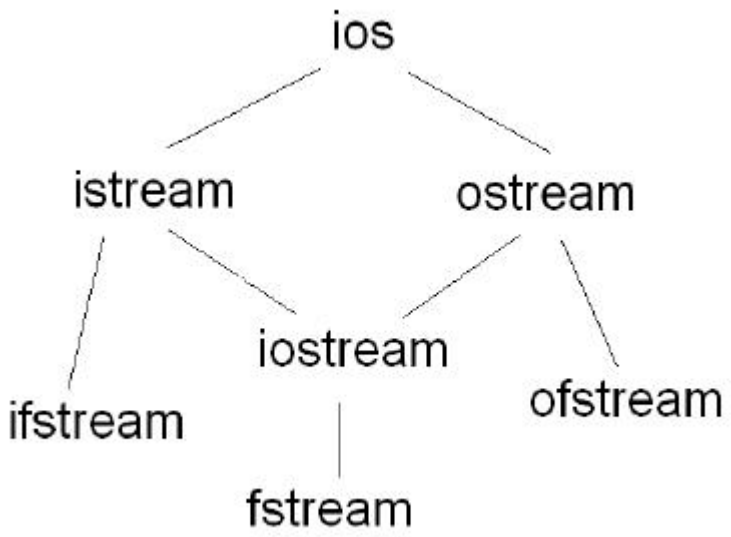
As variáveis de programas são temporárias, isto é, serão armazenadas na memória do seu computador e, assim que seu programa finalizar, elas serão perdidas. Contudo, existem diversas situações em que pode ser bem interessante guardar algumas informações para a próxima vez que o programa executar (imagine não poder salvar seu jogo no meio e continuá-lo depois), ou mesmo para utilizar estas informações de alguma outra forma.

Para fazer isto utilizamos arquivos, que são armazenados em dispositivos secundários de armazenamento, como o disco rígido do seu computador, e, desta forma, torna-se bem mais difícil perder os dados. E neste capítulo examinaremos a manipulação de arquivos em C++.

7.0.3 Arquivos em C++

Para realizar o processamento de arquivos em C++, devem ser incluídos os arquivos de cabeçalho <iostream> e <fstream>. O cabeçalho <iostream> inclui objetos e estruturas que são utilizados na entrada e saída padrão de dados, mas que também são válidos para manipular arquivos.

Já o cabeçalho <fstream> inclui as definições das classes ifstream (para realizar a entrada de um arquivo), ofstream (para realizar a saída para um arquivo) e fstream (para entrada e saída de um arquivo). Todas essas classes herdam das classes definidas em <iostream> de acordo com o esquema abaixo.



7.0.4 Arquivo Seqüencial

Como o nome já diz, um arquivo seqüencial é simplesmente um arquivo de acesso seqüencial, isto é, para acessar um dado no fim do arquivo é preciso percorrer todo o arquivo antes. Para criar um arquivo seqüencial, é preciso criar um objeto 'ofstream' chamando o seu construtor com dois parâmetros: o nome do arquivo e o modo como este aquivo será aberto.

7.0.5 Modos de Abertura

ios::app Grava toda saída no fim do arquivo.

ios::ate Abre um arquivo para saída e avança até o fim do arquivo.

ios::in Abre um arquivo para entrada.

ios::out Abre um arquivo para saída.

ios::trunc Abre um novo arquivo, eliminando antigo se este já existir.

ios::binary Abre um arquivo para entrada ou saída binária.

Em seguida, a entrada para o arquivo é idêntica a entrada padrão de dados. Observe o código 7.1 de exemplo, que realiza um cadastro de filmes com os seguintes campos: Código, Nome e Nota.

Agora, quando criamos um arquivo seqüencial, é bem provável que queiramos abri-lo em uma próxima oportunidade. Para realizar isso, o procedimento é análogo ao de criação de um arquivo seqüencial. Primeiro, deve-se criar um objeto ifstream chamando o seu construtor com os seus parâmetros: nome do arquivo

Código 7.1: Escrevendo em Arquivos

```
#include <cstdlib>
#include <iostream>
#include <ofstream>

int main()
{
  // construtor de ofstream abre o arquivo
  ofstream file( "Movie.dat", ios::out );

  if ( !file )
  // operador ! sobrecarregado
  {
        cerr << "Arquivo nao encontrado..." << endl;
        exit( 1 );
  }

  cout << "Digite o codigo do filme, o nome e a nota.\n" << endl
      ";

  int cod;
  char nome[ 50 ];
  int nota;

  while ( cin >> cod >> nome >> nota )
  {
        file << cod << ' ' << nome << ' ' << nota << '\n'; cout
           << "? ";
  }

  return 0;
}
```

e modo de abertura do arquivo. Em seguida, basta ler os valores do arquivo do mesmo modo que se faz com a entrada de dados padrão, trocando cin pelo objeto criado.

'objeto criado' » 'variáveis que armazenarão os valores lidos do arquivo'

Exercício:

Primeiramente, utilize o código dado para criar um arquivo seqüencial de filmes. Em seguida, faça um programa que leia deste arquivo seqüencial e escreva na tela apenas o nome dos filmes com nota acima de 6.

7.1 Arquivo Aleatório

Um arquivo seqüencial resolve muitas vezes o nosso problema, contudo, apresenta um grave problema de tempo: seu mecanismo é pouco eficiente. Imagine-se na frente de um caixa eletrônico, querendo realizar uma retirada de dinheiro, e o sistema verificando seqüencialmente num arquivo de todas as contas do banco, se sua conta possui o saldo suficiente. É bem provável que você desista de retirar o dinheiro. Para resolver este tipo de problema, utilizam-se arquivos de acesso aleatório.

A idéia básica consiste em saber de antemão onde se encontra a informação desejada, e buscá-la diretamente no local apropriado. Isto pode ser feito utilizando-se estruturas de tamanho fixo. Dessa forma, é possível calcular de forma rápida que o registro de chave 5 iniciará na posição 5 vezes o tamanho da estrutura (considerando iniciar na chave 0).

Implementar este tipo de arquivo em C++ também não impõe muitas dificuldades. Os próprios objetos ostream e istream possuem métodos write e read, respectivamente, nos quais é possível especificar um offset para ir direto para determinado ponto do arquivo. Entretanto, foge um pouco do escopo do curso detalhar todas as técnicas e métodos de manipulação de arquivos.

Capítulo 8

Standard Template Library

A STL é uma parte extremamente importante da biblioteca padrão do C++. Na prática, programadores passam a maior parte do tempo programando somente com a STL, usando pouco as outras facilidades oferecidas pela biblioteca padrão completa. Na STL, estão declaradas estruturas de dados muito versáteis e diversos algoritmos úteis, de forma a facilitar a programação.

O uso dessa biblioteca evita que seja necessário, por exemplo, implementar e depurar listas encadeadas.Além disso, houve uma grande preocupação com o desempenho das classes e funções, de modo que dificilmente uma implementação própria pode oferecer vantagens consistentes de performance, sem falar nas facilidades que a STL provê.

8.1 Conhecendo a STL

Pode-se dividir a STL em três tipos de componentes: containers, iteradores e algoritmos. Containers são estruturas que contém objetos de outras classes, como por exemplo listas, pilhas, mapas, árvores, vetores, etc.. Para que se entenda como funcionam essas classes, é extremamente necessário que se esteja familiarizado com o conceito de templates.

Iteradores são, de certa forma, ponteiros. Na realidade eles não são ponteiros, porque eles não apontam simplesmente para uma palavra na memória, e existem operações que podem ser feitas sobre ponteiros que não podem ser feitas sobre iteradores, como aritmética de ponteiros, por exemplo. Entretanto, mesmo com estas diferenças, iteradores têm um uso semelhante a ponteiros: eles fornecem uma interface de acesso a elementos internos de containers. Algoritmos são, como o nome diz, algoritmos prontos que a STL fornece. Eles permitem, por exemplo, ordenar containers, obter elementos máximos de conjuntos, e outras coisas interessantes. Com algoritmos STL, pode-se acumular todos os valores de um vetor, essencialmente resolvendo um problema de integração numérica em uma linha de código.

Pode-se também aplicar uma função arbitrária a cada elemento de um con-

Contêiner	Categoria	Descrição
Vector	Sequencial	Inserções/Retiradas rápidas, apenas no fim. Acesso direto a qualquer elemento.
List	Sequencial	Inserções/Retiradas rápidas.
Deque	Sequencial	Inserções/Retiradas rápidas, no início e no fim. Acesso direto a qualquer elemento.
String	Sequencial	Vetor de caracters típico.
Set	Associativo	Pesquisa rápida. Duplicatas não permitidas.
Multiset	Associativo	Pesquisa rápida. Duplicatas permitidas.
Map	Associativo	Mapeamento um para um. Pesquisa rápida usando chaves. Duplicatas não permitidas.
Multimap	Associativo	Mapeamento um para um. Pesquisa rápida usando chaves. Duplicatas permitidas.
Stack	Adaptador	Primeiro a entrar, último a sair (FILO)
Queue	Adaptador	Primeiro a entrar, primeiro a sair (FIFO)
Priority_Queue	Adaptador	Mais alta prioridade, primeiro a sair

junto, também com uma linha de código. Uma característica interessante dos algoritmos STL é que eles são genéricos. Por exemplo, o algoritmo sort, que implementa ordenação de listas, é versátil o suficiente para ordenar quase qualquer vetor de elementos arbitrários. Este "quase" deve ser levado em consideração. Para que o sort funcione, a classe dos elementos internos ao vetor deve prover um operador $<$ bem definido. Além disso, não existem restrições, o que implica que o sort pode ordenar elementos de qualquer classe, desde que o predicado de ordem seja especificado para esta classe!

8.2 Contêiners

Contêiners são inseridos, de acordo com suas características, em três categorias: seqüenciais, associativos e adaptadores de containers. A justificativa para essa separação provém do uso de cada um dos grupos de containers. Abaixo, uma lista com os principais containers, alocados em seus respectivos grupos, seguidos de uma descrição breve de sua utilização.

8.3 Funções Membro Comuns

Existem alguns membros básicos que são comuns a todos os containers da STL. Estes membros permitem fazer checagens simples, como obter o número de elementos em um container, testar se este está vazio, e etc..

A lista abaixo mostra os métodos comuns mais importantes:

Contêiner	#include
Vector	<vector>
List	<list>
Deque	<deque>
String	<string>
Set e Multiset	<set>
Map e Multimap	<map>
Stack	<stack>
Queue e Priority Queue	<queue>

::size() retorna um size_ type (convertível para unsigned int) que representa o número de elementos do container

::empty() retorna um valor booleano dizendo se o container está vazio

::begin() retorna um iterador para o início do container

::end() retorna um iterador para um elemento além do final do container

::rbegin() retorna um iterador reverso para o início reverso do container (ou seja, o último elemento do container)

::rend() retorna um iterador reverso para um elemento antes do início do container

Além destes, os operadores típicos de comparação != e == também estão implementados para contêiners, bem como construtores de cópia.

8.4 Funções Membro Específicas

Obviamente, existem algumas funcionalidades específicas de cada container, que precisam ser implementadas com métodos não-genéricos. Um exemplo disso é a função clear(), que apaga todos os valores de um container, e só é função-membro dos containers seqüenciais ou associativos (também conhecidos como containers de primeira classe).

8.4.1 Arquivos de Cabeçalho

Para utilizar os containers da STL, é necessáro incluir os arquivos de cabeçalho daquele container no programa. Abaixo, os arquivos necessários para cada um dos containers discutido acima.

8.5 Vector

Um dos containers mais simples e utilizados da biblioteca de templates do C++ é a classe Vector. Apesar de ser fundamentalmente diferente de arrays em C, o vector serve para muitos dos mesmos fins. Uma das principais vantagens de se utilizar vector sobre arrays pré-determinados é que um vector não tem limite de tamanho (estritamente falando, um vector tem limite de tamanho máximo, mas este valor é limitado por uma constante normalmente desprezivelmente grande, definida pela quantidade de memória disponível na máquina. Este valor pode ser obtido com o método ::max _ _ size())

A maioria dos conceitos relativos aos objetos Vector já é bastante consolidada em programadores C, de modo que essa apostila mostrará, principalmente, a sintaxe e diferenças básicas entre o uso de arrays e de vetores da STL. Comecemos com um exemplo básico, que utiliza vectors apenas para armazenar e mostrar dados. A classe Aluno deve receber um número indefinido de notas, armazenar em um vector, e imprimir um histórico escolar, a partir desses dados. Sua implementação está em 8.1, 8.2.

Código 8.1: Header da classe Aluno

```
// trecho de aluno.h
#include <vector>
#include <iostream>
using std::vector;
using std::cout;
using std::endl;

class Aluno
{
  public:
  // Construtor
  Aluno();
  // Destrutor
  ~Aluno();
  // Adicionar nota
  void adicionarNota(int nota1);
  // Imprimir historico
  void imprimirHistorico();

  private:
  vector<int> notas;
};
```

8.5.1 Teste de Aprendizado

- Adicione na classe Aluno uma função calculaMedia().
- Troque o uso de push_back() pelo simples notas[n] = nota1.

Código 8.2: Classe usando um vetor

```
// trecho de aluno.cpp
void Aluno::adicionarNota(int nota1)
{
  notas.push_back(nota1);
}

void Aluno::imprimirHistorico()
{
  for(int i = 0; i < notas.size(); ++i)
  cout << "Nota: " << notas[i] << endl;
}
```

O que acontece?

Obs.: Utilize vector::size() para pegar o número de notas já adicionadas, e, conseqüentemente, o índice da próxima variável a ser adicionada.

8.6 Como lidar com capacidade ilimitada, ou como o Vector não é mágico

À primeira vista, vectors parecem desafiar as leis básicas da computação. Eles oferecem sempre o melhor de dois mundos. Temos garantias teóricas de que vectors fornecem um tamanho ilimitado de elementos, sabemos que estes elementos estão contíguos na memória, e sabemos que o acesso a qualquer elemento tem complexidade constante, bem como a inserção no final e a remoção do final. Não é nem um pouco trivial desenvolver uma estrutura de dados com todas estas capacidades ao mesmo tempo. De fato, se levarmos estas definições ao pé da letra, tal estrutura de dados é impossível.

O que a STL "esconde" aqui é que, na realidade, a inserção não é feita em tempo constante, mas sim em tempo constante amortizado. O funcionamento desta inserção é bastante simples: Quando inicializamos um vector, a STL reserva um espaço de memória contíguo de tamanho, digamos, n, para alguns poucos elementos, e mantém este espaço disponível para nosso vector.

Conforme vamos populando ele com elementos, este espaço vai sendo ocupado e, quando queremos inserir o n+1-ésimo elemento, faltaria espaço contíguo. Quando isto acontece, a STL procura outro local de memória contíguo, desta vez com 2n espaços, e move todo o conteúdo do vector para lá. Como o tamanho da nova área de memória aumenta exponencialmente, a probabilidade de que, dado um push_back() qualquer, este precise uma realocação, tende a 0 conforme fazemos mais e mais push_backs(). Por isso que chamamos a complexidade de inserção de constante amortizada.

É interessante notar que esta realocação de memória pode invalidar ponteiros sobre o container, e é por isso que ponteiros sobre containers não devem ser

usados. Este processo de alocação amortizada justifica um método bastante útil da classe vector, que é o método reserve(). Este método serve para que possamos determinar um tamanho para o espaço de memória inicial que o vector deve ocupar.

Por exemplo, se soubermos que nosso vector irá ocupar "aproximadamente" 100 elementos, podemos fazer uma chamada a reserve(100), que irá reservar um espaço de memória contígua de 100 elementos para nosso vector. Se nos garantirmos que a inserção irá só até o centésimo elemento, aí sim esta terá complexidade constante. Note, porém, que o método reserve() não popula o vector com nenhum elemento, de forma que esta chamada não altera o resultado de size() e empty()!

O exemplo 8.3 mostra outro detalhe que deve ser observado quando se trata de containers de tamanho arbitrário: apesar de que podemos, em princípio, inserir quantos elementos quisermos dentro de um vector, só podemos acessar aqueles que já foram inseridos. Isto funciona da mesma forma que arrays em C tradicional, ou seja, se um array tem 5 elementos, o acesso ao elemento 6 (inclusive ao 5) gerará comportamento indefinido. É importante observar que isto sumariza a diferença entre infinitos elementos e número arbitrário de elementos. Um vector deve ser pensado exatamente como um array C que pode ter seu tamanho modificado em tempo de execução.

Código 8.3: Demonstração de Vetores

```
#include <iostream>
#include <vector>

using std::vector;
using std::cin;

int main(int argc, char **argv)
{
  vector<int> numeros;
  for(int i = 0; i < 4; ++i)
  {
    numeros.push_back(i);
  }
  vector<int> numeros2;
  numeros2.reserve(3);
  for(int i = 0; i < 4; ++i)
  {
    numeros2[i] = i;
  }

  numeros2[5000] = 5;

  return 0;
}
```

8.6.1 Observação

Ao testar no Dev-C++, que usa o gcc 3.4.2, o código acima só gerou erro ao acessar o elemento 5000. Porém, ao executar compilando com o g++ no Linux, um acesso ao item de índice 4 é suficiente para gerar um erro de execução. Dessa forma, esse tipo de acesso gera comportamento indefinido, que é suficiente para que seu uso não seja recomendado.

8.7 List

Além dos vetores, uma estrutura muito utilizada em programação é a lista encadeada. Observe que o nome list não quer dizer que é uma lista simplesmente encadeada. De fato, este template STL é uma lista duplamente encadeada (para uma lista especificamente simplesmente encadeada, use a estrutura não-padrão slist).

Listas são recomendadas para tarefas onde é necessário muita performance para inserções no início ou no final, e consulta no início ou no final. De fato, estas operações têm complexidade O(1). Operações arbitrárias no meio da lista, entretanto, como inserção, remoção e consulta, têm complexidade O(n). Uma vantagem das listas para os vectors, por outro lado, é que o tempo constante de listas de fato é constante, e não constante amortizado, pois a estrutura de dados subjacente a uma lista é um encadeamento de ponteiros.

Quando começamos a trabalhar com listas, a utilização de iteradores passa a ser necessária. Enquanto, para varrer um vector, é possível utilizar o operador [], da mesma forma que se varre arrays em C, quando queremos varrer o conteúdo de uma lista, somos obrigados a utilizar iteradores. O exemplo 8.4 mostra o conceito de iteradores em uma varredura linear simples de uma lista:

Inicialmente, a sintaxe destes comandos pode parecer complicada. Lembre-se, entretanto, de exatamente como funciona o comando for: O primeiro comando passado dentro dos parênteses é executado exatamente uma vez antes do laço ser processado. Neste caso, estamos declarando uma variável, chamada pos, de tipo list<int>::iterator, e setando esta variável ao valor especial foo.begin(), que aponta para o primeiro elemento da lista. O segundo comando no parêntese do for é um teste que será executado antes de cada passada do for, inclusive da primeira, e que se falhar, o for será terminado.

Aqui, estamos testando se o valor do iterador é igual ao valor fixo foo.end(), que aponta para um elemento além do último elemento da lista, de forma que este for varre a lista inteira. Finalmente, o último comando no parêntese do for chama o operador++ sobre o iterador, que está definido internamente e significa que o iterador deve receber o valor de seu sucessor imediato. Note que não existe o operador– para iteradores. Para varrer uma lista em ordem reversa, reverse_iterators devem ser utilizados. Veremos iteradores em mais detalhes na seção apropriada.

Código 8.4: Iterando sobre uma lista

```
#include <list>
#include <iostream>
using std::list;
using std::cout;
using std::cin;

int main()
{
  list<int> foo;
  for (list<int>::iterator pos = foo.begin(); // note que este
      for
    pos != foo.end(); ++pos){                 // esta separado em
        duas linhas!

  cout << (*pos) << endl;
  }

  return 0;
}
```

8.8 Outros Containers

A seguir, faremos um breve resumo sobre os outros containers mais específicos da STL, e ainda sobre um tipo de dados específico, o pair.

Deque Um deque é funcionalmente semelhante a um vector. Observe que deque significa double-ended queue, mas isto não quer dizer que um deque seja implementado como uma lista encadeada. De fato, deques são implementados como espaços contíguos na memória, de forma que inserções no meio levam tempo linear, mas acesso ao meio leva tempo constante. A principal diferença entre um vector e um deque é que o deque possui os métodos push_front e pop_front, que fazem papéis análogos ao push_back e pop_back, mas atuam na frente da lista. Da mesma forma que os vectors, estes métodos são implementados com uma técnica de tempo constante amortizado.

String Strings implementam um vetor de caracteres, para possibilitar formas de entrada e saída mais alto-nível do que null-terminated chars em C. Uma string possui complexidades semelhantes ao vector, de forma que também é mantida como área contígua de memória, mas o tipo de seus elementos é char. Existe uma variante interessante de strings, que são as wstrings, que permitem trabalhar com chars de 16 bits. Normalmente isto acontece quando a interface com algum outro programa exige a utilização de wide characters, ou seja, caracteres de 16 bits

Set Um set é, primeiramente, um container associativo. Isto significa que

acesso randômico aos seus elementos não é possível. Entretanto, inserção e remoção são executados em tempo O(log n). É importante notar que para containers associativos, não é possível especificar o local onde o elemento será adicionado, pois estes containers são implementados como árvores binárias balanceadas. Isto significa, adicionalmente, que os elementos são automaticamente guardados em ordem dentro do container. Observe que sets não permitem elementos duplicados. As funções únicas mais importantes de sets são insert, remove e find, que inserem, removem, e localizam, respectivamente, um elemento no set.

Multiset Muito semelhante ao set, a principal diferença entre eles é que enquanto o set não permite colocação de elementos duplicados, um multiset permite.

Map Um Map é um container mais interessante, no sentido de que ele permite buscas do tipo chave-informação. Dessa forma, os elementos de um map são pares (std::pair) onde o primeiro elemento representa a chave de busca e o segundo o valor desejado. O map é implementado também como uma árvore binária balanceada onde a ordenação é feita com respeito às chaves. Dessa forma, é possível buscar um valor informando somente uma chave, com complexidade O(log n). Não permitem duplicatas

Multimap Multimaps são análogos a multisets, no sentido de que são maps que permitem multiplicidade de elementos.

Modificadores de containeres Stacks, Queues e Priority Queues não são containeres propriamente ditos, mas sim modificadores que atuam sobre containeres. Dessa forma, para especificarmos uma stack, por exemplo, primeiramente temos que definir um container normal (um vector, por exemplo), e depois construir a stack informando este vector. O que acontece é que os métodos acessados pelo objeto stack serão limitados a acesso e escrita ao primeiro elemento da pilha, como desejamos. O mesmo acontece para queues e priority queues.

Pair Um Pair não é um container. Ao invés disso, um pair é uma estrutura de dados declarada pela STL que permite formarmos pares de tipos genéricos. Para especificar um par inteiro-string, por exemplo, devemos fazer o seguinte: pair<int,string> foo. Os tipos de dados que podem formar pares são arbitrários, e para acessarmos elementos do pair, existem os atributos .first e .second.

8.9 Iteradores

A definição mais simples de iteradores é razoavelmente óbvia: iteradores iteram sobre o conteúdo de containers STL. Ou seja, iteradores são tipos de dados

específicos que a STL implementa para possibilitar uma forma de acesso uniforme a elementos de qualquer tipo de container. A idéia de interface uniforme é muito importante aqui: se fôssemos implementar um conjunto de containers diferentes a partir do zero, teríamos, possivelmente, classes para listas, filas, arrays, árvores binárias, e etc.

Cada uma de nossas classes possuiria um método diferente para acessar os elementos de dentro dos nossos containers, e o programador que fosse utilizá-los teria que aprender cada um individualmente. O conceito de iteradores da STL resolve este conflito, pois fornece uma única forma de acesso uniforme a elementos de qualquer container.

De certa forma, iteradores podem ser pensados como ponteiros para os elementos dos containers. Esta analogia deve ser tomada com muito cuidado, porém, pois existem coisas que podem ser feitas com ponteiros (apesar de que não são necessariamente recomendadas) que a STL não fornece. O exemplo mais clássico de uma aplicação assim é aritmética de ponteiros.

Já vimos, na seção que trata com listas, um exemplo de declaração de iteradores. Por isso, vamos mostrar um exemplo que utiliza iteradores sobre sets. Fica bastante claro que a estrutura do comando é exatamente igual. A única coisa que muda é o tipo do iterador, que passa a ser do tipo set<int>::iterator.

Código 8.5: Demonstrando Iteradores

```
#include <set>
#include <iostream>

using std::list;
using std::cout;
using std::cin;

int main()
{
  set<int> foo;
  for (list<int>::iterator pos = foo.begin(); pos != foo.end();
      ++pos)
  {
        cout << (*pos) << endl;
  }
  return 0;
}
```

No exemplo 8.5, podemos notar alguns detalhes interessantes: em primeiro lugar, veja que a STL também segue, de certa forma, a analogia de que iteradores são ponteiros. Isto quer dizer que o operator* está definido para iteradores, e tem a mesma semântica que ele tem para ponteiros (ele retorna o valor apontado pelo ponteiro, ou, neste caso, pelo iterador). Além disso, o operador++ também está definido, e ele serve para passar de um iterador para o próximo. Observe que, mesmo que sets não sejam lineares (lembre que são

árvores binárias), o operador++ está bem definido. Finalmente, devemos observar o significado dos comandos foo.begin() e foo.end() no exemplo acima. foo.begin() é um iterador específico que aponta para o primeiro elemento do container trabalhado. O foo.end(), entretanto, não aponta para o último elemento, e sim para um elemento além do último, seguindo de certa forma o conceito de que índices em C variam de 0 até n-1, e não até n. Abaixo, mostramos os tipos principais de iteradores e suas utilidades:

::iterator Este tipo de iterador declara os iteradores mais comuns, que acessam elementos do container de forma seqüencial da esquerda para a direita e que podem modificar os elementos apontados por eles

::reverse_iterator Este tipo de iterador serve para varrer containers de forma reversa. Como o operador– não está declarado para iteradores normais, não seria possível iterar reversamente por um container sem este iterador especial. Observe que o operador++ está definido para este tipo de iterador da mesma forma que para ::iterator's, mas para ::reverse_iterator's a varredura é feita em ordem reversa!

::const_iterator Semelhante ao ::iterator, mas com a restrição de que os elementos só podem ser acessados, e não modificados, utilizando este iterador. O uso deste iterador existe para podermos varrer containers que foram declarados const. Se tentarmos varrer um const vector<int> com um ::iteretor normal, encontraremos um erro de compilação.

::const_reverse_iterator Análogo ao ::const_iterator, mas se aplica a varredura reversa do container. Além disso, cada container tem definido quatro iteradores especiais que servem para limitar laços for, entre outros usos.

Iremos assumir, para os exemplos abaixo, a existência de um vector<int> chamado foo, mas de fato estes iteradores estão definidos para qualquer container de qualquer tipo.

foo.begin() este iterador é do tipo ::iterator, e aponta para o primeiro elemento do container

foo.end() iterador do tipo ::iterator, aponta para um elemento além do último elemento do container. Acesso a (*foo.end()) gera comportamento indefinido.

foo.rbegin() iterador do tipo ::reverse_iterator, e aponta para o último elemento do container.

foo.rend() iterador do tipo ::reverse_iterator que aponta para um elemento antes do primeiro elemento do container. Acesso a *(*foo.rend())* gera comportamento indefinido.

8.10 Algoritmos

Algoritmos também são uma parte bastante importante da Standard Template Library, já que implementam funções usadas muito freqüentemente de forma eficiente, facilitando muito o trabalho do programador. Um exemplo de algoritmo extremamente utilizado é o de ordenação. Através desse algoritmo, fica fácil organizar um vetor (ou algum outro tipo de estrutura) usando qualquer tipo de relação de ordem. No exemplo 8.6, usaremos a função sort para ordenar inteiros de forma crescente.

Código 8.6: STL Sort

```
#include <algorithm>

int main()
{
  vector<int> numeros;

  for(int i = 0; i < 6; ++i)
  {
     numeros.push_back(50 - 10*i);
  }
  cout << "Imprimir fora de ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
          cout << numeros[i] << endl;
  }

  std::sort(numeros.begin(), numeros.end());
  cout << "Imprimir em ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }
}
```

O algoritmo sort é normalmente o algoritmo mais utilizado da STL e, portanto, merece alguma atenção especial. Internamente, o padrão STL dita que o algoritmo utilizado para ordenamento é o algoritmo introsort. Basicamente, o introsort ordena os elementos inicialmente utilizando um quicksort, e quando o nível de recursão atinge um determinado limite, o algoritmo passa a executar um heapsort.

A vantagem desta abordagem é que a complexidade deste algoritmo é garantidamente O(n log n). Outro detalhe importante é que o algoritmo utilizado pela STL não é um algoritmo de ordenamento estável. Para um algoritmo estável, utilize a alternativa stable_sort. Abaixo, mostramos alguns algoritmos comuns que a STL implementa, com uma breve descrição de sua utilidade:

count(iterator first, itertor last, const T &value) a função count da STL recebe dois iteradores (semelhante ao sort) e mais um valor, e retorna o número de ocorrências do valor value dentro do intervalo definido pelos dois iteradores

count_if(iterator first, iterator last, predicate pred) esta função é semelhante à função count, mas ela retorna o número de elementos dentro do intervalo definido pelos iteradores que satisfaz à condição imposta pelo predicado pred, que deve ser uma função booleana de um argumento do tipo do container dos iteradores utilizados

max_element(iterator first, iterator last) retorna um iterador para o elemento de valor máximo dentro do intervalo definido por first e last. Esta função utiliza o comparador operator< do tipo do container para fazer as comparações.

min_element(iterator first, iterator last) semelhante a max_element, mas retornando um iterador para o elemento mínimo

max(const T &v1, const T &v2) dados dois valores de um mesmo tipo, como por exemplo max(1,2), retorna o valor máximo entre eles

min(const T &v1, const T &v2) semelhante a max, mas retornando o valor mínimo

8.10.1 Usando o Algoritmo Sort para qualquer Ordenação

O algoritmo sort é extremamente amplo, de modo que não se limita a ordenações crescentes e decrescentes de inteiros. Nessa seção, veremos como utilizar funções próprias para ordenar estruturas através do sort. Para a demonstração, será usado o exemplo 8.7, que coloca os inteiros em ordem decrescente.

8.11 Usando a STL com Classes Próprias

A biblioteca padrão do C++ tem como alicerce, em sua construção, a facilidade de extensão do seu uso. A partir disso, é redundante dizer que as estruturas criadas podem ser utilizadas não só com os tipos básicos da linguagem, mas também com classes criadas pelo programador. No exemplo 8.8, ordenaremos nossos Livros pelo ano de publicação (o maior antes), e em caso de igualdade, pelo título do livro.

Como se pode ver, basta declarar um operador do tipo <, que determina qual será o critério utilizado para definir se um objeto Livro é ou não menor do que o outro. Depois, é só usar a função sort no vetor. O C++ ordena automaticamente o vetor utilizando o critério definido pelo operador.

Código 8.7: Usando Função para Sort

```
#include <algorithm>
// fun
bool antes(int i, int j) { return (i > j); }

int main()
{
  vector<int> numeros;
  for(int i = 0; i < 6; ++i)
  {
    numeros.push_back(10*i);
  }

  cout << "Imprimir fora de ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }

  std::sort(numeros.begin(), numeros.end(), antes);
  cout << "Imprimir em ordem" << endl;
  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }
}
```

Código 8.8: STL com Classes Próprias

```
// trecho de livro.cpp
// Operador <
// (vai determinar se um Livro deve ser menor na ordem ou nao)

bool Livro::operator<(const Livro &book) const
{
  if(anoDePublicacao > book.getAno())
  return true;
  else
  {
   if(anoDePublicacao == book.getAno())
        {
          if(titulo < book.getTitulo())
          return true;
        }
  }
  return false;
};

/*
trecho de main.cpp

std::sort(livros.begin(), livros.end());

Ok! os livros estarao ordenados.
*/
```

8.11.1 Teste de Aprendizado

1. Crie o operador $<$ na classe Aluno, e ordene os alunos pela média das notas

2. Crie uma classe Time, com número de vitórias, empates e derrotas, e organize n times em uma tabela de classificação usando: Maior número de pontos(vitórias*3 + empates), menos número de jogos, maior número de vitórias

8.12 Últimas Considerações

A seguir, falamos sobre algumas considerações importantes para escrever qualquer código que utilize a STL. Estes itens são resumos de alguns capítulos do livro Effective STL, Revised, de Scott Meyers. Seguir estes itens é extremamente importante, para garantir que o programa execute de forma previsível, já que devido à complexidade da STL, se não tomarmos cuidado, o programa pode gerar comportamento indefinido facilmente. O número entre colchetes é a numeração de cada item no livro original.

- Sempre garanta que o objeto usado como tipo de um contâiner possui cópia bem definida[3]: Grande parte da filosofia da STL é baseada em cópias dos objetos que utilizamos. Por exemplo, se declaramos um vector<foo> bar, onde foo é uma classe declarada por nós, quando inserimos um elemento neste vector (com um push_back, por exemplo), a STL não insere exatamente o elemento que queríamos, mas sim uma cópia deste.

  Para tipos de dados básicos, como ints, float, e etc., isto não é importante. Para classes arbitrárias, por outro lado, isso pode ser um problema. Para estas classes, quando uma cópia é feita, o compilador invoca métodos específicos desta classe: o seu construtor de cópia ou seu operador de cópia, também conhecido como operator=. O detalhe aqui é que se nós não implementarmos estes métodos em nossa classe, o compilador assume uma implementação padrão que pode ou não fazer o que queremos. Outro problema é que se nós de fato implementarmos estes métodos, devemos nos garantir de que eles estão corretos, senão as operações feitas sobre contâiners da STL podem gerar comportamento inesperado e dificultar enormemente a depuração do programa.

- Em contâiners, use o membro empty(), ao invés de checar size() $==$ 0[4]; O motivo para esta recomendação é bastante simples. A operação empty() tem complexidade O(1), enquanto o teste de size() $==$ 0 pode, em alguns casos, custar O(n). Para contâiners numerosos, isto se torna um problema.

- Lembre de chamar delete em iteradores antes de apagar o contâiner[7]; Suponha que tenhamos, por exemplo, um vector<int *> foo (um vector de ponteiros para inteiros). Além disso, imagine que para popular este vector, utilizamos um laço for que aloca dinamicamente cada elemento com um operador new e coloca o valor no vector, como o exemplo:

```
vector<int *> foo;
for (int i = 0; i < 5; ++i)
{
  int *x = new int; *x = 3; foo.push_back(x);
}
```

- Cada elemento do vector foo representa uma posição de memória dinâmica alocada pelo Sistema Operacional. Quando apagarmos este vector (seja explicitamente ou seja porque seu escopo local está esgotado), o compilador não chama os deletes correspondentes a esses new's! Isto significa que, ao declarar um código como o acima, nós somos obrigados a fazer um outro laço semelhante, chamando os deletes correspondentes a cada iterador:

  ```
  for (int i = 0; i < 5; ++i){ delete foo[i]; }
  ```

  O exemplo que foi mostrado usa somente vector, mas este raciocínio vale para qualquer container STL.

- Prefira vectors e strings a vetores dinamicamente alocados[13]; Ao utilizar vetores dinamicamente alocados, o programador deve se responsabilizar inteiramente pelo gerenciamento de memória dinâmica feito. Para programas simples, isto pode parecer trivial, mas para um projeto mais complexo, gerenciamento de memória dinâmica é um problema bastante custoso.

  O programador deve garantir, em seu código, que existe exatamente uma chamada delete para cada chamada new correspondente. Se esta chamada delete não estiver presente, haverá com certeza vazamento de memória. Se, por outro lado, a chamada for feita duplicada, o comportamento, de acordo com a definição C++ é, novamente, indefinido. Com estes problemas, não parece fazer sentido adotar vetores dinâmicos ao invés de vectors ou strings, que fazem todo este gerenciamento automaticamente e de forma segura.

- Garanta que operadores para contâiners associativos retornam falso para valores iguais[21]; É bastante comum, como vimos em um exemplo acima, declarar operadores booleanos para poder implementar contâiners associativos (como sets, multisets, etc.) de classes arbitrárias. Dessa forma, implementaremos um operator< dentro de uma classe arbitrária qualquer. Em princípio, a STL permite que o comparador utilizado em um set não seja necessariamente o operator<, mas sim qualquer função que

informarmos, desde que esta satisfaça algumas condições. Assim, é possível, em princípio, estabelecer um set onde o comparador utilizado é um operator$<=$, ao invés de um operator$<$. Isto é uma péssima idéia, e o motivo disto é simplesmente que a STL espera que elementos iguais retornem falso a uma comparação, para poder, entre outras coisas, não inserir duplicatas em sets. Assim, se nosso comparador retornar qualquer coisa diferente de falso para valores iguais, a STL gerará comportamento indefinido.