Introdução à Programação Orientada para Objetos em Linguagem C++

Prof. Jorge Díaz Calle

Laboratório de Modelagem e Simulação Numérica “Juca Costa”
Departamento de Ciências Básicas – ZAB
Faculdade de Zootecnia e Engenharia de Alimentos – FZEA
Universidade de São Paulo – USP

Novembro de 2017

INTRODUÇÃO

O presente material visa fornecer os fundamentos da programação orientada para objetos, utilizando particularmente a linguagem C++. Estas ferramentas permitem desenvolver software de qualidade no desenvolvimento de projetos.

Inicia-se com comentários sobre algumas filosofias de programação, vantagens e desvantagens, a importância de programar tipos abstratos definidos pelo usuário, a característica do encapsulamento, e outros conceitos da programação orientada para objetos.

É desejável o conhecimento de uma linguagem procedural como o C ou o Pascal, então, a apresentação atual da sintaxe e os fundamentos da linguagem C++ estão priorizados nos conceitos da programação orientada para objetos, e nos melhoramentos do C++ respeito ao C. Os participantes recebem no início do treinamento um conjunto de exemplos para analisar, modificar, compilar e executar. As descobertas que o participante fará com esses códigos serão de valor inestimável.

O treinamento não tenta abordar a programação orientada para objetos e o C++ em todos os detalhes, mas dar os conceitos básicos para iniciar o desenvolvimento dos primeiros programas orientados a objetos em C++ do participante.

O presente material aproveita a experiência obtida na elaboração das apostilas elaboradas para os treinamentos em linguagem de programação durante minha passagem no CENAPAD-SP, Centro Nacional de Processamento de Alto Desempenho em São Paulo:

1. Introdução à linguagem C. 1998.
2. Introdução à linguagem C++. Programação Orientada para Objetos. 1999.

UM POUCO DE HISTÓRIA

A programação orientada para objetos não é uma filosofia de programação nova. Os conceitos básicos deste enfoque foram já introduzidos pela linguagem de programação **Simula** desenvolvida na década de 1960 por O. J. Dahl e Kristen Nygaard. O Simula foi desenvolvido para realizar simulações computacionais de processos reais, e nela, a elaboração de módulos é central, e a construção destes módulos baseia-se nos objetos físicos a ser modelados.

O Simula não teve sucesso para desenvolver programas de propósito geral, mas os conceitos que ele utilizou foram aproveitados por várias linguagens posteriores, entre as quais se destacam o Smalltalk e o C++. A **Smalltalk** foi desenvolvida na Xerox PARC (Palo Alto, California) na década de 1970 por uma equipe coordenada por Alan Kay.

O desenvolvimento de **C++** se deu em 1980, nos Laboratórios Bell por Bjarne Strousstrup. A linguagem C++ originalmente foi desenvolvida para solucionar algumas simulações motivadas por eventos, muito rigorosas, para as quais as considerações sobre eficiência impediam o uso de outras linguagens.

Um objetivo chave do projeto C++ era manter a compatibilidade com o C. A idéia era preservar a integridade de milhões de linhas de programas já escritos e depurados, a integridade de muitas bibliotecas C existentes e a utilidade de ferramentas C já desenvolvidas.

Devido ao alto grau de sucesso na obtenção desse objetivo, a transição do C para o C++ é muito simples. A melhoria mais significativa da linguagem C++ é seu suporte à programação orientada para objetos (abreviadamente OOP). Neste caso, muda-se a abordagem na solução de problemas para aproveitar-se de todos os benefícios de C++, isto é, utiliza-se outra filosofia de programação.

O Smalltalk e o C++ são muito utilizados atualmente, embora eles tenham dois diferentes enfoques para realizar a orientação para objetos. O Smalltalk é considerado como a linguagem implementada na metodologia da orientação para objetos, cada entidade nela é implementada como um objeto; enquanto o C++ é híbrido, no sentido que nele co-existem características de uma linguagem convencional (o C) e características da orientação para objetos.

Comentários sobre Filosofias de programação

Programação procedural

Quando é necessário repetir várias vezes uma seqüência de instruções, no desenvolvimento de um programa, utiliza-se a idéia de procedimento ou função para encapsular esta seqüência. A cada vez que for preciso executar essa seqüência de instruções, é inserida apenas uma linha de código chamando à função ou procedimento desejado. Assim, o fluxo de controle é passado para o procedimento e quando finalizado o fluxo continua exatamente na linha de código seguinte a chamada à função.

Utilizando procedimentos desenvolve-se programas melhor estruturados, mais legíveis e fáceis de depurar. Quando um erro é acusado no código não é preciso verificar os procedimentos dos quais existe certeza que estejam corretos, a busca se reduz apenas ao código que ainda não foi revisado.

O programa fica mais legível, enxerga-se ele como uma seqüência de chamadas a funções ou procedimentos.

Programação em Módulos

Os procedimentos desenvolvidos utilizando a programação procedural podem ser dos mais variados, mas eles podem ser agrupados em módulos identificando neles alguma funcionalidade comum. Cada módulo pode conter dados próprios, permitindo que o módulo manipule um estado interno que pode ser modificado pelas chamadas aos procedimentos do módulo.

O principal problema da programação modular é o desacoplamento entre os dados e os procedimentos do módulo.

Programação Orientada para Objetos

A programação é realizada definindo e implementando tipos de dados abstratos e as suas relações, tentando modelar a realidade, desenvolvendo programas que fazem interagir objetos. Os objetos são instâncias de tipos de dados abstratos obtidos a partir da abstração de conjuntos de objetos os quais foram modelados identificando as suas características essenciais e ignorando as características acidentais.

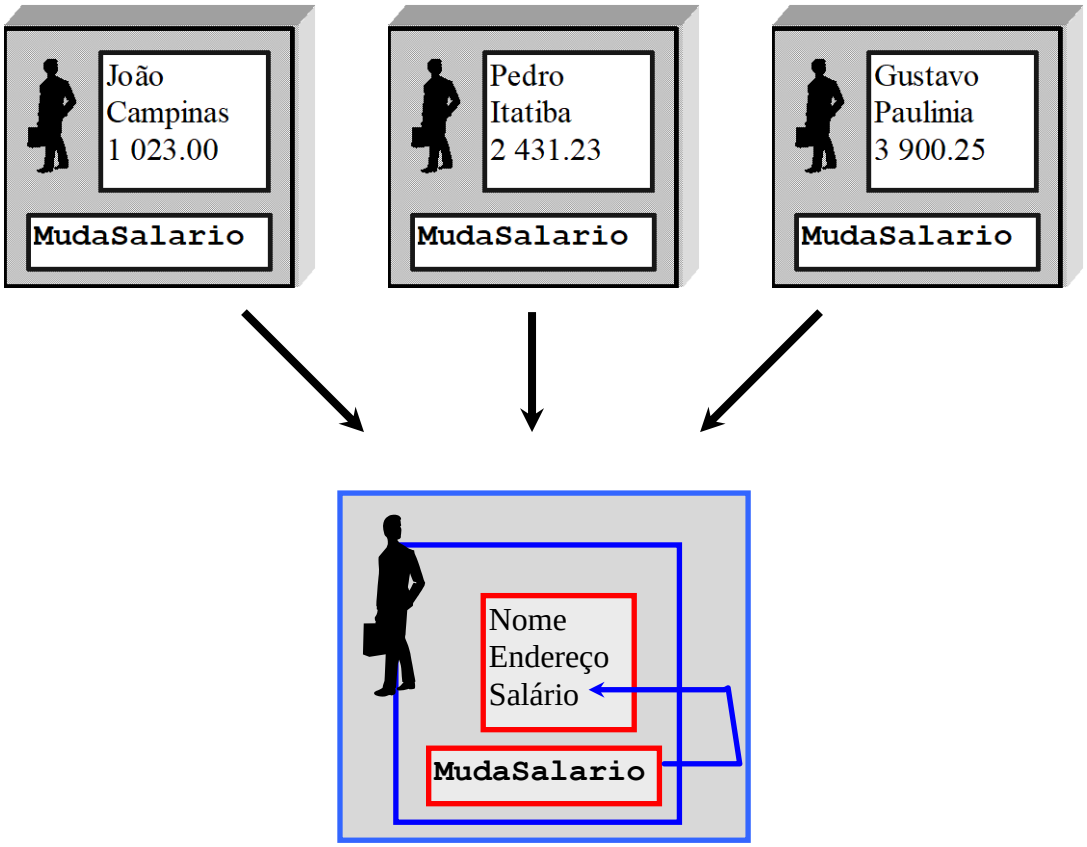
Dez são os conceitos principais que é preciso aprender para entender a programação orientada para objetos: objeto, método, mensagem, classe, sub-classe, instância, herança, encapsulamento, abstração e polimorfismo.

Tipo de dado abstrato

A vantagem de programar orientado para objetos, é que permite pensar no nível do sistema do mundo real que se está enfrentando.

O primeiro passo é entender o problema a resolver para identificar os detalhes essenciais e desconsiderar os superficiais. Isto significa obter uma visão própria (abstrata) do problema, o modelo. Neste modelo são reconhecidos os dados envolvidos e as operações a ser realizadas com estes dados.

Por exemplo, suponha que você tem a enfrentar o problema da administração de uma empresa. Os principais objetos a ser considerados são os funcionários. Os funcionários são pessoas caracterizadas por um conjunto de propriedades como: nome, endereço de residência, data de nascimento, salário, cargo na empresa, profissão, estatura, etc. Algumas destas propriedades são irrelevantes para o problema, como a estatura, mas outras são essenciais, como o salário. Agora é possível definir um modelo abstrato do funcionário considerando apenas as propriedades essenciais dele, as que serão chamadas de dados do modelo funcionário. O modelo funcionário como um conjunto de dados é insuficiente, é adequado identificar operações que possam ser realizadas com um funcionário e/ou com os dados dele.



O processo de modelar um conjunto de objetos da realidade em um tipo de dado abstrato é conhecido como *abstração*, e nele identificam-se as propriedades dos objetos. Estas propriedades são os dados a ser considerados e o procedimento que ele precisa. Então, com a abstração é possível criar entidades bem definidas que consideram duas partes. A primeira é a estrutura de dados comum no conjunto de elementos abstraídos, nomeada de dados membro; e a segunda, é um conjunto de operações que acessa os dados membro e determina o comportamento do tipo de dado abstrato, denominada de interface ou métodos membro.

As linguagens de programação fornecem um conjunto de tipos de dados básicos e o programador define suas variáveis em termos destes tipos de dados básicos. Nas linguagens de programação orientadas para objetos os programadores podem definir novos tipos de dados, chamados tipos de dados abstratos. A ferramenta que permite criar novos tipos de dados é a **classe.** Uma classe pode utilizar nos seus dados membro tipos de dados básicos da linguagem de programação ou qualquer outra classe previamente definida, construindo assim tipos de dados cada vez mais complexos.

Por exemplo, um tipo de dado *avião*, é composto de outros tipos de dados como : quatro *turbinas*, uma *fuselagem*, duas *asas*, etc. O tipo de dado fuselagem está composto de outros dados como : seis *portas*, n *janelas*, material (que pode ser um inteiro, o código do material), espessura da parede, etc.

Na programação orientada para objetos é utilizado o princípio de esconder a estrutura de dados membro de uma classe e fornecer uma bem elaborada interface que permita o acesso indireto a estes dados. Isto é conhecido como **encapsulamento.** Observe que os dados principais da estrutura criada não devem ser acessados por qualquer objeto que poderia modificar os valores acidentalmente. Por exemplo, o dado salário da classe funcionário deve estar adequadamente protegido. Este valor não pode ser modificado a qualquer momento (apenas com ordem expressa da gerência).

No encapsulamento existem vários níveis, encapsulamento privado, protegido e público. Nos dados encapsulados como privados é permitido apenas o acesso do próprio objeto e os seus métodos. Nos dados protegidos é permitido o acesso de classes relacionadas com a propriedade da herança. Os dados públicos são totalmente disponibilizados. Esta proteção pode ser burlada, quando necessário, via outro conceito da programação orientada para objetos, a **amizade (friend)**.

A partir de um tipo de dado abstrato que declara uma estrutura de dados particular e um comportamento, preenchendo valores determinados na estrutura de dados interna se cria

uma instância deste tipo de dado abstrato. Esta instância é chamada de objeto. Muitas instâncias (objetos) podem ser construídas independentemente. Para que a construção dos objetos de um tipo de dado abstrato seja bem sucedida é necessário definir um método especial, método **construtor,** que descreva sequencialmente as ações a executar para montar a estrutura de dados interna do objeto. Por outro lado, a sequência de ações a executar quando o objeto deve ser destruído (p.e. no final do seu escopo) devem ser dados no método **destrutor**.

Em resumo, para bem utilizar a programação orientada para objetos, o usuário analisa os objetos concretos do mundo real a ser trabalhados, abstrai um tipo de dado abstrato em uma classe, e a partir dela cria num código as instâncias da classe que necessita valendo-se do método construtor da classe.

Bibliotecas padrão

Antes de iniciar o trabalho com as classes no C++, comenta-se um pouco sobre as bibliotecas disponibilizadas no C++. Isto facilitará os testes e permitirá acompanhar o que esteja acontecendo na construção ou manipulação de objetos.

Seria uma perda de tempo se cada programador tivesse que idealizar e codificar uma rotina para calcular a raiz quadrada e depois tivesse que incorporar a rotina no programa. C e C++ resolvem estas dificuldades fornecendo ao programador bibliotecas de funções que executam cálculos comuns. Com o auxílio das bibliotecas, é necessária apenas uma única instrução para invocar a função.

A lista a seguir destaca as categorias de bibliotecas fornecidas pelo compilador C++:

Rotinas de entrada e saída	Rotinas de conversão
Rotinas de controle	Rotinas matemáticas
Rotinas de manipulação	Rotinas de alocação de memória
Rotinas de exibição de janela de texto	Rotinas de hora e data
Rotinas de controle de diretório	Rotinas gráficas
Rotinas de classificação	Rotinas padrão

A maioria das funções de biblioteca destina-se a usar informações contidas em arquivos especiais que são fornecidos com o sistema. Estes arquivos, portanto deverão ser incluídos quando você usar as funções de biblioteca. Estes arquivos são fornecidos também com o compilador C++. Eles têm geralmente a extensão .h e são chamados de arquivos de cabeçalho *(header files)*.

Arquivos de Cabeçalho

Arquivos de cabeçalho contêm tipicamente declarações, definições e outras instruções necessárias para vários programas.

Os arquivos de cabeçalho, fornecidos com o compilador, disponibilizam tipos de dados e funções comuns em bibliotecas, e são armazenados em um diretório especial geralmente nomeado de **include**. Para você utilizar uma biblioteca pronta do C++, deve incluir no seu código o arquivo cabeçalho correspondente à biblioteca. Isto é feito em uma linha que inicia com #, estas linhas serão chamadas de diretivas para o pré-processador, pois não é uma linha de código, são instruções realizadas antes da compilação do programa.

Por exemplo, para utilizar a biblioteca de rotinas matemáticas, precisa-se incluir o arquivo cabeçalho **math.h** com a seguinte linha

#include <math.h>

O resultado de incluirmos um arquivo de cabeçalho é o mesmo que sería obtido se incluíssemos o texto do arquivo, *math.h,* nessa mesma posição. Essa estratégia economiza espaço no seu arquivo fonte, pois você não precisa copiar as linhas do *math.h,* em todos os programas que requerem as rotinas matemáticas, o pre-processador o fará antes de passar para o compilador.

Se a diretiva utiliza < >, o pré-processador procura o arquivo cabeçalho no diretório include(por default) fornecido pelo compilador em uso. Se for preciso utilizar um arquivo cabeçalho no diretório ativo (do usuário) deve-se utilizar na diretiva as aspas duplas como limitadores, como no seguinte exemplo:

#include "myheader.h"

A seguir encontra-se uma lista de alguns arquivos de cabeçalho fornecidos com o C++.

Arquivo de Cabeçalho	**Descrição resumida**
iostream.h	declara funções de manipulação de entrada e saída
stdlib.h	miscelânea
ctype.h	para conversão de caracteres
math.h	declara funções matemáticas
string.h	declara funções de manipulação de strings
times.h	rotinas de conversão de tempo
fstream.h	manipulação de arquivos de entrada e saída
iomanip.h	manipuladores parametrizados para entrada e saída

Biblioteca de entrada e saída

Na linguagem C, existe a biblioteca padrão de entrada e saída, que para ser utilizada apenas é necessário incluir uma diretiva no código fonte : *#include <stdio.h>*.
Na linguagem C++, por ser orientada para objetos, não é difícil pensar que existe uma biblioteca para entrada e saída que trabalha com objetos e não simplesmente com funções. Esta biblioteca é um conjunto de objetos e de funções que fazem a interface entre o usuário e o computador, através da tela do vídeo ou do teclado. Para utilizar esta biblioteca no C++, inclui-se no código a diretiva : *#include <iostream.h>*
Para entender melhor isto, leia atentamente e teste o seu primeiro programa em C++

```
/* Primeiro programa exemplo no C++ */
#include <iostream.h>
main( ) {
        cout << “Alô pessoal !\n”;
}
```

Observe que para fazer a saída da mensagem “Alô pessoal!” é utilizado o **cout**. O cout não é uma função que mostra a mensagem na tela, ele é um objeto, uma instância de um tipo de dado abstrato ou classe, já criado na biblioteca standard de entrada e saída.
O tipo de dado do cout é chamado de **ostream** e está declarado no arquivo iostream.h. O objeto cout direciona à saída standard (vídeo) o que for passado para ele. No exemplo foi passada uma string. O cout reconhece o tipo de dado para ele passado, sem precisar especificar um formato.
Para realizar entrada de dados no C++ existe o tipo de dado **istream**, também no arquivo iostream.h. Mais ainda, nele está pronto um objeto que armazena em variáveis os valores fornecidos a partir de um dispositivo de entrada, (o dispositivo standard de entrada é o teclado). Este objeto é chamado **cin**.
O cout faz saída buferizada, isto é, os valores passados são acumulados antes de ser mostrados na saída standard. Buferizar segue o princípio que é eficiente escrever um grupo grande de valores e logo mostrar todos juntos no lugar de fazer uma saída serial destes valores. Mas isto tem uma grande desvantagem quando a saída imediata é necessária, por exemplo, quando no percurso do programa estabelecemos mensagens

informativas. Se o programa tem um erro fatal e é fechado, as saídas que estavam sendo buferizadas não são mostradas mais, elas são perdidas e portanto não se terá informação completa da localização do bug dentro do programa. Por outro lado, é importante identificar no código as mensagens referentes ao propósito do programa, das mensagens informativas para fins de depuração e das mensagens de erro para um provável usuário do programa.

Isto é solucionado no arquivo iostream.h, onde são fornecidos não apenas o objeto cout para saída, mas também outros dois objetos para saída **cerr** e **clog**.

Fluxo	**Propósito**	**Exemplo**
cin	Entrada de teclado (stdin)	cin >> nome;
cout	Saída de dados (stdout)	cout << "teste de saída";
cerr	Saída de erro (não buferizada)	cerr << "Erro crítico!";
clog	Para saída padrão de erro (buferizada)	clog << "msg erro!";

Ao realizar saída para *cout*, *cerr* ou *clog*, pode-se usar os caracteres especiais da tabela abaixo:

Caractere	**Propósito**
\a	Alerta
\b	Backspace
\f	Avanço de página
\n	Caractere de nova linha (equivalente endl)
\t	Tabulação horizontal
\v	Tabulação vertical
\\	Backslash
\?	Ponto de interrogação
\'	Aspas simples
\0	Caractere nulo
\ooo	Valor octal, tal como \0333
\xhhh	Valor hexadecimal, tal como \x1B

Utilizando os objetos para entrada e saída

Os programas a seguir, usam *cout* para exibir mensagens e *cin* para atribuir letras digitadas a uma variável.

```
#include <iostream.h>
void main(void) {
        cout << "Testando entrada e saída em C++, << endl;
}
```

```
#include <iostream.h>
void main(void) {
     char nome[64];
     cout << "Digite Nome: ";
     cin >> nome;
     cout << "Nome digitado: " << nome;
}
```

Todo programa que você criar, indiferente de seu propósito, sempre irá realizar algum tipo de operação de entrada e saída, portanto torna-se indispensável entender fluxos de entrada e saída, e suas capacidades. Por ser este um treinamento muito prático, mesmo antes de entender os conceitos envolvidos na programação orientada para objetos, vamos conhecer vários dos métodos (funções membro) dos objetos de entrada e saída.

Controlando flags de fluxo de entrada e saída (manipuladores)

Um manipulador de fluxo é um item através do qual filtra-se a saída de um fluxo de entrada e saída. Por exemplo, três manipuladores comuns são *hex, oct, dec*, os quais permitem que você exiba valores em formato hexadecimal, octal e decimal. Alguns destes manipuladores de fluxo permanecem em efeito após uma operação de entrada e saída completar.

Ao exibir valores de ponto flutuante, existem situações nas quais deseja-se exibir valores utilizando um formato decimal fixo, tal como 345.452, enquanto que em outras vezes, deseja-se exibir valores em formato cientifico exponencial, tal como 2.12343e4. O manipulador *setiosflags*, por exemplo, possibilita controlar muitos flags, tais como:

ios::fixed e *ios::scientific*. Esses deixam que você controle a saída em ponto flutuante de um programa.

Manipulador	Efeito
dec	Exibe um valor em decimal
oct	Exibe um valor em octal
hex	Exibe um valor em hexadecimal
setbase	Define uma base de conversão
setiosflags	Manipular diversos tipos de arquivos
setfill	Permite especificar largura mínima a usar para exibir um campo de valor
setprecision	Controlar o número dos dígitos exibidos

Exemplo:

```
#include <iostream.h>
#include <iomanip.h>
void main(void) {
     cout << setiosflags(ios::fixed) << 234.45 << endl;
     cout << 23445.8967 << endl;
     cout << resetiosflags(ios::fixed);
     cout << setiosflags(ios::scientific);
     cout << 234.45 << endl;
     cout << setprecision(4);
     cout << 23445.8967 << endl;
}
```

Funções-membro de entrada

Os programas mais simples utilizam-se do *cin* e do operador de inserção (>>) para realizar operações de entrada. À medida que seus programas tornam-se mais complexos sobre as operações de entrada, você necessitará de um controle mais rigoroso do que o operador de inserção pode oferecer. Em tais casos, nossos programas podem utilizar funções-membro de entrada aqui discutidas. As principais funções-membro são:

Função-membro	Efeito
gcount	Determinar a quantidade de caracteres extraídos
getline	Ler uma linha de texto até encontrar retomo de carro
get	Realizar entrada de um caracter por vez
eof	Detectar o fim de arquivo
peek	Ler caracteres até que um caractere específico seja achado
putback	Colocar um caracter qualquer dentro do buffer de entrada
ignore	Ignorar caracteres até que uma dada condição seja satisfeita

Exemplo:

```
#include <iostream.h>
#include <ctype.h>
void main(void) {
     char letra;
     while (!cin.eof()) {
          letra = cin.get();
          letra = toupper(letra);
          cout << letra;
     }
}
```

O programa anterior usa a função-membro *get* para ler entrada redirecionada de um caracter por vez, convertendo a entrada para letras maiúsculas, no decorrer usa a função *eof* para detectar o fim do arquivo (fim da entrada direcionada).

A função *getline* pega todos os caracteres até encontrar o caractere "nova linha" **\n**, mesmo tendo espaços em branco ou caracteres especiais na linha. O caractere final \n, pode ser substituído fornecendo um terceiro argumento na função getline. Esta função deve ser utilizada com cuidado após a inserção de um valor numérico.

O exemplo a seguir faz uso de da função-membro *getline.* Esse programa termina a primeira operação de entrada em um caractere X. A seguir, o programa realiza uma segunda operação de entrada.

```
#include <iostream.h>

void main(void) {
     char linha[128];
     cout << "Digite uma linha do texto e pressione <CR>";
     cout << endl;
     cin.getline(linha, sizeof(linha), 'X');
     cout << "Primeira linha: " << linha << endl;
     cin.getline(linha, sizeof(linha));
     cout << "Segunda linha: " << linha;
}
```

Funções membro de saída

Fluxos de entrada oferecem algumas funções-membro que os programas podem utilizar para realizar operações de entrada. De forma análoga analisaremos aqui as funções-membro de fluxo de saída. *O cout é* um objeto do tipo *ostream.* Como tal, suporta algumas funções-membro. O programa a seguir utiliza a função-membro *put* para exibir uma letra por vez no vídeo.

```
#include <iostream.h>

void main(void)  {
     char letra;
     for (letra = 'A'; letra <= 'Z'; letra++)
          cout.put(letra);
}
```

Outras funções-membro de fluxo de saída são*:*

flush( ), flags( ), setf( ), unsetf( ), etc.

RAPIDAMENTE OS FUNDAMENTOS DO C

Palavras-chaves

Palavras-chave são identificadores predefinidos que possuem significados especiais para o compilador. Exemplos: *char*, *float*, *signed*, *void*, *double*, etc. Esses são símbolos que não podemos usar para nossos próprios propósitos, pois possuem significados especiais para o compilador e não podem ser modificados. Pode-se usá-las apenas como foram definidas.

Maiúsculas e minúsculas

A linguagem C++ é sensível ao tipo de letra utilizado, distingue maiúsculas de minúsculas (case sensitive). Isso significa que, por exemplo, a variável auxvar e auxVar são diferentes.

Palavras-chave C

asm	far	short	break	float	signed
case	for	sizeof	goto	static	char
huge	struct	const	if	switch	continue
int	typedef	default	interrupt	union	do
long	unsigned	double	near	void	else
pascal	volatile	enum	register	while	retum

Palavras-chaves C++ não encontradas em ANSI C

friend	private	this	catch	inline	protected
virtual	class	new	public	delete	operator
template					

Comentários

Todo comentário em C++ pode ser começado por /* e prosseguindo da esquerda para a direita até alcançar o símbolo */. O compilador ignora tudo que estiver entre esses símbolos. Existe ainda um segundo modo de se fazer comentários em C++ (não

disponível no C), são os comentários para linha, feito com duas barras //. Neste caso o compilador ignora todo que estiver após as duas barras até o final da linha.

```
/* Este é um exemplo de se fazer um comentário
na linguagem C .
*/
cout << variavel01 << /*comentario*/ variavel02 << endl;      // imprimindo variaveis
```

Tipos de dados

Em C++ uma variável é uma posição de memória que recebeu um nome. Variáveis podem armazenar todos os tipos de dados: strings, números, estruturas, etc. Todas as variáveis possuem uma característica em comum: um tipo de dados associado. Isso significa que, além de escolhermos um nome apropriado para a variável, devemos também dizer ao compilador que tipo de informação deseja-se armazenar.

Uma observação importante é que nem todas as versões do C++ armazenam as variáveis obrigatoriamente no mesmo número de bytes. Entretanto, uma variável *char* sempre ocupa um byte. Abaixo, alguns exemplos do uso de variáveis em C++:

```
long populacao, conta;
int agencia, fone;
long populacao = 5000;
long conta        = 124243;
```

Escopo

Escopo de uma variável refere-se aos limites que uma variável tem validade. Instruções podem se referir a variáveis, mas apenas quando as variáveis estão dentro do mesmo escopo, ou nível.

Constantes

Constantes são variáveis cujo conteúdo não pode ser alterado durante a execução de um programa. As constantes em C++ são tipicamente escritas em letras maiúsculas, ainda que isso não seja obrigatório. Por exemplo, um programa que armazena notas de exames. Podemos criar uma constante chamada NOTA_MAXIMA e associar esse identificador com um valor, talvez 7. Se a nota máxima mais tarde for alterada para 10, tudo o que necessita ser feito é revisar o valor da constante e recompilar o programa.

Não havendo a necessidade de procurar toda ocorrência de 7 no programa e alterá-la para 10. Existem em C++ quatro tipos de constantes:

Constantes literais

Constantes definidas

Constantes declaradas

Constantes enumeradas

Constantes Literais

Constantes literais formam a variedade mais comum. Elas compõem-se de valores como 34, 3.1415 ou ainda "informações de string" digitados diretamente no texto do programa. Podem ser do tipo: caracter, string, composta por números inteiros (short, int e long), ponto flutuante.

Constantes Definidas

É uma boa idéia dar aos valores, nomes que ajudem sua memória a se lembrar do que esses valores representam. Um modo de dar um nome a uma constante é usar uma macro #define. Observa-se que aqui, o #define não especifica tipos de dados, não usa o símbolo de atribuição (=), e não termina com ponto-e-vírgula,

#define NOTAMAXIMA 10

Constantes Declaradas

Um segundo modo preferido, de se criar constantes em programas é anteceder uma definição normal de variável com a palavra-chave *const*. O compilador rejeita quaisquer instruções que tentem alterar valores definidos dessa forma. Vale a pena observar que as definições com *const* especificam tipos de dados, terminam com ponto-e-vírgulas e são inicializadas como variáveis, como pode ser visto:

const char CHARACTER = 'S' ;

const int DECIMAL = 10;

Constantes Enumeradas

Ao invés de se definir constantes individuais que recebem valores subjacentes, pode-se usar constantes enumeradas para criar listas categorizadas que atingem o mesmo objetivo. O exemplo clássico de constantes enumeradas é o de uma lista de cores que pode ser declarada dessa forma:

enum ***cores*** *{VERMELHO. AMARELO, AZUL, LARANJA, VERDE, VIOLETA};*

A palavra-chave *enum* diz ao compilador que estes itens devem ser enumerados, isto é, associados a números inteiros seqüenciais. Assim, o compilador atribui um valor começando por 0, para cada elemento enumerado, de forma que VERMELHO seja equivalente a 0, AMARELO a 1, etc. Você pode atribuir um valor expresso a uma constante enumerada, mesmo um valor inteiro negativo, e as seguintes constantes enumeradas assumem valores consecutivos.

enum ***status*** *{FALSE, TRUE, NOTRUN=-1, FAIL, OK } estado;*

Observe que FAIL é atribuido com o valor zero e OK com o valor 1.

Na instrução, além de declarar a enumeração, está-se criando uma variável que pode assumir os valores de status.

Nota: Um elemento de uma enumeração não tem endereço de armazenamento em memória. Não deve se utilizar o operador &(endereço) com eles.

Operadores

C++ é repleto de operadores (símbolos que executam várias operações sobre os seus argumentos. Além dos usuais + (mais), - (menos), / (divisão) e * (multiplicação), existem operadores que criam funções, operadores para vetores, operadores que incrementam e decrementam valores e operadores mais complexos que executam mais do que uma tarefa de uma só vez.

Precedência refere-se à ordem em que o C e o C++ avaliam os operadores quando existem dois ou mais deles em uma sentença. Existe um conjunto de regras incorporadas para determinar a ordem em que os operadores são avaliados, e é preciso decorá-las para redigir códigos que realizem corretamente as operações. Dizer que um operador tem precedência maior que um outro operador significa que será avaliado antes.

A associação refere-se à ordem de avaliação de operadores de igual precedência. Eles podem ser avaliados primeiro de direita à esquerda ou de esquerda à direita.

A seguir está uma tabela com os operadores, a sua precedência e a associação respectiva. A ordem de precedência é de cima para baixo, sendo avaliados primeiro aqueles que estão mais acima.

Precedência	Associação
( ) [ ] . ->	Da esquerda para a direita
! ~ -(unário) ++ -- *(unário) &(unário) (cast) sizeof	Da direita para a esquerda
* / %	Da esquerda para a direita
+ -	Da esquerda para a direita
<< >>	Da esquerda para a direita
< <= > >=	Da esquerda para a direita
== !=	Da esquerda para a direita
&	Da esquerda para a direita
^	Da esquerda para a direita
|	Da esquerda para a direita
&&	Da esquerda para a direita
||	Da esquerda para a direita
? :	Da direita para a esquerda
= += -= *= /= %= (operadores de atribuição)	Da direita para a esquerda
,	Da esquerda para a direita

O problema da precedência e a associação é superado utilizando parênteses no código, e apenas decorando as precedências mais importantes.

ESTRUTURAS DE CONTROLE

Instruções condicionais

A linguagem C suporta quatro instruções condicionais básicas:

if, if-else, condicional ?: e *switch.*

A maioria das instruções condicionais pode ser usada para executar seletivamente uma única linha de código relacionado. Sempre que uma instrução condicional estiver associada com apenas uma linha, não são necessárias chaves { } envolvendo a instrução. No entanto, se a instrução condicional estiver associada a mais do que uma instrução (linha), torna-se necessário o uso das chaves.

Instrução if / else if

É usada para executar condicionalmente um segmento de código. A forma mais simples dessa instrução é:

if (expressão)

ação;

Onde a *expressão* deve ser: verdadeiro(!0) ou falso(0).

```
#include <iostream.h>
main() {
     double salario;
     int nivel;
     cout << "Digite salario: ";
     cin >> salario;
     if(salario < 500.) {
          cout << "Nivel 1" << endl;
          nivel = 1;
     } else if(salario >= 500. && salario <= 1000.) {
          cout << "Nivel 2" << endl;
          nivel = 2;
      } else if(salario > 1000.) {
          cout << "Nivel 3" << endl;
          nivel = 3;
      }
```

```
      if(nivel==2 || nivel==3)
          cout << "Empregado satisfeito" << endl;
     else
          cout << "Empregado insatisfeito" << endl;
}
```

Instrução switch

Uma instrução *switch* torna-se prática sempre que um programa necessita selecionar algumas ações dentre as diversas possíveis, tendo como base o resultado de uma expressão ou uma variável, equivalentes a um valor inteiro ou a um caracter.

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#define TIPO 1
main() {
     char escolha;
     while(TIPO) {
          cout << "\nMenu: Adicionar Deletar Ordenar Sair: ";
          cin >> escolha;
          switch(toupper(escolha)) {
               case 'A':
                    cout << "Você selecionou adicionar\n";
                    break;
                case 'B':
                    cout << "Você selecionou deletar\n";
                    break;
                case 'C':
                    cout << "Você selecionou ordenar\n";
                    break;
                case 'S':
```

```
                    exit(O);
                default:
                    cout << "Tecla inválida. Novamente!\n";
            }
        }
}
```

Preste atenção particularmente a instrução *break*. A instrução *break* faz com que a parte restante das instruções *switch* sejam puladas.

Instrução condicional

A instrução condicional ? proporciona uma maneira rápida de escrever uma condição de teste. São executadas ações associadas dependendo se a expressão for avaliada como VERDADEIRA ou FALSA Você pode usar o operador ? para substituir uma instrução if-else equivalente. A sintaxe para uma instrução condicional é:

expressão? ação1: ação2;

O operador ? também é conhecido como operador ternário porque precisa de três operadores. A listagem a seguir demonstra como reescrever uma instrução *if-else* usando o operador condicional:

if(opção)
    x=y;
else
    x=y/2;

A mesma instrução reescrita com o operador condicional:

x = opcao ? y : y/2

```
#include <iostream.h>
main() {
     char ch;
     int resp, vall,val2;
     cout << "\nDigite dois valores inteiros: ";
     cin >> vall >> val2;
```

```
     cout << "\nDigite '+' para somar e outra tecla para
subtrair: ";
     cin >> ch;
     resp = (ch == '+') ? val1+val2 : val1-val2;
     cout << "\nO resultado ,: " << resp << endl;
     return 0;
}
```

Instruções em loop

A linguagem C++ assim como a linguagem C contém uma série-padrão de instruções de controle de repetição: laços *for*, *while*, *do-while* (conhecido como *repeat-until* em várias linguagens de alto nível). Todos os loops (laços) podem terminar naturalmente baseados na condição de teste booleano. No entanto, em C/C++, um loop de repetição pode terminar devido a uma condição de erro antecipado usando instrução *break ou exit*. *Os* loops de repetição podem também ter seu fluxo de controle lógico alterado por instruções *break e continue*.

Loop for

A sintaxe de um loop *for* é dada por:

for(inicialização_expressão; teste_expressão; incremento_expressão) {

instrução_a;

instrução_b;

}

O programa a seguir calcula o fatorial de um número.

```
#include <iostream.h>

main() {
     int numero, fatorial = 1;
     cout << "Digite o número: ";
     cin >> numero;
```

```
      for(int i=numero; i>0; i--)
           fatorial *= i;
      cout << "Fatorial de " << numero << " = " << fatorial;
      cout << endl;
      return 0;
}
```

Quando se tem mais do que uma instrução para um loop *for*, deve-se utilizar chaves para sinalizar ao compilador quais instruções devem ser incorporadas ao loop.

Loop while

Exatamente como o loop *for*, o *while* de C/C++ é um laço de pré-teste. Isto significa que é avaliada a condição antes que sejam interpretadas as instruções que estão internas ao corpo do loop. Daí, que pode ser executado desde zero até muitas vezes.

```
while(condição){
      instrução a;
      instrução n;
}
```

Este tipo de loop é usado sempre que é esperado um número indefinido de repetições. O programa usa um laço *while* para calcular uma lista de números definida pelo usuário:

```
#include <iostream.h>
#define VERDADEIRO 1
#define FALSO 0
main() {
      int quantidade, sinal = FALSO;
      double soma = 0.0, valor = 0.0, media = 0.0;
      while(!sinal) {
         cout << "\nNúmero p/ calcular média e 0 p/ sair:";
         cin >> valor;
         if(valor I= 0.0) {
```

```
            soma += valor;
            quantidade++;
         } else
            sinal = VERDADEIRO;
      }
      quantidade>0 ? (media=soma/quantidade) : (media = 0);
      cout << "A soma de " << quantidade << " números é ";
      cout << soma << "A média é " << media;
      return 0;
}
```

O programa inicia definindo duas constantes *int* **VERDADEIRO** e **FALSO**, que serão usadas como sinalizador para determinar quando um loop *while* terminará. O sinalizador sinal é inicializado em **FALSO**, como o loop irá repetir até que o usuário introduza o valor 0, que mudará o sinalizador sinal para **VERDADEIRO**. Como os loops *while* somente se repetem enquanto a condição de teste é avaliada como **VERDADEIRO**, a condição ***!VERDADEIRO*** pára as repetições.

Loop do-while

O loop *do-while* difere tanto do loop *for* quanto do *while* pelo fato de que é um loop do tipo pós-teste. Em outras palavras, o laço é interpretado ao menos uma vez, e a condição do loop é testada ao fim da primeira iteração. A sintaxe para um loop *do-while* é:

```
do {
        ação _1;
        ação_2;
} while(condição_de_teste);
```

O laço do-while é uma boa escolha para menus, porque sempre você deseja que uma função do menu execute ao menos uma vez. Depois que as opções forem mostradas, o programa será executado até que uma opção válida seja selecionada.

O programa a seguir utiliza um loop *do-while* para imprimir um menu e obter uma resposta válida do usuário:

```
#include <iostream.h>
void menu( ) {
  char c;
  cout << "1. Verificar ortografia\n";
  cout << "2. Corrigir erros de ortografia\n";
  cout << "3. Mostrar erros de ortografia\n";
  cout << "Entre com a sua escolha : ";
  do {
     cin >> c;   // Lê do teclado a seleção
     switch(c) {
       case '1':
         check_spelling();
         break;
       case '2':
         correct_erros();
         break;
       case '3':
         display_erros();
         break;
       default:
         cout << "Escolha inválida. Tente novamente.\n";
         break;
     }
  }while(c!='1' && c!='2' && c!='3');
}
```

Instruções break, continue, exit

A instrução *break* de C/C++ pode ser usada para sair de um loop antes que a condição de teste se torne falsa. Quando se sai de um loop através de uma instrução *break*, a execução do programa continua na primeira instrução que vem logo após o loop. Também é utilizada para quebrar o fluxo dentro de um switch.

O *break* pára todas a execução de um loop, em contraste, *continue* faz com que todas as instruções dentro do loop que vêm após ela sejam ignoradas, mas não impede o incremento da variável de controle do loop ou condição de teste de controle do loop. Em resumo, o continue apenas quebra o processo de uma iteração e permite continuar com a próxima iteração sempre que a condição seja verdadeira.

Algumas vezes pode ser interessante que um programa encerre a execução muito antes que instruções tenham sido examinadas e/ou executadas. Para estas situações, C/C++ oferece a função de biblioteca *exit.* A função *exit* espera por um argumento inteiro chamado de *status* (valor de estado). Os sistemas operacionais UNIX e WINDOWS interpretam um valor de status 0 como terminação normal de programa e valores do status diferentes de zero como diferentes tipos de erro.

FUNÇÕES EM C++

As funções no C++ são identicamente tratadas como no C, elas devem ser concisas, para tornar os conceitos fáceis de entender e para evitar que você se perca numa profusão de códigos.

Modelagem de função

Embora sejam permitidas outras variações, sempre que possível você deverá usar a forma do modelo de função que é uma réplica da linha de declaração de função.

retorna_tipo nome_função(tipo(s)_argumento nome(s)_argumento);

```
#include <iostream.h>
int subtrair(int x, int y);          // modelo de funcao
main() {
     int a = 3;
     int b = 45;
     int c;
     c = subtrair(a,b);
     cout << "A diferenca b-a = , " << c << endl;
     cout << "a = " << a << endl;
     cout << "b = " << b << endl;
```

```
     //Perceba que os valores de a e b não foram alterados
      return 0;
}
int subtrair(int x, int y) {     // declaração da função
     int r;
     r = y-x;
     x++;
     y--;
     return r;           // tipo de retorno da função
}
```

A função é chamada de subtrair. O modelo estabelece que a função aceitará dois argumentos *int* e retomará um tipo *int.* Na realidade, o padrão ANSI sugere que cada função seja modelada num arquivo de cabeçalho separado, é assim que os arquivos de cabeçalho são associados com as bibliotecas apropriadas.

A função pode ser do tipo *void, char, int, long, double*, ou qualquer outro tipo de dado declarado previamente. E estas especificações são dadas no *retorna_tipo*.

O *nome_função* é qualquer nome que tenha significado e que você escolhe para descrever a função. Se qualquer informação é passada à função, você dará *um tipo(s)_argumento* seguido de um *nome(s)_argumento*. *tipo(s)_argumento* podem também ser qualquer tipo de dado existente. Você pode passar muitos valores a uma função repetindo o tipo do argumento e nome do argumento separados por uma vírgula. É correto também listar apenas o tipo do argumento, mas, essa forma é mais usada nos arquivos cabeçalho.

Chamada por valor / referência

Quando são passados argumentos por valor para uma função, uma cópia do valor da variável é na realidade, passada à função. Como é passada uma cópia, não é alterada a variável no programa que chama a função. Chamar uma função por valor é um meio comum de passar informação a uma função e é o método-padrão em C e C++.

```
#include <iostream.h>
int subtrair(int &x,int &y); //int subtrair(int *x,int *y);
```

```
main() {
     int a = 25;
     int b = 67;
     int c;
     c = subtrair(a,b);            //c = subtrair(&a,&b);
     cout << "A diferenca b-a = " << c << endl;
     cout << "a = " << a << endl << "b = " << b << endl;
     // valores de a e b alterados na função subtrair
    return 0;
     //retornam com seus respectivos valores alterados
}
int subtrair(int &x,int &y) { //int subtrair(int *x,int *y)
     int r;
     r = y-x;
     x++;
     y--;
     return r;
}
```

Em C++ além de usar variáveis e ponteiros como argumentos, acrescenta um terceiro tipo de argumento chamado tipo referência.

Numa chamada por referência, é passada à função o endereço do argumento, e não o seu valor. Esta abordagem requer menos memória de programa do que a chamada por valor. Quando se usa a chamada por referência, as variáveis no programa que chama a função podem ser alteradas. O mesmo acontece quando é passado o ponteiro para um variável. Compare os valores das variáveis no exemplo acima.

Recorrência

A recorrência acontece quando uma função chama a si própria.

```
#include <iostream.h>
main() {
  double numero = 5;
```

```
  double fat;
  fat = fatorial(numero);
  cout << "Fatorial de " << numero << " é " << fat << endl;
  return O;
}
double fatorial(double resp) {
  if(resp <=1.0)
    return 1.0;
  else
          return (resp*fatorial(resp-1.0));
}
```

A recorrência é permitida tanto em C como em C++. O exemplo acima utiliza-se da recorrência de uma função para calcular o fatorial de um número.

Tipos de função

Um tipo de função é o tipo de valor retomado pela função. Nenhum dos exemplos anteriores retomavam informações da função e, portanto, eram do tipo *void*.

A seguir, está uma função de tipo double chamada trigoseno. A função trigoseno utiliza a função *sin* descrita em *math.h*, biblioteca de rotinas matemáticas, para obter a resposta. Os ângulos devem ser convertidos de graus para radianos para todas as funções trigonométricas.

```
#include <iostream.h>
#include <math.h>
double trigoseno(double angulo);
main() {
    double seno;
    for(int i=0; i<91; i++){
      seno = trigoseno((double) i);
      cout << "Seno de " << i << " é = " << seno << endl;
    }
    return 0;
}
```

```
double trigoseno(double angulo) {
    double temp;
    temp = sin((M_PI/180.0)*angulo);
    retur temp;
}
```

Argumentos de função para main

C e C++ podem aceitar argumentos de linha de comando. Os argumentos de linha de comando são passados quando o programa é chamado através do DOS/UNIX.

*C:>**programa** marcos jose maria joana lisandra 20 40 50*

No exemplo acima, são passados oito valores pela linha de comando para *programa*. De fato, essa informação específica é dada a *main.* Um argumento recebido por *main, argc, é* um *int* dando o número de termos da linha de comando mais um. O título do programa é contado como o primeiro termo. O segundo argumento é um ponteiro para as strings chamado *argv*. Todos os argumentos são cadeias de caracter (strings), portanto *argv* é do tipo *char*[argc]*. Como todos os programas têm um nome, *argc* é sempre um ou maior do que um. Os dois exemplos a seguir explicam várias técnicas para se recuperarem informações da linha de comando. Os nomes de argumentos argc *e argv* são necessários ao compilador e não podem ser alterados.

Se forem digitados números na linha de comando, eles serão interpretados como strings ASCII e deverão ser impressos como dados caracteres. O programa a seguir permite que vários ângulos sejam digitados na linha de comando. O seno dos ângulos será calculado e apresentado na tela.

Perceba que os ângulos são ingressados ao programa como strings, então é necessário inserir o arquivo stdlib.h para poder utilizar as funções de biblioteca que transformam strings em float, *atof*. Nesse arquivo existem muitas outras funções para conversão, por exemplo, *atoi*, para converter uma string (argumento) em um inteiro.

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
void main(int argc, char *argv[]) {
```

```
   double angulo;
   if(argc<2) {
     cout << "Digite varios angulos na linha de comando\n";
     cout << "O programa retorna o seno dos angulos\n";
     exit(0);
   }
   for(int i=0; i<argc; i++) {
     angulo = (double)atof(argv[i]);
     cout << "O seno de " << angulo;
     cout << " eh " << sin((PI/180.)*angulo) << endl;
   }
}
```

CARACTERÍSTICAS DE C++

C++ oferece características especiais para funções, permitindo declarar as funções como inline ou outline. C++ permite também a sobrecarga de função, que possibilita se dar o mesmo nome de função a várias funções. Os modelos individuais de função são reconhecidos pelo seu tipo e pela lista de argumentos, não exatamente pelo nome. É útil onde uma função precisa funcionar com diferentes tipos de dados.

Função inline (em linha)

A palavra-chave *inline* (em linha) é uma diretiva, para que o compilador C++ insira uma função em uma linha. A codificação para uma função em linha (*inline*) é copiada onde a função é chamada no programa. Colocando-se a codificação no ponto de chamada da função, consegue-se economizar tempo de execução em funções pequenas chamadas com freqüência.

```
#include <iostream.h>
void imprime(void);
main() {
     for(int i=0; i<10; i++)
        imprime;
     return 0;
```

```
}
inline void imprime(void) {
       cout << "Exemplo de uma função em linha\n";
}
```

Sobrecarga de funções

O exemplo ilustra a sobrecarga de função. Note que duas funções com mesmo nome são modeladas dentro do mesmo escopo. A função correta será selecionada com base nos argumentos fornecidos. Existem algumas coisas que têm que ser evitadas quando se sobrecarrega uma função. Por exemplo, se uma função somente difere do tipo (e não nos argumentos), ela não pode ser sobrecarregada.

```
#include <iostream.h>
#include <iostream.h>
int multiplica(int matriz[]);
double multiplica(double matriz[]);
main() {
  int matriz1[5] = {1, 2, 3, 4, 5};
  double matriz2[5] = {1.2, 2.1, 3.5, 5.5, 6.5};
  int prodl = multiplica(matriz1);
  double prod2 = multiplica(matriz2);
  cout << "Produto de numeros inteiros: " << prodl << endl;
  cout << "Produto dos numeros double: " << prod2 << endl;
  return 0;
}
int multiplica(int matriz[]) {
  int temp;
  temp = matriz[0];
  for(int i=1; i<5; i++)
    temp *= matriz[i];
  return(temp);
}
double multiplica(double matriz[]) {
  double temp;
```

```
  temp = matriz[0];
  for(int i=1; i<5; i++)
    temp *= matriz[i];
  return(temp);
}
```

Uma função sobrecarregada, também não aceita a seguinte lista de argumentos:

double carga funcao(int valor)

double carga funcao(int &numero)

Esta lista de argumentos não seria permitida porque cada função aceitaria o mesmo tipo de argumentos. A sobrecarga de funções será discutida novamente quando estivermos trabalhando com classes, especificamente com funções-membro de classes.

Ponteiros

Este capítulo descreve os fundamentos dos ponteiros em C++. Nesse capitulo você aprenderá como declarar variáveis-ponteiro, usar ponteiros de forma segura e correta, o que é alocação dinâmica de memória e como usá-la e por que os ponteiros *void* são tão poderosos.

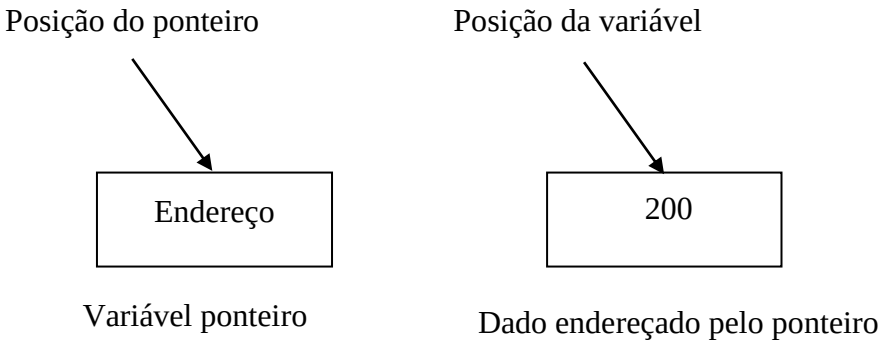
Nosso mundo está repleto de ponteiros. As idéias de uma flecha apontando o sentido de uma rua de mão única e a de um dedo apontando para uma direção são naturais e intuitivas. Por que então vários programadores acreditam que ponteiros em programação são como flechas envenenadas com seus nomes escritos nelas?

Esqueça o que tenha ouvido sobre ponteiros provocando bugs. Ponteiros não causam bugs, as pessoas sim. Você pode escrever programas sem ponteiros. Mas, como aprenderemos, eles oferecem um controle praticamente ilimitado sobre o armazenamento de memória, deixando outras técnicas muito para trás.

Declarando variáveis-ponteiro

Para criar um ponteiro, adicione um espaço e um asterisco após qualquer identificador de tipo de dado. Por exemplo, *int* p cria um inteiro simples, mas *int* *p cria um ponteiro para um inteiro. Assim:

- Um ponteiro é uma variável assim como outra qualquer;
- Uma variável ponteiro contém um endereço que aponta para outra posição de memória;
- Armazenado nessa outra posição fica o dado que o ponteiro endereça;



Um ponteiro aponta para uma variável armazenada na memória. Então se você declarar uma variável como:

*int *ptr;*

estará dizendo ao compilador que a variável ptr é um ponteiro para um inteiro armazenado em algum lugar da RAM. Existe uma diferença significativa, entretanto, entre variáveis normais e aquelas endereçadas por ponteiros. Os programas podem manipular os ponteiros de forma que não são aplicáveis às variáveis normais. Alterando-se, manipulando-se e fazendo novas atribuições ao endereço armazenado em um ponteiro, toma-se possível criar várias estruturas de dados interessantes tais como listas, árvores e pilhas que usam ponteiros como elos para itens na memória. Dessa maneira ponteiros operam como correntes conectando variáveis que podem ficar armazenadas em qualquer lugar, tornando possível projetarmos estruturas que crescem e encolhem de modo a acomodarem o que quer que você deseje armazenar dentro delas. Com ponteiros, todos os bytes de sua memória ficam a sua disposição.

O trecho do programa a seguir mostra como declarar e inicializar um ponteiro string:

```
#include <iostream.h>
#include <string.h>
main() {
     int indice;
     char *palindromo = "Coitado Dan, ele está desanimado";
```

```
     for(indice=strlen(palindromo)-1; indice >=0; indice--)
          cout << palindromo[indice];
     return 0;
}
```

A listagem abaixo demonstra os conceitos de *criação*, *inicialização* e *derreferenciamento* de uma variável ponteiro. Estes são os detalhes mais importantes a serem aprendidos sobre ponteiros.

Escrito como **ptr*, a variável *ptr* se torna um ponteiro para uma variável *char* armazenada em algum lugar da memória. Entretanto, nesse estágio inicial do programa, *ptr* ainda não endereça qualquer posição válida.

```
#include <iostream.h>
main() {
     char c;
     char *ptr;
     ptr = &c;
     cout << "ALFABETO" << endl;
     for(c='A'; c <= 'Z'; c++)
          cout << *ptr;
     return 0;
}
```

Como todas as variáveis, o ponteiro requer inicialização antes de ser usado. Essa tarefa é cumprida ao atribuirmos a *ptr* o endereço da variável *c*. A expressão *&c* significa "endereço da variável *c*". Agora, se a variável *c* está armazenada em algum lugar da memória, e *ptr* aponta para a mesma posição de memória, então, referem-se ao mesmo dado, alterar um deles certamente afetará o outro.

Outro fato que deve ser observado é o asterisco em frente ao nome do ponteiro. Esse símbolo diz ao compilador C++ para derreferenciar um ponteiro, em outras palavras, para usar o dado que o ponteiro endereça. Nesse caso, o dado é um caracter e, portanto, a instrução exibe o valor deste caracter.

Ponteiros void e NULL

Agora que você já sabe como declarar um ponteiro, inicializá-lo e derreferenciá-lo para usar o dado endereçado, existem dois outros conceitos importantes e relacionados que são os ponteiros *void e NULL*. Um ponteiro *NULL* não aponta para qualquer lugar em particular. É como uma flecha sem a ponta, é o modo do C++ dizer que, ao menos para o momento "este ponteiro não endereça qualquer dado válido na memória".

O propósito de um ponteiro *NULL* é dar aos programas um modo de saberem quando um ponteiro endereça uma informação válida. *NULL* é uma macro equivalente a 0 e é definida no arquivo de cabeçalho *stdlib.h*, devemos incluir-lo antes de poder usar o identificador *NULL*. Ou define-se *NULL* no topo de nosso programa (o que é muito melhor e seguro) com a seguinte linha:

#define NULL 0

Como em uma função *void*, que não retorna qualquer valor, mas realiza uma ação, um ponteiro *void* não aponta para um tipo específico de dados, mas armazena um endereço. Pode-se criar um ponteiro *void* da seguinte forma:

*void *var;*

O ponteiro *void var* pode endereçar qualquer posição de memória, mas o ponteiro não fica amarrado a um tipo de dados específico. Ponteiros *void* semelhantes a este podem endereçar uma variável, float, *char, double, int,* etc. Ou seja, um ponteiro *void* é um ponteiro para qualquer tipo de dado. O programa C++ a seguir demonstra o uso de ponteiros *void*:

```
#include <iostream.h>
void main() {
     void *ptr;
     double a,b,c;
     ptr = &a;
     b  = 10;
     c  = 7;
     a  = b+c;
     cout << "O valor de a é " << *ptr << endl;
     return 0;
}
```

Alocação dinâmica de memória

Quando um programa C é compilado, a memória do computador é dividida em quatro zonas que contêm: a codificação do programa, todos os dados globais, a pilha e o "*heap*". *Heap* é uma área de memória livre (armazenagem livre) que é manipulada com as funções de alocação dinâmica *malloc* e *free*.

A maioria dos compiladores C usa as funções de biblioteca *malloc* e *free* para proporcionar alocação dinâmica de memória. No entanto, em C++, estas propriedades foram consideradas tão importantes que se tornaram parte do núcleo da linguagem. C++ usa *new* e *delete* para alocar e liberar memória do heap. O argumento para *new*, é uma expressão que retorna o número de bytes alocados; o valor retornado é um ponteiro para o início desse bloco de memória. O argumento para *delete* é o endereço inicial do bloco de memória a ser liberado. Os dois programas a seguir ilustram as semelhanças e diferenças entre as aplicações C e C++ que usam alocação dinâmica de memória, respectivamente.

```
#include <stdio.h>  //Programa em C
#include <stdlib.h> //contém definições para malloc e free
#define MAXIMO 256
void main() {
     int *bloco;
     bloco = (int *)malloc(MAXIMO*sizeof(int));
     if (bloco == NULL)
          printf("Memoria insuficiente\n");
     else
          printf("Memoria alocada\n");
     free(bloco);
     return 0;
}
```

```
#include <iostream.h>    //Programa em c++
#define MAXIMO 256
main() {
     int *bloco;
     bloco = new int[MAXIMO];
```

```
      if(bloco == NULL)
            cout << "Memoria insuficiente\n";
      else
            cout << "Memoria alocada\n";
      delete(bloco);
      return 0;
}
```

PROGRAMAÇÃO ORIENTADA PARA OBJETOS

A programação orientada para objetos é baseada na escrita de programas em termos de objetos (coisas) que compõe um sistema. Um objeto representa uma entidade do mundo real (coisa) que pode armazenar dados (variáveis-membros) e possui um conjunto especifico de operações (funções-membro) que são realizadas nele, ou ainda, é um conjunto de dados e procedimentos para trabalhar com esses dados. As funções-membro também conhecidas como métodos, definem o que pode ou não ser feito com os dados.

Uma vez que em um sistema objetos armazenam tipos específicos de informação e operações específicas, o primeiro passo na criação de um programa baseado em objetos é identificar os objetos do sistema, a informação chave do objeto e as operações realizadas nos objetos.

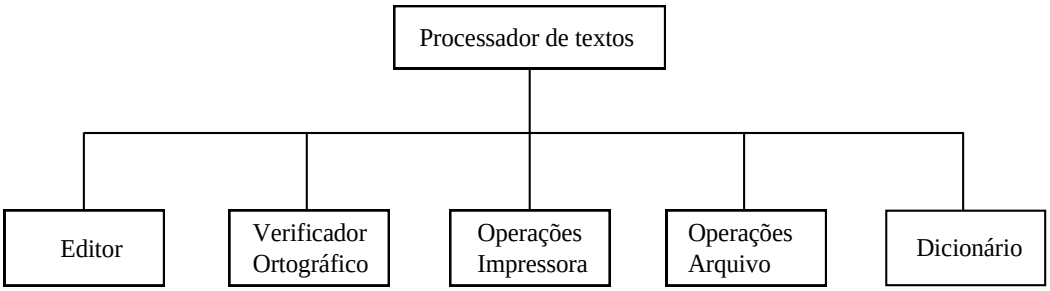
A programação orientada para objetos é um técnica relativamente nova para a concepção e implementação de sistemas de software, centraliza-se nos principais conceitos: tipos de dados abstratos e classes, hierarquias de tipos (subclasses), herança e polimorfismo. Acredita-se que o ponto marcante da programação orientada para objetos seja a possibilidade de se reutilizar código, ou seja, um objeto pode usar as implementações de um outro objeto e estendê-las para alcançar um objetivo específico.

Quando se trabalha com orientação para objeto, todos os objetos são organizados em classes. São definidas classes bases e estas são estendidas de forma a criar novas classes, o que torna o software orientado para objeto muito, mais fcil de ser

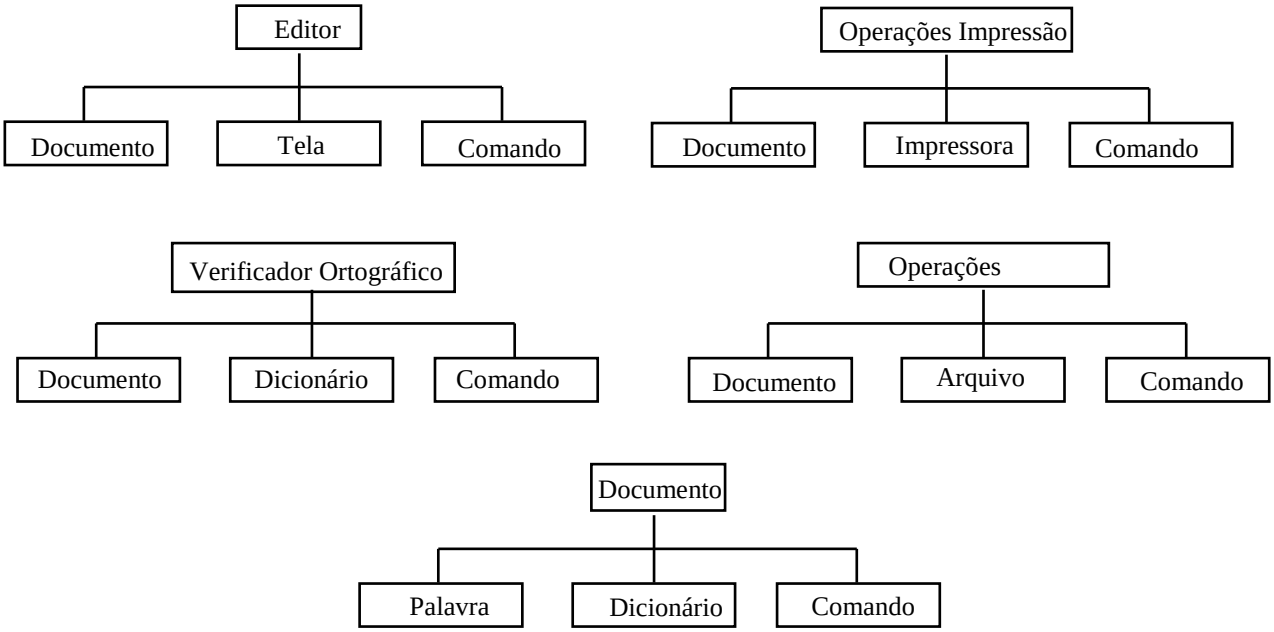
documentado e entendido. O principal objetivo da programação orientada para objetos é aumentar a produtividade do programador através de uma maior reutilização de software, além de diminuir a complexidade e o custo da manutenção do mesmo.

Objetos

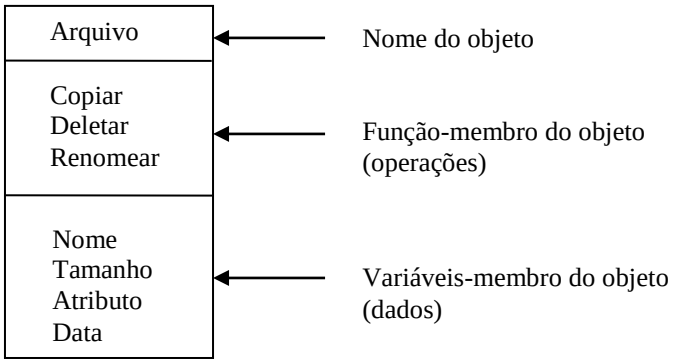
Voltando ao passado, os programadores visavam programas como longas listas de instruções que realizam uma específica tarefa. Quando você criar programas baseados em objetos, você olhará para os objetos que fazem parte do seu programa. Por exemplo, assuma que você está escrevendo um programa que implementa um simples processador de textos. Se você pensar em todas as funções que um processador de textos realiza, pode rapidamente desanimar. De qualquer modo, se você visualizar o processador de textos como uma coleção de distintos objetos, o projeto toma-se menos intimidante. Por exemplo, a figura abaixo ilustra os principais objetos do processador de textos.



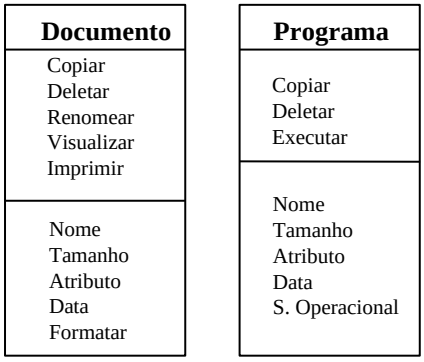
A medida que você começa a identificar os objetos utilizados em seu sistema, você irá descobrir que muitas partes diferentes do seu programa usam os mesmos tipos de objetos. Como resultado, para escrever seus programas em termos de objetos, você pode facilmente (e rapidamente) reutilizar o código que escreveu para uma parte diferente do seu programa ou até mesmo em um diferente programa.



Após você identificar os objetos, deve determinar o propósito da cada objeto. Para isso, imagine as operações que o objeto realiza ou as que são realizadas no objeto. Dado um arquivo objeto, um programa pode copiar, deletar, renomear o arquivo. Tais operações irão se tornar as funções-membro do objeto. Na seqüência, identifique a informação que você deve conhecer a respeito do objeto. No caso do arquivo objeto, você deve saber o nome do arquivo, o tamanho, atributo de proteção, data da última atualização, por exemplo. Estes itens de dados irão tornar-se nas variáveis-membro do arquivo objeto. A figura abaixo mostra como pode ser imaginado nosso arquivo objeto.

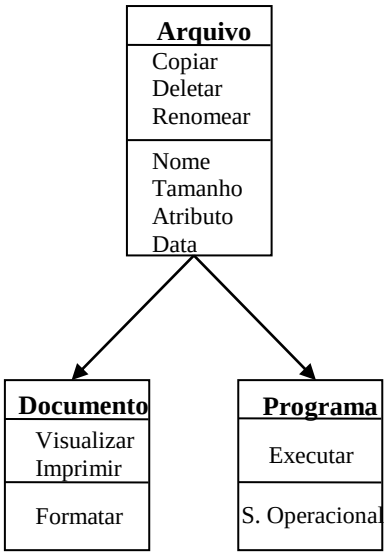


Você pode pensar que seu programa precisa visualizar e imprimir seus arquivos e que às vezes os arquivos contêm programas executáveis que você precisa executar. Existirá uma forte tendência em querer adicionar métodos (funções) para a definição do objeto arquivo acima. Entretanto, existem formas melhores para tratar tais casos. Uma solução seria construir duas novas classes de objeto, as quais, poderemos chamar de documentos e programas, e simplesmente adicionar os campos exigidos para cada classe, como mostra a figura a seguir.



Uma outra solução, muito melhor, seria construir as duas classes de objetos baseadas na classe de arquivo, como ilustrado a seguir:

Quando seus programas constróem uma classe a partir de outra, desta forma, a nova classe herda as funções-membro e variáveis da classe básica. No caso das classes documento e programa, os objetos que você criar destes tipos de classe não somente podem usar suas próprias funções-membro e variáveis, mas também as da classe arquivo. Assim, seu programa pode visualizar, imprimir, copiar, deletar, renomear um arquivo de documento.



Definições de programação orientada a objetos

Classes

A linguagem de programação C++ é uma derivação da linguagem de programação C. Logo, o conjunto de instruções que fazem parte da linguagem C, também está incorporada em C++. As principais implementações adicionadas à linguagem C para dar origem ao C++ consistem de classes, nos objetos e na idéia de programação orientada para objeto.

Na discussão anterior, os termos objetos e classes foram usados livremente. Em geral, uma classe oferece uma máscara com a qual seu programa pode mais tarde criar objetos. Por exemplo, a classe arquivo previamente discutida, descreve as funções-membro e variáveis que irão ser usadas pelos objetos-arquivo que seu programa criar no futuro.

Um objeto por esse motivo é uma instância de uma máscara da classe. Em outras palavras, um objeto é a entidade ou coisa com a qual seus programas trabalham.

Uma definição de classe em C++ é muito similar a uma definição de estrutura em C. Por exemplo, as seguintes declarações definem a classe arquivo.

```
class arquivo {
   public:
     char nome_arquivo[32];  //  variável-membro publica
     long tamanho;
   private:
     unsigned proteção;      // variável-membro privada
     long data;
     int copiar(char *nome_destino); //função-membro privada
     int renomear(char *nome_destino);
     int deletar(void);
}
```

Uma classe é considerada um tipo de dado como os tipos que existem predefinidos em compiladores de diversas linguagens de programação. Como exemplo, considere um

tipo *int* que é predefinido em C e C++. Podemos declarar tantas variáveis do tipo *int* quantas forem necessárias ao programa. De modo similar, pode-se declarar quantas variáveis quisermos de uma classe já definida. Como visto, uma classe não cria um objeto. Para criar um objeto, você simplesmente especifica o tipo da classe, seguido pelo nome da variável objeto, como mostrado abaixo:

```
main() {

     nome_da_classe nome_do_objeto; // cria um objeto
                                    // nome_do_objeto do
                                   // tipo nome_da_classe
          :
          :
     int var;  // definição de variável simples do tipo int

}
```

Assim, variável de uma classe é chamada objeto e conterá campos de dados e funções. Definir uma classe não cria nenhum objeto, do mesmo modo que a existência do tipo *int* não cria nenhuma variável. As funções de um objeto são chamadas funções-membro ou métodos e, de modo geral, são o único meio de acesso às variáveis-membro (campos de dados) também chamados de variáveis de instância. As classes são muito mais poderosos que as estruturas e uniões. Elas são formadas por variáveis-membro e funções-membro que operam esses dados.

Encapsulamento

Se um programa precisa atribuir valor a alguma variável de instância, deve chamar uma função-membro que recebe o valor como argumento e fazer a alteração. Não se pode acessar variáveis de instância diretamente. Assim, os campos de dados ficam escondidos para nós, o que previne alterações acidentais. Diz-se então, que os campos de dados e suas funções estão encapsulados, ou seja, tenham uma única identidade.

Esse processo de impedir a visibilidade dos dados para o mundo externo ao objeto é chamado de encapsulamento. Com a encapsulamento de dados os detalhes de

implementação ficam escondidos dos demais objetos, assim o programador preocupa-se apenas com “o que" um objeto faz, sem perder tempo entendendo "como" o objeto faz.
Desta forma, conclui-se que a encapsulamento aumenta a modularidade e simplifica a escrita, manutenção e alteração dos programas.

Mensagem

Objetos só são úteis quando associados com outros objetos para alcançar um resultado. E como os objetos interagem entre si? A resposta para essa pergunta é, através de mensagens. Para melhor entendimento, pense num objeto carro com um método para mudar sua velocidade. Em um programa para modelar o uso de um carro, poderíamos criar um objeto do tipo motorista e usá-lo associado ao objeto carro. Mas, como nosso motorista aumentaria a velocidade do carro? Com uma mensagem adequada para tal, uma mensagem "ACELERAR", por exemplo.
Uma mensagem possui basicamente três componentes:

- objeto para receber a mensagem (carro);
- um nome da operação (método/função-membro) a ser executada pelo objeto (acelerar);
- um parâmetro (argumento) do qual a velocidade pode depender (velocidade).

Um programa em C++ consiste em um conjunto de objetos que se comunicam por meio das chamadas às funções-membro e para chamar um método de um objeto deve-se enviar uma mensagem ao objeto da função-membro em questão.

Herança

A programação orientada a objetos oferece uma maneira de relacionar classes uma com as outras por meio de hierarquias. Logo, pode-se dividir classes em sub-classes, mantendo-se o princípio de que cada sub-classe herda características da classe, da qual foi derivada. Partindo-se do reino animal, a classe de animais é dividida nas sub-classes mamíferos, aves, peixes, etc. Uma das características dessa classe é a reprodução. Toda sub-classe tem essa característica. Em C++, a classe de origem é chamada classe-base e as classes que compartilham as características de uma classe-base e tem outras características adicionais são chamadas classes derivadas.
Uma classe-base representa os elementos comuns a um grupo de classes derivadas. Pode-se pensar em herança como algo semelhante ao uso de funções para simplificar

tarefas tradicionais. Escreve-se uma função quando identifica várias seções diferentes de um programa, em parte executam a mesma coisa. Em C++, define-se uma classe-base quando identifica-se características comuns em um grupo de classes derivadas. Da mesma forma você pode criar uma biblioteca de funções úteis a diversos programas, pode formar uma biblioteca de classes que poderão vir a ser o núcleo de muitos programas. O uso de biblioteca de classes oferece uma grande vantagem sobre o uso de uma biblioteca. Isto significa que, sem alterar a classe-base, é possível adicionar a ela características diferentes que a tornarão capaz de executar exatamente o que desejamos.
Uma classe que é derivada de uma classe-base pode, por sua vez, ser classe-base de outra classe. O uso de classes derivadas aumenta a eficiência da programação pela não necessidade da criação de códigos repetitivos. A uma função de biblioteca não se pode adicionar outras implementações a não ser que ela seja reescrita ou que tenhamos seu código-fonte para alterá-la e recompilá-la.
A facilidade com que classes existentes podem ser reutilizadas sem serem alteradas é uma dos maiores benefícios oferecidos por linguagens orientadas ao objeto.
A herança é o processo pelo qual criamos novas classes, chamadas classes derivadas.

Polimorfismo

O sentido da palavra polimorfismo provém do uso de um único nome para definir várias formas distintas. Em C++ dá-se o nome de polimorfismo à criação de uma família de funções virtuais. Uma função virtual é uma função membro de classe que é projetada para trabalhar com virtualmente quaisquer membros de classe-básica ou derivada (de tipo ainda desconhecido). Você pode pensar numa função virtual como alguma coisa como uma "função variável", uma função-membro derivada e ainda desconhecida. Retornaremos com esse assunto nos próximos capítulos, quando tivermos que colocar a mão na massa.
Por hora devemos nos lembrar apenas que, essa capacidade da linguagem C++ possibilitar várias formas a um dado objeto torna o C++ uma ferramenta muito poderosa e flexível para o desenvolvimento das aplicações.

Construtores e destrutores de classe

Ao criar um objeto, existem situações em que você precisa alocar memória para buffers usados pelo objeto. Por exemplo, assuma que você esta trabalhando com um arquivo objeto e necessita que armazene o nome do arquivo como uma cadeia de caracteres.

Ao criar o objeto, você quer que ele aloque memória dinamicamente para a cadeia, e quando ele não for mais necessário você desejará liberar a memória. Para auxiliar seus programas a realizar operações cada vez que um objeto é criado (construtor) e cada vez que ele for destruído (destrutor), C++ suporta funções de construção e de destruição. Ou seja, construtor nada mais é do que uma função-membro que automaticamente é executada cada vez que você cria um objeto e destrutor é uma função-membro que é automaticamente executada quando um objeto é destruído.

Assim o construtor é uma função-membro que C++ automaticamente executa cada vez que você criar um objeto de uma classe específica. O construtor é a única função que usa o mesmo nome da classe. Como tal, para uma classe de nome aluno, o construtor seria definido aluno. Além disso, a função não retorna um valor e não é do tipo *void*.

O destrutor é uma função que automaticamente executa quando um objeto é destruído. Como a função construtor, a função destrutor tem o mesmo nome que a classe. Também, a função destrutor não retorna um valor e não é do tipo *void*.

```
class Triangulo  {
   private:
       double fb, fh;
   public:
       /* função construtor */
       Triangulo(double base, double altura);

       /* função destrutor */
       ~Triangulo();

       double area(double b, double h);
};
```

```
#include <iostream.h>
#include <string.h>
class mensagem {
      public:
          mensagem(char *titulo);
          mensagem();
          void mostra_texto();
      private:
```

```
          char texto[64];
};
mensagem::mensagem(char *titulo) {
     strcpy(texto, titulo);
}
mensagem::mensagem(void) {
      strcpy(texto, "Os Sertoes");
}
void mensagem::mostra_texto(void) {
      cout << "Obra: " << texto << endl;
}
void main(void) {
       mensagem livro1;
       mensagem livro2("Introducao ao C++");
       livro1.mostra_texto();
       livro2.mostra_texto();
}
```

Seus programas não podem passar parâmetros para uma função destrutor. Distingue-se uma função construtor de uma função devido ao fato que o destrutor é precedido por um caractere til(~), mostrado acima.

Quando você define funções construtor nos seus programas, pode especificar múltiplas funções das quais o compilador irá selecionar função correta a invocar baseado no que seu programa usa. Por exemplo, o programa a seguir usa das funções construtor para a classe mensagem. O primeiro construtor atribui a mensagem especificada através do parâmetro mensagem, enquanto o segundo parâmetro utiliza a mensagem default “Os Sertões".

No exemplo acima, a criação do primeiro objeto *livro1* não especifica parâmetros. Portanto, o programa chama o construtor que funciona como *void* (sem parâmetros). Ao criar o objeto *livro2*, o programa passa uma cadeia de caracteres para o construtor. Por essa razão, o programa chama o construtor que suporta um parâmetro string (cadeia de caracteres).

Uma maneira de se obter o mesmo resultado seria trabalhar com argumentos *default* em construtores. Nesse caso você pode especificar valores para cada parâmetro do construtor. Assim a mesma rotina poderia ser reescrita como:

```
#include <iostream.h>
#include <string.h>
class mensagem {
      public:
          mensagem(char *titulo="O Alquimista");
         void mostra_texto();
      private:
          char texto[64];
};
mensagem::mensagem(char *titulo) {
     strcpy(texto, titulo);
}
void mensagem::mostra_texto(void) {
      cout << "Obra: " << texto << endl;
}

void main(void) {
       mensagem livro1;
       mensagem livro2("Introducao ao C++");
       livro1.mostra_texto();
       livro2.mostra_texto();
}
```

Inicializando variáveis em construtores

As funções construtor existem para auxiliar na inicialização dos membros da classe quando a classe é criada. Da forma como foi definido, C++ permite que se inicialize variáveis-membro, colocando o nome da variável e o valor desejado após um sinal de dois pontos simples, antes da declaração da função. O programa a seguir usa este formato de inicialização do construtor para inicializar três variáveis-membro com os valores 5, 6 e 7:

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

```
class objeto {
      public:
        objeto(void);
        void mostra_objeto(void);
     private:
        int n1;
        int n2;
        int n3;
        double n4;
};
objeto::objeto(void) : n1(5), n2(6), n3(7) {
     n4 = 125.35;
}
void objeto::mostra_objeto(void) {
     cout << "n1: " << n1 << endl;
      cout << "n2: " << n2 << endl;
      cout << "n3: " << n3 << endl;
     cout << "n4: " << n4 << endl;
}

void main(void) {
      objeto numero;
      numero.mostra_objeto();
}
```

Acesso a variáveis nas classes

Quando você declara uma classe, pode definir membros como *public* ou *private* ou *protected*. Membros públicos da classe estão disponíveis através de seu programa utilizando o nome do objeto com os operadores ponto (*objeto.membro*) ou indireção (*objeto->membro*). Membros privados da classe, por outro lado, somente podem ser acessados utilizando funções-membro da classe. Ao utilizar membros privados da classe, seus programas podem melhor controlar os valores atribuídos aos membros da classe e a forma como esses valores são utilizados. Por default, todos os membros da classe são privados. Assim, colocando o rótulo *private* na declaração da classe, como já experimentado em exemplos anteriores, as variáveis a seguir serão restritas ao uso somente pela classe a que elas pertencem, como mostrado aqui:

```
class qualquer_classes {
      /* métodos públicos */
      public:
           void qualquer_funcao(char *paramatro);
           void qualquer_outra(int a, int b, int c);
      /* variáveis privadas */
```

```
      private:
           int valor;
           char senha[32];
};
```

Objetos e funções

Você pode passar um objeto para uma função por valor ou por referência (caso seja necessário mudar o valor de um membro). Da mesma forma, funções podem retornar objetos. O programa a seguir passa um objeto *mensagem* para duas diferentes funções. A primeira função utiliza chamada por valor para exibir as variáveis-membro do objeto. A segunda função utiliza por referência para alterar as variáveis-membro:

```
#include <iostream.h>
#include <string.h>
class Mensagem {
      public:
         Mensagem(char *mensagem, char *autor);
      ~Mensagem();
         void Mostra_mensagem(void);
         char mensagem_tit[64];
      private:
      char mensagem_autor[64];
};

Mensagem::Mensagem(char *mensagem, char *autor) {
      strcpy(mensagem_tit, mensagem);
      strcpy(mensagem_autor, autor);
}

Mensagem::~Mensagem() {
}

void Mensagem::Mostra_mensagem(void) {
      cout << "Usuario: " << mensagem_autor << endl;
      cout << "Mensagem: " << mensagem_tit << endl;
```

```
}

void Apoio_usuario(Mensagem observe) {
      observe.Mostra_mensagem();
}

void Resposta_curso(Mensagem *observe) {
      strcpy(observe->mensagem_tit, "Curso Lotado!");
      observe->Mostra_mensagem();
}

void main(void) {
      Mensagem usuario("Inscricao curso C++", "Marcos");
      Apoio_usuario(usuario);
      Resposta_curso(&usuario);
}
```

Arrays de classes

Como existem casos em que sua classe pode conter um array de valores, existem outras situações em que seu programa usa um array para pegar múltiplos objetos. Por exemplo, o programa a seguir, usa um array para pegar cinco objetos data:

```
#include <iostream.h>
#include <string.h>
class Data {
      public:
          Data(int mes, int dia, int ano);
          Data(void);
          ~Data(void);
          void Mostra_data(void);
      private:
          int mes;
          int dia;
          int ano;
```

```
};

Data::Data(int mes, int dia, int ano) {
          Data::mes = mes;
          Data::dia = dia;
          Data::ano = ano;
          cout << "Construtor de data: ";
          Mostra_data();
}
Data::Data(void) {
          cout << "Construtor de data sem data " << endl;
}

Data::~Data(void) {
          cout << "Destrutor de data: ";
          Mostra_data();
}
void Data::Mostra_data(void) {
          cout << mes << '/' << dia << '/' << ano << endl;
}
void main(void) {
          Data feriados[3];
          Data ano_novo(01, 01, 97);
          Data carnaval(02, 15, 97);
          Data natal(12, 25, 97);
          feriados[0] = ano_novo;
          feriados[1] = carnaval;
          feriados[2] = natal;
}
```

Quando você usa um array de objetos, precisa ficar claro que C++ chama as funções construtor e destrutor para cada elemento do array. No caso acima, as funções construtor e destrutor só exibem uma mensagem, deixando você perceber que elas foram chamadas.

Herança de códigos em C++

Conforme define-se classes nos seus programas, você talvez ache que classes geralmente possuem características similares. Levando vantagem de tais relacionamentos de objeto, você pode significativamente reduzir a quantidade de código de programa, não só reduz seu tempo de programação, mas também melhora a legibilidade, pois menos código resulta em menor possibilidade de se cometer erros.

Classes derivadas

Ao construir classes baseadas nos relacionamentos entre objetos, seus programas levam vantagem a herança - uma classe derivada pode herdar características de sua classe básica. O exemplo a seguir utiliza-se da herança para adicionar as coordenadas x e y a uma classe que é derivada da classe que contém as variáveis-membro x e y.

```
#include <iostream.h>
class Ponto2D {
       public:
          Ponto2D(double x, double y);
         ~Ponto2D();
       protected:
          double fx;
          double fy;
};
class Ponto3D: public Ponto2D {
     public:
        Ponto3D(double x, double y, double z);
        ~Ponto3D();
        double coord_x() { return fx; }
        double coord_y() { return fy; }
        double coord_z() { return fz; }
     private:
        double fz;
};
```

```
Ponto2D::Ponto2D(double x, double y) {
       fx = x;
       fy = y;
}
Ponto2D::~Ponto2D() {
}
Ponto3D::Ponto3D(double x, double y, double z) : Ponto2D(x,
y) {
           fz = z;
}
Ponto3D::~Ponto3D() {
}
int main(void) {
     double x,y,z;
     cout << "Coordenadas: " << "\n" << "x = ";
     cin >> x;
     cout << "y = ";
     cin >> y;
     cout << "z = ";
     cin >> z;
     Ponto3D *c = new Ponto3D(x,y,z);
     cout << "(" << c->coord_x() << "," << c->coord_y() <<
",";
     cout << c->coord_z() << ")" << endl;
     return 0;
}
```

Ao criar uma classe derivada, você especifica o nome da classe derivada, seguido de dois pontos e o nome da classe básica, como mostrado aqui:

class derivado: public base{
funções e variáveis-membros
};

Quando você usa a palavra-chave *public* desta forma, os membros públicos da classe básica são consideradas públicos na classe derivada. Igualmente, membros *protected* da classe básica são tratados como membros *protected* na classe derivada. Se suas

definições de classe usarem a palavra chave *private*, por sua vez, os membros *public* e *protected* da classe básica serão tratados como membros *private* da classe derivada.

Quando você define classes em seus programas, existem situações em que duas ou mais classes compartilham características (membros) similares. Antes de duplicar os membros de cada classe, defina uma classe básica que contém as funções-membro comuns. Em seguida você pode derivar classes a partir da classe básica. Quando você define classes dessa forma, a classe derivada herda as características da classe básica.

Suponha que você tenha criado uma classe veículo:

```
class veículo {

  public:
    veiculo(char *nome,int rodas,int potencia,int velocidade);
    void mostra_veiculo(void);

  private:
    char nome[64];
    int potencial;
    int velocidade;

};
```

Com o auxílio da herança, pode-se declarar uma classe automóvel, como segue:

```
class automovel: public veiculo {


     public:
        automovel(char *name, int potencia, int velocidade,
int portas);
        void mostra_auto(void);
     private:
        int portas;


};
```

De forma similar, pode-se declarar uma classe motocicleta, como segue:

```
class motociclo : public veiculo {
```

```
 public:
   motociclo(char *name,int potencia,int velocidade,int portas);
    void mostra_ciclo(void);
 private:
    int assentos;
};
```

Herança múltipla

Herança múltipla é o uso de duas ou mais classes básicas para derivar uma nova classe. Quando você deriva uma classe utilizando duas ou mais classes básicas, especifica o nome da classe derivada, seguido pelas classes básicas, como mostrado abaixo:

class derivada : public basical, public basica2 {

funções e variáveis-membros

};

Por exemplo, suponha que você tenha declarado as seguintes classes *livro* e *disco:*

```
class livro {
      public:
            livro(char *titulo, char *autor, int edicao);
            void mostra_livro(void);
      private:
            char titulo[64];
            char autor [64];
            int edicao;
};
class disco {
      public:
             disco(double espaco);
             void mostra_disco(void);
      private:
            double espaco;
};
```

A seguir, criar-se uma classe de nome *pacote* que é a combinação de *livro* e *disco*:

```
class pacote : public livro, public disco {
   public:
     pacote(char *tit, char *autor, int edicao, double
cap,double preco);
      void mostra_pacote(void);

   private:
      double preco;
};
```

Assim, *pacote* é a classe derivada, e *disco* e *livro* são as classes básicas.

Observe o uso da palavra-chave *public* que antecede cada classe básica. *public* permite que as classes derivadas tratem os membros *public* e *protected* da classe básica como membros *public* ou *protected* da classe derivada.

```
derivada::derivada(int a, int b) : basica1(a), basica2(b) {
        declarações
};
```

Nesse caso, quando seu programa chamar o construtor da classe derivada, C++ irá chamar primeiro o construtor da classe básica1, seguido pelo construtor da classe básica2.

Herança de múltiplos níveis

Como visto, herança múltipla é o uso de duas ou mais classes básicas derivadas em uma nova classe. Herança de múltiplos níveis, por outro lado, ocorre quando você deriva uma classe de uma classe básica que na realidade é uma classe derivada de uma outra.

A seguir mostra-se um exemplo de herança por múltiplos níveis. A classe *pesquisador* é baseado na classe *professor*, que por sua vez, é baseado na classe *servidor*.

Inicialmente, para criar estas classes deve-se fazer uma análise de quais dados elas possuem em comum. A seguir colocar estes dados comuns nas classes básicas e com o aumento da necessidade ir expandindo o código. Você pode começar com a classe

servidor, a seguir define-se a classe *professor* derivada de *servidor* e finalmente deriva-se a classe *pesquisador* da classe *professor*:

```
1:  class servidor {
2:     public:
3:          servidor(char *nome, int idade) ;
4:          void mostra_servidor(void) ;
5:     private:
6:          char nome[64] ;
7:          int idade;
8:   };
9:
10:  class professor : public servidor {
11:    public:
12:         professor(char *nome, int idade, double salario) ;
13:         void mostra_professor(void) ;
14:    private:
15:         double salario;
16:  };
17:  class pesquisador : public professor {
18:    public:
19:         pesquisador(char *nome, int idade, double salario, char *pesquisa) ;
20:         void mostra_pesquisador(void) ;
21:    private:
22:         char pesquisa[64];
23:  };
24:
25:  servidor::servidor(char *nome, int idade) {
26:          strcpy(servidor::nome, nome) ;
27:          servidor::idade = idade;
28:  }
29:
30:  void servidor: :mostra_servidor(void) {
31:          cout << "Nome: " << nome << endl;
32:          cout << "Idade: " << idade << endl;
33:  }
34:
35:  professor::professor(char *nome, int idade, double salario) : servidor(nome, idade)
{
36:          professor::salario = salario;
37:  }
38:
39:  void professor::mostra_professor(void)  {
40:          mostra_servidor();
41:          cout << "Salário: " << salario << endl;
42:  }
43:
44:  pesquisador::pesquisador(char *nome, int idade, double salario, char *pesquisa)
45:                                                  : professor(nome, idade, salario) {
46:          strcpy(pesquisador::pesquisa, pesquisa);
47:  }
48:
49:  void pesquisador::mostra_pesquisador(void) {
50:      mostra_professor();
51:      cout  <<  "Linha de Pesquisa: " << pesquisa << endl;
52:  }
53:
54:  void main(void) {
55:     professor quimica("Fulano", 56, 3000.00);
56:     pesquisador mecanica("Beltrano", 34,  2400.00, "elementos finitos");
57:     química.mostra_professor();
58:     pesquisador.mostra_pesquisador();
59: }
```

Como você deve ter percebido no exemplo acima, quando seus programas usam herança, você permanece utilizando construtor para inicializar as classes. A única diferença, entretanto, é que o construtor da classe derivada deve chamar o construtor da classe básica primeiro, como ilustra as linhas de código 35 e 44/45. A função

construtora da classe derivada usa uma sintaxe similar para a definição de classe, com um dois pontos seguido pela função construtora da classe básica. Desta forma, os membros da classe básica são corretamente inicializados e a memória para os buffers da classe básica é corretamente alocada antes que a classe derivada os referencie. Ainda nestas mesmas linhas, como se pode ver, a chamada ao construtor da classe básica usa os nomes de parâmetros idênticos aos que foram passados para o construtor da classe derivada.

Quando você deriva uma classe de uma outra, a classe derivada não pode acessar membros *private* da classe básica. Dependendo de seus programas, existem situações em que você quer dar acesso especial aos membros das classes básicas às classes derivadas. Em tais casos, você pode usar membros *protected* da classe, os quais são acessíveis pelas classes derivadas mas não pelo resto do programa. Por exemplo, dada a seguinte definição de classe, classes derivadas podem acessar os variáveis-membros *nome* e *profissao*, mas não o membro private *salario*:

```
class base  {
      public:
            base(char nome[64], char profissao[64]);
            void mostra_base(void) ;
      protected:
            char nome[64];
            char profissao[64];
      private:
            double salario;
};
```

Funções virtuais

Como já mencionado anteriormente, polimorfismo em C++ é baseado em funções virtuais. Em seu sentido mais simples, uma função *virtual* é uma função-membro que as classes derivadas podem modificar, permitindo que sejam adaptadas às necessidades do programa.

Muitas vezes essas funções não fazem absolutamente nada na classe básica, estão ali somente para que possam ser referenciadas na classe básica e implementada

posteriormente pelas classes derivadas. Para declarar uma função *virtual*, coloque antes da declaração, a palavra-chave *virtual*. Por exemplo, a seguinte classe *servidor* utiliza uma função-membro *virtual void mostra_servidor*:

```
class servidor {
      public:
            servidor(char *nome, int idade);
            virtual void mostra_servidor(void);
      private:
            char nome[64];
            int idade;
};
```

Desta forma, seu programa pode derivar classes *professor* e *pesquisador*, cada uma das quais referenciando uma função-membro *mostra_servidor* específica.

Se você não antecipar o nome de uma função-membro da classe básica com a palavra-chave *virtual*, polimorfismo não irá ocorrer. Além disso, se o tipo de retorno ou os tipos dos argumentos da função-membro da classe derivada não são os mesmos, o polimorfismo também não ocorrerá. O exemplo a seguir mostra o bom uso de uma função *virtual* para realizar o polimorfismo.

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

#define PI 3.141592
#define CICLO 1

class Elemento {
   public:

     virtual void Area()  = 0;
     virtual void Dados() = 0;
     virtual void Print() = 0;

};


class Retangulo : public Elemento {
   public:
     Retangulo();
```

```
     ~Retangulo();
      void Dados();
      void Area();
      void Print();
   private:
     double fb;
     double fh;
};

class Circulo : public Elemento {
   public:
     Circulo();
     ~Circulo();
     void Dados();
      void Area();
      void Print();
   private:
     double fr;
};

class Triangulo : public Elemento {
   public:
     Triangulo();
     ~Triangulo();
     void Dados();
      void Area();
      void Print();
   private:
     double fb;
     double fh;
};

Retangulo::Retangulo() {
}
Retangulo::~Retangulo() {
}

void Retangulo::Dados() {
     int b,h;
     cout << "Retangulo" << endl;
     cout << "b = ";
     cin >> b;
     cout << "h = ";
     cin >> h ;
     fb = b;
     fh = h;
}

void Retangulo::Area() {
     cout << "Area: " << fb*fh << endl;
}
```

```
void Retangulo::Print() {
     cout << "Dados do Retangulo:" << endl;
     cout << "b= " << fb << " h= " << fh << endl;
     cout << "Area: " << fb*fh << endl;
}
Circulo::Circulo() {
}

Circulo::~Circulo() {
}

void Circulo::Dados() {
     int r;
     cout << "Circulo" << endl;
      cout << "r = ";
      cin >> r;
     fr = r;
}

void Circulo::Area() {
      cout << "Area: " << PI*fr*fr << endl;
}

void Circulo::Print() {
     cout << "Dados do Circulo:" << endl;
     cout << "r= " << fr << endl;
      cout << "Area: " << PI*fr*fr << endl;
}

Triangulo::Triangulo() {
}

Triangulo::~Triangulo() {
}

void Triangulo::Dados() {
     int b,h;
      cout << "Triangulo" << endl;
      cout << "b = ";
      cin >> b;
     cout << "h = ";
     cin >> h;
     fb = b;
     fh = h;
}

void Triangulo::Area() {
     cout << "Area: " << fb*fh/2.0 << endl;
}
```

```
void Triangulo::Print() {
     cout << "Dados do Triangulo:" << endl;
     cout << "b= " << fb << " h= " << fh << endl;
     cout << "Area: " << fb*fh/2.0 << endl;
}

main() {
     Elemento *forma[2];
     char tipo = ' ';
     for(int i=0;i<3;i++) {
          cout<<"Digite uma letra R, C, T ou zero(0) para
Sair"<<endl;
          cin >> tipo;
          switch(tipo) {
               case 'r':
               case 'R':
                    forma[i] = new Retangulo;
               break;
               case 'c':
               case 'C':
                    forma[i] = new Circulo;
               break;
               case 't':
               case 'T':
                    forma[i] = new Triangulo;
               break;
               case '0':
                    exit(0);
               default:
                    cout << endl;
               continue;
          }
          forma[i]->Dados();
          forma[i]->Area();
     }
     for(i=0;i<3;i++) {
          forma[i]->Print();
          delete forma[i];
     }
}
```

Não confunda *polimorfismo* com *sobrecarga* de funções. Uma vez que a definição da classe básica não usa a palavra-chave *virtual*, as declarações do membro na classe derivada simplesmente sobrecarregam a declaração da função. As funções não são virtuais.

Sobrecarga de funções é o processo de definir duas ou mais funções, como mesmo nome, nos seus programas. As funções diferem somente pelo número ou tipo dos

parâmetros que elas suportam. Ao compilar seu programa, o compilador C++ determina qual função invocar, baseado nos parâmetros da função. Assim, seus programas sempre chamam a função correta automaticamente.

Funções amigas

O conceito de isolamento dos itens de uma classe, onde funções não membro não teriam o privilégio de acesso aos dados privados ou protegidos desta classe, é violado por um mecanismo oferecido pelo C++. Este mecanismo dá permissão especial de acesso aos itens por essas funções não membros especiais. Declarando uma função como *friend* (amiga), ela passa a ter os mesmos privilégios de uma função-membro da classe, mas não está associada a um objeto da classe.
O programa a seguir faz uso de funções *friend*:

```
#include <iostream.h>

class tempo {
 private:
   long segundos;
 public:
   tempo(long h,long m,long s) {segundos = h*3600+m*60+s; }
   friend double prntm(tempo); //declaracao da funcao amiga
};

double prntm(tempo); // declaracao da funcao

void main() {
     tempo tm(5,8,20);
     cout << prntm(tm) << " % do dia" << endl;
}

double prntm(tempo tm) {
     double x;
     x = tm.segundos/86400.;  x *=100;
     return x;
}
```

Neste exemplo, ao declararmos a função *prntm* como *friend* da classe *tempo*, foi possível ter acesso ao dado privado *segundos* da classe *tempo*. A declaração da classe *friend* pode ser em qualquer posição da classe.

Classes amigas

Além de ser possível declarar funções independentes como *friend*, pode-se declarar uma classe toda como *friend* de outra. Neste caso, as funções-membro da classe serão todas *friend* da outra classe. A diferença é que estas funções-membros possuem acesso à parte privada ou protegida de sua própria classe. O programa a seguir usa esse conceito para disponibilizar os membros da classe *friend*:

```
#include <iostream.h>
class tempo {
 private:
   long h, m, s;
 public:
   tempo(int hh, int mm, int ss) { h=hh; m=mm; s=ss; }
   friend class data;    // data , uma classe amiga
};
class data {
 private:
   int d, m, a;
 public:
   data(int dd, int mm, int aa) { d=dd; m=mm; a=aa; }
   void prndt(tempo tm) {
     cout << "\nData: " << d << "/" << m << "/" << a;
     cout<<"\nHora: "<<tm.h<<":"<<tm.m<<":"<<tm.s<<endl;
   }
};
void main() {
     tempo tm(07,18,30), tm1(23,18,30);
     data dt(02,07,97);
     dt.prndt(tm);
     dt.prndt(tm1);
}
```

Outras ferramentas e características

O ponteiro this

A palavra-chave *this* é um ponteiro disponível para todos os métodos de uma classe. O ponteiro *this* endereça a instância da classe que chamou a função. A palavra-chave *this* indica um ponteiro auto-referido que é declarado implicitamente pela linguagem. Por exemplo:

*TClasse *this; // onde TClasse é o tipo da classe*

O ponteiro *this* aponta para o objeto para o qual a função-membro é invocada. No exemplo a seguir o ponteiro *this* acessa a variável de classe *caracter*.

```
class Teste {
      char caracter;
      Teste(char ch) { caracter=ch; }
      char SetChar(char a)
        { caracter = a; return this->caracter; }
      void Start(Teste *ptr) { ptr->caracter = ' '; }
   public:
      void Reset() { Start(this); }
};
```

No exemplo acima a função *Reset* usa o ponteiro *this* para passar o objeto em questão como parâmetro da função *Start*.

Variáveis estáticas (static)

Uma variável estática tem comportamento semelhante ao uma variável global. Ao declarar uma variável como estática (*static*), apenas uma cópia da variável (variável-membro) existe na memória. Uma variável-membro estática pode ser de qualquer tipo comum do C++ (por exemplo, *int*, *double*, *float*) ou uma outra classe.

Se uma variável-membro estática é declarada como pública, as instruções externas à classe podem se referir a esta variável como uma instrução tal como *classe::variável_estática = x*.

Onde x é um valor a ser atribuído a variável-membro de acesso público da classe. Entretanto, assim como com a maioria dos campos de dados e pelas mesmas razões

discutidas anteriormente (acesso às variáveis de classe, encapsulação), em geral é mais apropriado declararmos variáveis estáticas em áreas privadas e protegidas, permitindo assim apenas às funções-membro da classe ou suas derivações acesso as variáveis estáticas.

Dependendo do propósito do uso de seu objeto, pode existir a necessidade de dois ou mais objetos compartilharem a mesma variável-membro. Em tais casos, seus programas podem criar uma variável-membro compartilhada e isto se faz precedendo a declaração da variável na classe com a palavra-chave *static*. Por exemplo, a seguinte classe *Trabalhador*, objetos compartilham as variáveis *companhia_seguro* e *plano_saude*:

```
class empregado
     public:
          Trabalhador(char *nome, long id, int tarefa);
          void mostra_trabalhador(void);
     private:
          char nome[64];
          long id;
          int tarefa;
          static char *companhia_seguro;
          static char *plano_saude;
};
```

Em seguida, externamente à declaração da classe, deve-se especificar declarações para as variáveis, como as que seguem:

*char *Trabalhador::companhia_segura = " ";*

*char *Trabalhador::plano_saude = " ";*

Onde cada objeto do tipo *Trabalhador* que for criado mais tarde no programa irá compartilhar estas duas variáveis.

Funções-membro estáticas

Funções estáticas tem acesso apenas a variáveis de dados estáticas da classe. Uma função-membro estática não recebe um ponteiro *this* e portanto, não pode fazer referência a quaisquer variáveis de dados de um objeto da classe.

O principal uso de funções-membro estáticas é para inicialização de um aspecto global da classe que deveria ser aplicado igualmente a todas as variáveis de sua classe. Armazenar os passos globais de inicialização em uma função-membro estática torna possível para um programa inicializar a classe para todas as variáveis futuras.
Funções *static* podem ser chamadas diretamente sem o uso de um objeto:

```
class Test{
     static void Init() { ..... }
};

void main(){
     Test::Init();
}
```

Sobreposição de métodos

Muitas vezes é necessário redefinir uma ação quando se faz uma derivação, ou seja, a classe base age de uma maneira à uma mensagem mas a classe derivada deve agir de uma outra forma. Em C++ podemos redefinir um método da classe base na classe derivada. Se o nosso ponto 2D possuir um método para mudar a posição:

```
class Ponto2D {
     double x;
     double y;
     void posicao(double valx, double valy) {
          x = valx;
           y = valy;
     }
};
```

A nossa classe de ponto 3D deve ter um método diferente para acertar a posição:

```
class Ponto3D : public Ponto {
     double z;
```

```
     void posicao(double valx, double valy, double valz) {
          x = valx;
          y = valy;
          z = valz;
     }
};
```

Sobreposição de operadores

A sobreposição de operadores é um mecanismo que nos permite acrescentar novos tipos de dados àqueles que o C++ foi projetado para manipular. Com a sobreposição de operadores, pode-se criar uma classe e escrever funções que implementam uma operação de soma para duas instâncias do tipo da classe. Após sobrepor o operador de soma, podemos escrever expressões usando o sinal de mais familiar, e o C++ chama funções de operador personalizado para adicionar as instâncias da classe.

As funções amigas a pouco discutidas são especialmente convenientes quando usadas para redefinir operadores. O programa seguinte mostra a limitação do uso de funções operadoras quando funções amigas não são usadas:

```
#include <iostream.h>

class ponto {
   private:
     int x, y;
   public:
     ponto() { x=0; y=0; }
     ponto(int  x1, int y1) { x=x1; y=y1; }
     ponto operator + (ponto p) {       // funcao operadora
           return ponto(x+p.x, y+p.y);
     }
     ponto operator +(int n) {          // funcao operadora
       return ponto (x+n, y+n);
     }
     void printpt() {
```

```
        cout << "(" << x << "," << y << ")" << endl;
     }
};

void main() {
     ponto p1(5,1), p2(2,3), p3;
     p3 = p1 + p2;
     p3.printpt();
     p3 = p1 + 5;
//     p3 = 5 + p1;     Erro, pois o primeiro operando , uma
constante
     p3.printpt();
}
```

LEITURA E GRAVAÇÃO EM ARQUIVOS

Leitura de dados em um arquivo

Para ler entrada a partir de um arquivo seu programa deve criar um objeto do tipo *ifsteam*, o qual é definido no arquivo de cabeçalho *fstream.h*. Em seguida, o programa deve abrir o fluxo de arquivo utilizando a função-membro *open* ou a função construtora *ifstream*. Seu programa pode então usar o operador de extração para ler dados do arquivo. Após ter realizado a operação de leitura do arquivo, ele deve ser fechado utilizando a função-membro *close*.

O programa a seguir ilustra os passos envolvidos na leitura de um arquivo:

```
#include <iostream.h>
#include <fstream.h>  //Inclui o arquivo FSTREAM.H
void main(void) {
     int dado01, dado02, dado03;
     char texto[256];
```

```
     ifstream entrada("dados.dat");
     entrada.getline(texto, sizeof(texto));
     entrada >> dado01 >> dado02 >> dado03;
     cout << "dado01: " << dado01 << " dado02: ";
     cout << dado02 << endl;
     entrada.close();
}
```

Gravação de dados em um arquivo

Para escrever a saída em um arquivo, seu programa deve criar um objeto do tipo *ofstream*, o qual é definido no arquivo de cabeçalho *fstream.h*. A seguir o programa pode usar a instância (objeto) criada do tipo *ofstream* para associar o objeto ao arquivo em disco. O programa pode então usar o operador de inserção para escrever os dados para o arquivo. Após o programa ter escrito sua saída, deve fechar o arquivo utilizando a função-membro *close*. O programa a seguir ilustra os passos envolvidos para gravar os dados no arquivo:

```
#include <iostream.h>
#include <fstream.h>
void main(void) {
     /* torna saída um tipo ofstream */
     ofstream saida("dados.res");
     saida << "Informacoes a serem armazenadas no arquivo";
     saída << end1;
     saida << "Corpo do texto ..." << endl;
     saida << "Ultima linha do texto!" << endl;
     saida.close();
}
```

CLASSE TMatrix ORIENTADA A OBJETOS

VISÃO GERAL

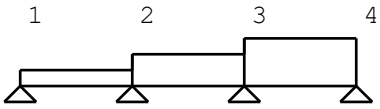
Existem atualmente muitas bibliotecas que manipulam matrizes com muita eficiência, tais como LAPACK, LINPACK, EISPACK[1]. Porém, essas bibliotecas foram concebidas com uma filosofia estrutural, o que as torna difíceis de serem manipuladas e estendidas. Suas novas versões "orientadas a objeto", como por exemplo LAPACK++, são adaptações das versões estruturadas e não incorporam as vantagens oferecidas por um projeto orientado a objetos. Outro inconveniente dessas bibliotecas é usar muitos argumentos na chamada de suas funções, tornando seu uso tedioso.

O objetivo da criação da classe **TMatrix** foi o desenvolvimento de uma biblioteca de uso geral, orientada a objetos, que:

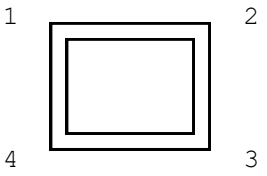
- manipule matrizes com diferentes padrões de armazenamento;
- seja fácil de usar e portátil para diferentes plataformas;
- que use os conceitos de encapsulação, herança, abstração de dados;
- que seja, sobretudo, de fácil manutenção.

Os cálculos de elementos finitos geram sistemas lineares de equações [K][u]=[F], onde a matriz de rigidez (K) pode ser simétrica ou não-simétrica, dependendo da equação diferencial modelada. Ainda segundo a estrutura da malha, K pode ser melhor armazenada em estrutura de Banda ou Skyline (ver FIGURA 0-1). Isso mostra a necessidade de ter uma biblioteca que manipule diferentes tipos de matrizes.

[1] Maiores informaçoes sobre essas bibliotecas podem ser encontradas em ANDERSON et al. (1995) e BUNCH et al. (1979).



	V_1	θ_1	V_2	θ_2	V_3	θ_3	V_4	θ_4
V_1	x	x	x	x				
θ_1	x	x	x	x				
V_2	x	x	x	x	x	x		
θ_2	x	x	x	x	x	x		
V_3			x	x	x	x	x	x
θ_3			x	x	x	x	x	x
V_4					x	x	x	x
θ_4					x	x	x	x



	V_1	θ_1	V_2	θ_2	V_3	θ_3	V_4	θ_4
V_1	x	x	x	x			x	x
θ_1	x	x	x	x			x	x
V_2	x	x	x	x	x	x		
θ_2	x	x	x	x	x	x		
V_3			x	x	x	x	x	x
θ_3			x	x	x	x	x	x
V_4	x	x			x	x	x	x
θ_4	x	x			x	x	x	x

FIGURA 0-1- Exemplos de matriz de rigidez.

A biblioteca chamada de ***TMatrix*** contém as seguintes matrizes:

1. matrizes não simétricas:
    - matriz cheia;
    - matriz banda;
    - matriz esparsa;
2. matrizes simétricas:
    - matriz cheia;
    - matriz banda;
    - matriz skyline;
    - matriz esparsa;
3. condensação estática.

ESTRUTURA DAS CLASSES TMatrix

Uma matriz é um arranjo ordenado de dados que permite diferentes tipos de operações sobre o mesmo. Com a intenção de abstrair a idéia de matriz, existe uma classe base que define um conjunto de métodos padrão para todos os tipos de matrizes. Porém, cada um destes métodos poderá ter uma implementação apropriada segundo o tipo de matriz.

Esta biblioteca possui vários tipos de matrizes[2], cada uma com uma alocação de memória apropriada, assim como algoritmos especializados para manipular estas alocações. Estas particularidades são ou não implementadas nas classes derivadas. Existe ainda outra classe derivada que serve de classe base para as matrizes do tipo simétricas. Estas duas classes base são do tipo virtual, isto é, não fazem nenhum tipo de alocação, apenas perfilam o comportamento das classes filhas.

Na FIGURA 0-2 pode-se observar a estrutura desta classe. Tem-se ainda que o usuário é livre para derivar suas próprias classes.

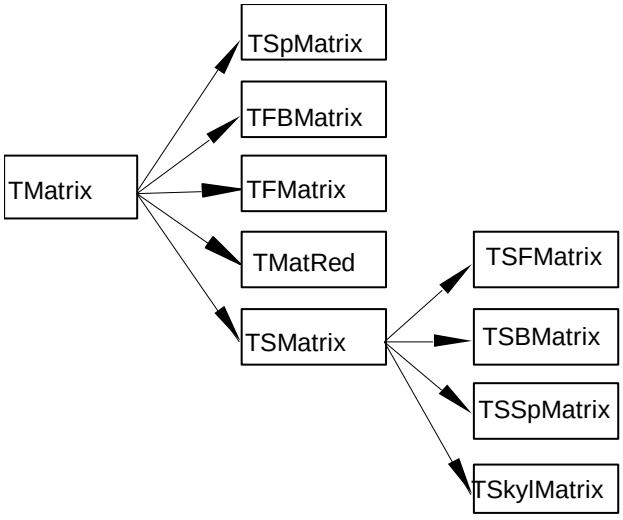


FIGURA 0-2- A estrutura das classes derivadas de TMatrix.

A CLASSE BASE TMatrix

Esta é a classe base da qual os diferentes tipos de matrizes foram derivadas. Nesta classe define-se o comportamento que será implementado pelas classes derivadas. Na classe **TMatrix** também estão definidos vários tipos de *flags*, que contêm informações sobre o "estado" da matriz, como por exemplo, se a matriz foi decomposta e qual decomposição

[2] A classe TMatrix é a classe base dessa biblioteca, que contém diferentes tipos de matrizes derivadas. Daqui em diante, o termo as classes TMatrix estará se referindo à biblioteca inteira.

Exemplos de utilização desta biblioteca podem ser obtidas na pagina internet http://www.cenapad.unicamp.br/~phil/matrix.htlm.

foi utilizada. A obtenção deste dado é justificável, porque a matriz resultante de uma decomposição (Cholesky, LU, LDLt, etc.) é guardada na própria matriz.

Também foi admitido que qualquer operação algébrica entre matrizes retorna uma matriz cheia. Esta limitação cobre 99% dos casos e evita a implementação de um grande número de métodos virtualmente inúteis (por exemplo multiplicação matriz "skyline" com matriz esparsa, banda com "skyline", etc.).

Várias operações com sub-matrizes foram implementadas, como por exemplo colocar uma sub-matriz dentro de uma matriz, copiar uma sub-matriz para uma matriz e somar uma sub-matriz com uma matriz. As mesmas operações com sub-matrizes também poderiam ser feitas com objetos do tipo **TBlock**, porém foram implementadas para matrizes para que o programador não precise definir um objeto do tipo **TBlock** se for realizar uma operação simples com uma sub-matriz.

A classe **TMatrix** não aloca memória para armazenar os elementos, é uma classe virtual. Mesmo assim, utilizando os métodos GetVal e PutVal (ou o operador () sobrecarregado), vários métodos de decomposição foram implementadas. Isso implica que qualquer classe derivada da classe **TMatrix** dispõe de um conjunto de algoritmos de decomposição. Porém, para melhorar a eficiência da implementação, a maioria dos métodos de decomposição são redefinidos nas classes derivadas. As decomposições simétricas, Cholesky, LDLt, usam a parte inferior da diagonal da matriz, assumindo que a matriz que a está chamando é simétrica.

Existem três métodos que precisam ser implementados nas classes derivadas. Estes são:

virtual void PutVal(int row, int col, REAL value).

-Torna o elemento (row,col) igual a value;

virtual REAL &GetVal(int row,int col).

-Retorna uma referência ao elemento (row,col);

virtual void Mult(TFMatrix &matin, TFMatrix &matout, int opt).

-Este método implementa a multiplicação entre a matriz corrente e matin; o resultado desta operação é retornado em matout. A opção opt indica se a matriz corrente ou sua transposta será usada na multiplicação.

Quando uma classe é derivada da classe **TMatrix**, o usuário tem a opção de definir PutVal e GetVal e/ou Mult. Dependendo do método a ser implementado, esta classe derivada pode ganhar certa funcionalidade ou não. Em um certo nível de abstração, uma matriz é uma transformação de um vetor em R^n em outro vetor em R^m. Assim, é suficiente definir esta transformação e a matriz existirá sem a capacidade de retornar e

colocar elementos dentro de si mesma. A matriz que implementa esta transformação em Mult apenas pode ser invertida somente com métodos iterativos. Classes que implementam PutVal e GetVal não precisam implementar Mult**,** porque a classe **TMatrix** implementa este comportamento. Não obstante, para aumentar o desempenho da classe derivada pode-se redefinir Mult tirando vantagem da esparsividade da matriz. A chamada de um método que use PutVal e GetVal dentro de uma classe em que não tenham sido implementadas, pode causar erro e a saída da execução. A seguir serão apresentadas as classes derivadas da classe **TMatrix**.

TFMatrix

Implementa o esquema de armazenamento de matrizes cheias. Os elementos desta matriz são alocados dinamicamente e armazenados num vetor. Este tipo de armazenamento inclui matrizes quadradas e retangulares.

O armazenamento dos elementos da matriz é feito num vetor, e são guardados por coluna. A transformação desta alocação é dada por *i+j*Num_Colunas*, conforme esquema mostrado na FIGURA 0-3.

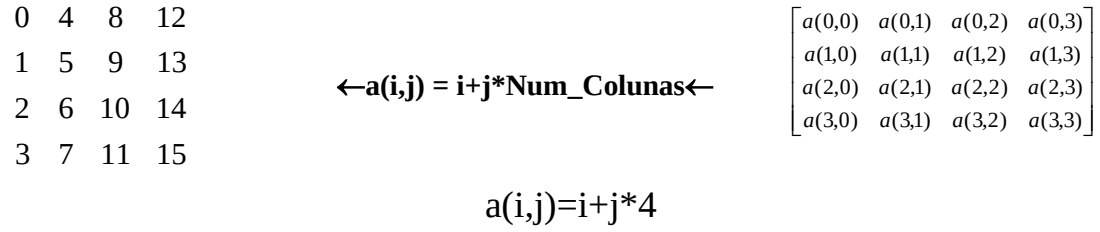


FIGURA 0-3- Esquema da alocação da classe TFMatrix.

Este objeto permite usar um "pedaço" de memória dada pelo usuário para alocar seus elementos. Isto é possível pelo fato que os elementos são alocados num vetor de *doubles*. Porém o tamanho da memória fornecida pelo usuário tem que ser coerente com o tamanho requerido pela matriz.

TFBMatrix

Esta classe gerencia matrizes do tipo banda não simétrica, porém trabalha somente com matrizes quadradas. A correspondência entre a posição dos elementos na matriz e a posição dos mesmos na memória alocada é dada pela seguinte expressão:

a(i,j)=elemento_na_memoria[tamanho_da_Banda(2*i+1)+j]*

Na elaboração da classe para este tipo de matriz, foi admitido que seriam manipuladas apenas matrizes quadradas. Como argumentos do construtor tem-se a dimensão da matriz e o tamanho da banda. Assim, uma matriz quadrada com dimensão 6 e banda 2 terá que ser declarada como *TFBMatrix a(6,2).* Na FIGURA 0-4 mostra-se a correspondente alocação de memória para esta declaração.

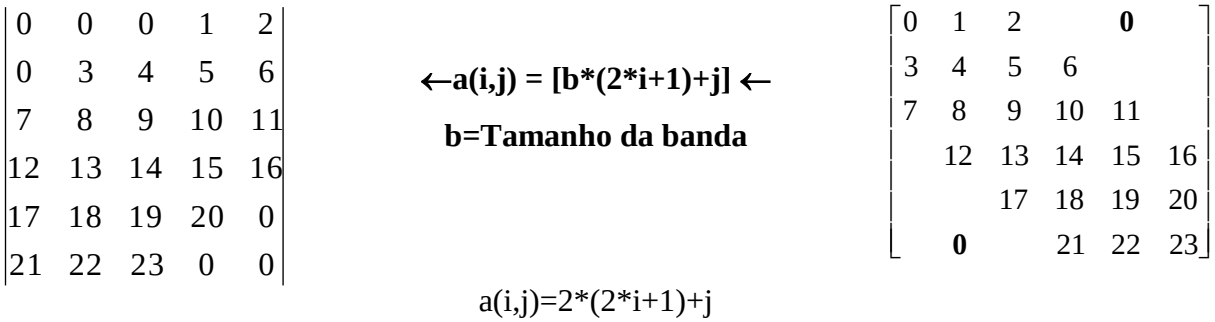


FIGURA 0-4- Esquema da alocação da classe TFBMatrix.

Ao se estudar a quantidade de alocação requerida para uma matriz, tem-se que para uma matriz com dimensão igual a *n* e banda igual a *b* são alocados *(2*b+1)*n* elementos. Para uma matriz cheia quadrada, o número de elementos é igual a n^2. Se for definida a memória poupada como sendo a diferença entre uma alocação tipo banda e uma alocação tipo matriz cheia, pode-se escrever a seguinte expressão:

$$memória\_poupada = n^2 - (2*b+1)*n > 0$$

Resolvendo-se a expressão acima, tem-se que $b < (n+1)/2$, ou seja, só é possível realmente poupar memória se o tamanho da banda for menor que a metade da dimensão da matriz.

Assim, não é vantajoso do ponto de vista da alocação de memória usar um objeto do tipo **TFBMatrix** se a banda for maior que a metade da dimensão da matriz.

TSpMatrix

Esta classe utiliza esquema de armazenamento do tipo esparsa não simétrica. Para a alocação dos elementos deste tipo de matriz foi criada a classe **TLink**, que implementa uma lista ligada com número indefinido de elementos. Na classe **TLink** são feitas operações de inserir, adicionar, remover e retornar elementos da lista. O tempo de busca de um elemento da matriz é otimizado pela utilização da função GetLast, que retorna o último elemento pedido. Este artifício aumenta o desempenho, principalmente na decomposição de matrizes.

Por exemplo, na lista ligada da FIGURA 0-5, no caso de se estar trabalhando com o elemento 6, e for necessário usar o elemento anterior, que é o elemento 5, seria necessário percorrer a lista inteira para retorná-lo. Mas com a função GetLast, retorna-se o elemento 5 imediatamente, sem a necessidade de percorrer toda a lista. Quando o elemento anterior não existir na alocação de memória, é retornado zero.

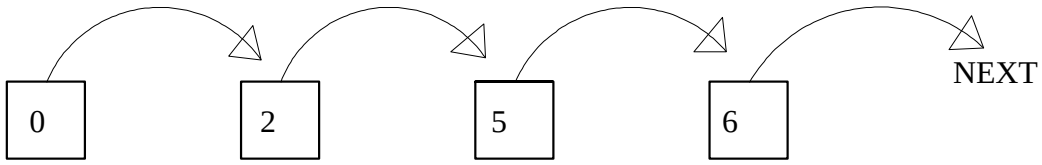


FIGURA 0-5- Lista ligada.

Em seguida é mostrado um exemplo de alocação de memória para uma matriz de 5x8, mostrada na FIGURA 0-6.

$$\begin{bmatrix} X & X & 0 & X & X & 1 & X & X \\ X & X & X & 2 & X & X & X & X \\ X & X & X & X & X & X & X & X \\ X & 3 & X & X & X & X & X & X \\ X & X & X & 4 & 5 & X & X & 6 \end{bmatrix}$$

FIGURA 0-6- Exemplo de uma matriz esparsa.

Esta matriz de 40 elementos só tem 7 elementos alocados, de acordo com o esquema das seguintes listas ligadas, mostrado na FIGURA 0-7. Os números inferiores indicam a posição na lista.

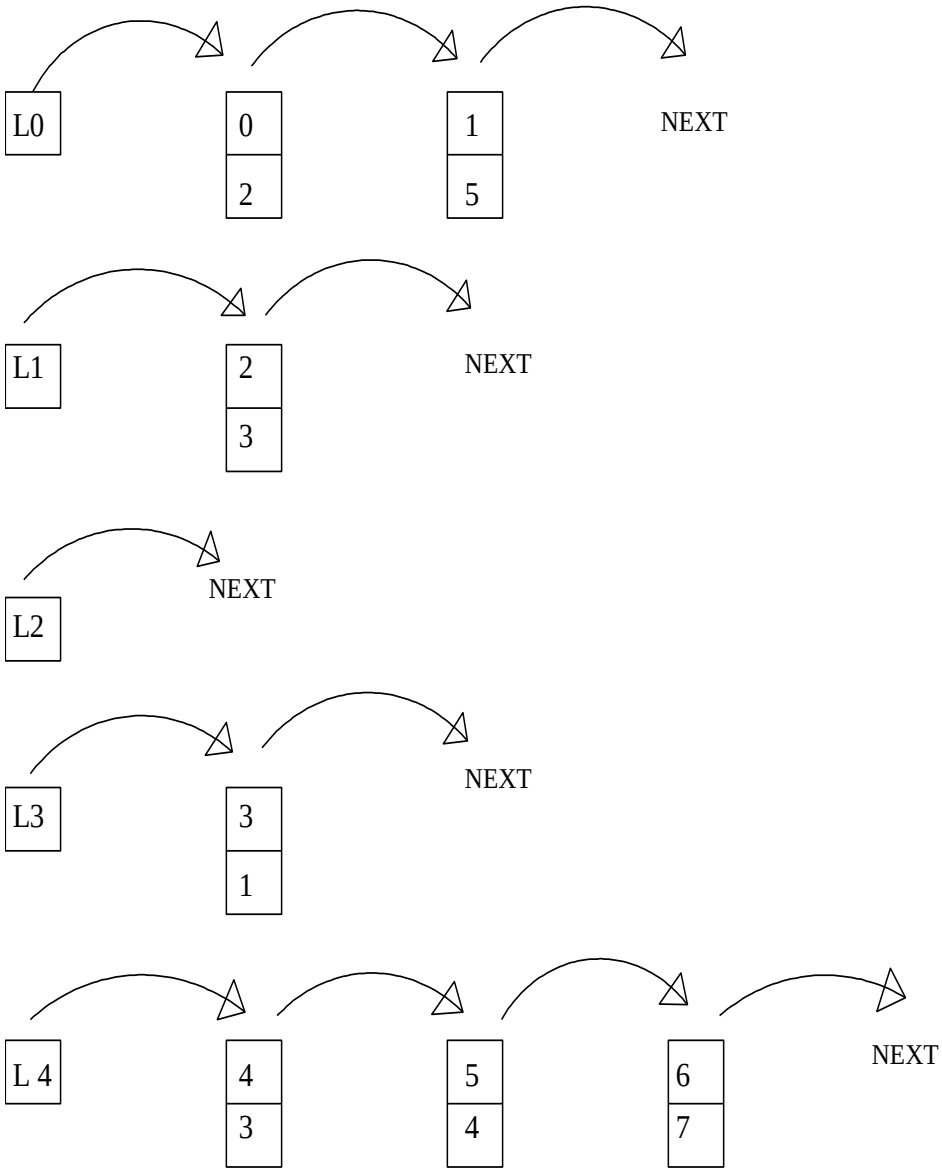


FIGURA 0-7- Esquema de alocação dos elementos da matriz esparsa.

Na FIGURA 0-7, L0, L1, L2, L3, e L4 são ponteiros que apontam para o primeiro elemento de cada linha. Assim, quando uma matriz é declarada, simplesmente deixa-se os ponteiros apontando para os primeiros elementos de cada linha, sem ser feita nenhuma alocação. Os elementos vão sendo alocados segundo vão sendo criados. Não é verificado se o elemento é diferente de zero antes de ser alocado, pois isso diminuiria o desempenho da alocação, e além disso, não há necessidade de alocar elementos nulos em uma matriz esparsa.

A entrada e a saída de elementos da matriz podem ser efetivadas através das funções Put e Get, respectivamente (que verificam se os elementos manipulados pertencem ao domínio da matriz) ou PutVal e GetVal, que não fazem nenhum tipo de verificação. Também poderia ser usada a função Input da classe **TMatrix**, que faz uma varredura da

matriz inteira para ler os elementos. Porém, isso não teria sentido, já que uma matriz esparsa contem apenas alguns elementos diferentes de zero.
Esta matriz pode ser quadrada ou não, por isso, para declarar uma matriz deste tipo, é passado o número de linhas e o número de colunas da matriz. Para o exemplo anterior, tem-se a declaração *TSpMatrix a(5,8).*

TMatRed

Esta classe é usada para condensação estática. Maiores informações encontram-se no capítulo "Subestruturação".

TSimMatrix

Para a implementação de matrizes simétricas foi derivada da classe base **TMatrix** a classe **TSimMatrix**. Esta classe também é uma classe base, na qual foram definidas todas as operações permitidas entre matrizes simétricas e matrizes não simétricas. As matrizes simétricas são as seguintes: **TSFMatrix** (Symetric Full Matrix), **TSBMatrix** (Symetric Banded Matrix), **TSSpMatrix** (Symetric Sparse Matrix) e **TSSkylMatrix** (Symetric Sky-Line Matrix). Dentro da classe **TSimMatrix** estão implementadas operações de álgebra matricial e métodos para resolução de sistemas de equações usando as funções GetVal e PutVal. Para as matrizes simétricas, estão implementados dois métodos de resolução de sistemas, a decomposição de Cholesky, só se aplica a matrizes definidas positivas, e a decomposição LDLt, que se aplica a matrizes não definidas positivas também.

TSFMatrix

Esta classe gerencia matrizes do tipo simétricas cheias. A correspondência entre um elemento da matriz e um elemento alocado é dado pela seguinte expressão:

elemento_a(i,j)=alocação[i(i+1)/2+j]*

A FIGURA 0-8 mostra esquematicamente as alocações da matriz *A(4,4)*.

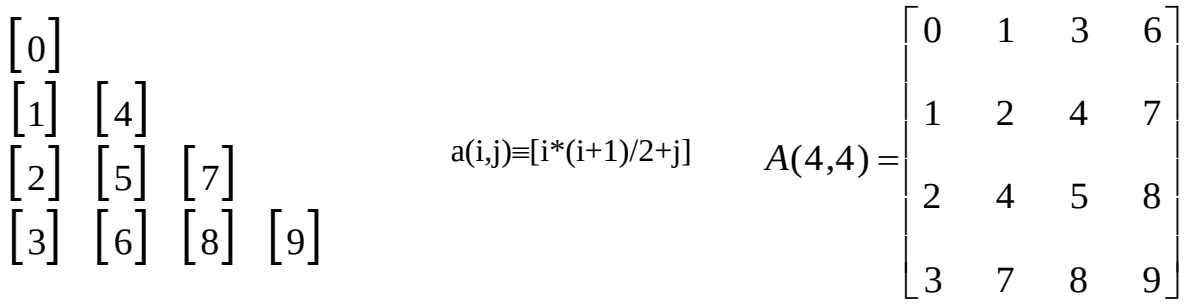


FIGURA 0-8- Esquema de alocação da classe TSFMatrix.

O construtor de uma matriz simétrica cheia tem apenas um argumento, pois toda matriz simétrica é quadrada. Dessa forma, ao invés de se declarar *TSFMatrix a(4,4)*, declara-se *TSFMatrix a(4).*

TSBMatrix

Esta classe implementa o esquema de armazenamento para matrizes de banda simétrica. Para cada elemento na diagonal aloca-se dinamicamente um vetor do tamanho igual ao da banda da matriz. Isto foi feito com a intenção de se ganhar um pouco mais de desempenho na decomposição de matrizes. Os algoritmos usados para a decomposição por Cholesky e LDLt trabalham por colunas, portanto, uma alocação por colunas facilita o retorno da posição de cada elemento.

A localização do elemento em cada coluna é dada por uma simples operação de subtração entre a posição da coluna e a posição do elemento da diagonal, conforme mostrado na FIGURA 0-9. Por exemplo, para o elemento *a(3,4)* tem-se que a posição do mesmo é 4-3=1, no vetor correspondente à coluna 4.

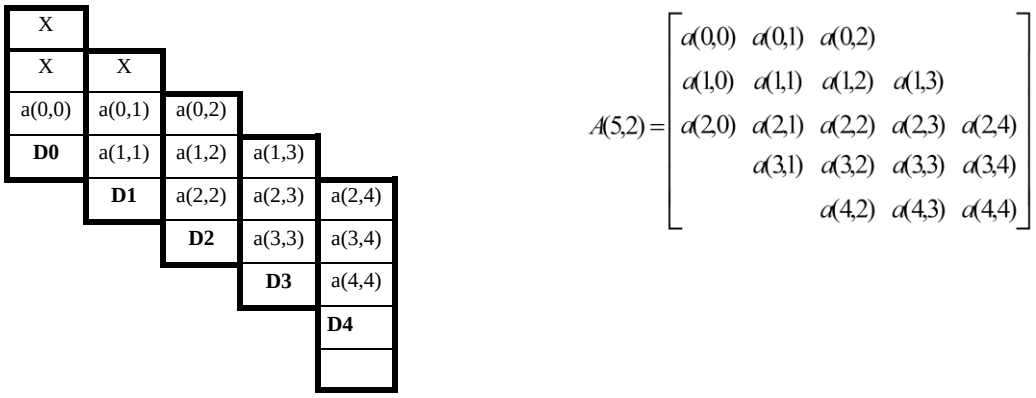


FIGURA 0-9- Esquema da alocação da classe TSBMatrix.

TSSpMatrix

Esta classe gerencia matrizes simétricas do tipo esparsa. A filosofia utilizada para a alocação deste tipo de matriz é igual à da classe **TSpMatrix** (matrizes esparsas não simétricas), porém nesta classe as listas ligadas partem dos elementos da diagonal da matriz.

A declaração de uma matriz do tipo **TSSpMatrix** requer apenas um argumento, ou seja, a dimensão da matriz, uma vez que a matriz é simétrica e, portanto, quadrada.

TSkylMatrix

Esta classe gerencia matrizes do tipo "skyline". Quando um objeto do tipo **TSkylMatrix** é criado, são alocados vetores de tamanho igual a 1 sobre os elementos da diagonal. Isso é conseguido com auxílio de uma classe que foi denominada **TColuna**. A classe **TColuna** gerencia a alocação de um vetor redimensionável, sendo possível apenas incrementar o número de elementos do vetor e nunca diminuir, pois seria dispendioso em termos de tempo construir um outro vetor de tamanho menor, copiar os elementos do vetor um a um e eliminar o vetor anterior.

Um objeto do tipo **TColuna** guarda três informações importantes: os elementos da coluna (Vet), o tamanho da coluna (VetSize) e o tamanho máximo da coluna (Size), que depende da posição na diagonal onde a coluna será alocada. Os elementos da matriz são retornados segundo a seguinte expressão:

$$A(i,j)\ \forall\ i < j\ \begin{cases} 0.0 & se\ Dj.Size \le (j-i) \\ Dj.Vet(j-i) & se\ Dj.Size > (j-i) \end{cases}$$

GERENCIAMENTO DOS METODOS DE SOLUÇÃO DE SISTEMAS LINEARES

Conforme já foi mencionado, a classe **TMatrix** define um comportamento que as classes derivadas precisam implementar ou não dependendo de suas características próprias. A classe **TMatrix** declara e implementa diferentes métodos para a resolução de sistemas lineares, que as classes derivadas podem redefinir ou não, com a intenção de melhorar sua eficiência. Além dos métodos para solução de sistemas lineares definidos

e implementados, esta classe possui uma interface para os arquivos *header* da biblioteca TEMPLATES[3]. Como tal, a classe **TMatrix** define e implementa os seguintes métodos:
métodos diretos:

- decomposição de LU;
- decomposição de Cholesky;
- decomposição de LDLt;

métodos iterativos:

- método de Jacobi;
- SOR (*Successive Overrelaxation Method*);
- SSOR (*SymmetricSuccessive Overrelaxation Method*);
- CG (*Conjugate Gradient*) pré-condicionado;
- Método GMRES (*Generalized Minimal Residual Method*) pré-condicionado.

O pré-condicionador do CG e do GMRES é implementado como um objeto da classe **TSolver**. Geralmente é um método direto ou iterativo aplicado na mesma matriz objeto. Para facilitar o gerenciamento desta enorme quantidade de combinações, foi criada separadamente uma classe **TSolver**, que associa um objeto do tipo **TMatrix** a um método de solução. Como exemplo, pode-se imaginar o GMRES sendo condicionado por duas iterações do GC que por sua vez é pré-condicionado por uma iteração do SSOR.

TRATAMENTO DE BLOCOS.

A classe **TBlock** implementa uma divisão lógica da matriz em blocos de tamanho variável. O construtor tem três argumentos: um ponteiro para um objeto do tipo **TMatrix**, um inteiro que é o número de blocos na diagonal da matriz, e outro que é a dimensão de cada bloco na diagonal. A classe **TBlock** não modifica a matriz. Ela aumenta a capacidade de endereçar os elementos da matriz por blocos.
O objeto do tipo **TBlock** contem duas informações básicas: a posição do bloco na diagonal e sua dimensão. A FIGURA 0-10 é um exemplo de uma matriz dividida em três blocos de diferentes dimensões: o bloco 0, de dimensões 2×2, o bloco 1, de 3×3, e o bloco 2 de 2×2.

[3] Sobre a biblioteca TEMPLATES, ver BARRET et al. (1994).

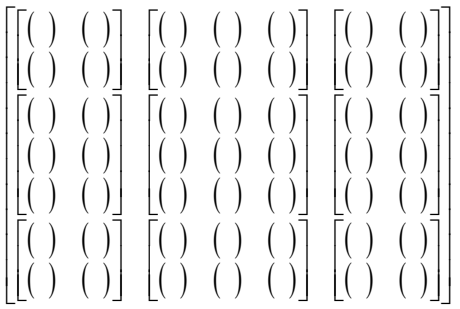


FIGURA 0-10- Esquema de matriz dividida em blocos.

As informações básicas são suficientes para calcular também a posição e a dimensão de qualquer bloco fora da diagonal, de acordo com o esquema mostrado na FIG. 11. Por exemplo, o bloco cuja posição é (1,2) tem dimensão 3x2 (no esquema, a linha 1 corresponde à dimensão 3, e a coluna 2 corresponde à dimensão 2).

Posição na Diagonal	0	1	2
Dimensão	2	3	2

FIGURA 0-11- Informações contidas no objeto Tblock.

Este construtor inicialmente so permite que os blocos da diagonal sejam todos do mesmo tamanho. Por exemplo, se forem feitas as seguintes declarações:

TFMatrix A(5,5) *//declaração de uma matriz cheia*
TBlock b(&A,3,2) *//declaração do bloco b com 3 blocos na diagonal*
//cada bloco com dimensão 2x2
//este bloco esta apontando para a matriz A

esquematicamente, será obtido o arranjo mostrado na FIGURA 0-12.

$$\begin{bmatrix} \begin{bmatrix} A(0,0) & A(0,1) \\ A(1,0) & A(1,1) \end{bmatrix} & \begin{bmatrix} A(0,2) & A(0,3) \\ A(1,2) & A(1,3) \end{bmatrix} & \begin{bmatrix} A(0,4) & A(0,5) \\ A(1,4) & A(1,5) \end{bmatrix} \\ \begin{bmatrix} A(2,0) & A(2,1) \\ A(3,0) & A(3,1) \end{bmatrix} & \begin{bmatrix} A(2,2) & A(2,3) \\ A(3,2) & A(3,3) \end{bmatrix} & \begin{bmatrix} A(2,4) & A(2,5) \\ A(3,4) & A(3,5) \end{bmatrix} \\ \begin{bmatrix} A(4,0) & A(4,1) \\ A(5,0) & A(5,1) \end{bmatrix} & \begin{bmatrix} A(4,2) & A(4,3) \\ A(5,2) & A(5,3) \end{bmatrix} & \begin{bmatrix} A(4,4) & A(4,5) \\ A(5,4) & A(5,5) \end{bmatrix} \end{bmatrix}$$

FIGURA 0-12- Arranjo em blocos obtido para a matriz exemplo A.

Agora, se na segunda declaração acima, os dois últimos argumentos do construtor não forem passados, o construtor assumirá o número de blocos na diagonal igual ao número de elementos na diagonal, ou seja, assumirá blocos de dimensão 1x1. Se forem feitas as declarações:

```
TFMatrix A(8,8)      //declaração de uma matriz cheia
TBlock b(&A)         //declaração do bloco b sem especificar
                     //o número de blocos nem o tamanho dos blocos
```

o arranjo dos blocos na matriz será o mostrado na FIGURA 0-13.

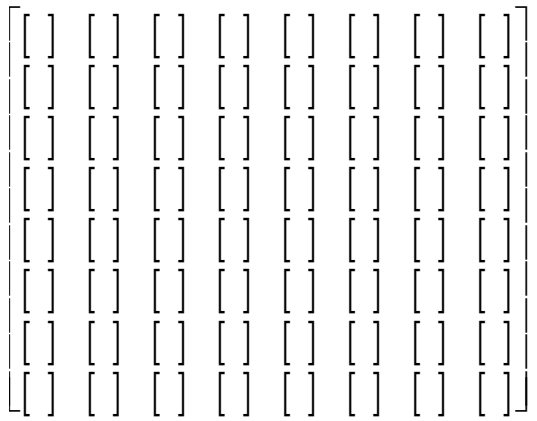


FIGURA 0-13- Novo arranjo obtido para a matriz exemplo.

Este tipo de "visualização" da matriz pode ser modificado utilizando-se duas funções. A primeira modificação, o número de blocos na diagonal, pode ser realizada com a função SetBlocks, que tem como argumento o novo número de blocos que o objeto do tipo **TBlock** terá.

Outra função utilizada para mudar o objeto do tipo **TBlock** é Set, que utiliza três argumentos. O primeiro argumento fornece a posição na diagonal do bloco a ser modificado, o segundo argumento, o novo tamanho do bloco a ser modificado; por último, com o terceiro argumento é fornecida a posição relativa do primeiro elemento do bloco, com respeito à sua posição na diagonal da matriz.

Em seguida mostra-se como modificar o arranjo dos blocos da FIGURA 0-12.

```
b.SetBlocks(4);          //modificação do número de blocos na diagonal
```

```
b.Set(0,2,0);        //o bloco 0 tem dimensão 2
                     //e o primeiro elemento do bloco está na
                     //posição 0 da diagonal da matriz.
b.Set(1,3,2);        //o bloco1 tem dimensão 3
                     //e o primeiro elemento do bloco está na
                     //posição 2 da diagonal da matriz.
b.Set(2,1,5);        //o bloco 2 tem dimensão 1
                     //e o primeiro elemento do bloco está na
                     //posição 5 da diagonal da matriz
b.Set(3,2,6);        //o bloco 3 tem dimensão 2
                     //e o primeiro elemento do bloco está na
                     //posição 6 da diagonal da matriz
```

Esquematicamente, será obtido o novo arranjo mostrado na FIGURA 0-14.

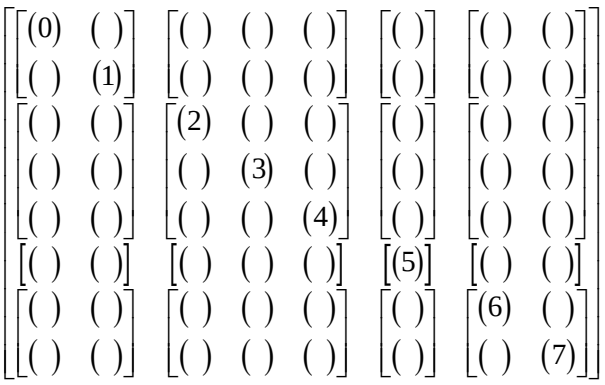


FIGURA 0-14- Novo arranjo para a matriz exemplo.

Uma observação importante é que esta modificação tem que ser de uma maneira ordenada, ou seja, primeiro tem que ser mudado o número de blocos, depois o tamanho do primeiro bloco, depois o tamanho do segundo bloco, e assim por diante até o último bloco.

O mesmo procedimento anterior pode ser feito usando-se a função SetAll(dimensions) e a função Resequence(start). A primeira função passa para o objeto do tipo **TBlock** as dimensões dos blocos na diagonal através do vetor de inteiros ***dimensions***. A

segunda função faz a seqüência dos blocos a partir da posição start (este argumento tem como padrão a posição 0).

Observa-se que a matriz não foi modificada; ela continua existindo na mesma forma com que foi criada.

Outras funções também permitem que os diferentes blocos sejam copiados para outras matrizes. Também é permitida a operação de igualar dois objetos do tipo **TBlock** e de extrair um dado bloco da matriz. Todas essas e outras funções estão documentadas no arquivo *header* da classe.