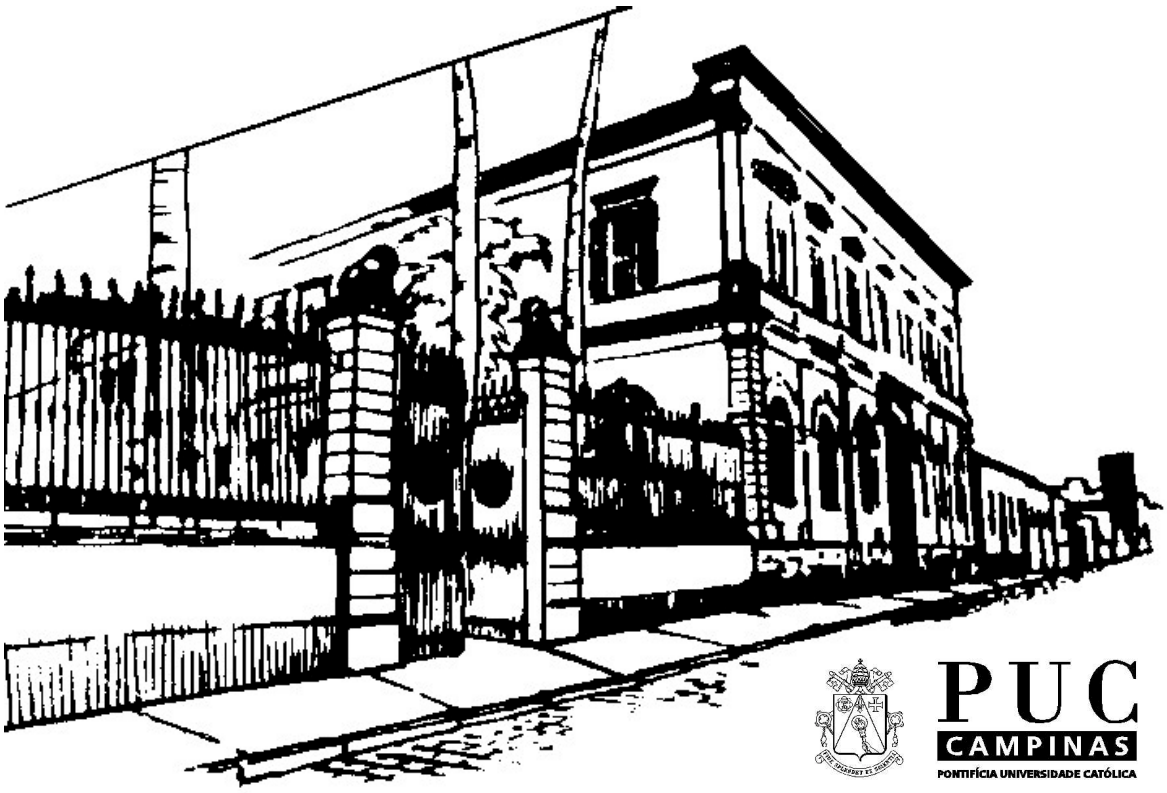

Centro de Ciências Exatas, Ambientais e de Tecnologias

Faculdade de Engenharia de Computação

Paradigmas de Linguagens de Programação II

2º Semestre de 2002
Volume 3

Prof. André Luís dos R.G. de Carvalho

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS
INSTITUTO DE INFORMÁTICA
PLANO DE ENSINO

Faculdade:	Disciplina:	Docente:	C.h. Semanal	Período Letivo:
Engenharia de Computação	Paradigmas de Linguagens de Programação II	André Luís dos Reis Gomes de Carvalho	04 (quatro horas aula)	2º Semestre de 2002

Objetivo Geral da Disciplina:

- Estudar o paradigma de programação orientada a objetos, bem como linguagens representativas deste paradigma.

Avaliação:

- 70% da nota será dada pela média aritmética de duas provas.
- 30% da nota será dada pela média ponderada das notas de uma série de trabalhos a serem desenvolvidos durante o semestre com pesos compatíveis com sua complexidade.

Freqüência:

- Nenhum abono de falta será concedido pelo professor. Qualquer problema neste sentido deverá ser encaminhada ao PA que tomará todas as providências no sentido de encaminhar a solução do mesmo.
- Para fins de controle de freqüência, não será permitido que alunos desta turma assistam aulas em outras turma, e nem o contrário.

Bibliografia Utilizada :

- Concepts of Programming Languages
  Sebesta, R.W.
- Conceitos de Linguagens de Programação
  Ghesi, C.; e Jazayeri, M.
- Programming Languages: Design and Implementation
  Pratt, T.W.
- Object Orientede System Analysis: Modelling the World in Data
  Shlaer, S.; e Mellor, S.J.
- Object Lifecycles: Modelling the World in States
  Shlaer, S.; e Mellor, S.J.
- Introdução à Programação Orientada a Objetos
  Takahashi, T.
- Programação Orientada a Objetos
  Takahashi T.; e Liesemberg, H.K.
- Programação Orientada para Objeto
  Cox, B.J.
- The TAO of Objects: A Beginner's Guide to Object Oriented Programming
  Entsminger, G.
- A Complete Object Oriented Design Example
  Richardson, J.E.; Schulz, R.C.; e Berard, E.V.
- Java Unleashed
  Morrison, M.; December, J.; et alii
- Dominando o Java
  Naughton, P.
- C++: Manual de Referência Comentado (Documento Oficial do ANSI)
  Ellis, M.A., Stroustrup, B.
- Visual C++ - Guia de Desenvolvimento Avançado
  Barbakati, N.
- Programação Orientada para Objeto e C++
  Wiener, R.S.; e Pinson, L.J.
- Object-Oriented Programming using C++
  Pohl, I.
- Programação Orientada para Objeto com Turbo C++
  Perry, G.
- A Complete Object Oriented Design Example
  Richardson, J.E.; Schulz, R.C.; e Berard, E.V.

Nº aulas	Conteúdos selecionados

04	O PARADIGMA DE ORIENTAÇÃO A OBJETOS:
	- INTRODUÇÃO;
	- CONCEITOS BÁSICOS:
	  - Objeto → (1) estado interno; (2) comportamento: métodos / mensagens.
	  - Classes → (1) estado; (2) comportamento: métodos / mensagens; (3) instâncias de classe.
	- FUNDAMENTOS:
	  - Encapsulamento.
	  - Herança.
	  - Polimorfismo.
	- CONCLUSÃO.

18	A LINGUAGEM DE PROGRAMAÇÃO JAVA (ASPECTOS BÁSICOS):
	
	- INTRODUÇÃO À JAVA
	  - Execução de Conteúdo → (1) o que pode-se fazer com Java: o que é java / o que é execução de conteúdo / como Java muda a WWW; (2) origens e futuro de Java: o estado corrente de Java / as possibilidades futuras de Java; (3) potencial da linguagem Java: animação / interação / interatividade e computação / comunicação / aplicações e handlers; (4) o que se torna possível com Java.
	  - Projeto Flexível e Dinâmico → (1) primeiro contato com Java: conexao com a WWW / alguns programas simples; (2) visão geral de Java: suporte a comunicação de rede / características de Java enquanto uma linguagem de programação / HotJava / Java em ação / componentes de software de Java / especificação da máquina virtual de Java / segurança em Java.
	  - Impactos na WWW → (1) visão geral da WWW: ideias que levaram à WWW / uma definição de WWW; (2) como Java transforma a WWW: suporte à interatividade / eliminação da necessidade aplicações auxiliares; (3) contribuições à comunicação na WWW; (4) impactos no potencial da WWW.
	  - Páginas Animadas → (1) applets em movimento; (2) animação; (3) sites comerciais.
	  - Páginas Interativas → (1) jogos interativos; (2) aplicações educacionais.
	  - Distribuição de Conteúdo → (1) significado de distribuição e recuperação em rede; (2) como lidar com novos protocolos e formatos; (3) recuperação e compartilhamento de informações em rede.
	
	- PRIMEIRO CONTATO

	- Ferramentas → (1) visão geral; (2) navegadors; (3) ambientes de desenvolvimento; (4) bibliotecas de programação; (5) recursos online. - O Kit de Desenvolvimento Java → (1) como obter a última versão; (2) visão geral; (3) o compilador; (4) o interpretador para a execução; (5) o visualizador de applets; (6) o depurador; (7) engenharia reversa em arquivos de classe; (8) o gerador de arquivos header e strub; (9) o gerador de documentação. - Outros Ambientes e Ferramentas → (1) ambientes de desenvolvimento; (2) bibliotecas para programação. - A LINGUAGEM DE PROGRAMAÇÃO JAVA - Fundamentos da Linguagem Java → (1) um primeiro programa; (2) tokens: identificadores / palavras chave / literais / operadores / separadores / comentarios / espaços em branco; (3) tipos de dados: inteiros / reais / logico / caractere; (4) conversão de tipo; (5) blocos e escopo; (6) vetores; (7) strings. - Expressões, Operadores e Estruturas de Controle → (1) expressões e operadores: precedência de operadores / operadores de inteiros / operadores de reais / operadores lógicos / operadores de strings / o operador de atribuição; (2) estruturas de controle: seleções / repetições / break e continue. - Classes, Pacotes e Interfaces → (1) revisão de conceitos de programação orientada a objetos: objetos / encapsulamento / mensagens / classes / herança; (2) classes: declaração / derivação / redefinição de métodos / sobrecarga de métodos / modificadores de acesso / classes e métodos abstratos; (2) criação de objetos: o método de criação / o operador new; (3) destruição de objetos; (4) pacotes: declaração / importação / visibilidade das classes; (5) interfaces: declaração / implementação.
02	PRIMEIRA PROVA
20	A LINGUAGEM DE PROGRAMAÇÃO JAVA (ASPECTOS AVANÇADOS): - RESISTÊNCIA A FALHAS - Tratamento de Excessões → (1) programação em alto nível de abstração; (2) programação em baixo nível de abstração; (3) limitações do programador; (4) a cláusula finally. - AS BIBLIOTECAS DE CLASSES DA LINGUAGEM JAVA - Visão Geral das Bibliotecas de Classes → (1) o pacote Language; (2) o pacote Utilities; (3) o pacote I/O; (4) classes relacionadas com threads; (10) classes relacionadas com tratamento de erros; (5) classes relacionadas com processos. - O Pacote Language → (1) a classe Object; (2) classes relacionadas com os tipos de dados: a classe Boolean / a classe character / classes relacionadas com inteiros / classes relacionadas com reais; (3) a classe Math; (4) classes relacionadas com Strings: a classe String / a classe

	StringBuffer; (5) a classe System; (6) a classe Runtime; (7) a classe Class; (8) a classe ClassLoader. - O Pacote Utilities → (1) interfaces: enumeration / observer; (2) classes: BitSet / Date / Random / StringTokenizer / Vector / Stack / Dictionary / Hashtable / Properties / Observable. - O Pacote I/O → (1) classes de entrada: a classe InputSream / o objeto System.in / a classe BufferedInputStream / a classe DataInputStream / a classe FileInputStream / a classe StringBufferInputStream; (2) classes de saída: a classe OutputStream / a classe PrintSream / o objeto System.out / a classe BufferedOutputStream / a classe DataOutputSream / a classe FileOutputStream; (3) classes relacionadas com arquivos: a classe File / a classe RandomAccessFile. - PROGRAMAÇÃO DE APPLETS - Visão Geral de Programação de Applets → (1) o que é uma applet: applets e a WWW / diferença entre applets e aplicações; (2) os limites das applets: limites funcionais / limites impostos pelo navegador; (3) noções básicas sobre applets: herança da classe Applet / HTML; (4) exemplos básicos de applets. - O Pacote AWT (abstract window toolkit) → (1) uma applet AWT simples; (2) tratamento de eventos: tratamento de eventos em detalhes / handleEvent () ou action () / geração de eventos; (3) componentes: componentes de interface / containers / métodos comuns a todos os componentes; (4) como projetar uma interface de usuário. - O Pacote *Applets e Gráficos* → (1) características das applets; o ciclo de vida de uma applet: init () / start () / stop () / destroy (); (2) como explorar o navegador: como localizar arquivos / imagens / como usar o MediaTracker / audio; (3) contextos de uma applet: showDocument () / showStatus () / como obter parâmetros; (4) gráficos; (5) uma applet simples. - Como programar Applets → (1) projeto básico de applets: interface do usuário / projeto das classes; (2) applets no munto real: applets devem ser pequenas / como responder ao usuário. - MULTIPROGRAMAÇÃO - Threads e Multithreading → (1) o que são threads e para que elas servem; (2) como escrever applets com threads; (3) o problema do conceito de paralelismo; (4) como pensar em termos de multithreads; (5) como criar e usar threads; (6) como saber que uma thread parou; (7) thread scheduling: preemptivo X não preemptivo; como testar. - ACESSO A BASES DE DADOS - O Pacote SQL → (1) drivers; (2) classes: Connection, Statement, ResultSet. - PROGRAMAÇÃO DISTRIBUÍDA - ServerSockets e Sockets & RMI.

04	A LINGUAGEM DE PROGRAMAÇÃO C++ (ASPECTOS BÁSICOS):
	
	- FUNDAMENTOS:
	  - Classes, membros e tipos de membros.
	  - Funções sobrecarregadas.
	  - Classes e funções amigas.
	  - Construtores e destrutores.
	  - Operadores como funções, sobrecarga de operadores.
	
	- HERANÇA:
	  - Classes base e classes derivadas.
	  - Classes abstratas.
	  - Derivação múltipla.
	  - Classes base virtuais.

10	A LINGUAGEM DE PROGRAMAÇÃO C++ (ASPECTOS AVANÇADOS):
	
	- STREAMS:
	  - E/S.
	  - Streams.
	  - Formatação.
	  - Dispositivos padrão de E/S e streams.
	  - Arquivos e streams.
	  - Strings e streams.
	  - E/S em streams de tipos do usuário
	
	- TEMPLATES:
	  - Templates de função.
	  - Templates de classe.

	- EXCEÇÕES:
	  - Tratamento de exceções.
	  - Discriminação e nomeação de exceções.
	  - Exceções que não são erros.
	  - Especificação de interface.
	  - Exceções não tratadas.
	  - Alternativas para tratamento de erros.

02	SEGUNDA PROVA

Índice

Capítulo IV:
A Linguagem de Programação C++

Introdução

C++ é uma linguagem de programação de propósito geral baseada na linguagem C. De certa forma, podemos considerar a linguagem C++ como sendo uma extensão da linguagem C, mas esta não é uma abordagem recomendável.

Isto porque a linguagem C++ não se limita a acrescentar à linguagem C novos comandos e estruturas sintáticas. Naturalmente isto também acontece, mas não é o fundamental.

Na verdade C++ é uma linguagem que se baseia em um paradigma de programação diferente daquele que serve de base para a linguagem C e para a maioria das linguagens de programação convencionais.

É importante notar que uma mudança deste porte impactua profundamente na concepção que se tem de programa, e conseqüentemente, obrigam o programador a rever e mudar a sua forma de escrever programas.

Assim, em termos práticos, o benefício que conhecer a linguagem C traz para quem se propõe a aprender a linguagem C++ se limita ao aproveitamento do conhecimento que se tem dos comandos e das estruturas sintáticas daquela linguagem, já que a forma de agrupá-los para constituir um programa difere fundamentalmente nas duas linguagens.

Generalidades

Sendo de fato uma extensão da linguagem C, a linguagem C++ mantém intactas todas as características gerais daquela linguagem. Em virtude disto, um compilador C++ compila um programa C exigindo pouca ou nenhuma mudança.

Mas um programa genuíno em C++ difere estruturalmente bastante de um programa em C. Em programas de ambas as linguagens encontraremos dados e funções, mas o encaixamento e o significados destes na estrutura do programa diferem fundamentalmente.

Programas imperativos são organizados como uma hierarquia de procedimentos. O que importa quando se escreve um programa imperativo é o processo para a obtenção de um determinado resultado.

No paradigma imperativo, programas são entendidos como uma hierarquia de processos. Sempre há um processo de mais alto nível que, virtualmente, é entendido como sendo aquele que realiza todas as funções atribuídas ao programa, muito embora não as realize em pessoa, apenas controle processos de nível hierárquico imediatamente inferior.

Cada um destes processos, por sua vez, virtualmente, também é entendido como sendo aquele que realiza toda função que lhe é atribuída, apesar de também apenas controlar processos de nível mais baixo.

Este padrão se repete até encontrarmos procedimentos muito simples, que realmente realizam as tarefas que lhes são atribuídas sem contar com a ajuda de outros procedimentos para tanto.

Diferentemente das linguagens convencionais, em C++ não importa o processo para a obtenção de um determinado resultado, importam os objetos envolvidos em um problema, suas características e seu comportamento.

Identificadores

Identificadores introduzem nomes no programa. Em C++ (e também em C) podem ser uma seqüência arbitrariamente longa de letras, dígitos e sublinhados (_). O primeiro caractere de um identificador deve necessariamente ser uma letra ou um sublinhado (_).

É importante ressaltar que, diferentemente de outras linguagens de programação, a linguagem C++ (e também a linguagem C) diferencia letras maiúsculas de letras minúsculas. Em C++ (e também em C), identificadores que são lidos da mesma forma, mas que foram escritos de forma diferente no que tange ao emprego de letras maiúsculas e minúsculas, são considerados identificadores diferentes.

Identificadores que se iniciam por sublinhado (_) devem ser evitados, tendo em vista que muitas implementações da linguagem C++ (e também da linguagem C) os reservam para seu próprio uso. Identificadores que se iniciam por duplo sublinhado (__) também devem ser evitados, tendo em vista que muitas implementações da linguagem C++ os reservam para seu próprio uso.

Palavras-Chave

Os palavras listadas a seguir são reservadas pela linguagem C++ (e também pela linguagem C) para serem usadas como palavras-chave:

asm	auto	break	case
char	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	while	

C++, no entanto, acrescenta novas palavras-chave àquelas da linguagem C. São elas:

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

Operadores e Outros Separadores

Os caracteres e as combinações de caracteres a seguir são reservados pela linguagem C++ (e também pela linguagem C) para uso como operadores, sinais de pontuação ou são reservados para o pré-processador, e não podem ser utilizados com outra finalidade:

+	-	*	/	%	++	--
<	>	<=	>=	==	!=	&&
||	!	&	|	^	~	<<
>>	+=	-=	*=	/=	%=	&=
|=	^=	<<=	>>=	?	:	(
)	'	"	=	.	->	,
\	[	]	;	{	}	#

C++, no entanto, acrescenta novos operadores àqueles da linguagem C. São eles:

##	.*	->*	::

Cada um deles deve ser considerado como um único símbolo.

Comentários

Na linguagem C++ (e também na linguagem C), os caracteres /* iniciam um comentário que terminará com os caracteres */. Além dessa forma de escrever comentários, temos que, em

C++, os caracteres // também iniciam um comentário que terminará no final da linha onde apareceram.

Tipos Básicos

Não são muitos os tipos de que dispomos em C++; na verdade dispomos apenas dos mesmos quatro tipos básicos disponíveis na linguagem C, a saber: char (caractere), int (inteiro), float (real) e void (vazio).

No entanto, através de qualificadores, podemos criar interessantes variações desses tipos, ampliando assim o conjunto dos tipos básicos.

Observe abaixo os possíveis tipos numéricos existentes em Turbo C++ 3.0, uma particular implementação de C++:

1. **Inteiros:**

   Os tipos inteiros permitem a declaração de variáveis capazes de armazenar números inteiros.

   Os inteiros simples têm o tamanho natural sugerido pela arquitetura da máquina; os demais tamanhos são fornecidos para atender a necessidades especiais e podem ser tornados equivalentes entre si ou a um inteiro simples.

   - char: 8 bits com sinal (de -128 a 127);
   - signed char: como char;
   - unsigned char: 8 bits sem sinal (de 0 a 255);
   - int: 16 bits com sinal (de -32768 a 32767);
   - signed int: como int;
   - unsigned int: 16 bits sem sinal (de 0 a 65535);
   - short int: como int;
   - signed short int: como short int;

- unsigned short int: como unsigned int;
- long int: 32 bits com sinal (de -2147483648 a 2147483647);
- signed long int: como long int;
- unsigned long int: 32 bits sem sinal (de 0 a 4294967295).

2. Reais:

Os tipos reais permitem a declaração de variáveis capazes de armazenar números reais. As implementações de C podem decidir livremente as características a conferir a estes tipos.

- float: 32 bits (de 3,4 x 10^{-38} a 3,4 x 10^{+38});
- double: 64 bits (de 1,7 x 10^{-308} a 1,7 x 10^{+308});
- long double: 80 bits (de 3,4 x 10^{-4932} a 1.1 x 10^{+4932}).

Observe abaixo os possíveis tipos numéricos existentes em Visual C++ 6.0, uma particular implementação de C++:

1. Inteiros:

Os tipos inteiros permitem a declaração de variáveis capazes de armazenar números inteiros. Existem em diversos tamanhos, a saber:

- char: 8 bits com sinal (de -128 a 127);
- signed char: como char;
- unsigned char: 8 bits sem sinal (de 0 a 255);
- short int: 16 bits com sinal (de -32.768 a 32.767);
- signed short int: como short int;
- unsigned short int: 16 bits sem sinal (de 0 a 65.535);
- int: 32 bits com sinal (de – 2.147.483.648 a 2.147.483.647);
- signed int: como int;

- unsigned int: 32 bits sem sinal (de 0 a 4.294.967.295);
- long int: como int;
- signed long int: como long int;
- unsigned long int: como unsigned int.

2. Reais:

Os tipos reais permitem a declaração de variáveis capazes de armazenar números reais. Existem em três tamanhos, a saber:

- float: 32 bits (de 3,4 x 10^{-38} a 3,4 x 10^{+38});
- double: 64 bits (de 1,7 x 10^{-308} a 1,7 x 10^{+308});
- long double: como double.

Observe abaixo outros tipos típicos da linguagem C++, independentemente de implementação:

3. Vazio:

O tipo void especifica um conjunto vazio de valores. Ele é usado como tipo de retorno para funções que não retornam nenhum valor. Não faz sentido declarar variáveis do tipo void.

4. Booleanos:

Vale ressaltar que na linguagem C++ não existe um tipo específico para armazenar valores lógicos. Para tanto, qualquer tipo integral pode ser empregado. Faz-se a seguinte convenção: o valor zero simboliza a falsidade, e qualquer valor diferente de zero, simboliza a verdade.

Constantes

Constantes são instâncias de tipos das linguagens. Temos duas variedades de constantes, a saber, constantes literais e constantes simbólicas.

Constantes Literais

Constantes literais são aquelas que especificam literalmente uma instância de um tipo da linguagem. A seguir apresentaremos as constantes literais existentes na linguagem C++ (e também na linguagem C).

Constantes Literais do Tipo Inteiro

Uma constantes inteira constituída de uma seqüência de dígitos é considerada decimal, a menos que inicie por um dígito 0, caso em que será considerada uma constante octal. É importante notar que os caracteres 8 e 9 não são dígitos octais válidos, e por isso não podem ser empregados em tais constantes.

Uma seqüência de dígitos precedida por 0x ou 0X é considerada uma constante hexadecimal. Além dos dígitos, também as letras de a (ou A) até h (ou H) são consideradas válidas para compor tais constantes.

O tipo de uma constante inteira depende de sua forma, valor e sufixo. Se ela for decimal e não tiver nenhum sufixo, ela será do primeiro desses tipos no qual seu valor puder ser representado: int, long int, unsigned long int.

Se ela for octal ou hexadecimal e não tiver nenhum sufixo, ela será do primeiro desses tipos no qual seu valor puder ser representado: int, unsigned int, long int, unsigned long int.

Independentemente da base, se ela for seguida pelo caractere u ou U, ela será do primeiro desses tipos no qual seu valor puder ser representado: unsigned int, unsigned long int.

Também independentemente da base, se ela for seguida pelo caractere l ou L, ela será do primeiro desses tipos no qual seu valor puder ser representado: long int, unsigned long int.

Por fim, ainda independentemente da base, se ela for seguida por ul, lu, uL, Lu, Ul, lU, UL ou LU, ela será unsigned long int.

Constantes Literais do Tipo Real

Uma constante real consiste numa parte inteira com sinal opcional, um ponto decimal e uma parte fracionária, um e (ou E), um expoente inteiro com sinal opcional, e um sufixo de tipo.

Tanto a parte inteira como a parte fracionária podem ser omitidas (mas não ambas). Tanto o ponto decimal seguida da parte fracionária como o e (ou E) e o expoente podem ser omitidos (mas não ambos). O sufixo de tipo também pode ser omitido.

O tipo de uma constante real é double float, a menos que explicitamente seja especificado um outro tipo. Os sufixos f (ou F) e l (ou L) especificam, respectivamente float e long double.

Constantes Literais do Tipo Caractere

Uma constantes caractere é constituída por um único caracteres delimitado por apostrofes (`'c'`). Constantes caractere são do tipo `char`, que é um tipo integral. O valor de uma constante caractere é o valor numérico do caractere no conjunto de caracteres da máquina.

Certos caracteres não visíveis, o apóstrofe (`'`), as aspas (`"`), o ponto de interrogação (`?`) e a barra invertida (\) podem ser representados de acordo com a seguinte tabela de seqüências de caracteres:

'\b'	retrocesso (backspace)
'\t'	tabulação (tab)
'\v'	tabulação vertical (vtab)
'\n'	nova linha (new line)
'\f'	avanço de formulário (form feed)
'\r'	retorno do carro (carriage return)
'\a'	alerta (bell)
'\0'	nulo (null)
'\''	apóstrofe (single quote)
'\"'	aspas (double quote)
'\\'	barra invertida (backslash)
'\OOO'	caractére ASCII (OOO em octal)
'\xHH'	caractére ASCII (HH em hexadecimal)

Uma seqüência como estas, apesar de constituídas por mais de um caractere, representam um único caractere.

Vale ressaltar que o emprego de uma barra invertida (\) seguida por um outro caractere que não um dos especificados acima implicará em um comportamento indefinido em C++ (e também em C).

Constantes Literais do Tipo *String*

Uma constante *string* é constituída por uma seqüência de caracteres (podendo incluir caracteres visíveis e invisíveis; neste último caso, os caracteres serão representados segundo a tabela discutida acima) delimitada por aspas (”).

Constantes string são vetores de char que contém toda a seqüência dos caracteres constituintes do *string* seguida pelo caractere \0 que funciona como marcador de final de *string*.

Constantes Simbólicas

Constantes simbólicas são aquelas que associam um nome a uma constante literal, de forma que as referenciamos no programa pelo nome, e não pela menção de seu valor literal.

Constantes simbólicas são aquelas que associam um nome a uma constante literal, de forma que as referenciamos no programa pelo nome, e não pela menção de seu valor literal.

Em C++ (e também em C), isto é feito através do uso da instrução `#define`. Sendo Tipo um tipo, Const o identificador de uma constante e Lit uma constante literal, temos que a forma geral de uma declaração de constante simbólica é como segue:

#define Const Lit

Entretanto, a linguagem C++ possui uma outra forma de declarar constantes simbólicas, cuja principal finalidade é possibilitar a declaração de objetos constantes. É importante ressaltar que essas duas formas de declarar constantes tem significados e usos completamente diferentes. Esta forma será apresentada em um momento mais apropriado.

Variáveis

Convencionalmente entendemos que variáveis são células de memória capazes de armazenarem valores para serem futuramente recuperados.

Num programa orientado a objetos, no entanto, o principal papel das variáveis é o de representar o estado interno de uma classe ou de um objeto.

Variáveis são definidas quando mencionamos o nome de um tipo, e em seguida uma série de identificadores separados por vírgulas (,) e tendo no final um ponto-e-vírgula (;). Cada um dos identificadores será uma variável do tipo que encabeçou a definição.

Sendo Tipo um tipo e Var_i nomes de identificadores, temos que a forma geral de uma declaração de variáveis é como segue:

$$Tipo\ Var_1, Var_2, \ldots, Var_n;$$

Variáveis podem ser iniciadas no ato de sua definição, i.e., podem ser definidas e receber um valor inicial. Isto pode ser feito acrescentando um sinal de igual (=) e o valor inicial desejado imediatamente após o identificador da variável que desejamos iniciar.

Sendo Tipo um tipo, Var_i nomes de identificadores e $Expr_i$ expressões que resultam em valores do tipo Tipo, temos que a forma geral de uma declaração de variáveis iniciadas é como segue:

$$Tipo\ Var_1 = Expr_1, Var_2 = Expr_2, \ldots, Var_n = Expr_n;$$

Escopo

A área de código onde um identificador é visível é chamada de escopo do identificador. O escopo de um nome é determinado pela localização de sua declaração. Em C++ (como em C) existe o escopo local e o escopo global.

Em C++, no entanto, existe o chamado escopo de classe, cujo significado está intimamente ligado aos conceitos de orientação a objetos. Aprofundaremos nosso conhecimento sobre escopo de classe a medida que avançarmos no estudo de C++.

Expressões

Uma expressão é uma seqüência de operadores e operandos que especifica um computação e resulta um valor.

A ordem de avaliação de subexpresssões é determinada pela precedência e pelo agrupamento dos operadores. As regras matemáticas usuais para associatividade e comutatividade de

operadores podem ser aplicadas somente nos casos em que os operadores sejam realmente associativos e comutativos.

A ordem de avaliação dos operandos de operadores individuais é indefinida. Em particular, se um valor é modificado duas vezes numa expressão, o resultado da expressão é indefinido, exceto no caso dos operadores envolvidos garantirem alguma ordem.

Operadores Aritméticos Convencionais

Os operadores aritméticos da linguagem C++ (e também da linguagem C) são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam números. São eles:

1. + (soma);
2. - (subtração);
3. * (multiplicação);
4. / (divisão);
5. % (resto da divisão inteira); e
6. - (menos unário).

Não existe em C++ (e nem em C) um operador que realize a divisão inteira, ou, em outras palavras, em C++ (e também em C) uma divisão sempre resulta um número real. Para resolver o problema, podemos empregar um conversão de tipo para forçar o resultado da divisão a ser um inteiro. Neste caso, o real resultante terá truncada a sua parte fracionária, se transformando em um inteiro.

O operador % (resto da divisão inteira) opera sobre dois valores integrais. Seu resultado também será um valor integral. Ele resulta o resto da divisão inteira de seu primeiro operando por seu segundo operando.

O operador - (menos unário) opera sobre um único operando, e resulta o número que se obtém trocando o sinal de seu operando.

Operadores de Incremento e Decremento

Os operadores de incremento e decremento da linguagem C++ (e também da linguagem C) não são usualmente encontrados em outras linguagens de programação. Todos eles são operadores unários e operam sobre variáveis inteiras e resultam números inteiros. São eles:

1. ++ (préfixo);
2. - - (préfixo);
3. ++ (pósfixo);
4. - - (pósfixo).

O operador ++ (préfixo) incrementa seu operando e produz como resultado seu valor (já incrementado).

O operador -- (préfixo) decrementa seu operando e produz como resultado seu valor (já decrementado).

O operador ++ (pósfixo) incrementa seu operando e produz como resultado seu valor original (antes do incremento).

O operador -- (pósfixo) decrementa seu operando e produz como resultado seu valor original (antes do decremento).

Operadores Relacionais

Os operadores relacionais da linguagem C++ (e também da linguagem C) são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre números e resultam valores lógicos. São eles:

1. == (igualdade);
2. != (desigualdade);
3. < (inferioridade);
4. <= (inferioridade ou igualdade);
5. > (superioridade); e

6. >= (superioridade ou igualdade).

Operadores Lógicos

Os operadores lógicos da linguagem C++ (e também da linguagem C) são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores lógicos e resultam valores lógicos. São eles:

1. && (conjunção ou *and*);
2. || (disjunção ou *or*); e
3. ! (negação ou *not*).

Operadores de Bit

Os operadores de bit da linguagem C++ (e também da linguagem C) são, em sua maioria, aqueles que usualmente encontramos nas linguagens de programação imperativas. Todos eles operam sobre valores integrais e resultam valores integrais. São eles:

1. & (and bit a bit);
2. | (or bit a bit);
3. ^ (xor bit a bit);
4. ~ (not bit a bit);
5. << (shift left bit a bit); e
6. >> (shift right bit a bit).

Operador de Atribuição com Operação Embutida

Os operadores aritméticos e de bit podem ser combinados com o operador de atribuição para formar um operador de atribuição com operação embutida.

Sendo Var uma variável, Opr um operador aritmético ou de bit e Expr uma expressão, temos que:

Var Opr= Expr;

é equivalente a

Var = Var Opr (Expr);

Expressões Condicionais

Expressões condicionais representam uma forma compacta de escrever comandos a escolha de um dentre uma série possivelmente grande de valores. Substituem uma seqüência de ifs.

Sendo Cond uma condição booleana e $Expr_i$ expressões, temos que:

Condição? $Expr_1$: $Expr_2$

resulta $Expr_1$ no caso de Cond ser satisfeita, e $Expr_2$, caso contrário.

Conversões de Tipo

Expressões podem ser forçadas a resultar um certo tipo se usarmos conversão de tipo. Existem duas formas para fazer isso em C++.

A primeira, também encontrada na linguagem C, consiste em preceder a expressão que se deseja converter pelo tipo desejado entre parenteses.

Sendo Tipo um tipo e Expr uma expressão que resulta um valor que não é do tipo Tipo, temos que:

(Tipo) Expr

resulta a expressão Expr convertida para o tipo Tipo.

A segunda forma consiste em usar o nome do tipo para o qual se deseja converter a expressão como se fosse uma função e a expressão em questão como se fosse seu parâmetro.

Sendo Tipo um tipo e Expr uma expressão que resulta um valor que não é do tipo Tipo, temos que:

Tipo (Expr)

resulta a expressão Expr convertida para o tipo Tipo.

Entrada e Saída Básica

Em C++ podemos fazer E/S de duas formas, uma herdada da linguagem C, e outra própria da linguagem C++. Discutiremos a seguir esta última forma de fazer E/S a qual chamamos de `streams` (objetos abstratos que representam dispositivos de entrada e saída).

A biblioteca padrão de E/S em streams, que acompanha todos os ambientes C++, implementa um meio seguro, flexível e eficiente para fazer E/S de inteiros, ponto flutuante, e cadeias de caracteres.

Para usar essa biblioteca, basta incluí-la com o comando `#include <iostream.h>`.

Associados aos dispositivos padrão de entrada, saída, e erro, temos definidos, respectivamente, os *streams* `cin`, `cout`, e `cerr`.

Entrada e saída para os tipos básicos da linguagem é feita pelos operadores `<<` e `>>` respectivamente.

Sendo Var_i variáveis, temos que a leitura das mesmas poderia ser feita conforme o comando abaixo:

$$\texttt{cin >> } Var_1 \texttt{ >> } Var_2 \texttt{ >> … >> } Var_n\texttt{;}$$

Sendo $Expr_i$ expressões, temos que a escrita das mesmas poderia ser feita conforme o comando abaixo:

$$\texttt{cout << } Expr_1 \texttt{ << } Expr_2 \texttt{ << … << } Expr_n\texttt{;}$$

Sendo $Erro_i$ mensagens de erro, temos que a escrita das mesmas poderia ser feita conforme o comando abaixo:

$$\texttt{cerr << } Erro_1 \texttt{ << } Erro_2 \texttt{ << … << } Erro_n\texttt{;}$$

Comandos

Comandos são a unidade básica de processamento. Em C++ (assim como em C) não existe um repertório muito vasto de comandos. Muitos comandos que em outras linguagens existem, em C++ (e também em C) são encontrados em bibliotecas de classe.

Comandos em C++ (assim como em C), salvo raras exceções, são terminados por um ponto-e-vírgula (;).

O Comando de Atribuição

Comandos são a unidade básica de processamento de uma linguagem de programação imperativa.

O comando de atribuição tem a seguinte forma básica: em primeiro lugar vem o identificador da variável receptora da atribuição, em seguida vem o operador de atribuição (=), em seguida vem a expressão a ser atribuída, em seguida vem um ponto-e-vírgula (`;`).

Sendo Var o identificador de uma variável e Expr uma expressão, temos abaixo a forma geral do comando de atribuição:

```
Var = Expr;
```

Blocos de Comando

Já conhecemos o conceito de bloco de comandos, que nada mais é do que uma série de comandos delimitada por um par de chaves (`{}`).

Blocos de comando não são terminados por ponto-e-vírgula (`;`).

Blocos de comando são muito úteis para proporcionar a execução de uma série de subcomandos de comandos que aceitam apenas um subcomando.

O Comando `if`

O comando `if` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `if`, em seguida vem, entre parênteses, a condição a ser avaliada, em seguida vem o comando a ser executado no caso da referida condição se provar verdadeira.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando if (sem else):

```
if (Cond)
        Cmd;
```

O comando `if` pode também executar um comando, no caso de sua condição se provar falsa. Para tanto, tudo o que temos que fazer é continuar o comando `if` que já conhecemos, lhe acrescentando a palavra chave `else`, e em seguida o comando a ser executado, no caso da referida condição se provar falsa.

Sendo Cond uma condição booleana e Cmd_i dois comandos, temos abaixo forma geral do comando if com else:

```
if (Cond)
        Cmd1;
else
        Cmd2;
```

Observe que, tanto no caso da condição de um comando `if` se provar verdadeira, quanto no caso dela se provar falsa, o comando `if` somente pode executar um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a execução de mais de um comando.

Por isso, nos lugares onde se espera a especificação de um comando, podemos especificar um bloco de comandos.

O Comando `switch`

O comando `switch` tem a seguinte forma: em primeiro lugar vem a palavra-chave `switch`, em seguida vem, entre parênteses, uma expressão integral a ser avaliada, em seguida vem, entre chaves (`{}`), uma série de casos.

Um caso tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `case`, em seguida vem uma constante integral especificando o caso, em seguida vem o caractere dois pontos (`:`), em seguida vem uma série de comandos a serem executados no caso.

Sendo Expr uma expressão, $Const_I$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
        case Const1:    Cmd1a;
                        Cmd1b;
```

```
                    …
        case Const2:  Cmd2a;
                      Cmd2b;
                      …
        case Const3:  Cmd3a;
                      Cmd3b;
                      …
        …
}
```

Se, em alguma circunstância, desejarmos executar uma mesma série de comandos em mais de um caso, tudo o que temos a fazer é especificar em seqüência os casos em questão, deixando para indicar somente no último deles a referida seqüência de comandos.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
        case Const1,1:
       case Const1,2:
       …              Cmd1a;
                      Cmd1b;
                      …
        case Const2,1:
       case Const2,2:
       …              Cmd2a;
                      Cmd2b;
                      …
        case Const3,1:
       case Const3,2:
       …              Cmd3a;
                      Cmd3b;
                      …
        …
}
```

Se, em alguma circunstância, desejarmos executar uma série de comandos qualquer que seja o caso, tudo o que temos a fazer é especificar um caso da forma: em primeiro lugar vem a palavra-chave `default`, em seguida vem o caractere dois pontos (`:`), em seguida vem uma série de comandos a serem executados qualquer que seja o caso.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch:

```
switch (Expr)
{
        case Const1,1:
        case Const1,2:
        …               Cmd1a;
                        Cmd1b;
                        …
        case Const2,1:
        case Const2,2:
        …               Cmd2a;
                        Cmd2b;
                        …
        …
        default:
                CmdDa;
                        CmdDb;
                        …
}
```

É importante ficar claro que o comando switch não se encerra após a execução da seqüência de comandos associada a um caso; em vez disso, todas as seqüências de comandos, associadas aos casos subseqüentes, serão também executadas.

Se isso não for o desejado, basta terminar cada seqüência (exceto a última), com um comando break.

Sendo Expr uma expressão, $Const_{I,J}$ constantes literais do mesmo tipo que Expr e Cmd_{IJ} comandos, temos abaixo uma das formas gerais do comando switch com breaks:

```
switch (Expr)
{
        case Const1,1:
        case Const1,2:
        …               Cmd1a;
                        Cmd1b;
                        …
                        break;
        case Const2,1:
        case Const2,2:
        …               Cmd2a;
```

```
                    Cmd2b;
                    …
                    break;
        …
        default:
            CmdDa;
                    CmdDb;
                    …
}
```

Observe, abaixo, uma ilustração do uso deste comando:

```
...
int a, b, c;
char Operacao;
...
switch (Operacao)
{
    case '+': a = b + c;
              break;

    case '-': a = b - c;
              break;

    case '*': a = b * c;
              break;

    case '/': a = b / c;
}

cout << a << '\n';
...
```

O Comando `while`

O comando `while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração, em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando while:

```
while (Cond)
    Cmd;
```

Observe que, conforme especificado acima, o comando `while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim conseguiremos o efeito desejado.

O Comando `do-while`

O comando `do-while` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `do`, em seguida vem o comando a ser iterado, em seguida vem a palavra-chave `while`, em seguida vem, entre parênteses, a condição de iteração. A iteração se processa enquanto a condição de iteração se provar verdadeira.

A diferença que este comando tem com relação ao comando `while` é que este comando sempre executa o comando a ser iterado pelo menos uma vez, já que somente testa a condição de iteração após tê-lo executado. Já o comando `while` testa a condição de iteração antes de executar o comando a ser iterado, e por isso pode parar antes de executá-lo pela primeira vez.

Sendo Cond uma condição booleana e Cmd um comando, temos abaixo a forma geral do comando `do-while`:

```
do Cmd;
while (Cond);
```

Observe que, conforme especificado acima, o comando `do-while` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, em lugar de especificar o comando a ser iterado, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

O Comando `for`

O comando `for` tem a seguinte forma básica: em primeiro lugar vem a palavra-chave `for`, em seguida vem, entre parênteses e separadas por pontos-e-vírgulas (`;`), a sessão de iniciação,

a condição de iteração, e a sessão de reiniciação. Em seguida vem o comando a ser iterado. A iteração se processa enquanto a condição de iteração se provar verdadeira.

No caso de haver mais de um comando na sessão de iniciação ou na sessão de reiniciação, estes devem ser separados por virgulas (`,`).

Sendo $Cmd_{I,i}$, $Cmd_{R,i}$ e Cmd comandos, e Cond uma condição booleana, temos abaixo a forma geral do comando `for` (os números indicam a ordem de execução das partes do comando `for`):

for ($Cmd_{I,1}$, $Cmd_{I,2,...}$; Cond; $Cmd_{R,1}$, $Cmd_{R,2,...}$) Cmd;

1	**2**		**3**
	5	**4**	**6**
	8	**7**	**9**
	11	**10**	**12**
	...		
	n	**n-1**	

Observe que, conforme especificado acima, o comando `for` itera somente um comando. Isso nem sempre, ou melhor, quase nunca é satisfatório; é comum desejarmos a iteração de um conjunto não unitário de comandos.

Por isso, no lugar onde se espera a especificação de um comando, podemos especificar um bloco de comandos. Assim, conseguiremos o efeito desejado.

`Observe, abaixo`, uma ilustração do uso deste comando, que escreve na tela 10 vezes a frase "C e demais!":

```
...
for (int i=1; i<=10; i++)
    cout << "C++ e demais!\n";
...
```

equivale a

```
...
int i;
...
for (i=1; i<=10; i++)
    cout << "C++ e demais!\n";
...
```

equivale a

```
...
int i=1;

for (; i<=10; i++)
    cout << "C++ e demais!\n";
...
```

equivale a

```
...
int i=1;

for (; i<=10;)
{
    cout << "C++ e demais!\n";
    i++;
}
...
```

equivale a

```
...
int i=1;

for (;;)
{
    cout << "C++ e demais!\n";
    i++;
    if (i>10) break;
}
...
```

Observe, abaixo, outra ilustração do uso deste comando, que inverte um vetor v de 100 elementos:

```
...
int i, j;
...
for (i=0, j=99; i<j; i++, j--)
{
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
...
```

O Comando continue

O comando continue força o reinicio de uma nova iteração nos comandos while, do-while e for. Assim, o comando

```
continue;
```

provoca a saída imediata do comando while, do-while, for ou switch, dentro do qual se encontra encaixado mais diretamente.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive, seguidos pelos números de 17 a 20, inclusive:

```
...
int i = 0;

while (i<20)
{
    i++;
    if (i>7 && i<17) continue;
    cout << I << '\n';
}
...
```

O Comando break

O comando break força a saída imediata dos comandos while, do-while, for e switch (veja abaixo os comandos while, do-while e for). Assim, o comando

```
break;
```

provoca a saída imediata do comando while, do-while, for ou switch, dentro do qual se encontra encaixado mais diretamente.

Observe, abaixo, uma ilustração do uso deste comando, que escreve na tela os números de 1 a 7, inclusive:

```
...
int i = 0;

while (i<20)
{
    i++;
    if (i>7 && i<17) break;
    cout << I << '\n';
}
...
```

O Comando goto

O comando `goto`, também conhecido como desvio incondicional, força o desvio da execução para um ponto determinado do programa.

O ponto em questão deve ser marcado com um rótulo, ou, em outras palavras, o comando para onde se deseja desviar incondicionalmente a execução deve ser precedido por um identificador (que será o rótulo ao qual nos referimos) seguido pelo caractere dois pontos (`:`).

Sendo R um rótulo e Cmd_i comandos, temos abaixo a forma geral do comando goto:

```
        ...
R:      Cmd1;
        Cmd2;
        ...
        goto R;
        ...
```

Funções

Funções são definidas mencionando o tipo do retorno da função, seguido pelo identificador da função, seguido por um par de parênteses (`()`) contendo, opcionalmente, a lista dos parâmetros formais da função, seguido por um bloco de comandos representando o corpo da função.

Uma lista de parâmetros formais nada mais é do que uma série de definições de parâmetros formais separadas por vírgulas (`,`). A definição de um parâmetro formal é feita mencionando o nome do tipo do parametro e em seguida o identificador do parâmetro formal.

O tipo do retorno de uma função pode ser omitido, caso em que assumir-se-á que a mesma retorna um valor do tipo `int`.

Não existe em C++ o conceito de procedimento, muito embora possamos definir funções que não retornam valor nenhum, o que dá no mesmo. Isto pode ser feito dizendo que o tipo do retorno da função é `void`.

O comando `return` é empregado para fazer o retorno do resultado da função a seu chamante.

Parâmetros por Referência

Para passarmos um parâmetro por referência em C++, é preciso explicitamente usar ponteiros. Um parâmetro por referência, na realidade, é um ponteiro para a variável que desejamos

passar. Desta forma, quando desejamos tomar ou alterar o valor da referida variável, o fazemos sempre através do ponteiro.

A declaração de uma variável que é um ponteiro para um dado tipo é idêntica à declaração de uma variável do tipo dado, exceto pelo fato de que antepomos ao identificador da variável o caractere asterisco (`*`).

Como veremos com mais detalhes quando estivermos considerando o tipo ponteiro, existe um operador prefixo unário simbolizado pelo caractere `*` que opera sobre ponteiros. Trata-se do operador de derreferenciação. Este operador, quando aplicado a um ponteiro P, resulta no conteúdo da memória apontada por P.

Existe um operador prefixo unário simbolizado pelo caractere `&`. Este operador, quando aplicado a uma variável, resulta no endereço de memória que lhe próprio. Este operador é muito útil na passagem de parâmetros por referência.

Isto porque, quando desejamos passar uma variável como um parâmetro efetivo por referência, o que passamos não é de fato a variável, e sim o seu endereço, já que a função receptora espera receber um ponteiro.

Encapsulamento

Não podemos esquecer que a linguagem C++ é uma linguagem orientada a objetos e, como tal, permite a implementação de programas que seguem os princípios estabelecidos pelo paradigma de orientação a objetos.

Mas não podemos esquecer também que C++ é uma linguagem que vem da linguagem C e que no seu projeto foram feitas opções no sentido da manutenção das características gerais da linguagem C.

É exatamente este o caso das funções. Funções são escritas em C++ exatamente da mesma maneira que em C, embora o paradigma de orientação a objetos determine que todo processamento se dará dentro de objetos.

Desta forma, temos que, apesar de ser possível escrever funções encaixadas diretamente no programa, esta não é uma abordagem recomendada pelo paradigma de orientação a objetos. O recomendado seria que todas as funções fossem encaixadas dentro de objetos.

A função `main`

Um programa deve conter uma função de nome `main` que representa o local de início da execução do programa.

Recursão

Sabemos que o conceito de recursão se encontra entre os mais poderosos recursos de programação de que se dispõe e a linguagem C++, sendo uma linguagem completa, não poderia deixar de suportá-lo.

Trata-se da possibilidade de escrever funções que ativam outras instâncias de execução de si próprias na implementação de suas funcionalidades

Como pressupomos um bom conhecimento de programação em alguma linguagem estruturada completa (como Pascal, por exemplo), entendemos que o estudo minucioso desta técnica de programação foge ao escopo deste texto e, por isso, limitar-nos-emos a mencionar sua existência a assumir que o leitor saiba como empregá-la.

Instruções para o Pré-Processador

`#include <header.h>`

Já conhecemos a instrução `#include <header.h>`. Ela instrui o pré-processador para substituir a instrução pelo conteudo do arquivo nela especificado. O fato do nome do arquivo vim entre *angle brackets* (<>) indica que se trata de um arquivo *header* de um módulo do próprio ambiente de desenvolvimento.

Vale ressaltar que é permitido a inclusão aninhada de *header*s, i.e., é lícito incluir um *header*, sendo que este, por sua vez, inclua outros, e assim por diante.

`#include "header.h"`

A instrução `#include "header.h"` instrui o pré-processador para substituir a instrução pelo conteudo do arquivo nela especificado. O fato do nome do arquivo vim entre aspas (**""**) indica que se trata de um arquivo *header* de um módulo do usuário.

Vale ressaltar que é permitido a inclusão aninhada de *headers*, i.e., é lícito incluir um *header*, sendo que este, por sua vez, inclua outros, e assim por diante.

`#define Nome Literal`

Já conhecemos a instrução `#define Nome Literal`. Ela instrui o pré-processador sobre a definição de um nome para um determinado literal. A partir da definição, o nome será conhecido, e, sempre que mencionado no programa no decorrer da compilação, o pré-processador fará a substiuição do nome pelo literal.

Trata-se de uma forma de implementar constantes simbólicas, ou seja, constantes que, além de um valor, também têm um nome. Isto faz com que a programação se torne mais clara e mais limpa, já que, com isso, deixariamos de mencionar no programa literais sem nenhum significado, passando a mencionar nomes.

É interessante notar que o nome que esta instrução define, pode eventualmente ser parametrizado, implementando assim uma macro (algo semelhante a uma função que ao ser chamada provoca a substituição de sua invocação por sua definição).

O operador # pode ser usado na composição do `Literal` e faz com que o argumento que o sucede seja considerado uma `string`.

O operador ## pode ser usado na composição do `Literal` e tem a função de concatenar dois strings.

Caso a definição de uma macro seja muito longa, pode-se desmembrá-la em várias linhas. Para tanto, basta terminar cada uma delas (exceto a última) com o caractere \.

Vale ressaltar que é necessária atenção redobrada com macros deste tipo, pois a substituição ocorre exatamente da forma como foi definida. Isso pode levar a efeitos não imaginados.

`#define Nome`

A instrução `#define Nome` instrui o pré-processador para tornar definido o símbolo identificado por `Nome`. O propósito de definições como estas, sem a associação de valor ao símbolo definido é subsidiar as instruções de compilação condicional sobre as quais comentaremos logo mais.

#undef Nome

A instrução `#undef Nome` instrui o pré-processador para tornar indefinido o símbolo identificado por `Nome`. O propósito desta instrução é subsidiar as instruções de compilação condicional sobre as quais comentaremos logo mais.

#ifdef Nome ... #else ... #endif

Estas instruções, em conjunto com a instrução `#Define Nome` acima, promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com retiscências somente será compilado no caso do símbolo `Nome` estar definido, ao passo que o trecho de programa compreendido na segunda área marcada com retiscências somente será compilado no caso complementar.

#ifndef Nome ... #else ... #endif

Estas instruções, em conjunto com a instrução `#Define Nome` acima, promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com retiscências somente será compilado no caso do símbolo `Nome` não estar estar definido, ao passo que o trecho de programa compreendido na segunda área marcada com retiscências somente será compilado no caso complementar.

#if $ExpCte_1$…#elif $ExpCte_2$…else…#endif

Estas instruções promovem compilação condicional. O trecho de programa compreendido na primeira área marcada com retiscências somente será compilado no caso da $ExpCte_1$ ser satisfeita, o trecho de programa compreendido na segunda área marcada com retiscências somente será compilado no caso da $ExpCte_2$ ser satisfeita, e assim por diante, até que o trecho de programa compreendido na última área marcada com retiscências somente será compilado no caso de nenhuma das $ExpCte_i$ serem satisfeitas.

#line Nro NomArq

Estas instruções fazem com que o compilador pense estar compilando a linha de número `Nro` e do arquivo de nome `NomArq`. `NomArq` pode ser omitido.

`#pragma`

Várias instruções podem ser dadas ao compilador dependendo da implementação deste. Deve-se consultar o manual do compilador a fim de conhecer as possíveis diretivas deste tipo.

Organização de Programa

Uma dúvida que boa parte dos programadores iniciantes têm é como dividir um programa em módulos.

Embora não seja obrigatório particionar os programas em módulos, recomenda-se fortemente que isso seja feito, especialmente programas de um porte maior. Isso fará com que o programa se torne mais manutenível, além de minimizar a perda de tempo com compilações, já que módulos são unidades de compilação separada.

Declarações e Definições

As declarações introduzem nomes em um programa. Toda declaração é uma definição a menos que:

1. Declare uma função, externa ou não, sem incluir o corpo da função;
2. Declare uma variável externa sem iniciá-la;
3. Declare um tipo com `typedef`.

Uma variável, função ou enumerador pode ter somente uma definição, mas pode ter mais de uma declaração, naturalmente, desde que elas todas combinem entre si.

Modularização em C

Todo programa tem pelo menos um módulo, o módulo principal, que e onde encontramos a função `main`, podendo, eventualmente, ter também outros módulos.

Todo módulo, exceto o módulo principal, consiste de dois arquivos: (1) o arquivo principal do módulo; e (2) o cabeçalho do módulo. O módulo principal não tem cabeçalho, tem somente o arquivo principal do módulo.

O arquivo principal do módulo deve ter um nome com extensão `.c`, ao passo que o cabeçalho do módulo deve estar gravado em um arquivo com extensão `.h`.

Deve haver uma correspondência biunívoca entre módulos e *headers*. Isso porque o objetivo de um *header* é especificar a interface do seu módulo. Tudo o que for importante para o usuário de um módulo conseguir utilizar aquele módulo, deve estar no *header* do módulo.

Headers devem conter apenas declarações e não definições. Compoem os headers as declarações de constante, tipos, variáveis e funções. Mas note, apenas as declarações que constituirem a interface do módulo devem estar no *header* do módulo; declarações relacionadas com a implementação do módulo devem estar no próprio módulo, e portanto, ocultadas do usuário.

Para um programa poder ser executado, primeiramente cada um de seus módulos devem ser compilados a fim de gerar um arquivo objeto relocável. Os arquivos principais dos módulos são compilados diretamente, já os cabeçalhos são compilados indiretamente, sempre que incluídos no arquivo principal de um módulo.

Conceitualmente, a compilação de um módulo se dá em duas fases, a saber: o pré-processamento e a compilação efetiva.

A primeira fase realiza basicamente a inclusão de arquivos, a substituição de macros e compilações condicionais. O pré-processamento é controlado por diretivas que se iniciam pelo caractere # (não necessariamente na primeira coluna do arquivo).

O resultado do pré-processamento é uma seqüência de símbolos. Tal seqüência de símbolos é o que de fato vai ser compilado na fase seguinte, gerando um arquivo-objeto relocável.

Ato contínuo, todos os arquivos-objeto relocáveis, resultantes da compilação de cada um dos módulos do programa devem ser ligados para gerar o que chamamos de arquivo-executável.

Em ambientes integrados de desenvilvimento, a compilação dos módulos e a ligação dos arquivos objeto relocáveis se da de forma automática e relativamente transparente.

Todo módulo inclui o seu header e os headers de todos os módulos que lhe prestam serviços. O único módulo que não tem um header é o módulo principal, já que este módulo não presta

serviços a nenhum outro módulo; são os outros módulos que, diretamente ou indiretamente, lhe prestam serviços.

Quando nomes globais são definidos em um módulo, se estes não forem qualificados explicitamente como estáticos (através do qualificador `static`), outros módulos do mesmo programa poderão acessá-los se os declararem como nomes externos.

Variáveis são declaradas como externas se forem qualificadas com `extern`. Funções serão externas se as declararmos seus protótipos, i.e., as declararmos sem corpos.

Quando nomes globais são definidos em um módulo com o qualificador `static`, eles passam a ter seu escopo irremediavelmente restrito a seu módulo, i.e., outros módulos do mesmo programa não poderão em hipótese nenhuma acessá-los.

Modularização em C++

Todo programa tem pelo menos um módulo, o módulo principal, que e onde encontramos a função `main`, podendo, eventualmente, ter também outros módulos.

Todo módulo, exceto o módulo principal, consiste de dois arquivos: (1) o arquivo principal do módulo; e (2) o cabeçalho do módulo. O módulo principal não tem cabeçalho, tem somente o arquivo principal do módulo.

O arquivo principal do módulo deve ter um nome com extensão `.cpp`, ao passo que o cabeçalho do módulo deve estar gravado em um arquivo com extensão `.h`.

Deve haver uma correspondência biunívoca entre módulos e *headers*. Isso porque o objetivo de um *header* é especificar a interface do seu módulo. Tudo o que for importante para o usuário de um módulo conseguir utilizar aquele módulo, deve estar no *header* do módulo.

É recomendável que haja um módulo para cada classe implementada no programa. O *header* de um módulo deve conter apenas a declaração da classe. As definições das funções membro da classe, bem como as definições de seus dados membro de classe devem ocorrer no arquivo pricipal do módulo.

Para um programa poder ser executado, primeiramente cada um de seus módulos devem ser compilados a fim de gerar um arquivo objeto relocável. Os arquivos principais dos módulos são compilados diretamente, já os cabeçalhos são compilados indiretamente, sempre que incluídos no arquivo principal de um módulo.

Conceitualmente, a compilação de um módulo se dá em duas fases, a saber: o pré-processamento e a compilação efetiva.

A primeira fase realiza basicamente a inclusão de arquivos, a substituição de macros e compilações condicionais. O pré-processamento é controlado por diretivas que se iniciam pelo caractere # (não necessariamente na primeira coluna do arquivo).

O resultado do pré-processamento é uma seqüência de símbolos. Tal seqüência de símbolos é o que de fato vai ser compilado na fase seguinte, gerando um arquivo objeto relocável.

Ato contínuo, todos os arquivos objeto relocáveis, resultantes da compilação de cada um dos módulos do programa devem ser ligados para gerar o que chamamos de arquivo executável.

Em ambientes integrados de desenvolvimento, a compilação dos módulos e a ligação dos arquivos objeto relocáveis se da de forma automática e relativamente transparente.

Todo módulo inclui o seu header e os headers de todos os módulos que lhe prestam serviços. O único módulo que não tem um header é o módulo principal, já que este módulo não presta serviços a nenhum outro módulo; são os outros módulos que, diretamente ou indiretamente, lhe prestam serviços.

Não é recomendável que tenhamos variáveis e funções globais em C++, já que tais coisas constituiriam um mal uso da programação orienteda a objetos.

Classes

Em C++, a palavra chave `class` pode ser usada para criar uma abstração de dados. Classes são usadas da mesma forma que qualquer outro tipo, ou seja, na declaração de variáveis que armazenam objetos da classe.

Existe pouca diferença entre classes e estruturas. Os elementos de dados e as funções definidas em uma classe são conhecidos como membros da classe. As funções membro de uma classe são também chamadas de métodos.

Classes usualmente são divididas em duas partes, a saber: (1) a parte pública (normalmente contém somente métodos); e (2) a parte privativa (normalmente contém todos os elementos de dados).

Membros de Classe e de Instâncias de Classe

Podemos classificar os membros de uma classe em (1) dados e funções de classe; e (2) dados e funções de instâncias de classe.

Entendemos que membros de classe, sejam eles dados ou funções, são membros relativos à uma classe como um todo e não a nenhum objeto individual da classe. Membros de classe expressam propriedades e ações aplicáveis a toda uma classe de objetos e não a um objeto específico.

Entendemos que membros de instância de classe, sejam eles dados ou funções, são membros relativos aos objetos individuais da classe e não à classe como um todo. Membros de instância de classe expressam propriedades e ações que são aplicáveis aos objetos de uma classe individualmente e não à classe de modo geral.

Diferenciamos os dois tipos de membro por um qualificador. Variáveis e funções são de classe se forem precedidos pelo qualificador static, e são de instâncias de classe caso contrário.

Acesso aos Membros

1. **Acesso Simples:**

   Chamamos de acesso simples aqueles acesso nos quais, para acessar um membro, simplesmente mencionamos o nome do membro. São simples os seguintes acessos:

- Funções de uma classe acessando membros de classe de sua própria classe (neste caso o acesso também pode ser feito mencionando o nome de sua classe, seguido pelos caracteres: `:`, seguido pelo nome do membro que se deseja acessar);

- Funções de uma instância de classe acessando membros de classe sua própria classe (neste caso o acesso também pode ser feito mencionando o nome de sua classe, seguido pelos caracteres: `:`, seguido pelo nome do membro que se deseja acessar);

- Funções membro de uma instância de classe acessando membros de instância de classe de sua própria instância de classe (neste caso o acesso também pode ser feito mencionando a palavra chave this, seguida pelos caracteres `->`, seguido pelo nome do membro que se deseja acessar).

2. **Acesso com Nome de Classe:**

   Todo acesso não simples a um membro de classe se faz mencionando o nome da classe eu questão, seguido pelos caracteres `::`, seguido do nome do membro que se deseja acessar.

3. **Acesso com Nome de Instância de Classe:**

   Todo acesso não simples a um membro de uma instância de classe se faz mencionando o nome do objeto questão, seguido do caractere ponto (`.`), seguido do nome do membro que se deseja acessar.

Membros de Classe

Definindo a Classe Fracao

Veja abaixo a definição da classe Fracao:

[Exemplo 1 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        static int          Numerador;
        static unsigned int Denominador;
```

```
    public:
        static void AssumaValor (int,unsigned int);
        static void EscrevaSe ();
        static void SomeEmSi (int);
        static void SubtraiaDeSi (int);
};

#endif
```

Vale observar que para definir uma classe Fracao em C++ o programador terá que escolher uma forma para implementá-la fisicamente. Além disto, o programador também terá que implementar um conjunto de operações para permitir a interação de alto nível que podemos observar no exemplo acima.

Usando a Classe Fracao

Não faz sentido declarar objetos da classe Fracao, já que todos os seus membros são de classe. A classe Fracao somente pode ser usada através das operações que define. Dados privativos não podem ser acessados por funções externas.

[Exemplo 1 – princip.cpp]

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

void main ()
{
    cout << "Entre com o numerador da Fracao: ";
    int N;
    cin >> N;

    cout << "Entre com o denominador da Fracao: ";
    int D;
    cin >> D;

    cout << "\n\n";

    Fracao::AssumaValor (N,D);

    cout << "Entre com um inteiro para ser somado a Fracao: ";
    int I;
    cin >> I;

    Fracao::EscrevaSe ();
    cout << " + " << I << " = ";
    Fracao::SomeEmSi (I);
    Fracao::EscrevaSe ();
    cout << "\n\n";

    cout << "Entre com um inteiro para ser subraido da Fracao: ";
    cin >> I;
```

```
    Fracao::EscrevaSe ();
    cout << " - " << I << " = ";
    Fracao::SubtraiaDeSi (I);
    Fracao::EscrevaSe ();
    cout << "\n\n";
}
```

Escrevendo Funções Membro

Funções membro são definidas em C++ da mesma forma que se emprega em C para definir funções. O nome da função membro deve especificar sua classe através do operador de resolução de escopo (`::`).

Assim, o nome de uma função membro é sempre o nome de usa classe, seguido pelo operador de resolução de escopo (`::`), seguido pelo nome, propriamente dito, da função.

Uma função membro de classe pode referenciar: (1) seus argumentos (pelo nome); e (2) os membros estáticos de sua classe (diretamente pelos nomes dos membros ou usando o nome da classe seguido por ::).

[Exemplo 1 – fracao.cpp]

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

int          Fracao::Numerador;
unsigned int Fracao::Denominador;

void Fracao::AssumaValor (int N, unsigned int D)
{
    Fracao::Numerador   = N;
    Fracao::Denominador = D;
}

void Fracao::EscrevaSe ()
{
    cout << Fracao::Numerador << "/" << Fracao::Denominador;
}

void Fracao::SomeEmSi (int I)
{
    Fracao::Numerador = Fracao::Denominador * I + Fracao::Numerador;
}

void Fracao::SubtraiaDeSi (int I)
{
    Fracao::Numerador = Fracao::Numerador - Fracao::Denominador * I;

}
```

Membros de Instância

Definindo a Classe Fracao

Veja abaixo uma primeira definição da classe Fracao:

[Exemplo 2 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AssumaValor (int,unsigned int);
        void EscrevaSe ();
        void SomeEmSi (int);
        void SubtraiaDeSi (int);
};

#endif
```

Vale ressaltar que (1) para os usuários da classe, trata-se de um tipo abstrato de dados. Para utilizá-lo não é preciso conhecer como são representados na memória as instâncias do tipo, nem tampouco conhecer os detalhes da implementação das operações válidas para o tipo; e (2) o código que depende desta escolha fica isolado nas operações definidas para a classe e não influencia no código que trabalha com frações.

Usando Objetos da Classe Fracao

Objetos da classe Fracao podem somente ser manipulados via as operações definidas para sua classe. Dados privativos não podem ser acessados por funções externas.

[Exemplo 2 – princip.cpp]

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

void main ()
{
    Fracao F1, F2;
```

```
    cout << "Entre com o numerador de F1: ";
    int N;
    cin >> N;

    cout << "Entre com o denominador de F1: ";
    int D;
    cin >> D;

    cout << "\n\n";

    F1.AssumaValor (N,D);

    cout << "Entre com o numerador de F2: ";
    cin >> N;

    cout << "Entre com o denominador de F2: ";
    cin >> D;

    cout << "\n\n";

    F2.AssumaValor (N,D);

    cout << "Entre com um inteiro para ser somado a F1: ";
    int I;
    cin >> I;

    F1.EscrevaSe ();
    cout << " + " << I << " = ";
    F1.SomeEmSi (I);
    F1.EscrevaSe ();
    cout << "\n\n";

    cout << "Entre com um inteiro para ser subraido de F2: ";
    cin >> I;

    F2.EscrevaSe ();
    cout << " - " << I << " = ";
    F2.SubtraiaDeSi (I);
    F2.EscrevaSe ();
    cout << "\n\n";
}
```

Escrevendo Funções Membro

Funções membro são definidas em C++ da mesma forma que se emprega em C para definir funções. O nome da função membro deve especificar sua classe através do operador de resolução de escopo (`::`).

Assim, o nome de uma função membro é sempre o nome de usa classe, seguido pelo operador de resolução de escopo (`::`), seguido pelo nome, propriamente dito, da função.

Uma função membro pode referenciar: (1) seus argumentos (pelo nome); e (2) o objeto a que se vincula (diretamente pelos nomes dos membros ou usando a palavra chave `this`).

[Exemplo 2 – fracao.cpp]

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

void Fracao::AssumaValor (int N, unsigned int D)
{
    this->Numerador = N;
    this->Denominador = D;
}

void Fracao::EscrevaSe ()
{
    cout << this->Numerador << "/" << this->Denominador;
}

void Fracao::SomeEmSi (int I)
{
    this->Numerador = this->Denominador * I + this->Numerador;
}

void Fracao::SubtraiaDeSi (int I)
{
    this->Numerador = this->Numerador - this->Denominador * I;
}
```

Sobrecarga

C++ permite sobrecarga de nomes de função e de símbolos de operação, i.e., um único nome de função ou símbolo de operação para diversas operações diferentes.

Cada operação deve ter tipos de parâmetros diferentes. C++ seleciona uma operação com base nos tipos dos argumentos fornecidos.

[Exemplo 3 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AtribuaSe (int, unsigned int);

        Fracao SomeSeCom        (Fracao);
        Fracao SubtraiaDeSi     (Fracao);
        Fracao MultipliqueSePor (Fracao);
        Fracao DividaSePor      (Fracao);
```

```
        Fracao SomeSeCom        (int);
        Fracao SubtraiaDeSi     (int);
        Fracao MultipliqueSePor (int);
        Fracao DividaSePor      (int);

        void EscrevaSe ();
};

#endif
```

[Exemplo 3 – fracao.cpp]

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

void Fracao::AtribuaSe (int N, unsigned int D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }

    Numerador   = N;
    Denominador = D;
}

Fracao Fracao::SomeSeCom (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::SubtraiaDeSi (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::MultipliqueSePor (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;
```

```
    return R;
}

Fracao Fracao::DividaSePor (Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::SomeSeCom (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::SubtraiaDeSi (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::MultipliqueSePor (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::DividaSePor (int I)
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
```

```
}

void Fracao::EscrevaSe ()
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 3 – princip.cpp]

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

void main ()
{
    Fracao F, F1, F2;

    F1.AtribuaSe (1,2);
    F2.AtribuaSe (2,3);

    cout << "F1 = "; F1.EscrevaSe (); cout << '\n';
    cout << "F2 = "; F2.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1.SomeSeCom (F2);
    cout << "F1 + F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.SubtraiaDeSi (F2);
    cout << "F1 - F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.MultipliqueSePor (F2);
    cout << "F1 * F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.DividaSePor (F2);
    cout << "F1 / F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1.SomeSeCom (7);
    cout << "F1 + 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.SubtraiaDeSi (7);
    cout << "F1 - 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.MultipliqueSePor (7);
    cout << "F1 * 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.DividaSePor (7);
    cout << "F1 / 7 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';
}
```

Parâmetros com Valores Padrão

Um parâmetro com valor padrão é um parâmetro que pode ser omitido no ato da chamada da função que ele parametriza, caso em que, a linguagem assume que ele vale o seu valor padrão.

O valor padrão de um parâmetro é indicado no ato da declaração da função que ele parametriza.

Parâmetros que assumem valores padrão no caso de serem omitidos podem ser uma interessante alternativa em muitos casos de sobrecarga.

Tanto na declaração da função, quanto em sua chamada, somente os últimos parâmetros podem assumir valores padrão.

Sendo Tp_i tipos, Funcao o identificador de uma função, P_i identificadores de parâmetros e Cte_i constantes, veja a seguir a forma geral da declaração de uma função que tem parâmetros com valores padrão:

$$Tp\ Funcao\ (Tp_1\ P_1,\ldots,Tp_n\ P_n,\ Tp_{n+1}\ P_{n+1} = Cte_{n+1},\ldots,Tp_m\ P_m = Cte_m);$$

Amizade

As funções declaradas dentro de uma classe podem ser: (1) funções membro; ou (2) funções amigas.

Funções membro são declaradas dentro de uma classe e definidas vinculadas a ela. Podem acessar a parte privativa da classe à qual se vinculam e são chamadas sempre a partir de um objeto daquela classe.

Para acessarem os membros de sua classe, podem mencionar diretamente os seus nomes ou podem empregar a palavra-chave `this`.

Funções amigas são declaradas em uma ou mais classes e não são necessariamente definidas vinculadas a alguma classe. Têm acesso à parte privativa de todas as classes onde foram declaradas e não são chamadas a partir de objetos daquelas classes. Se distinguem das funções membro pela palavra-chave `friend`.

[Exemplo 4 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AtribuaSe (int, unsigned int);

        Fracao SomeSeCom        (Fracao);
        Fracao SubtraiaDeSi     (Fracao);
        Fracao MultipliqueSePor (Fracao);
        Fracao DividaSePor      (Fracao);

        Fracao SomeSeCom        (int);
        Fracao SubtraiaDeSi     (int);
        Fracao MultipliqueSePor (int);
        Fracao DividaSePor      (int);

        friend Fracao Some        (int, Fracao);
        friend Fracao Subtraia    (int, Fracao);
        friend Fracao Multiplique (int, Fracao);
        friend Fracao Divida      (int, Fracao);

        void EscrevaSe ();
};

#endif
```

[Exemplo 4 – fracao.cpp]

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

void Fracao::AtribuaSe (int N, unsigned int D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }

    Numerador   = N;
    Denominador = D;
}

Fracao Fracao::SomeSeCom (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
```

```
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::SubtraiaDeSi (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::MultipliqueSePor (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::DividaSePor (Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::SomeSeCom (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::SubtraiaDeSi (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
```

```
}

Fracao Fracao::MultipliqueSePor (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::DividaSePor (int I)
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao Some (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao Subtraia (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao Multiplique (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao Divida (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
```

```
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;

    return R;
}

void Fracao::EscrevaSe ()
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 4 – princip.cpp]

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

void main ()
{
    Fracao F, F1, F2;

    F1.AtribuaSe (1,2);
    F2.AtribuaSe (2,3);

    cout << "F1 = "; F1.EscrevaSe (); cout << '\n';
    cout << "F2 = "; F2.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1.SomeSeCom (F2);
    cout << "F1 + F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.SubtraiaDeSi (F2);
    cout << "F1 - F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.MultipliqueSePor (F2);
    cout << "F1 * F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1.DividaSePor (F2);
    cout << "F1 / F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1.SomeSeCom (7);
    cout << "F1 + 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.SubtraiaDeSi (7);
    cout << "F1 - 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.MultipliqueSePor (7);
    cout << "F1 * 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1.DividaSePor (7);
    cout << "F1 / 7 = "; F.EscrevaSe (); cout << '\n';
```

```
    cout << '\n';

    F = Some (7, F1);
    cout << "7 + F1 = "; F.EscrevaSe (); cout << '\n';

    F = Subtraia (7, F1);
    cout << "7 - F1 = "; F.EscrevaSe (); cout << '\n';

    F = Multiplique (7, F1);
    cout << "7 * F1 = "; F.EscrevaSe (); cout << '\n';

    F = Divida (7, F1);
    cout << "7 / F1 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';
}
```

Uma função pode ser amiga de mais de uma classe. Neste caso o seu protótipo aparece na declaração de cada uma das classes, sempre precedido pela palavra friend.

Tambem, uma função membro de uma classe pode ser uma função-amiga de uma outra classe. Neste caso, o seu protótipo aparece na declaração da classe da qual a função é amiga (com o operador de resolução de escopo indicando a qual classe à qual pertence a função), sempre precedido pela palavra friend.

É também possível fazer com que todas as funções membro de uma determinada classe sejam amigas de uma outra classe. Para tanto, basta que declaremos a primeira classe como amiga da segunda classe. Isto é feito mencionando o protótipo da primeira classe na declaração da segunda classe, sermpre precedido pela palavra friend. Classes-amigas representam uma alternativa para a declaração individual de cada função-amiga.

Funções Comuns

Funções que não precisam ter acesso direto à parte privativa de nenhuma classe, não precisam ser funções membro de nenhuma classe, podem ser funções soltas, como as da linguagem C.

Embora isso seja aceitável na linguagem C++, é desnecessário dizer que isso não constitui uma boa prática de programação orientada a objetos, haja visto fugir do recomendado por este paradigma.

Funções *Inline*

Qualquer tipo de função (membro, amiga e comum) pode ser *inline*.

Com funções inline é possível eliminar o tempo de processamento adicional imposto por chamadas de função.

Quando invocadas, em vez do controle ser transferido para elas, para posteriormente ser retornado para a função chamante, elas são macroexpandidas no local da chamada.

São normalmente usadas sempre que puderem ocorrer **muitas** chamadas a funções de tamanho **pequeno**.

Sobrecarga de Operadores

Para promover sobrecarga de operadores, deve-se definir funções com nomes que iniciam com a palavra-chave operator, seguida pelo símbolo do operador que se deseja definir, e.g., `+`, `*`, `[]`, etc.

Tal função pode ser definida como membro, caso em que seu operando mais a esquerda representa o objeto ao qual ela se vincula, ao passo que, todos os demais lhe são passados como argumentos (`S1 + S2` seria equivalente a `S1.operator+ (S2)`).

Tal função pode também ser definida como uma função comum, caso em que todos os seus operandos lhe são passados como argumentos (`S1 + S2` seria equivalente a `operator+ (S1, S2)`).

Quando se tem sobrecarga de operadores, fica a cargo do C++ decidir e chamar a função apropriada a cada caso, dependendo do tipo dos operandos envolvidos, livrando assim o operador de uma sobrecarga cognitiva desnecessária.

Veja o exemplo abaixo:

[Exemplo 5 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
```

```
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AtribuaSe (int, unsigned int);

        Fracao operator+ (Fracao);
        Fracao operator- (Fracao);
        Fracao operator* (Fracao);
        Fracao operator/ (Fracao);

        Fracao operator+ (int);
        Fracao operator- (int);
        Fracao operator* (int);
        Fracao operator/ (int);

        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        void EscrevaSe ();
};

#endif
```

[Exemplo 5 – fracao.cpp]

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

void Fracao::AtribuaSe (int N, unsigned int D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }

    Numerador   = N;
    Denominador = D;
}

Fracao Fracao::operator+ (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator- (Fracao F)
```

```
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator/ (Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator- (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;
```

```
    return R;
}

Fracao Fracao::operator/ (int I)
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;
```

```
    return R;
}

void Fracao::EscrevaSe ()
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 5 – princip.cpp]

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

void main ()
{
    Fracao F, F1, F2;

    F1.AtribuaSe (1,2);
    F2.AtribuaSe (2,3);

    cout << "F1 = "; F1.EscrevaSe (); cout << '\n';
    cout << "F2 = "; F2.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + F2;
    cout << "F1 + F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - F2;
    cout << "F1 - F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * F2;
    cout << "F1 * F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / F2;
    cout << "F1 / F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + 7;
    cout << "F1 + 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - 7;
    cout << "F1 - 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * 7;
    cout << "F1 * 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / 7;
    cout << "F1 / 7 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = 7 + F1;
    cout << "7 + F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 - F1;
    cout << "7 - F1 = "; F.EscrevaSe (); cout << '\n';
```

```
    F = 7 * F1;
    cout << "7 * F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 / F1;
    cout << "7 / F1 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';
}
```

Sobrecarga de Operadores Bifixos

Existem em C++ alguns operadores unários que podem ser usados, tanto como operadores prefixos, quanto como operadores posfixos (++ e --).

Tais operadores, quando usados prefixadamente têm significado diferente daquele que teria se usado posfixadamente.

Ao sobrecarregá-los, uma convenção simples diferencia um do outro:

Prefixo

É definido sem parâmetros;

Posfixo

É definido recebendo um único parâmetro do tipo int (tal parâmetro serve apenas para diferenciá-lo do operador prefixo).

[Exemplo 6 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AtribuaSe (int, unsigned int);

        Fracao operator+ (Fracao);
        Fracao operator- (Fracao);
        Fracao operator* (Fracao);
        Fracao operator/ (Fracao);

        Fracao operator+ (int);
        Fracao operator- (int);
        Fracao operator* (int);
        Fracao operator/ (int);
```

```
        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        Fracao operator++ ();
        Fracao operator-- ();

        Fracao operator ++ (int);
        Fracao operator -- (int);

        void EscrevaSe ();
};

#endif
```

[Exemplo 6 – fracao.cpp]

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

void Fracao::AtribuaSe (int N, unsigned int D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }

    Numerador   = N;
    Denominador = D;
}

Fracao Fracao::operator+ (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator- (Fracao F)
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F)
```

```
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator/ (Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator- (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I)
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator/ (int I)
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }
```

```
    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;

    return R;
}

Fracao Fracao::operator++ ()
{
    return *this = *this + 1;
}

Fracao Fracao::operator-- ()
{
    return *this = *this - 1;
}
```

```
Fracao Fracao::operator++ (int)
{
    Fracao R = *this;

    *this = *this + 1;

    return R;
}

Fracao Fracao::operator-- (int)
{
    Fracao R = *this;

    *this = *this - 1;

    return R;
}

void Fracao::EscrevaSe ()
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 6 – princip.cpp]

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

void main ()
{
    Fracao F, F1, F2;

    F1.AtribuaSe (1,2);
    F2.AtribuaSe (2,3);

    cout << "F1 = "; F1.EscrevaSe (); cout << '\n';
    cout << "F2 = "; F2.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + F2;
    cout << "F1 + F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - F2;
    cout << "F1 - F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * F2;
    cout << "F1 * F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / F2;
    cout << "F1 / F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + 7;
    cout << "F1 + 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - 7;
```

```
    cout << "F1 - 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * 7;
    cout << "F1 * 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / 7;
    cout << "F1 / 7 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = 7 + F1;
    cout << "7 + F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 - F1;
    cout << "7 - F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 * F1;
    cout << "7 * F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 / F1;
    cout << "7 / F1 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1++ + ++F2;
    cout << "F1++ + ++F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1-- - --F2;
    cout << "F1-- - --F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';
}
```

Constantes

A instrução #define do pré-processador somente pode ser usada para definir nomes simbólicos para constantes literais. Assim sendo, não é possível, com ela, definir constantes que não poderiam ser escritas literalmente.

Para este caso, C++ provê uma forma alternativa de definição de contantes. Tal forma é em tudo semelhante à declaração de variáveis, exceto porque: (1) são sempre precedidas pela palavra chave `const`; e (2) devem ser sempre iniciadas.

Naturalmente, em C++, o valor de uma constante não pode ser modificado.

Por esta razão, constantes somente podem ser passadas como parâmetro real para uma função, se o correspondente parâmetro formal for (1) passado por valor; ou (2) for qualificado com a palavra chave `const`.

Também por isto, constantes somente podem ser usadas para chamar uma função membro se esta tiver sido declarada constante pelo acrescimo da palavra chave `const` no final de seu cabeçalho.

[Exemplo 7 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        void AtribuaSe (int, unsigned int);

        Fracao operator+ (Fracao) const;
        Fracao operator- (Fracao) const;
        Fracao operator* (Fracao) const;
        Fracao operator/ (Fracao) const;

        Fracao operator+ (int) const;
        Fracao operator- (int) const;
        Fracao operator* (int) const;
        Fracao operator/ (int) const;

        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        Fracao operator++ ();
        Fracao operator-- ();

        Fracao operator ++ (int);
        Fracao operator -- (int);

        void EscrevaSe () const;
};

#endif
```

[Exemplo 7 – fracao.cpp]

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

void Fracao::AtribuaSe (int N, unsigned int D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
```

```
        exit (1);
    }

    Numerador   = N;
    Denominador = D;
}

Fracao Fracao::operator+ (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator- (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator/ (Fracao F) const
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;
```

```
    return R;
}

Fracao Fracao::operator- (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator/ (int I) const
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;
```

```
    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;

    return R;
}

Fracao Fracao::operator++ ()
{
    return *this = *this + 1;
}

Fracao Fracao::operator-- ()
{
    return *this = *this - 1;
}

Fracao Fracao::operator++ (int)
{
    Fracao R = *this;

    *this = *this + 1;

    return R;
}

Fracao Fracao::operator-- (int)
{
    Fracao R = *this;

    *this = *this - 1;

    return R;
}

void Fracao::EscrevaSe () const
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 7 – princip.cpp]

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

```
#include "fracao.h"

void main ()
{
    Fracao F, F2;

    F2.AtribuaSe (1,2);

    const Fracao F1 = F2;

    F2.AtribuaSe (2,3);

    cout << "F1 = "; F1.EscrevaSe (); cout << '\n';
    cout << "F2 = "; F2.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + F2;
    cout << "F1 + F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - F2;
    cout << "F1 - F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * F2;
    cout << "F1 * F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / F2;
    cout << "F1 / F2 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + 7;
    cout << "F1 + 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - 7;
    cout << "F1 - 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 * 7;
    cout << "F1 * 7 = "; F.EscrevaSe (); cout << '\n';

    F = F1 / 7;
    cout << "F1 / 7 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = 7 + F1;
    cout << "7 + F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 - F1;
    cout << "7 - F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 * F1;
    cout << "7 * F1 = "; F.EscrevaSe (); cout << '\n';

    F = 7 / F1;
    cout << "7 / F1 = "; F.EscrevaSe (); cout << '\n';

    cout << '\n';

    F = F1 + ++F2;
    cout << "F1 + ++F2 = "; F.EscrevaSe (); cout << '\n';

    F = F1 - F2--;
```

```
    cout << "F1 - F2-- = "; F.EscrevaSe (); cout << '\n';
    cout << '\n';
}
```

Encapsulamento de Dados

Por questões de compatibilidade com C, em C++ é possível realizar 4 operações sobre os elementos de dados (exceto vetores e funções): (1) criação (em geral com a declaração de uma variável do tipo); (2) iniciação; (3) atribuição; e (4) destruição (em geral no final do bloco onde foi declarada a variável do tipo).

Estas operações não requerem operações de definição na classe e podem depender de sutilezas da implementação escolhida, entrando em conflito com o conceito de encapsulamento de dados.

C++ resolve este conflito permitindo que o autor de uma classe proveja operações especiais para a classe (além das 4 operações padrão).

Atribuição

A atribuição de um valor a um objeto do seu próprio tipo pode ser feita através da operação padrão de atribuição (faz-se cópia de memória, como em C) ou através da definição de um `operator=`, caso em que, este Substituaria a operação padrão.

Nem sempre é conveniente o uso da operação padrão de atribuição. Nos casos em que existirem na classe ponteiros que dão acesso a estruturas dinâmicas, o uso da operação padrão de atribuição quase sempre resulta em problemas.

Atribuição de um valor a um objeto de tipo diferente do seu somente pode ser feita se um `operator=` tiver sido definido.

Criação e Destruição

O autor da classe pode controlar a criação e a destruição de objetos escrevendo funções construtoras e destrutoras.

Na criação de objetos (e.g., na declaração), aloca-se memória para o objeto e em seguida chama-se o construtor da classe (se houver um). Na destruição de objetos (e.g., quando deixa-

se o escopo da declaração) chama-se o destrutor da classe (se houver um) e libera-se a memória alocada para o objeto.

Quando um objeto é criado, o construtor de sua classe é chamado pelo próprio objeto criado. Quando um objeto é destruído, o destrutor de sua classe também é chamado pelo objeto a ser destruído.

Construtores são batizados com o nome da classe a que se referem. Destrutores são batizados com o nome da classe a que se referem precedido por um til (~)

Como os construtores e os destrutores são chamados automaticamente, não faz sentido que eles tenham retorno.

Iniciação

Diferentemente da atribuição, que dá um novo valor a um objeto existente, a iniciação cria um objeto com um valor específico. Passagem de parâmetros e retorno de resultado por valor são equivalentes a uma iniciação.

A iniciação de um objeto com um valor do seu próprio tipo será feita do seguinte modo: (1) se sua classe tem um construtor, este construtor vai ser chamado; (2) se sua classe não tem um construtor, mas seus membros tiverem, então serão chamados os construtores de seus membros (ou eventualmente os dos membros de seus membros, e assim por diante); na falta de um construtor, será usada cópia de memória como em C.

A iniciação de um objeto com um valor de tipo diferente do seu próprio tipo somente pode ser feita se sua classe tiver um construtor para tratar o caso.

[Exemplo 8 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        Fracao ();
        Fracao (int, unsigned int);
```

```
        Fracao operator+ (Fracao) const;
        Fracao operator- (Fracao) const;
        Fracao operator* (Fracao) const;
        Fracao operator/ (Fracao) const;

        Fracao operator+ (int) const;
        Fracao operator- (int) const;
        Fracao operator* (int) const;
        Fracao operator/ (int) const;

        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        Fracao operator++ ();
        Fracao operator-- ();

        Fracao operator ++ (int);
        Fracao operator -- (int);

        void EscrevaSe () const;
};

#endif
```

[Exemplo 8 – fracao.cpp]

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

Fracao::Fracao (): Numerador   (0),
                   Denominador (1)
{}

Fracao::Fracao (int N, unsigned int D): Numerador   (N),
                                        Denominador (D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }
}

Fracao Fracao::operator+ (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}
```

```
Fracao Fracao::operator- (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator/ (Fracao F) const
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator- (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
```

```
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator/ (int I) const
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;
```

```
    return R;
}

Fracao Fracao::operator++ ()
{
    return *this = *this + 1;
}

Fracao Fracao::operator-- ()
{
    return *this = *this - 1;
}

Fracao Fracao::operator++ (int)
{
    Fracao R = *this;

    *this = *this + 1;

    return R;
}

Fracao Fracao::operator-- (int)
{
    Fracao R = *this;

    *this = *this - 1;

    return R;
}

void Fracao::EscrevaSe () const
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 8 – princip.cpp]

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

void main ()
{
      Fracao F, F2 (2,3);
      const Fracao F1 (1,2);

      cout << "F1 =  "; F1.EscrevaSe (); cout << '\n';
      cout << "F2 =  "; F2.EscrevaSe (); cout << '\n';
      cout << '\n';

      F = F1 + F2;
      cout << "F1 + F2 =  "; F.EscrevaSe (); cout << '\n';

      F = F1 - F2;
      cout << "F1 - F2 =  "; F.EscrevaSe (); cout << '\n';

      F = F1 * F2;
      cout << "F1 * F2 =  "; F.EscrevaSe (); cout << '\n';
```

```
      F = F1 / F2;
      cout << "F1 / F2 =  "; F.EscrevaSe (); cout << '\n';
      cout << '\n';

      F = F1 + 7;
      cout<<"F1 + 7 = ";F.EscrevaSe();cout <<'\n';

      F = F1 - 7;
      cout<<"F1 - 7 = ";F.EscrevaSe();cout <<'\n';

      F = F1 * 7;
      cout<<"F1 * 7 = ";F.EscrevaSe();cout <<'\n';

      F = F1 / 7;
      cout<<"F1 / 7 = ";F.EscrevaSe();cout <<'\n';
      cout << '\n';

      F = 7 + F1;
      cout << "7 + F1 =  "; F.EscrevaSe (); cout << '\n';

      F = 7 - F1;
      cout << "7 - F1 =  "; F.EscrevaSe (); cout << '\n';

      F = 7 * F1;
      cout << "7 * F1 =  "; F.EscrevaSe (); cout << '\n';

      F = 7 / F1;
      cout << "7 / F1 =  "; F.EscrevaSe (); cout << '\n';
      cout << '\n';

      F = F1 + ++F2;
      cout << "F1 + ++F2 =  "; F.EscrevaSe (); cout << '\n';

      F = F1 - F2--;
      cout << "F1 - F2-- =  "; F.EscrevaSe (); cout << '\n';
      cout << '\n';
}
```

Iniciação de Membros

Pode acontecer de um construtor precisar iniciar um membro com um valor específico. C++ possui uma sintaxe específica para permitir o tratamento deste tipo de situação. Veja o exemplo abaixo:

[Exemplo 9 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

class Fracao
{
    private:
        int         Numerador;
        unsigned int Denominador;

    public:
        Fracao ();
```

```
        Fracao (int, unsigned int);

        Fracao operator+ (Fracao) const;
        Fracao operator- (Fracao) const;
        Fracao operator* (Fracao) const;
        Fracao operator/ (Fracao) const;

        Fracao operator+ (int) const;
        Fracao operator- (int) const;
        Fracao operator* (int) const;
        Fracao operator/ (int) const;

        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        Fracao operator++ ();
        Fracao operator-- ();

        Fracao operator ++ (int);
        Fracao operator -- (int);

        operator float () const;
        void  EscrevaSe () const;
};

#endif
```

[Exemplo 9 – fracao.cpp]

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

Fracao::Fracao (): Numerador   (0),
                   Denominador (1)
{}

Fracao::Fracao (int N, unsigned int D): Numerador   (N),
                                        Denominador (D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }
}

Fracao Fracao::operator+ (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
```

```
}

Fracao Fracao::operator- (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator/ (Fracao F) const
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator- (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I) const
{
    Fracao R;
```

```
    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator/ (int I) const
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}

Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;
```

```
    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;

    return R;
}

Fracao Fracao::operator++ ()
{
    return *this = *this + 1;
}

Fracao Fracao::operator-- ()
{
    return *this = *this - 1;
}

Fracao Fracao::operator++ (int)
{
    Fracao R = *this;

    *this = *this + 1;

    return R;
}

Fracao Fracao::operator-- (int)
{
    Fracao R = *this;

    *this = *this - 1;

    return R;
}

Fracao::operator float () const
{
    return (1.0 * Numerador) / (1.0 * Denominador);
}

void Fracao::EscrevaSe () const
{
    cout << Numerador << '/' << Denominador;
}
```

[Exemplo 9 – complexo.h]

```
#ifndef COMPLEXO
#define COMPLEXO

#include "fracao.h"

class Complexo
{
    private:
        Fracao PtReal, PtImag;

    public:
        Complexo ();
        Complexo (Fracao, Fracao);
```

```
        Complexo operator+ (Complexo) const;
        Complexo operator- (Complexo) const;
        Complexo operator* (Complexo) const;
        Complexo operator/ (Complexo) const;

        void EscrevaSe () const;
};

#endif
```

[Exemplo 9 – complexo.cpp]

```
#include <stdlib.h>
#include <iostream.h>
#include "fracao.h"
#include "complexo.h"


Complexo::Complexo (): PtReal (0,1), PtImag (0,1)
{}

Complexo::Complexo (Fracao PR, Fracao PI): PtReal (PR), PtImag (PI)
{}

Complexo Complexo::operator+ (Complexo C) const
{
    Complexo R ;

    R.PtReal = PtReal + C.PtReal;
    R.PtImag = PtImag + C.PtImag;

    return R;
}

Complexo Complexo::operator- (Complexo C) const
{
    Complexo R;

    R.PtReal = PtReal - C.PtReal;
    R.PtImag = PtImag - C.PtImag;

    return R;
}

Complexo Complexo::operator* (Complexo C) const
{
    Complexo R;

    R.PtReal = PtReal * C.PtReal - PtImag * C.PtImag;
    R.PtImag = PtReal * C.PtImag + PtImag * C.PtReal;

    return R;
}

Complexo Complexo::operator/ (Complexo C) const
{
    if (float (C.PtReal) == 0 && float (C.PtImag) == 0)
    {
```

```
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Complexo R;

    R.PtReal = (PtReal   * C.PtReal + PtImag   * C.PtImag) /
               (C.PtReal * C.PtReal + C.PtImag * C.PtImag);

    R.PtImag = (PtImag   * C.PtReal + PtReal   * C.PtImag) /
               (C.PtReal * C.PtReal + C.PtImag * C.PtImag);

    return R;
}

void Complexo::EscrevaSe () const
{
    PtReal.EscrevaSe ();
    cout << " + (";
    PtImag.EscrevaSe ();
    cout << ")i";
}
```

[Exemplo 9 – princip.cpp]

```
#include <iostream.h>
#include "fracao.h"
#include "complexo.h"

void main ()
{
    Complexo C, C2 (Fracao(2,3), Fracao (1,2));
    const Complexo C1 (Fracao (2,5), Fracao(3,7));

    cout << "C1 = "; C1.EscrevaSe (); cout << '\n';
    cout << "C2 = "; C2.EscrevaSe (); cout << '\n';

    cout << '\n';

    C = C1 + C2;
    cout << "C1 + C2 = "; C.EscrevaSe (); cout << '\n';

    C = C1 - C2;
    cout << "C1 - C2 = "; C.EscrevaSe (); cout << '\n';

    C = C1 * C2;
    cout << "C1 * C2 = "; C.EscrevaSe (); cout << '\n';

    C = C1 / C2;
    cout << "C1 / C2 = "; C.EscrevaSe (); cout << '\n';
}
```

Referências

Efetivamente implementam um nome adicional para um objeto já existente. A declaração de uma referência tem a seguinte forma: primeiramente vem o nome do tipo da referência, em seguida vem o caractere & (e comercial) e em seguida vem o nome da referência.

Geralmente são usadas em parâmetros e retornos de função. Têm o mesmo efeito que a passagem de um ponteiro.

[Exemplo 10 – lista.h]

```
#ifndef LISTA
#define LISTA

class Lista
{
    private:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista ();
        Lista (const Lista&);
       ~Lista ();

        Lista& operator=  (const Lista&);
        void   InsEmOrdem (int);
        int    Pertence   (int) const;
        void   Del        (int);
        void   EscrevaSe  () const;
};

#endif
```

[Exemplo 10 – lista.cpp]

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

Lista::Lista (): Inicio (NULL)
```

```
{}

Lista::Lista (const Lista& L):Inicio (NULL)
{
    *this = L;
}

Lista::~Lista ()
{
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
        Inicio = Inicio -> Prox;

        free (P);
    }

}

Lista& Lista::operator= (const Lista& L)
{
    pNo PT, PL;

    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
        Inicio = Inicio -> Prox;

        free (P);
    }


    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
        if (Inicio == NULL)
        {
            if ((Inicio = (pNo) malloc (sizeof (sNo))) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            Inicio -> Info = PL -> Info;
            Inicio -> Prox = NULL;
            PT             = Inicio;
        }
        else
        {
            if ((PT -> Prox = (pNo) malloc (sizeof (sNo))) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            PT         = PT -> Prox;
            PT -> Info = PL -> Info;
            PT -> Prox = NULL;
        }

        return *this;
}

void Lista::InsEmOrdem (int I)
{
    if (Inicio == NULL)
    {
```

```
        if ((Inicio = (pNo) malloc (sizeof (sNo))) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
        if (I < Inicio -> Info)
        {
            pNo N;

            if ((N = (pNo) malloc (sizeof (sNo))) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = Inicio;
            Inicio    = N;
        }
        else
        {
            pNo A, P;

            for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
            {
                if (P == NULL)     break;
                if (I < P -> Info) break;
            }

            pNo N;

            if ((N = (pNo) malloc (sizeof (sNo))) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = P;
            A -> Prox = N;
        }
}

int Lista::Pertence (int I) const
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I)
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }
```

```
    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P         == NULL) break;
      if (P -> Info == I   ) break;
    }

    if (P == NULL)
    {
        cerr << "\nErro: Remocao de elemento inexistente!\n\n";
        exit (1);
    }

    if (A == NULL)
        Inicio = P -> Prox;
    else
        A -> Prox = P -> Prox;

    free (P);
}

void Lista::EscrevaSe () const
{

    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        cout << P -> Info << ' ';
}
```

[Exemplo 10 – princip.cpp]

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

void main ()
{
    Lista L1;
    int   N;

    cout << "Entre com 10 numeros separados por espacos:\n";

    for (int I = 1; I <= 10; I++)
    {
        cin >> N;
        L1.InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n";

    cout << "L1 = "; L1.EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista
original:\n";

    cin >> N;

    L1.Del (N);
```

```
    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n";

    cout << "L1 = "; L1.EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco\n"
         << "para que seja verificado se pertencem ou nao a lista:\n";

    cin >> N;

    if (L1.Pertence (N))
        cout << "O primeiro numero pertence a lista";
    else
        cout << "O primeiro numero nao pertence a lista";

    cin >> N;

    if (L1.Pertence (N))
        cout << "\nO segundo numero pertence a lista";
    else
        cout << "\nO segundo numero nao pertence a lista";

    Lista L2 = L1;

    cout << "\n\nFoi feita uma copia de sua lista de numeros;";
    cout << "\nVerifique se a copia esta correta:\n";

    cout << "L2 = "; L2.EscrevaSe ();

    Lista L3;

    L3 = L1;

    cout << "\n\nFoi feita uma outra copia de sua lista de numeros;";
    cout << "\nVerifique se esta outra copia esta correta:\n";

    cout << "L3 = "; L3.EscrevaSe ();

    cout << "\n\n";
}
```

Como passar Parâmetros a uma Função

Se pergunte:

1. A função precisa alterar seus parâmetros reais?
   - Se sim, use ponteiros; eles deixarão claro este fato:

Se não, se pergunte:

2. A função precisa alterar sua cópia privativa dos seus parâmetros reais?
   - Se sim, use passagem por valor; ela criará a cópia que se faz necessária;
   - Se não, use uma referência constante.

Alocação Dinâmica

Para possibilitar um controle direto sobre a criação e a destruição de objetos, C++ provê dois operadores de alto nível especificamente projetados para este fim: `new` e `delete`.

Os operadores `new` e `delete` são definidos para todos os tipos da linguagem, mas, como todo operador, podem ser redefinidos.

Sendo Tp um tipo, Ptr um ponteiro para o tipo Tp, $Args_i$ argumentos para o construtor de Tp e Tam do vetor que pretende-se alocar, temos que os operadores `new` e `delete` têm a seguite forma geral:

```
Ptr = new Tp (Arg1,…, Argn);   // Para alocação de um objeto
Ptr = new Tp [Tam];            // Para alocação de um vetor

Delete Ptr;                    // Para desalocar um objeto
Delete [] Ptr;                 // Para desalocar um vetor
```

Herança

Um programa pode muitas vezes conter grupos de classes relacionadas (semelhantes, mas não idênticas).

Uma abordagem possível seria a criação de diversas classes independentes. Esta solução tem, no entanto, certos inconvenientes, tais como a repetição de funções membro e a impossibilidade de escrever funções polimórficas.

Outra alternativa seria a criação de uma classe única que abarcasse todas as variações. Esta solução tem também os seus problemas. Em primeiro lugar, nunca trabalharíamos, de fato, com um único tipo de classe. Em segundo lugar, esta abordagem implica necessariamente em desperdício de memória. Por fim, temos nesta abordagem uma imensa falta de transparência da implementação, o que a faz bem pouco recomendável.

Com o conceito de herança, grupos de classes relacionadas podem ser criados de maneira simples, flexível e eficiente.

Características comuns são agrupadas em uma classe base. Outras classes são derivadas da classe base.

Classes derivadas herdam todos os membros da classe base. Elas podem acrescentar novos membros àqueles herdados, bem como redefinir aqueles membros.

Posto que classes derivadas herdam as características de sua classe base, a descrição de uma classe derivada deve se concentrar nas diferenças que esta apresenta com relação a classe base, adicionando ou modificando características.

As funções acrescentadas por uma classe derivada somente podem ser usadas a partir de objetos da classe derivada. As funções definidas numa classe base podem ser usadas em objetos da classe base ou em objetos de qualquer de suas classes derivadas.

Para redefinir uma função membro de sua classe base, a classe derivada deve implementar uma função membro de mesmo nome e com parâmetros do mesmo tipo.

O conceito de herança permite a criação de classes relacionadas sem os problemas que existem com a dispersão em várias classes independentes e sem os problemas que existem com a agregação em uma classe única .

O conceito de herança elimina o problema da alta redundância de funções membro que teríamos no caso da dispersão em várias classes independentes, pois (1) a classe base declara as funções membro comuns uma única vez; e (2) as classes derivadas declaram cada função membro específica uma única vez.

Além disto, a possibilidade de redefinir as funções membro herdadas, permite que lidemos com as peculiaridades de cada caso em um nível conveniente de abstração o que não aconteceria no caso da agregação em uma classe única.

Membros Protegidos

Uma classe pode ter 3 tipos de membros. Dois deles já são nossos conhecidos. São eles: os públicos (acessíveis em toda parte) e os privativos (acessíveis somente nas operações definidas por sua classe).

Introduziremos agora o terceiro e último tipo de membro de uma classe: os protegidos (acessíveis nas operações definidas por sua classe e pelas classes derivadas dela).

[Exemplo 11 – lista.h]

```
#ifndef LISTA
#define LISTA

class Lista
{
    protected:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista ();
        Lista (const Lista&);
       ~Lista ();

        Lista& operator=  (const Lista&);
        void   InsEmOrdem (int);
        int    Pertence   (int) const;
        void   Del        (int);
        void   EscrevaSe  () const;
};

#endif
```

[Exemplo 11 – lista.cpp]

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

Lista::Lista (): Inicio (NULL)
{}

Lista::Lista (const Lista& L):Inicio (NULL)
{
    *this = L;
}

Lista::~Lista ()
{
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete P;
    }
}
```

```
Lista& Lista::operator= (const Lista& L)
{
    pNo PT, PL;

    for (PT = Inicio; Inicio != NULL; PT = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete PT;
    }

    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
        if (Inicio == NULL)
        {
            if ((Inicio = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            Inicio -> Info = PL -> Info;
            Inicio -> Prox = NULL;
            PT             = Inicio;
        }
        else
        {
            if ((PT -> Prox = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            PT        = PT -> Prox;
            PT -> Info = PL -> Info;
            PT -> Prox = NULL;
        }

        return *this;
}

void Lista::InsEmOrdem (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
        if (I < Inicio -> Info)
        {
            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }
```

```
            N -> Info = I;
            N -> Prox = Inicio;
            Inicio    = N;
        }
        else
        {
            pNo A, P;

            for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
            {
                if (P == NULL)     break;
                if (I < P -> Info) break;
            }

            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = P;
            A -> Prox = N;
        }
}

int Lista::Pertence (int I) const
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I)
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P         == NULL) break;
      if (P -> Info == I   ) break;
    }

    if (P == NULL)
    {
        cerr << "\nErro: Remocao de elemento inexistente!\n\n";
        exit (1);
    }

    if (A == NULL)
        Inicio = P -> Prox;
    else
        A -> Prox = P -> Prox;
```

```
    delete P;
}

void Lista::EscrevaSe () const
{

    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        cout << P -> Info << ' ';
}
```

[Exemplo 11 – listacpt.h]

```
#ifndef LISTACOMPLETA
#define LISTACOMPLETA

#include "lista.h"

class ListaCompleta: public Lista
{
    public:
      ListaCompleta ();
      ListaCompleta (const ListaCompleta&);

      ListaCompleta& operator= (const ListaCompleta&);

      void InsInicio (int);
      int DelInicio ();

      void InsFinal (int);
      int DelFinal ();

      void EscrevaSe () const;
};

#endif
```

[Exemplo 11 – listacpt.cpp]

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

ListaCompleta::ListaCompleta (): Lista ()
{}

ListaCompleta::ListaCompleta (const ListaCompleta& L): Lista ()
{
    *this = L;
}

ListaCompleta& ListaCompleta::operator= (const ListaCompleta& L)
{
    pNo P;

    for (P = Inicio; Inicio != NULL; P = Inicio)
```

```
    {
        Inicio = Inicio -> Prox;

        delete P;
    }

    for (P = L.Inicio; P != NULL; P = P -> Prox)
        InsEmOrdem (P -> Info);

    return *this;
}

void ListaCompleta::InsInicio (int I)
{
    pNo N;

    if ((N = new sNo) == NULL)
    {
        cerr << "\nErro: Memoria esgotada!\n\n";
        exit (1);
    }

    N -> Info = I;
    N -> Prox = Inicio;
    Inicio    = N;
}

int ListaCompleta::DelInicio ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R = Inicio -> Info;

    pNo P  = Inicio;
    Inicio = Inicio -> Prox;
    delete P;

    return R;
}

void ListaCompleta::InsFinal (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
    {
        pNo A, P, N;

        for (A = NULL, P = Inicio; P != NULL; A = P, P = P -> Prox);
```

```
        if ((N = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        N -> Info = I;
        N -> Prox = NULL;
        A -> Prox = N;
    }
}

int ListaCompleta::DelFinal ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R;

    if (Inicio -> Prox == NULL)
    {
        R = Inicio -> Info;

        delete Inicio;
        Inicio = NULL;
    }
    else
    {
        pNo A, P, D;

        for (A = Inicio, P = Inicio -> Prox, D = P -> Prox;
             D != NULL;
             A = P, P = D, D = D -> Prox);

        R = P -> Info;

        A -> Prox = NULL;
        delete P;
    }

    return R;
}

void ListaCompleta::EscrevaSe () const
{
    cout << '{';

    int Entrou = 0;

    for (pNo P = Inicio; P != NULL; P = P -> Prox, Entrou = 1)
        cout << P -> Info << ", ";

    if (Entrou)
        cout << "\b\b";

    cout << '}';
}
```

[Exemplo 11 – princip.cpp]

```
#include <iostream.h>
#include "lista.h"
#include "listacpt.h"

void main ()
{
    int   I, N;

    cout << "\n================================\n"
         << "Teste com objeto da classe Lista\n"
         << "================================\n\n";

    Lista L1;

    cout << "Entre com 10 numeros separados por espacos\n"
         << "para que sejam incluidos na lista L1:\n";

    for (I = 1; I <= 10; I++)
    {
        cin >> N;
        L1.InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n"
         << "L1 = ";

    L1.EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista L1:\n";
    cin  >> N;

    L1.Del (N);

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
         << "L1 = ";

    L1.EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco\n"
         << "para que seja verificado se pertencem ou nao a lista L1:\n";
    cin  >> N;

    if (L1.Pertence (N))
        cout << "O primeiro numero pertence a lista L1";
    else
        cout << "O primeiro numero nao pertence a lista L1";

    cin >> N;

    if (L1.Pertence (N))
        cout << "\nO segundo numero pertence a lista L1";
    else
        cout << "\nO segundo numero nao pertence a lista L1";

    Lista L2 = L1;

    cout << "\n\nFoi feita uma copia da lista L1;"
         << "\nVerifique se a copia esta correta:\n"
         << "L2 = ";
```

```
    L2.EscrevaSe ();

    Lista L3;

    L3 = L1;

    cout << "\n\nFoi feita uma outra copia da lista L1;"
         << "\nVerifique se esta outra copia esta correta:\n"
         << "L3 = ";

    L3.EscrevaSe ();

    cout << "\n\n=======================================\n"
         << "Teste com objeto da classe ListaCompleta\n"
         << "=======================================\n\n";

    ListaCompleta LC1;

    cout << "Entre com 10 numeros separados por espacos\n"
         << "para que sejam incluidos na lista LC1:\n";

    for (I = 1; I <= 10; I++)
    {
        cin >> N;
        LC1.InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n"
         << "LC1 = ";

    LC1.EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista LC1:\n";
    cin  >> N;

    LC1.Del (N);

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
         << "LC1 = ";

    LC1.EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco para\n"
         << "que seja verificado se pertencem ou nao a lista LC1:\n";
    cin  >> N;

    if (LC1.Pertence (N))
        cout << "O primeiro numero pertence a lista LC1";
    else
        cout << "O primeiro numero nao pertence a lista LC1";

    cin >> N;

    if (LC1.Pertence (N))
        cout << "\nO segundo numero pertence a lista LC1";
    else
        cout << "\nO segundo numero nao pertence a lista LC1";

    cout << "\n\nEntre com dois numeros separados por espacos\n"
         << "(um para ser incluido no inicio "
         << "e outro no final da lista LC1):\n";
```

```
    cin >> N;

    LC1.InsInicio (N);

    cin >> N;

    LC1.InsFinal (N);

    cout << "\nApos as inclusoes, a lista ficou:\n"
         << "LC1 = ";

    LC1.EscrevaSe ();

    ListaCompleta LC2 = LC1;

    cout << "\n\nFoi feita uma copia da lista LC1;"
         << "\nVerifique se a copia esta correta:\n"
         << "LC2 = ";

    LC2.EscrevaSe ();

    ListaCompleta LC3;

    LC3 = LC1;

    cout << "\n\nFoi feita uma outra copia da lista LC1;"
         << "\nVerifique se esta outra copia esta correta:\n"
         << "LC3 = ";

    LC3.EscrevaSe ();

    LC1.DelInicio ();
    LC1.DelFinal  ();

    cout << "\n\nApos a exclusao do primeiro e do ultimo, "
         << "a lista ficou:\n"
         << "LC1 = ";

    LC1.EscrevaSe ();

    cout << "\n\n====================================\n"
         << "Teste com ponteiro da classe Lista\n"
         << "apontando para objeto da classe Lista\n"
         << "====================================\n\n";

    Lista* PL1 = new Lista;

    cout << "Entre com 10 numeros separados por espacos\n"
         << "para que sejam incluidos na lista *PL1:\n";

    for (I = 1; I <= 10; I++)
    {
        cin >> N;
        PL1 -> InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n"
         << "*PL1 = ";

    PL1 -> EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista *PL1:\n";
    cin  >> N;
```

```
PL1 -> Del (N);

cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
     << "*PL1 = ";

PL1 -> EscrevaSe ();

cout << "\n\nEntre com dois numeros separados por um espaco\n"
     << "para que seja verificado se pertencem ou nao a lista *PL1:\n";
cin  >> N;

if (PL1 -> Pertence (N))
    cout << "O primeiro numero pertence a lista *PL1";
else
    cout << "O primeiro numero nao pertence a lista *PL1";

cin >> N;

if (PL1 -> Pertence (N))
    cout << "\nO segundo numero pertence a lista *PL1";
else
    cout << "\nO segundo numero nao pertence a lista *PL1";

Lista* PL2 = new Lista (*PL1);

cout << "\n\nFoi feita uma copia da lista *PL1;"
     << "\nVerifique se a copia esta correta:\n"
     << "*PL2 = ";

PL2 -> EscrevaSe ();

Lista* PL3 = new Lista;

*PL3 = *PL1;

cout << "\n\nFoi feita uma outra copia da lista *PL1;"
     << "\nVerifique se esta outra copia esta correta:\n"
     << "*PL3 = ";

PL3 -> EscrevaSe ();

delete PL1; delete PL2; delete PL3;

cout << "\n\n=============================================\n"
     << "Teste com ponteiro da classe ListaCompleta\n"
     << "apontando para objeto da classe ListaCompleta\n"
     << "=============================================\n\n";

ListaCompleta* PLC1 = new ListaCompleta;

cout << "Entre com 10 numeros separados por espacos\n"
     << "para que sejam incluidos na lista *PLC1:\n";

for (I = 1; I <= 10; I++)
{
    cin >> N;
    PLC1 -> InsEmOrdem (N);
}

cout << "\nEm ordem, os numeros digitados sao:\n"
     << "*PLC1 = ";
```

```
    PLC1 -> EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista *PLC1:\n";
    cin  >> N;

    PLC1 -> Del (N);

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
         << "*PLC1 = ";

    PLC1 -> EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco para\n"
         << "que seja verificado se pertencem ou nao a lista *PLC1:\n";
    cin  >> N;

    if (PLC1 -> Pertence (N))
        cout << "O primeiro numero pertence a lista *PLC1";
    else
        cout << "O primeiro numero nao pertence a lista *PLC1";

    cin >> N;

    if (PLC1 -> Pertence (N))
        cout << "\nO segundo numero pertence a lista *PLC1";
    else
        cout << "\nO segundo numero nao pertence a lista *PLC1";

    cout << "\n\nEntre com dois numeros separados por espacos\n"
         << "(um para ser incluido no inicio "
         << "e outro no final da lista *PLC1):\n";
    cin  >> N;

    PLC1 -> InsInicio (N);

    cin >> N;

    PLC1 -> InsFinal (N);

    cout << "\nApos as inclusoes, a lista ficou:\n"
         << "*PLC1 = ";

    PLC1 -> EscrevaSe ();

    ListaCompleta* PLC2 = new ListaCompleta (*PLC1);

    cout << "\n\nFoi feita uma copia da lista *PLC1;"
         << "\nVerifique se a copia esta correta:\n"
         << "*PLC2 = ";

    PLC2 -> EscrevaSe ();

    ListaCompleta* PLC3 = new ListaCompleta;

    *PLC3 = *PLC1;

    cout << "\n\nFoi feita uma outra copia da lista *PLC1;"
         << "\nVerifique se esta outra copia esta correta:\n"
         << "*PLC3 = ";

    PLC3 -> EscrevaSe ();

    PLC1 -> DelInicio ();
```

```
    PLC1 -> DelFinal  ();

    cout << "\n\nApos a exclusao do primeiro e do ultimo, "
         << "a lista ficou:\n"
         << "*PLC1 = ";

    PLC1 -> EscrevaSe ();

    delete PLC1; delete PLC2; delete PLC3;

    cout << "\n\n=========================================\n"
         << "TESTE COM PONTEIRO DA CLASSE Lista\n"
         << "APONTANDO PARA OBJETO DA CLASSE ListaCompleta\n"
         << "=========================================\n\n";

    PL1 = new ListaCompleta; // OBSERVE COM ATENCAO!

    cout << "Entre com 10 numeros separados por espacos\n"
         << "para que sejam incluidos na lista *PL1:\n";

    for (I = 1; I <= 10; I++)
    {
        cin >> N;
        PL1 -> InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n"
         << "*PL1 = ";

    PL1 -> EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista *PL1:\n";
    cin  >> N;

    PL1 -> Del (N);

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
         << "*PL1 = ";

    PL1 -> EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco\n"
         << "para que seja verificado se pertencem ou nao a lista *PL1:\n";
    cin  >> N;

    if (PL1 -> Pertence (N))
        cout << "O primeiro numero pertence a lista *PL1";
    else
        cout << "O primeiro numero nao pertence a lista *PL1";

    cin >> N;

    if (PL1 -> Pertence (N))
        cout << "\nO segundo numero pertence a lista *PL1";
    else
        cout << "\nO segundo numero nao pertence a lista *PL1";

    PL2 = new ListaCompleta (*(ListaCompleta*)PL1);

    cout << "\n\nFoi feita uma copia da lista *PL1;"
         << "\nVerifique se a copia esta correta:\n"
         << "*PL2 = ";
```

```
    PL2 -> EscrevaSe ();

     PL3 = new ListaCompleta;
    *PL3 = *PL1;

    cout << "\n\nFoi feita uma outra copia da lista *PL1;"
         << "\nVerifique se esta outra copia esta correta:\n"
         << "*PL3 = ";

    PL3 -> EscrevaSe ();

    delete PL1; delete PL2; delete PL3;

    cout << "\n\n";
}
```

Redefinição de Construtores

Quando uma classe derivada redefine um construtor, ambos os construtores serão executados nos objetos da classe derivada. O construtor da classe base executa primeiro. O construtor da classe derivada pode controlar os parâmetros passados ao construtor da classe base.

Verificação de Tipo em C++

Em C++, sempre que um ponteiro para objeto de um certo tipo é esperado, é possível usar um ponteiro para um objeto daquele mesmo tipo, ou então, em seu lugar, um ponteiro para um objeto de qualquer classe derivada publicamente daquele tipo.

Esta característica da linguagem permite a criação de código polimórfico, o que permite a programação adequada tanto funções que precisam trabalhar especificamente com uma certa classe derivada, quanto funções que precisam trabalhar genericamente com qualquer classe da hierarquia.

A função a ser executada quando de uma chamada é determinada com base no tipo do objeto ao qual a chamada foi vinculada, no nome usado na chamada e nos tipos dos parâmetros fornecidos.

Se o tipo do objeto declarado não bate com o tipo do objeto fornecido, então, em tempo de execução, o objeto fornecido será examinado de modo a determinar qual função deve ser executada. Chamamos a isto *Binding* Dinâmico.

Este processo requer o armazenamento de algumas estruturas de dados em tempo de execução, além de causar algum acréscimo no tempo de execução do programa. Em virtude

disto, C++ somente o emprega quando instruído explicitamente a fazê-lo através da palavra chave `virtual`.

[Exemplo 12 – lista.h]

```
#ifndef LISTA
#define LISTA

class Lista
{
    protected:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista ();
        Lista (const Lista&);
       ~Lista ();

        virtual Lista& operator= (const Lista&);

        void   InsEmOrdem (int);
        int    Pertence   (int) const;
        void   Del        (int);

        virtual void EscrevaSe () const;
};

#endif
```

[Exemplo 12 – lista.cpp]

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

Lista::Lista (): Inicio (NULL)
{}

Lista::Lista (const Lista& L):Inicio (NULL)
{
    *this = L;
}
```

```
Lista::~Lista ()
{
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete P;
    }
}

Lista& Lista::operator= (const Lista& L)
{
    pNo PT, PL;

    for (PT = Inicio; Inicio != NULL; PT = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete PT;
    }

    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
        if (Inicio == NULL)
        {
            if ((Inicio = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            Inicio -> Info = PL -> Info;
            Inicio -> Prox = NULL;
            PT             = Inicio;
        }
        else
        {
            if ((PT -> Prox = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            PT         = PT -> Prox;
            PT -> Info = PL -> Info;
            PT -> Prox = NULL;
        }

        return *this;
}

void Lista::InsEmOrdem (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
```

```
    }
    else
        if (I < Inicio -> Info)
        {
            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = Inicio;
            Inicio    = N;
        }
        else
        {
            pNo A, P;

            for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
            {
                if (P == NULL)    break;
                if (I < P -> Info) break;
            }

            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = P;
            A -> Prox = N;
        }
}

int Lista::Pertence (int I) const
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I)
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P          == NULL) break;
      if (P -> Info == I   ) break;
    }
```

```
    if (P == NULL)
    {
        cerr << "\nErro: Remocao de elemento inexistente!\n\n";
        exit (1);
    }

    if (A == NULL)
        Inicio = P -> Prox;
    else
        A -> Prox = P -> Prox;

    delete P;
}

void Lista::EscrevaSe () const
{

    for (pNo P = Inicio; P != NULL; P = P -> Prox)
        cout << P -> Info << ' ';
}
```

[Exemplo 12 – listacpt.h]

```
#ifndef LISTACOMPLETA
#define LISTACOMPLETA

#include "lista.h"

class ListaCompleta: public Lista
{
    public:
      ListaCompleta ();
      ListaCompleta (const ListaCompleta&);

      ListaCompleta& operator= (const ListaCompleta&);

      void InsInicio (int);
      int DelInicio ();

      void InsFinal (int);
      int DelFinal ();

      void EscrevaSe () const;
};

#endif
```

[Exemplo 12 – listacpt.cpp]

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

ListaCompleta::ListaCompleta (): Lista ()
```

```
{}

ListaCompleta::ListaCompleta (const ListaCompleta& L): Lista ()
{
    *this = L;
}

ListaCompleta& ListaCompleta::operator= (const ListaCompleta& L)
{
    pNo P;

    for (P = Inicio; Inicio != NULL; P = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete P;
    }

    for (P = L.Inicio; P != NULL; P = P -> Prox)
        InsEmOrdem (P -> Info);

    return *this;
}

void ListaCompleta::InsInicio (int I)
{
    pNo N;

    if ((N = new sNo) == NULL)
    {
        cerr << "\nErro: Memoria esgotada!\n\n";
        exit (1);
    }

    N -> Info = I;
    N -> Prox = Inicio;
    Inicio    = N;
}

int ListaCompleta::DelInicio ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R = Inicio -> Info;

    pNo P  = Inicio;
    Inicio = Inicio -> Prox;
    delete P;

    return R;
}

void ListaCompleta::InsFinal (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
```

```
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
    {
        pNo A, P, N;

        for (A = NULL, P = Inicio; P != NULL; A = P, P = P -> Prox);

        if ((N = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        N -> Info = I;
        N -> Prox = NULL;
        A -> Prox = N;
    }
}

int ListaCompleta::DelFinal ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R;

    if (Inicio -> Prox == NULL)
    {
        R = Inicio -> Info;

        delete Inicio;
        Inicio = NULL;
    }
    else
    {
        pNo A, P, D;

        for (A = Inicio, P = Inicio -> Prox, D = P -> Prox;
             D != NULL;
             A = P, P = D, D = D -> Prox);

        R = P -> Info;

        A -> Prox = NULL;
        delete P;
    }

    return R;
}

void ListaCompleta::EscrevaSe () const
{
    cout << '{';

    int Entrou = 0;
```

```
    for (pNo P = Inicio; P != NULL; P = P -> Prox, Entrou = 1)
        cout << P -> Info << ", ";

    if (Entrou)
        cout << "\b\b";

    cout << '}';
}
```

[Exemplo 12 – princip.cpp]

```
#include <iostream.h>
#include "lista.h"
#include "listacpt.h"

void main ()
{
    int   I, N;

    cout << "\n\n=========================================\n"
         << "TESTE COM PONTEIRO DA CLASSE Lista\n"
         << "APONTANDO PARA OBJETO DA CLASSE ListaCompleta\n"
         << "=========================================\n\n";

    Lista* PL1 = new ListaCompleta; // OBSERVE COM ATENCAO!

    cout << "Entre com 10 numeros separados por espacos\n"
         << "para que sejam incluidos na lista *PL1:\n";

    for (I = 1; I <= 10; I++)
    {
        cin >> N;
        PL1 -> InsEmOrdem (N);
    }

    cout << "\nEm ordem, os numeros digitados sao:\n"
         << "*PL1 = ";

    PL1 -> EscrevaSe ();

    cout << "\n\nEntre com um numero para ser excluido da lista *PL1:\n";
    cin  >> N;

    PL1 -> Del (N);

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n"
         << "*PL1 = ";

    PL1 -> EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco\n"
         << "para que seja verificado se pertencem ou nao a lista *PL1:\n";
    cin  >> N;

    if (PL1 -> Pertence (N))
        cout << "O primeiro numero pertence a lista *PL1";
    else
        cout << "O primeiro numero nao pertence a lista *PL1";
```

```
    cin >> N;

    if (PL1 -> Pertence (N))
        cout << "\nO segundo numero pertence a lista *PL1";
    else
        cout << "\nO segundo numero nao pertence a lista *PL1";

    Lista* PL2 = new ListaCompleta (*(ListaCompleta*)PL1);

    cout << "\n\nFoi feita uma copia da lista *PL1;"
         << "\nVerifique se a copia esta correta:\n"
         << "*PL2 = ";

    PL2 -> EscrevaSe ();

    Lista* PL3 = new ListaCompleta;

    *PL3 = *PL1;

    cout << "\n\nFoi feita uma outra copia da lista *PL1;"
         << "\nVerifique se esta outra copia esta correta:\n"
         << "*PL3 = ";

    PL3 -> EscrevaSe ();

    delete PL1; delete PL2; delete PL3;

    cout << "\n\n";
}
```

Classes Abstratas

Até onde vimos, uma classe derivada herda de sua classe base a sua implementação e seu comportamento. No entanto, podem haver situações onde precisemos de grupos de classes que tenham comportamento comum mas diferentes implementações e vice-versa (implementações comuns mas diferentes comportamentos).

Para se ter classes que compartilham somente comportamento, criamos uma classe base com funções membro públicas e sem (ou com poucos) dados privativos. Geralmente não é possível escrever código para estas funções membro, e assim temos funções membro puramente virtuais (sem corpo).

Funções membro puramente virtuais são identificadas pela iniciação `= 0` e devem ser necessariamente redefinidas nas funções membro.

Classes com uma ou mais funções membro puramente virtuais são chamadas classes abstratas. Não é permitido criar objetos de classes abstratas. Muito embora classes abstratas não contenham código, elas são úteis por permitir a criação de funções comuns polimórficas.

Veja o exemplo abaixo:

[Exemplo 13 – lista.h]

```
#ifndef LISTA
#define LISTA

class Lista
{
    private:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista ();
        Lista (const Lista&);
       ~Lista ();

        Lista& operator= (const Lista&);

        void InsInicio (int);
        int  DelInicio ();

        void InsFinal (int);
        int  DelFinal ();

        void  InsEmOrdem (int);
        int   Pertence (int) const;
        void  Del       (int);

        void  EscrevaSe () const;
};

#endif
```

[Exemplo 13 – lista.cpp]

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

Lista::Lista (): Inicio (NULL)
{}

Lista::Lista (const Lista& L): Inicio (NULL)
{
```

```
    *this = L;
}

Lista::~Lista ()
{
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
      Inicio = Inicio -> Prox;

      delete P;
    }
}

Lista& Lista::operator= (const Lista& L)
{
    pNo PT, PL;

    for (PT = Inicio; Inicio != NULL; PT = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete PT;
    }

    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
        if (Inicio == NULL)
        {
            if ((Inicio = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            Inicio -> Info = PL -> Info;
            Inicio -> Prox = NULL;
            PT             = Inicio;
        }
        else
        {
            if ((PT -> Prox = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            PT         = PT -> Prox;
            PT -> Info = PL -> Info;
            PT -> Prox = NULL;
        }

        return *this;
}

void Lista::InsInicio (int I)
{
    pNo N;

    if ((N = new sNo) == NULL)
    {
        cerr << "\nErro: Memoria esgotada!\n\n";
        exit (1);
    }
```

```
    N -> Info = I;
    N -> Prox = Inicio;
    Inicio    = N;
}

int Lista::DelInicio ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R = Inicio -> Info;

    pNo P  = Inicio;
    Inicio = Inicio -> Prox;
    delete P;

    return R;
}

void Lista::InsFinal (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
    {
        pNo A, P, N;

        for (A = NULL, P = Inicio; P != NULL; A = P, P = P -> Prox);

        if ((N = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        N -> Info = I;
        N -> Prox = NULL;
        A -> Prox = N;
    }
}

int Lista::DelFinal ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R;
```

```
    if (Inicio -> Prox == NULL)
    {
        R = Inicio -> Info;

        delete Inicio;
        Inicio = NULL;
    }
    else
    {
        pNo A, P, D;

        for (A = Inicio, P = Inicio -> Prox, D = P -> Prox;
             D != NULL;
             A = P, P = D, D = D -> Prox);

        R = P -> Info;

        A -> Prox = NULL;
        delete P;
    }

    return R;
}

void Lista::InsEmOrdem (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
        if (I < Inicio -> Info)
        {
            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = Inicio;
            Inicio    = N;
        }
        else
        {
            pNo A, P;

            for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
            {
                if (P == NULL)     break;
                if (I < P -> Info) break;
            }

            pNo N;
```

```
            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = P;
            A -> Prox = N;
        }
}

int Lista::Pertence (int I) const
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
      if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I)
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P         == NULL) break;
      if (P -> Info == I   ) break;
    }

    if (P == NULL)
    {
        cerr << "\nErro: Remocao de elemento inexistente!\n\n";
        exit (1);
    }

    if (A == NULL)
        Inicio = P -> Prox;
    else
        A -> Prox = P -> Prox;

    delete P;
}

void Lista::EscrevaSe () const
{
    cout << '{';

    int Entrou = 0;

    for (pNo P = Inicio; P != NULL; P = P -> Prox, Entrou = 1)
        cout << P -> Info << ", ";

    if (Entrou)
        cout << "\b\b";
```

```
    cout << '}';
}
```

[Exemplo 13 – pilha.h]

```
#ifndef PILHA
#define PILHA

class Pilha
{
    public:
        virtual void Empilha    (int) = 0;
        virtual int  Desempilha ()    = 0;

        virtual int Tamanho () const = 0;
};

#endif
```

[Exemplo 13 – pillista.h]

```
#ifndef PILHAEMLISTA
#define PILHAEMLISTA

#include "pilha.h"
#include "lista.h"

class PilhaEmLista: public Pilha
{
    private:
        Lista Elems;
        int   Tam;

    public:
         PilhaEmLista ();

         void Empilha    (int);
         int  Desempilha ();

         int Tamanho () const;
};

#endif
```

[Exemplo 13 – pillista.cpp]

```
#include "pillista.h"

PilhaEmLista::PilhaEmLista (): Tam (0)
{}

void PilhaEmLista::Empilha (int E)
{
```

```
    Elems.InsInicio (E);
    Tam++;
}

int PilhaEmLista::Desempilha ()
{
    Tam--;
    return Elems.DelInicio ();
}

int PilhaEmLista::Tamanho () const
{
    return Tam;
}
```

[Exemplo 13 – pilvetor.h]

```
#ifndef PILHAEMVETOR
#define PILHAEMVETOR

#include "pilha.h"

class PilhaEmVetor: public Pilha
{
    private:
        int *Base, *Topo, TamMax;

        void Overflow  ();
        void Underflow ();

    public:
         PilhaEmVetor ();
         PilhaEmVetor (int);
        ~PilhaEmVetor ();

         void Empilha   (int);
         int  Desempilha ();

         int Tamanho () const;
};

#endif
```

[Exemplo 13 – pilvetor.cpp]

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

PilhaEmVetor::PilhaEmVetor (): TamMax (100),
                               Base   (new int [100]),
                               Topo   (Base)
{}

PilhaEmVetor::PilhaEmVetor (int TM): TamMax (TM),
                                     Base   (new int [TM]),
```

```
                                        Topo   (Base)
{}

PilhaEmVetor::~PilhaEmVetor ()
{
    delete [] Base;
}

void PilhaEmVetor::Overflow ()
{
    cerr << "\nErro: Estouro de pilha (pilha cheia)\n\n";
    exit (1);
}

void PilhaEmVetor::Underflow ()
{
    cerr << "\nErro: Estouro de pilha (pilha vazia)\n\n";
    exit (1);
}

void PilhaEmVetor::Empilha (int E)
{
    if (Tamanho () == TamMax) Overflow ();
    *Topo++ = E;
}

int PilhaEmVetor::Desempilha ()
{
    if (Tamanho () == 0) Underflow ();
    return *--Topo;
}

int PilhaEmVetor::Tamanho () const
{
    return Topo - Base;
}
```

[Exemplo 13 – princip.cpp]

```
#include <iostream.h>
#include "pillista.h"
#include "pilvetor.h"

void main ()
{
    int I;

    cout << "\n-----------------------------------------------------\n"
         << "Teste com um ponteiro da classe Pilha apontando para\n"
         << "uma instancia da classe PilhaEmLista\n"
         << "-----------------------------------------------------\n\n";

    Pilha* P = new PilhaEmLista;

    cout << "Em  ordem,  foi  empilhado: ";

    for (I = 0; I <= 9; I++)
    {
        P -> Empilha (I);
```

```
        cout << I << ' ';
    }

    cout << "\nEm ordem, foi desempilhado: ";

    while (P -> Tamanho () != 0)
        cout << P -> Desempilha () << ' ';

    delete P;

    cout << "\n\n-----------------------------------------------------\n"
         << "Teste com um ponteiro da classe Pilha apontando para\n"
         << "uma instancia da classe PilhaEmVetor\n"
         << "-----------------------------------------------------\n";

    P = new PilhaEmVetor (10);

    cout << "Em  ordem,  foi  empilhado: ";

    for (I = 0; I <= 9; I++)
    {
        P -> Empilha (I);
        cout << I << ' ';
    }

    cout << "\nEm ordem, foi desempilhado: ";

    while (P -> Tamanho () != 0)
        cout << P -> Desempilha () << ' ';

    delete P;

    cout << "\n\n";
}
```

Derivações Privativas e Dados Privativos

É possível existirem classes com implementações semelhantes, apesar de terem comportamentos diferentes. Por exemplo, pilhas e conjuntos são estruturas que podem ser implementadas com uma lista ligada, apesar de se comportarem de formas completamente diferentes.

Se derivarmos as classes Pilha e Conjunto da classe Lista, poderemos utilizar as operações de lista nas pilhas e nos conjuntos.

Para compartilhar implementação, uma das abordagens possíveis é lançar mão de derivações privativas. Veja o exemplo abaixo:

[Exemplo 14 – lista.h]

```
#ifndef LISTA
```

```
#define LISTA

class Lista
{
    private:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista ();
        Lista (const Lista&);
       ~Lista ();

        Lista& operator= (const Lista&);

        void InsInicio (int);
        int  DelInicio ();

        void InsFinal (int);
        int  DelFinal ();

        void  InsEmOrdem (int);
        int   Pertence (int) const;
        void  Del       (int);

        void  EscrevaSe () const;
};

#endif
```

[Exemplo 14 – lista.cpp]

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

Lista::Lista (): Inicio (NULL)
{}

Lista::Lista (const Lista& L): Inicio (NULL)
{
    *this = L;
}

Lista::~Lista ()
{
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
```

```
      Inicio = Inicio -> Prox;

      delete P;
    }
}

Lista& Lista::operator= (const Lista& L)
{
    pNo PT, PL;

    for (PT = Inicio; Inicio != NULL; PT = Inicio)
    {
        Inicio = Inicio -> Prox;

        delete PT;
    }

    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
        if (Inicio == NULL)
        {
            if ((Inicio = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            Inicio -> Info = PL -> Info;
            Inicio -> Prox = NULL;
            PT             = Inicio;
        }
        else
        {
            if ((PT -> Prox = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            PT        = PT -> Prox;
            PT -> Info = PL -> Info;
            PT -> Prox = NULL;
        }

        return *this;
}

void Lista::InsInicio (int I)
{
    pNo N;

    if ((N = new sNo) == NULL)
    {
        cerr << "\nErro: Memoria esgotada!\n\n";
        exit (1);
    }

    N -> Info = I;
    N -> Prox = Inicio;
    Inicio    = N;
}

int Lista::DelInicio ()
{
```

```
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R = Inicio -> Info;

    pNo P  = Inicio;
    Inicio = Inicio -> Prox;
    delete P;

    return R;
}

void Lista::InsFinal (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
    {
        pNo A, P, N;

        for (A = NULL, P = Inicio; P != NULL; A = P, P = P -> Prox);

        if ((N = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        N -> Info = I;
        N -> Prox = NULL;
        A -> Prox = N;
    }
}

int Lista::DelFinal ()
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    int R;

    if (Inicio -> Prox == NULL)
    {
        R = Inicio -> Info;

        delete Inicio;
        Inicio = NULL;
    }
```

```
    else
    {
        pNo A, P, D;

        for (A = Inicio, P = Inicio -> Prox, D = P -> Prox;
             D != NULL;
             A = P, P = D, D = D -> Prox);

        R = P -> Info;

        A -> Prox = NULL;
        delete P;
    }

    return R;
}

void Lista::InsEmOrdem (int I)
{
    if (Inicio == NULL)
    {
        if ((Inicio = new sNo) == NULL)
        {
            cerr << "\nErro: Memoria esgotada!\n\n";
            exit (1);
        }

        Inicio -> Info = I;
        Inicio -> Prox = NULL;
    }
    else
        if (I < Inicio -> Info)
        {
            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }

            N -> Info = I;
            N -> Prox = Inicio;
            Inicio    = N;
        }
        else
        {
            pNo A, P;

            for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
            {
                if (P == NULL)     break;
                if (I < P -> Info) break;
            }

            pNo N;

            if ((N = new sNo) == NULL)
            {
                cerr << "\nErro: Memoria esgotada!\n\n";
                exit (1);
            }
```

```
            N -> Info = I;
            N -> Prox = P;
            A -> Prox = N;
        }
}

int Lista::Pertence (int I) const
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
      if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I)
{
    if (Inicio == NULL)
    {
        cerr << "\nErro: Remocao de lista vazia!\n\n";
        exit (1);
    }

    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P         == NULL) break;
      if (P -> Info == I   ) break;
    }

    if (P == NULL)
    {
        cerr << "\nErro: Remocao de elemento inexistente!\n\n";
        exit (1);
    }

    if (A == NULL)
        Inicio = P -> Prox;
    else
        A -> Prox = P -> Prox;

    delete P;
}

void Lista::EscrevaSe () const
{
    cout << '{';

    int Entrou = 0;

    for (pNo P = Inicio; P != NULL; P = P -> Prox, Entrou = 1)
        cout << P -> Info << ", ";

    if (Entrou)
        cout << "\b\b";

    cout << '}';
}
```

[Exemplo 14 – pilha.h]

```
#ifndef PILHA
#define PILHA

#include "lista.h"

class Pilha: private Lista
{
    private:
        int Tam;

    public:
        Pilha ();

        void Empilha    (int);
        int  Desempilha ();

        int Tamanho () const;
};

#endif
```

[Exemplo 14 – pilha.cpp]

```
#include "lista.h"
#include "pilha.h"

Pilha::Pilha (): Lista (), Tam (0)
{}

void Pilha::Empilha (int E)
{
    InsInicio (E);
    Tam++;
}

int Pilha::Desempilha ()
{
    Tam--;
    return DelInicio ();
}

int Pilha::Tamanho () const
{
    return Tam;
}
```

[Exemplo 14 – princip.cpp]

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

void main ()
```

```
{
    Pilha P;

    cout << "\nEm ordem, foi empilhado:\n";

    for (int I = 0; I <= 9; I++)
    {
        P.Empilha (I);
        cout << I << ' ';
    }

    cout << "\n\nEm ordem, foi desempilhado:\n";

    while (P.Tamanho () != 0)
        cout << P.Desempilha () << ' ';

    cout << "\n\n";
}
```

Outra abordagem possível para o problema de compartilhar somente implementação é lançar mão de dados privativos.

Derivar de uma classe e incluí-la como dado membro são abordagens que diferem em muitos pontos. Com derivação pública: (1) as classes derivadas podem ser usadas sempre onde um objeto da classe base for esperado; (2) as classes derivadas podem redefinir funções membro; (3) as classes derivadas podem acessar os membros protegidos da classe base; e (4) a sintaxe das funções membro das classes derivadas ficam ligeiramente simplificadas.

Mas, em ambas as abordagens (derivação privativa ou dados membro privativos) é possível compartilhar somente a implementação, e assim, fica proibido usar um tipo quando outro tipo é esperado.

Herança Múltipla

Mesmo dispondo de um conjunto rico de classes base já desenvolvidas, pode ser que num dado momento nos vejamos confrontados com a necessidade de um novo tipo de objeto. Pode ser ainda que este novo tipo de objeto guarde muita semelhança, não com uma, mas com muitas das classes que já temos.

Trata-se de um caso típico a ser resolvido com herança múltipla (classe derivada de mais de uma classe).

Funções membro puramente virtuais de uma classe devem ser necessariamente redefinidas em todas as classes derivadas dela.

Quando uma classe é derivada de mais de uma classe, ela pode ser usada em todos os lugares onde for esperada uma de suas classes base

No caso de uma herança múltipla, é possível que uma classe herde funções membro de mesmo nome e que tenham parâmetros formais de tipos coincidentes. Isto leva a uma ambigüidade ilegal, que pode ser resolvida de duas formas: (1) com o uso do operador de resolução de escopo (`::`); e (2) pela redefinição da função ambígua na classe derivada.

Classes Base Virtuais

Numa situação de herança múltipla, pode ser que ocorra a criação de uma classe derivada de classes base que tenham um ancestral comum. Neste caso, a classe derivada teria várias cópias dos membros do ancestral comum.

Pode-se resolver este problema fazendo com que as classes base derivem do ancestral comum através de derivações virtuais. Quando isto é feito, chamamos o ancestral comum de classe base virtual.

Derivações virtuais são úteis quando se deseja fazer muitas extensões independentes a partir de uma mesma classe base.

Quando uma classe redefine uma função membro da classe da qual ela deriva, é comum que ela chame a função original na implementação de sua redefinição.

Classes derivadas de classes base virtuais não podem simplesmente se comportar desta maneira, pois isto poderia acarretar múltiplas chamadas de uma mesma função membro da classe base virtual.

Ao redefinir uma função membro herdada, todas as classes derivadas de uma classe base virtual: (1) devem isolar em funções auxiliares todo código adicional utilizado na implementação de sua redefinição; e (2) podem somente chamar a função membro da classe base virtual e as funções auxiliares de suas classes bases imediatas.

Construtores, como qualquer outra função membro, precisam ser redefinidos de uma forma especial quando falamos de herança múltipla com classes base virtuais.

Quando não temos uma base virtual, os membros dos ancestrais comuns são duplicados, as classes derivadas passam parâmetros para os construtores das classes base imediatamente superiores, e assim por diante, até a passagem de parâmetros para o construtor do ancestral comum. O construtor do ancestral comum é chamado uma vez para cada cópia de seus membros.

Com uma base virtual, os membros dos ancestrais comuns não são duplicados e as classes derivadas passam argumentos diretamente ao construtor da classe base virtual. Os construtores das classes intermediárias não passam argumentos para o construtor da classe base virtual. O construtor do ancestral comum é chamado somente uma vez pelas classes derivadas de mais baixo nível.

Observações Finais

Herança múltipla é um conceito poderoso, apesar de ser complicado de usar e de comprometer o desempenho do programa. Em virtude disto, não deve ser usada indiscriminadamente (pense bem se ela é realmente necessária antes de usá-la).

[Exemplo 15 – pessoa.h]

```
#ifndef PESSOA
#define PESSOA

class Pessoa
{
    private:
        char DataNasc [ 9];
        char Nome     [31];
        char Endereco [51];
        char Telefone [15];
        char Fax      [15];
        char Celular  [15];

    public:
        Pessoa ();
        Pessoa (char*, char*, char*, char*, char*, char*);

        void CadastreSe ();
        void EscrevaSe  () const;
};

#endif
```

[Exemplo 15 – pessoa.cpp]

```
#include <string.h>
#include <iostream.h>
#include "pessoa.h"

Pessoa::Pessoa ()
{
    DataNasc [0] = '\0';
    Nome     [0] = '\0';
    Endereco [0] = '\0';
    Telefone [0] = '\0';
    Fax      [0] = '\0';
    Celular  [0] = '\0';
}

Pessoa::Pessoa (char* DN, char* N, char* E, char* T, char* F, char* C)
{
    strcpy (DataNasc, DN);
    strcpy (Nome    ,  N);
    strcpy (Endereco,  E);
    strcpy (Telefone,  T);
    strcpy (Fax     ,  F);
    strcpy (Celular ,  C);
}

void Pessoa::CadastreSe ()
{
    cout << "Dt Nasc.: ";
    cin  >> DataNasc;
    cout << "Nome....: ";
    cin  >> Nome;
    cout << "Endereco: ";
    cin  >> Endereco;
    cout << "Telefone: ";
    cin  >> Telefone;
    cout << "FAX.....: ";
    cin  >> Fax;
    cout << "Celular.: ";
    cin  >> Celular;
}

void Pessoa::EscrevaSe () const
{
    cout << "Dt Nasc.: " << DataNasc << '\n'
         << "Nome....: " << Nome     << '\n'
         << "Endereco: " << Endereco << '\n'
         << "Telefone: " << Telefone << '\n'
         << "FAX.....: " << Fax      << '\n'
         << "Celular.: " << Celular  << '\n';
}
```

[Exemplo 15 – aluno.h]

```
#ifndef ALUNO
```

```
#define ALUNO

#include "pessoa.h"

class Aluno: virtual public Pessoa
{
    private:
        unsigned long int RA;
        char              Curso  [41];
        char              e_mail [41];

    protected:
        void  CadPtPropria ();
        void  EscPtPropria () const;

    public:
        Aluno ();
        Aluno (char*, char*, char*, char*, char*, char*,
               unsigned long int  , char*, char*);

        void CadastreSe ();
        void EscrevaSe  () const;
};

#endif
```

[Exemplo 15 – aluno.cpp]

```
#include <string.h>
#include <iostream.h>
#include "aluno.h"

Aluno::Aluno (): Pessoa  (),
                 RA      (0L)
{
    Curso  [0] = '\0';
    e_mail [0] = '\0';
}

Aluno::Aluno (char* DN, char*   N,
              char*  E, char*   T,
              char* Fx, char*   C,
              unsigned long int R,
              char* Cs, char*  em): Pessoa (DN, N, E, T, Fx, C),
                                    RA     (R)
{
    strcpy (Curso , Cs);
    strcpy (e_mail, em);
}

void Aluno::CadPtPropria ()
{
    cout << "RA......: ";
    cin  >> RA;
    cout << "Curso...: ";
    cin  >> Curso;
    cout << "e-mail..: ";
    cin  >> e_mail;
}
```

```
void Aluno::CadastreSe ()
{
    Pessoa::CadastreSe ();

    CadPtPropria ();
}

void Aluno::EscPtPropria () const
{
    cout << "RA......: " << RA     << '\n'
         << "Curso...: " << Curso  << '\n'
         << "e-mail..: " << e_mail << '\n';
}

void Aluno::EscrevaSe () const
{
    Pessoa::EscrevaSe ();

    EscPtPropria ();
}
```

[Exemplo 15 – func.h]

```
#ifndef FUNCIONARIO
#define FUNCIONARIO

#include "pessoa.h"

class Funcionario: virtual public Pessoa
{
    private:
        char  Chefe [9];
        float Salario;
        int   QFaltasMes;
        int   QHrExtrMes;

    protected:
        void  CadPtPropria ();
        void  EscPtPropria () const;

    public:
        Funcionario ();
        Funcionario (char*, char*, char*, char*, char*, char*,
                     char*, float, int  , int);

        void CadastreSe ();
        void EscrevaSe  () const;
};

#endif
```

[Exemplo 15 – func.cpp]

```
#include <string.h>
#include <iostream.h>
#include "func.h"
```

```
Funcionario::Funcionario (): Pessoa     (),
                             Salario    (0.0),
                             QFaltasMes (0),
                             QHrExtrMes (0)
{
    Chefe [0] = '\0';
}

Funcionario::Funcionario (char* DN, char* N,
                          char*  E, char* T,
                          char* Fx, char* C,
                          char* Ch, float S,
                          int    F, int   HE): Pessoa     (DN,
                                                           N,
                                                           E,
                                                           T,
                                                          Fx,
                                                           C),
                                                Salario    ( S),
                                                QFaltasMes ( F),
                                                QHrExtrMes (HE)
{
    strcpy (Chefe, Ch);
}

void Funcionario::CadPtPropria ()
{
    cout << "Chefe...: ";
    cin  >> Chefe;
    cout << "Salario.: ";
    cin  >> Salario;
    cout << "Q.Faltas: ";
    cin  >> QFaltasMes;
    cout << "Q.H.Ext.: ";
    cin  >> QHrExtrMes;
}
void Funcionario::CadastreSe ()
{
    Pessoa::CadastreSe ();

    CadPtPropria ();
}

void Funcionario::EscPtPropria () const
{
    cout << "Chefe...: " << Chefe      << '\n'
         << "Salario.: " << Salario    << '\n'
         << "Q.Faltas: " << QFaltasMes << '\n'
         << "Q.H.Ext.: " << QHrExtrMes << '\n';
}

void Funcionario::EscrevaSe () const
{
    Pessoa::EscrevaSe ();

    EscPtPropria ();
}
```

[Exemplo 15 – monitor.h]

```
#ifndef MONITOR
#define MONITOR

#include "aluno.h"
#include "func.h"

class Monitor: public Aluno, public Funcionario
{
    private:
        char Materia  [21];
        char Depto    [31];
        char PrfSuper [31];

    public:
        Monitor ();
        Monitor (char*, char*, char*, char*, char*, char*,
                 unsigned long int  , char*, char*,
                 char*, float, int  , int,
                 char*, char*, char*);

        void CadastreSe ();
        void EscrevaSe  () const;
};

#endif
```

[Exemplo 15 – monitor.cpp]

```
#include <string.h>
#include <iostream.h>
#include "monitor.h"

Monitor::Monitor (): Pessoa (), Aluno (), Funcionario ()
{
    Materia  [0] = '\0';
    Depto    [0] = '\0';
    PrfSuper [0] = '\0';
}

Monitor::Monitor (char* DN, char*   N,
                  char*  E, char*   T,
                  char* Fx, char*   C,
                  unsigned long int R,
                  char* Cs, char*  em,
                  char* Cf, float   S,
                  int    F, int    HE,
                  char*  M, char*   D,
                  char*  PS)          : Pessoa      (DN, N,
                                                     E, T,
                                                     Fx, C),
                                        Aluno       (DN, N,
                                                     E, T,
                                                     Fx, C,
                                                     R, Cs,
                                                     em),
```

```
                                          Funcionario (DN,  N,
                                                       E,  T,
                                                      Fx,  C,
                                                      Cf,  S,
                                                       F, HE)
{
    strcpy (Materia ,  M);
    strcpy (Depto   ,  D);
    strcpy (PrfSuper, PS);
}

void Monitor::CadastreSe ()
{
    Pessoa     ::CadastreSe   ();
    Aluno      ::CadPtPropria ();
    Funcionario::CadPtPropria ();

    cout << "Materia.: ";
    cin  >> Materia;
    cout << "Depto...: ";
    cin  >> Depto;
    cout << "Prf.Sup.: ";
    cin  >> PrfSuper;
}

void Monitor::EscrevaSe () const
{
    Pessoa     ::EscrevaSe ();
    Aluno      ::EscPtPropria ();
    Funcionario::EscPtPropria ();

    cout << "Materia.: " << Materia  << '\n'
         << "Depto...: " << Depto    << '\n'
         << "Prf.Sup.: " << PrfSuper << '\n';
}
```

[Exemplo 15 – princip.cpp]

```
#include <iostream.h>
#include "pessoa.h"
#include "aluno.h"
#include "func.h"
#include "monitor.h"

void main ()
{
    Pessoa      P ("19/01/66",
                   "Jose da Silva",
                   "R Oscar R Alves, 949",
                   "623.1656",
                   "623.0796",
                   "998.2307");

    Aluno       A ("19/01/66",
                   "Jose da Silva",
                   "R Oscar R Alves, 949",
                   "623.1656",
                   "623.0796",
                   "998.2307",
```

```
                    830078,
                    "Ciencia da Computacao",
                    "jose@provedor.com.br");

    Funcionario F ("19/01/66",
                    "Jose da Silva",
                    "R Oscar R Alves, 949",
                    "623.1656",
                    "623.0796",
                    "998.2307",
                    "Joao de Souza",
                    2000.00, 2, 7);

    Monitor     M ("19/01/66",
                    "Jose da Silva",
                    "R Oscar R Alves, 949",
                    "623.1656",
                    "623.0796",
                    "998.2307",
                    830078,
                    "Ciencia da Computacao",
                    "jose@provedor.com.br",
                    "Joao de Souza",
                    2000.00f, 2, 7,
                    "POO", "Dep.Ling.Prog.",
                    "Maria Alves");

    cout << "\nObjeto Pessoa construido com o construtor:\n";

    P.EscrevaSe ();

    cout << "\n\nObjeto Aluno construido com o construtor:\n";

    A.EscrevaSe ();

    cout << "\n\nObjeto Funcionario construido com o construtor:\n";

    F.EscrevaSe ();

    cout << "\n\nObjeto Monitor construido com o construtor:\n";

    M.EscrevaSe ();

    cout << "\n\nCadastro de Pessoa:\n";

    P.CadastreSe ();

    cout << "\n\nCadastro de Aluno:\n";

    A.CadastreSe ();

    cout << "\n\nObjeto Funcionario:\n";

    F.CadastreSe ();

    cout << "\n\nCadastro de Monitor:\n";

    M.CadastreSe ();

    cout << "\n\nObjeto Pessoa cadastrado:\n";

    P.EscrevaSe ();
```

```
    cout << "\n\nObjeto Aluno cadastrado:\n";
    A.EscrevaSe ();
    cout << "\n\nObjeto Funcionario cadastrado:\n";
    F.EscrevaSe ();
    cout << "\n\nObjeto Monitor cadastrado:\n";
    M.EscrevaSe ();
    cout << '\n';
}
```

Streams

C++ não possui facilidades específicas para E/S. Tais facilidades podem ser facilmente implementadas através da própria linguagem.

A biblioteca padrão de E/S em streams, que acompanha todos os ambientes C++, implementa um meio seguro, flexível e eficiente para fazer E/S de inteiros, ponto flutuante, e cadeias de caracteres.

A interface desta biblioteca é encontrada em `<iostream.h>`.

Estrutura Básica

`istream` e `ostream` são classes que contém operadores de conversão de formatos. `iostream` é uma classe derivada de `istream` e `ostream`, e essencialmente não tem membros.

`ios` é a classe base virtual das classes `istream` e `ostream`, fatorando funções comuns às duas classes, e.g., associação a arquivos, estado do *stream*, etc.

A classe `streambuf` é apontada pela classe `ios`. Implementa mecanismos de entrada e saída básicos. Lida com a acumulação de caracteres em buffers.

Dispositivos Padrão

Associados aos dispositivos padrão de entrada, saída, e erro, temos definidos, respectivamente, os *streams* `cin`, `cout`, e `cerr`.

Entrada e saída para os tipos básicos da linguagem (`char*`, `char&`, `short int&`, `int&`, `long int&`, `float&`, `double float&`) feita pelos operadores << e >> respectivamente.

Recursos

1. Destacamos abaixo alguns dentre os mais importantes e úteis métodos disponibilizados pala classe `ios`:

   – `int bad ()`:

   Predicado que verifica se ocorreu um erro de E/S.

   – `void clear ()`:

   Reajusta os flags controladores de erros e falhas de operação sobre streams para indicar estado normal de operação.

   – `int eof ()`:

   Predicado que verifica se foi encontrado fim de arquivo.

   – `int fail ()`:

   Predicado que verifica se a última operação sobre um stream falhou.

   – `char fill ()`:

   Retorna o caractere correntemente usado para o preenchimento de áreas de escrita inutilizadas.

   – `char fill (char)`:

   Assume um novo caractere a ser usado no preenchimento de áreas de escrita inutilizadas, bem como retorna o caractere correntemente em uso para tal finalidade.

   – `int good ()`:

   Predicado que verifica se não ocorreu um erro de E/S.

   – `void precision (int)`:

   Função sem retorno que ajusta a quantidade de casas decimais a serem empregadas para a escrita de um número real (seu efeito é limitado à próxima operação de escrita).

– `long setf (long)`:

Ajusta as flags de formatação de um stream. Retorna o valor atual dessas flags. No caso de uma formatação composta, i.e., uma formatação que envolve mais de uma flag, tudo o que precisa ser feito é empregá-las a todas, sempre separadas por um ous-lógicos (`|`).As flags de formatação válidas são dadas pela lista abaixo:

- `ios::skipws`: escolhe ignorar (pular) espaços (brancos e tabs);
- `ios::left`: escolhe alinhamento a esquerda;
- `ios::right`: escolhe alinhamento a direita;
- `ios::internal`: escolher alinhamento de sinal a esquerda e de valor numérico a direita;
- `ios::dec`: escolhe operar na base 10;
- `ios::oct`: escolhe operar na base 8;
- `ios::hex`: escolhe operar na base 16;
- `ios::showbase`: escolhe indicar a base numérica na qual está operando (precedendo números escritos com 0x se a base for a 16 e com 0 se a base for 8);
- `ios::showpoint`: escolhe exibir números com zeros iniciais;
- `ios::showcase`: escolhe usar letras maiusculas para escrever números na base 16 e em notação científica;
- `ios::showpos`: escolhe mostrar o sinal de '+' na escrita de números positivos;
- `ios::scientific`: escolhe escrever números reais em notação científica;
- `ios::fixed`: escolhe escrever números reais em notação convencional (não em notação científica);
- `ios::unitbuf`: escolhe esvaziar o buffer após cada operação de saída;

- `ios::stdio`: escolhe esvaziar o buffer após a escrita de cada caractere.

– `ostream* tie ()`:

Dissocia um stream de um stream de saída; devolve um ponteiro para o ostream presentemente associado.

– `ostream* tie (ostream*)`:

Associa um stream a um stream de saída; devolve um ponteiro para o ostream presentemente associado.

– `void width (int)`:

Função sem retorno que ajusta o tamanho da área de escrita (seu efeito é limitado à próxima operação de escrita).

2. Destacamos abaixo alguns dentre os mais importantes e úteis métodos disponibilizados pala classe `istream`:

– `int gcount ()`:

Retorna a quantidade de caracteres lidos pela última invocação de um método de leitura não formatada (`get`, `getline` e `read`).

– `int get ()`:

Lê e retorna um caractere do `istream` (ou `EOF`).

– `istream& get (signed char* Buf, int Len, char Del = '\n')`:

Lê caracteres e os posiciona em Buf. Para quando Len-1 caracteres já tiverem sido lidos, ou quando for encontrado EOF, ou ainda quando for encontrada o delimitador Del. O delimitador não é lido.

– `istream& get (unsigned char* Buf, int Len,char Del = '\n')`:

Lê caracteres e os posiciona em Buf. Para quando Len-1 caracteres já tiverem sido lidos, ou quando for encontrado EOF, ou ainda quando for encontrada o delimitador Del. O delimitador não é lido; acrescenta sempre um '\0' no final do Buf.

- `istream& get (unsigned char&)`:

  Lê um caractere do `istream` (ou `EOF`).

- `istream& get (signed char&)`:

  Lê um caractere do `istream` (ou `EOF`).

- `istream& get (streambuf&, char = '\n')`:

  Lê caracteres e os posiciona no streambuf dado ate que o caracrere delimitador seja encontrado.

- `streampos& getline (signed char* Buf, int Len,char = '\n')`:

  Lê caracteres e os posiciona em `Buf`. Para quando `Len-1` caracteres já tiverem sido lidos, ou quando for encontrado `EOF`, ou ainda quando for encontrada o delimitador `Del`. O delimitador é lido mas não é colocado em `Buf`.

- `istream& getline (unsigned char*Buf,int Len,char Del='\n')`:

  Lê caracteres e os posiciona em `Buf`. Para quando `Len-1` caracteres já tiverem sido lidos, ou quando for encontrado `EOF`, ou ainda quando for encontrada o delimitador `Del. O` delimitador é lido; acrescenta sempre um `'\0'` no final do `Buf`.

- `istream& ignore (int, int)`

  Despreza no máximo a quantidade de bytes expressa por seu primeiro parâmetro (default = 1) parando o processo caso o caractere expresso por seu segundo parâmetro seja encontrado.

- `int peek ()`:

  Análogo a um `get` seguido por um `putback`.

- `istream& putback (int)`:

  Recoloca um caractere lido em um `istream`.

- `istream& read (signed char*, int)`

  Lê no máximo a quantidade de caracteres expressa por seu segundo parâmetro e os

armazena no vetor que é seu primeiro parâmetro. Use o método `gcount` para saber a quantidade de caracteres de fato lidos no caso de erro.

- `istream& read (unsigned char*, int)`
  Lê no máximo a quantidade de caracteres expressa por seu segundo parâmetro e os armazena no vetor que é seu primeiro parâmetro. Use o método `gcount` para saber a quantidade de caracteres de fato lidos no caso de erro.

- `istream& seekg (long)`:
  Posiciona um `istream` em uma posição absoluta.

- `istream& seekg (long, seek_dir)`:
  Posiciona um `istream` em uma posição dada por um deslocamento com relação a um referencial (pode valer `ios::beg`, `ios::cur` ou `ios::end`).

- `streampos tellg ()`:
  Retorna a posição absoluta corrente de um `istream`.

3. Destacamos abaixo alguns dentre os mais importantes e úteis métodos disponibilizados pala classe `ostream`:

   - `ostream &put (char)`:
     Escreve um caractere do `ostream`.

   - `ostream &flush ()`:
     Força a efetivação imediata de escritas pendentes no `ostream`.

   - `istream& seekp (long)`:
     Posiciona um `istream` em uma posição absoluta.

   - `istream& seekp (long, seek_dir)`:
     Posiciona um `istream` em uma posição dada por um deslocamento com relação a um referencial (pode valer `ios::beg`, `ios::cur` ou `ios::end`).

   - `streampos tellg ()`:
     Retorna a posição absoluta corrente de um `istream`.

- `ostream& write (const signed char*, int)`:

  Escreve a quantidade de caracteres expressa por seu segundo parâmetro. Os caracteres em questão devem encontrar-se em seu primeiro parâmetro.

- `ostream& write (const unsigned char*, int)`:

  Escreve a quantidade de caracteres expressa por seu segundo parâmetro. Os caracteres em questão devem encontrar-se em seu primeiro parâmetro.

Extensibilidade

C++ provê ainda um modelo de extensão da biblioteca para contemplar tipos definidos pelo usuário.

Veja o exemplo abaixo:

[Exemplo 16 – fracao.h]

```
#ifndef FRACAO
#define FRACAO

#include <iostream.h>

class Fracao
{
    private:
        int          Numerador;
        unsigned int Denominador;

    public:
        Fracao ();
        Fracao (int, unsigned int);

        Fracao operator+ (Fracao) const;
        Fracao operator- (Fracao) const;
        Fracao operator* (Fracao) const;
        Fracao operator/ (Fracao) const;

        Fracao operator+ (int) const;
        Fracao operator- (int) const;
        Fracao operator* (int) const;
        Fracao operator/ (int) const;

        friend Fracao operator+ (int, Fracao);
        friend Fracao operator- (int, Fracao);
        friend Fracao operator* (int, Fracao);
        friend Fracao operator/ (int, Fracao);

        Fracao operator++ ();
        Fracao operator-- ();

        Fracao operator ++ (int);
```

```
        Fracao operator -- (int);

        friend ostream& operator<< (ostream&, const Fracao&);
        friend istream& operator>> (istream&, Fracao&);
};

#endif
```

[Exemplo 16 – fracao.cpp]

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

Fracao::Fracao (): Numerador   (0),
                   Denominador (1)
{}
Fracao::Fracao (int N, unsigned int D): Numerador   (N),
                                        Denominador (D)
{
    if (D == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }
}

Fracao Fracao::operator+ (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador +
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator- (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador -
                    this -> Denominador * F.Numerador;

    R.Denominador = this -> Denominador * F.Denominador;

    return R;
}

Fracao Fracao::operator* (Fracao F) const
{
    Fracao R;

    R.Numerador   = this -> Numerador   * F.Numerador;
    R.Denominador = this -> Denominador * F.Denominador;

    return R;
```

```
}

Fracao Fracao::operator/ (Fracao F) const
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador   * F.Denominador;
    R.Denominador = this -> Denominador * F.Numerador;

    return R;
}

Fracao Fracao::operator+ (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador + this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator- (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador - this -> Denominador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator* (int I) const
{
    Fracao R;

    R.Numerador   = this -> Numerador * I;
    R.Denominador = this -> Denominador;

    return R;
}

Fracao Fracao::operator/ (int I) const
{
    if (I == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = this -> Numerador;
    R.Denominador = this -> Denominador * I;

    return R;
}
```

```
Fracao operator+ (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador + F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator- (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Denominador - F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator* (int I, Fracao F)
{
    Fracao R;

    R.Numerador   = I * F.Numerador;
    R.Denominador = F.Denominador;

    return R;
}

Fracao operator/ (int I, Fracao F)
{
    if (F.Numerador == 0)
    {
        cerr << "\nErro: Divisao por zero!\n\n";
        exit (1);
    }

    Fracao R;

    R.Numerador   = I * F.Denominador;
    R.Denominador = F.Numerador;

    return R;
}

Fracao Fracao::operator++ ()
{
    return *this = *this + 1;
}

Fracao Fracao::operator-- ()
{
    return *this = *this - 1;
}

Fracao Fracao::operator++ (int)
{
    Fracao R = *this;

    *this = *this + 1;
```

```
    return R;
}

Fracao Fracao::operator-- (int)
{
    Fracao R = *this;

    *this = *this - 1;

    return R;
}

ostream& operator<< (ostream& OS, const Fracao& F)
{
    return (OS << F.Numerador << '/' << F.Denominador);
}

istream& operator>> (istream& IS, Fracao& F)
{
    char C;

    IS >> F.Numerador >> C >> F.Denominador;

    if (C != '/')
    {
        cerr << "\nErro: Barra (/) esperada!\n\n";
        exit (1);
    }

    if (F.Denominador == 0)
    {
        cerr << "\nErro: Denominador zero!\n\n";
        exit (1);
    }

    return IS;
}
```

[Exemplo 16 – princip.cpp]

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

void main ()
{
    Fracao F1, F2;

    cout << "Entre com F1: "; cin >> F1;
    cout << "Entre com F2: "; cin >> F2;

    cout << '\n';

    cout << "F1 = " << F1 << '\n';
    cout << "F2 = " << F2 << '\n';

    cout << '\n';

    cout << "F1 + F2 = " << F1 + F2 << '\n';
    cout << "F1 - F2 = " << F1 - F2 << '\n';
```

```
    cout << "F1 * F2 = " << F1 * F2 << '\n';
    cout << "F1 / F2 = " << F1 / F2 << '\n';

    cout << '\n';

    cout << "F1 + 7 = " << F1 + 7 << '\n';
    cout << "F1 - 7 = " << F1 - 7 << '\n';
    cout << "F1 * 7 = " << F1 * 7 << '\n';
    cout << "F1 / 7 = " << F1 / 7 << '\n';

    cout << '\n';

    cout << "7 + F1 = " << 7 + F1 << '\n';
    cout << "7 - F1 = " << 7 - F1 << '\n';
    cout << "7 * F1 = " << 7 * F1 << '\n';
    cout << "7 / F1 = " << 7 / F1 << '\n';

    cout << '\n';

    cout << "F1 + ++F2 = " << F1 + ++F2 << '\n';
    cout << "F1 - F2-- = " << F1 - --F2 << '\n';

    cout << '\n';
}
```

Manipuladores

Manipuladores têm como finalidade simplificar a invocação de métodos de streams. Isto é alcançado fazendo com que essas invocações tomem forma de uma escrita (para o caso dos streams de saída) ou de uma leitura (para o caso dos streams de entrada).

Por exemplo, se desejássemos esvaziar o buffer de saída de um stream, em vez de escrever

```
cout << V;
cout.flush ();
```

escreveríamos simplesmente

```
cout << V << flush;
```

Manipuladores Padrão

- `dec`: escolhe operar na base 10;
- `oct`: escolhe operar na base 8;
- `hex`: escolhe operar na base 16;
- `flush`: força a efetivação imediata de escritas pendentes no `ostream`.

- `endl`: análogo à escrita de um `'\n'` seguido por um `flush`;
- `ends`: análogo à escrita de um `'\0'` seguido por um `flush`; e
- `ws`: escolhe ignorar (pular) espaços (brancos e tabs);.

Manipuladores Definidos em `<iomanip.h>`

- `setbase (int)`: escolhe operar em determinada base numérica (bases válidas: 8, 10 e 16);
- `setw (int)`: escolhe o tamanho da área de escrita (seu efeito é limitado à próxima operação de escrita);
- `setprecision (int)`: escolhe a quantidade de casas decimais a serem empregadas para a escrita de um número real (seu efeito é limitado à próxima operação de escrita);
- `setfill (char)`: escolhe o caractere a ser usado no preenchimento de áreas de escrita inutilizadas;
- `setiosflags (long)`: escolhe um ajuste nas flags de formatação de um stream. No caso de uma formatação composta, i.e., uma formatação que envolve mais de uma flag, tudo o que precisa ser feito é empregá-las a todas, sempre separadas por um ous-lógicos (`|`).As flags de formatação válidas são a mesmas informadas acima para a função `setf` da classe `ios`.

Apesar de encontrarmos nas bibliotecas da linguagem C++ diversos manipuladores definidos, podemos, se assim desejarmos, criar nossos próprios manipuladores. Para isso podemos empregar a própria linguagem C++.

Veja abaixo dois exemplos que ilustram, respectivamente, como implementar manipuladores sem parâmetros e manipuladores com parâmetros.

[Exemplo 17 – manip.h]

```
#ifndef MANIP
#define MANIP

#include <iostream.h>

class Manipulador
```

```
{};

extern Manipulador MeuFormato;

ostream& operator<< (ostream& S, const Manipulador&);

#endif
```

[Exemplo 17 – manip.cpp]

```
#include "manip.h"

Manipulador MeuFormato;

ostream& operator<< (ostream& S, const Manipulador&)
{
    S.precision (3);
    S.width (4);

    return S;
}
```

[Exemplo 17 – princip.cpp]

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

void main ()
{
    cout << "\nPI = " << MeuFormato << 3.1415926536 << "\n\n";
}
```

[Exemplo 18 – manip.h]

```
#ifndef MANIP
#define MANIP

#include <iostream.h>

class Manipulador
{
    private:
        int I;
        ostream& (*F) (ostream&, int);

    public:
        Manipulador (ostream& (*) (ostream&, int), int);
        friend ostream& operator<< (ostream&, const Manipulador&);
};

extern Manipulador AjustaPrecisao (int);
extern ostream& operator<< (ostream&, const Manipulador&);
```

```
#endif
```

[Exemplo 18 – manip.cpp]

```
#include "manip.h"

Manipulador::Manipulador (ostream& (*FF) (ostream&, int), int II): F (FF),
                                                                   I (II)
{}

static ostream& Precisao (ostream& S, int I)
{
    S.precision (I);
    return S;
}

Manipulador AjustaPrecisao (int P)
{
    return Manipulador (&Precisao, P);
}

ostream& operator<< (ostream& S, const Manipulador& M)
{
    return (*M.F) (S, M.I);
}
```

[Exemplo 18 – princip.cpp]

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

void main ()
{
    cout << "\n1/3 = " << AjustaPrecisao (3) << 1.0/3.0 << "\n\n";
}
```

Especialização para Arquivos

Deriva-se `filebuf` de `streambuf`. Deriva-se virtualmente `fstreambase` de `ios`. Deriva-se `ifstream` de `istream` e `fstreambase`. Deriva-se `ofstream` de `ostream` e `fstreambase`. Deriva-se `fstream` de `iostream` e `fstreambase`.

`ifstreams` são por *default* abertos para leitura, e `ofstreams` são por *default* abertos para escrita. Os construtores de ambas as classes podem receber um argumento adicional especificando o tipo de abertura.

Os possíveis tipos de abertura são:

- ios::in: abre para leitura;
- ios::out: abre para escrita;
- ios::ate: abre e posiciona no eof;
- ios::app: abre para acrescentar;
- ios::trunc: esvazia o arquivo
- ios::nocreate: falha se o arquivo não existir;
- ios::noreplace: falha se o arquivo existir;
- ios::binary: arquivo binário (nao texto).

Todas as operações aplicáveis sobre `ifstreams` e `ofstreams` são também aplicáveis também a `fstreams`.

Streams não precisam ser explicitamente abertos, pois isto é automaticamente feito pelo construtor da classe caso lhe seja fornecido o nome do arquivo. Caso isto precise ser feito em um momento diferente do de criação do *stream*, então pode-se empregar a função `open ()` que promove explicitamente a abertura do *stream*.

Streams não precisam ser explicitamente fechados, pois isto é automaticamente feito pelo destrutor da classe. Caso isto precise ser feito antes do fim do escopo da definição do *stream*, então pode-se empregar a função `close ()` que promove explicitamente o fechamento do *stream*.

São declarados em `<fstream.h>`.

Veja o exemplo abaixo:

[Exemplo 19 – princip.cpp]

```
#include <fstream.h>

int main (int argc, char *argv [])
{
    if (argc != 3)
```

```
    {
        cerr << "\nErro: Numero Invalido de Argumentos!\n\n";
        return 1;
    }

    ifstream E (argv [1], ios::in | ios::binary | ios::nocreate);

    if (!E)
    {
        cerr << "\nErro: Impossivel Abrir o Arquivo "
             << argv [1]
             << "!\n\n";

        return 1;
    }

    ofstream S (argv [2], ios::out | ios::binary);

    if (!S)
    {
        cerr << "\nErro: Impossivel Abrir o Arquivo "
             << argv [2]
             << "!\n\n";

        return 1;
    }

    char C;

    while (!E.eof ())
    {
        E.get (C);
        S.put (C);
    }

    return 0;
}
```

Especialização para Strings

De maneira parecida à que acontece na especialização para arquivos, as classes `istrstreams` e `ostrstreams` implementam operações de entrada e saída em cadeias de caracteres.

O caractere '\0' que marca o final das cadeias de caracteres é interpretado como eof.

São declarados em `<strstrea.h>`.

Veja o exemplo abaixo:

[Exemplo 20 – princip.cpp]

```
#include <string.h>
#include <strstrea.h>
```

```
void main ()
{
    cout << '\n';

    char*      S  = "O essencial e invisivel aos olhos";
    istrstream E (S, strlen (S) + 1);

    char P [30];

    while (!E.eof ())
    {
        E >> P;
        cout << P << endl;
    }

    cout << '\n';
}
```

Templates

Template é um conceito avançado de programação C++ que permite a definição de classes, funções membro e funções comuns genéricas.

São especialmente úteis para a definição de classes container, i.e., classes que armazenam objetos de outro tipo (listas, pilhas, etc).

Veja o exemplo abaixo:

[Exemplo 21 – pilha.h]

```
#include <stdlib.h>

template <class TpElem> class Pilha
{
    private:
        TpElem *Base, *Topo;
        int TamMax;

        void Overflow  ()
        {
            cout << "\nErro: Estouro de pilha (pilha cheia)\n\n";
            exit (1);
        }

        void Underflow ()
        {
            cout << "\nErro: Estouro de pilha (pilha vazia)\n\n";
            exit (1);
        }


    public:
        Pilha (int TM): TamMax (TM),
```

```
                        Base   (new int [TM]),

                        Topo   (Base)
        {}

        ~Pilha ()
        {
            delete [] Base;
        }

        void Empilha (TpElem E)
        {
            if (Tamanho () == TamMax) Overflow ();
            *Topo++ = E;
        }

        TpElem Desempilha ()
        {
            if (Tamanho () == 0) Underflow ();
            return *--Topo;
        }

        int Tamanho () const
        {
            return Topo - Base;
        }
};
```

[Exemplo 21 – princip.cpp]

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

void main ()
{
    cout << '\n';

    int I;
    Pilha<int> P (10);

    for (I = 1991; I < 2001; I++)
    {
        P.Empilha (I);
        cout << I << ' ';
    }

    cout << '\n';

    while (P.Tamanho () != 0)
        cout << P.Desempilha () << ' ';

    cout << "\n\n";
}
```

Para poder observar a definição das funções membro de um template de classe no caso de suas funções não serem *inline*, veja o exemplo abaixo:

[Exemplo 22 – pilha.h]

```
#include <stdlib.h>

template <class TpElem> class Pilha
{
    private:
        TpElem *Base, *Topo;
        int TamMax;

        void Overflow  ();
        void Underflow ();

    public:
         Pilha (int);
        ~Pilha ();
         void Empilha (TpElem);
         TpElem Desempilha ();
         int Tamanho () const;
};

template<class TpElem> void Pilha<TpElem>::Overflow ()
{
    cout << "\nErro: Estouro de pilha (pilha cheia)\n\n";
    exit (1);
}

template<class TpElem> void Pilha<TpElem>::Underflow ()
{
    cout << "\nErro: Estouro de pilha (pilha vazia)\n\n";
    exit (1);
}

template<class TpElem> Pilha<TpElem>::Pilha (int TM): TamMax (TM),
                                                      Base   (new int
[TM]),
                                                      Topo   (Base)
{}

template<class TpElem> Pilha<TpElem>::~Pilha ()
{
    delete [] Base;
}

template<class TpElem> void Pilha<TpElem>::Empilha (TpElem E)
{
    if (Tamanho () == TamMax) Overflow ();
    *Topo++ = E;
}

template<class TpElem> TpElem Pilha<TpElem>::Desempilha ()
{
    if (Tamanho () == 0) Underflow ();
    return *--Topo;
}

template<class TpElem> int Pilha<TpElem>::Tamanho () const
{
    return Topo - Base;
}
```

[Exemplo 22 – princip.cpp]

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

void main ()
{
    cout << '\n';

    int I;
    Pilha<int> P (10);

    for (I = 1991; I < 2001; I++)
    {
        P.Empilha (I);
        cout << I << ' ';
    }

    cout << '\n';

    while (P.Tamanho () != 0)
        cout << P.Desempilha () << ' ';

    cout << "\n\n";
}
```

Templates de Função

O uso de classes template implica no uso de funções membro template. Alem destas, podem também ser definidas templates de funções comuns. Um template de função define uma família de funções da mesma forma que um template de classe define uma família de classes. Note que as funções membro de um template de classe são templates de função.

Conversões de Tipo

Conversões implícitas de tipo nunca são aplicadas aos argumentos dos templates de função; sempre que possível novas versões de função são geradas. Quando isto não for o desejado, deve-se aplicar conversão de tipo explicitamente.

Argumento de um Template

Os argumentos de um template não precisam ser necessariamente o nome de um tipo (podem também ser nomes de variáveis, de função e expressões constantes). Argumentos de templates

funcionais necessariamente precisam afetar o tipo de pelo menos um dos argumentos das funções geradas pelo template. Templates de classe não são afetados por esta restrição.

Exceções

O autor de uma biblioteca pode detectar erros com os quais não consegue lidar. O usuário de uma biblioteca pode querer tratar adequadamente erros que não é capaz de detectar. O conceito de exceção vem para prover meios para resolver este tipo de situação.

A idéia fundamental é que uma função que detecta um problema que não é capaz de tratar **lança** uma exceção que pode ser **pega** por uma função que quer tratar o problema mas que não é capaz de detectar.

Uma exceção em C++ é uma instância de um tipo primitivo da linguagem ou de um tipo definido pelo usuário.

Métodos que podem lançar exceções devem deixar este fato claro no seu cabeçalho. Quando declaramos o protótipo de uma função devemos especificar as exceções que são eventualmente lançadas pela função. Isto é feito pelo acréscimo da palavra chave throw no final da declaração do protótipo:

1. **Função que pode lançar qualquer exceção:**
   ```
   TpRet Fcao ();
   ```
2. **Função que não lança exceções:**
   ```
   TpRet Fcao () throw ();
   ```
3. **Função que lança somente as exceções x, y, e z:**
   ```
   TpRet Fcao () throw (x, y, z);
   ```

Quando um método chama outro que, por sua vez pode lançar uma exceção, o primeiro deve:

- Tratar a referida exceção; ou
- Avisar através da palavra chave throws o possível lançamento da referida exceção.

Sendo Excecao o identificador de exceção, veja abaixo a forma geral do comando que lança uma exceção.

```
throw Excecao ()
```

A chamada de um métodos que lançam exceções deve ocorrer da seguinte maneira: primeiro deve vir a palavra chave try; em seguida deve vir, entre chaves ({}) as chamadas dos métodos que podem causar o lançamento de exceções.

O processo de **lançar** e **pegar** exceções envolve uma busca por um tratador na cadeia de ativações a partir do ponto onde a exceção foi **lançada**.

A partir do momento que um tratador **pega** uma exceção já considera-se a exceção tratada, e qualquer outro tratador que possa ser posteriormente encontrado se torna neutralizado.

Exceções são pegas pelo comando catch. Um try pode ser sucedido por um número arbitrariamente grande de catches, cada qual com o objetivo de tratar uma das várias exceções que podem ser lançadas dentro do try.

O comando catch tem a seguinte forma geral: primeiro vem a palavra chave catch e, em seguida, entre parênteses (()), o nome da exceção que deve ser tratada. Logo após deve vir, entre chaves ({}), o código que trata a referida exceção.

Uma classe pode **lançar** várias exceções, e as funções usuárias podem ou não **pegá-las** a todas. Exceções não pegas em um certo nível podem ser pegas em um nível mais alto.

Tratadores de exceção podem **pegar** uma exceção mas não conseguir tratá-la completamente. Neste caso pode ser necessário um novo **lançamento** de exceção para provocar um tratamento complementar em um nível mais alto.

Tratadores mais sofisticados podem precisar receber dados. Como exceções nada mais são objetos de um certo tipo, pode-se perfeitamente implementar exceções que carreguem dados consigo.

Sendo Excecao o identificador de exceção e Dd_i os dados que ela carrega, veja abaixo a forma geral do comando que lança uma exceção que transporta dados.

```
throw Excecao (Dd1,…,Ddn)
```

Podem existir conjuntos de exceções cuja natureza e tratamento sejam semelhantes. Podemos implementar exceções deste tipo como um agrupamento de exceções .

Eventualmente pode-se desejar construir um tratador que **pegue** e trate um grupo de exceções **lançado** pelo bloco try que o precede. Para tanto basta escrever um catch que trata uma exceção de alto nível na hierarquia de exceções (talvez a raiz).

Pode ser ainda que seja desejável construir um tratador que **pegue** e trate qualquer exceção **lançada** pelo bloco try que o precede. Para tanto basta escrever um catch que trata uma exceção representada por retiscências (…). Tais tratadores podem funcionar como uma espécie de "caso contrário" em um seqüência de tratadores.

Pode acontecer que um tratador, após ter **pego** uma exceção, perceba que não tem condições de tratá-la. Neste caso ele pode abdicar de seu tratamento, **relançando-a** para ser tratada em um nível léxico superior. Isto pode ser feito através do seguinte comando:

```
throw;
```

Posto que construtores são chamados automaticamente, temos que construtores não retornam resultado. Exceções podem ser uma forma interessante e única de retornar a informação de uma falha no processo de construção.

A função `unexpected ()` é chamada sempre que uma função lança uma exceção fora daquelas especificadas em seu protótipo.

A função `unexpected ()` executa a última função passada como parâmetro para uma chamada da função `set_unexpected ()`.

O padrão é que a função `unexpected ()` chame a função `terminate ()`.

A função `terminate ()` é chamada sempre que uma exceção é **lançada** e não é **pega**. A função `terminate ()` executa a última função passada como parâmetro para uma chamada da função `set_terminate ()`.

O padrão é que a função `terminate ()` chame a função `abort ()`.

[Exemplo 23 – lista.h]

```
#ifndef LISTA
#define LISTA

class ExcessaoDeLista                                {};
class MemoriaEsgotada            : public ExcessaoDeLista {};
```

```
class RemocaoDeListaVazia         : public ExcessaoDeLista {};
class RemocaoDeElementoInexistente: public ExcessaoDeLista {};

class Lista
{
    private:
        typedef
            struct sNo
            {
                int Info;
                struct sNo *Prox;
            }
            sNo;

        typedef
            sNo* pNo;


        pNo Inicio;

    public:
        Lista () throw ();
        Lista (const Lista&) throw (MemoriaEsgotada);
       ~Lista () throw ();

        Lista& operator= (const Lista&) throw (MemoriaEsgotada);

        void InsInicio (int) throw (MemoriaEsgotada);
        int  DelInicio () throw (RemocaoDeListaVazia);

        void InsFinal (int) throw (MemoriaEsgotada);
        int  DelFinal () throw (RemocaoDeListaVazia);

        void  InsEmOrdem (int) throw (MemoriaEsgotada);
        int   Pertence   (int) const throw ();
        void  Del        (int) throw (RemocaoDeListaVazia,
                                      RemocaoDeElementoInexistente);

        void  EscrevaSe () const throw ();
};

#endif
```

[Exemplo 23 – lista.cpp]

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

Lista::Lista () throw (): Inicio (NULL)
{}

Lista::Lista (const Lista& L) throw (MemoriaEsgotada): Inicio (NULL)
{
    *this = L;
}

Lista::~Lista () throw ()
{
```

```
    for (pNo P = Inicio; Inicio != NULL; P = Inicio)
    {
      Inicio = Inicio -> Prox;

      free (P);
    }
}

Lista& Lista::operator= (const Lista& L) throw (MemoriaEsgotada)
{
    pNo PT, PL;

    for (PT = Inicio; Inicio != NULL; PT = Inicio)
    {
      Inicio = Inicio -> Prox;

      free (PT);
    }

    for (PL = L.Inicio; PL != NULL; PL = PL -> Prox)
      if (Inicio == NULL)
      {
          if ((Inicio = new sNo) == NULL)
             throw MemoriaEsgotada ();

          Inicio -> Info = PL -> Info;
          Inicio -> Prox = NULL;
          PT             = Inicio;
      }
      else
      {
          if ((PT -> Prox = new sNo) == NULL)
             throw MemoriaEsgotada ();

          PT         = PT -> Prox;
          PT -> Info = PL -> Info;
          PT -> Prox = NULL;
      }

    return *this;
}

void Lista::InsInicio (int I) throw (MemoriaEsgotada)
{
    pNo N;

    if ((N = new sNo) == NULL)
       throw MemoriaEsgotada ();

    N -> Info = I;
    N -> Prox = Inicio;
    Inicio    = N;
}

int Lista::DelInicio () throw (RemocaoDeListaVazia)
{
    if (Inicio == NULL)
      throw RemocaoDeListaVazia ();

    int R  = Inicio -> Info;
    pNo P  = Inicio;
    Inicio = Inicio -> Prox;
```

```
    delete P;
    return R;
}

void Lista::InsFinal (int I) throw (MemoriaEsgotada)
{
    if (Inicio == NULL)
    {
      if ((Inicio = new sNo) == NULL)
           throw MemoriaEsgotada ();

      Inicio -> Info = I;
      Inicio -> Prox = NULL;
    }
    else
    {
      pNo A, P, N;

      for (A = NULL, P = Inicio; P != NULL; A = P, P = P -> Prox);

      if ((N = new sNo) == NULL)
          throw MemoriaEsgotada ();

      N -> Info = I;
      N -> Prox = NULL;
      A -> Prox = N;
    }
}

int Lista::DelFinal () throw (RemocaoDeListaVazia)
{
    if (Inicio == NULL)
      throw RemocaoDeListaVazia ();

    int R;

    if (Inicio -> Prox == NULL)
    {
      R = Inicio -> Info;

      delete Inicio;
      Inicio = NULL;
    }
    else
    {
      pNo A, P, D;

      for (A = Inicio, P = Inicio -> Prox, D = P -> Prox;
           D != NULL;
           A = P, P = D, D = D -> Prox);

      R = P -> Info;

      A -> Prox = NULL;
      delete P;
    }

    return R;
}

void Lista::InsEmOrdem (int I) throw (MemoriaEsgotada)
{
    if (Inicio == NULL)
```

```
    {
      if ((Inicio = new sNo) == NULL)
           throw MemoriaEsgotada ();

      Inicio -> Info = I;
      Inicio -> Prox = NULL;
    }
    else

      if (I < Inicio -> Info)
      {
          pNo P;

          if ((P = new sNo) == NULL)
            throw MemoriaEsgotada ();

          P -> Info = I;
          P -> Prox = Inicio;
          Inicio    = P;
      }
      else
      {
          pNo A, P;

          for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
          {
            if (P == NULL)    break;
            if (I < P -> Info) break;
          }

          pNo N;

          if ((N = new sNo) == NULL)
            throw MemoriaEsgotada ();

          N -> Info = I;
          N -> Prox = P;
          A -> Prox = N;
      }
}

int Lista::Pertence (int I) const throw ()
{
    for (pNo P = Inicio; P != NULL; P = P -> Prox)
      if (I == P -> Info) return 1;

    return 0;
}

void Lista::Del (int I) throw (RemocaoDeListaVazia,
                               RemocaoDeElementoInexistente)
{
    if (Inicio == NULL)
      throw RemocaoDeListaVazia ();

    pNo A, P;

    for (A = NULL, P = Inicio;; A = P, P = P -> Prox)
    {
      if (P         == NULL) break;
      if (P -> Info == I   ) break;
    }
```

```
    if (P == NULL)
      throw RemocaoDeElementoInexistente ();

    if (A == NULL)
      Inicio = P -> Prox;
    else
      A -> Prox = P -> Prox;

    free (P);
}

void Lista::EscrevaSe () const throw ()
{
    cout << '{';

    int Entrou = 0;

    for (pNo P = Inicio; P != NULL; P = P -> Prox, Entrou = 1)
      cout << P -> Info << ", ";

    if (Entrou)
      cout << "\b\b";

    cout << '}';
}
```

[Exemplo 23 – princip.cpp]

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

void main () throw ()
{
    Lista L1;
    int   N;

    cout << "Entre com 10 numeros separados por espacos:\n";

    for (int I = 1; I <= 10; I++)
    {
        cin >> N;
        try
        {
            L1.InsEmOrdem (N);
        }
        catch (MemoriaEsgotada E)
        {
            cerr << "\nErro: Nao ha memoria disponivel "
                 << "para executar esta operacao\n\n";
            exit (1);
        }
    }

    cout << "\nEm ordem, os numeros digitados sao:\n";

    cout << "L1 = "; L1.EscrevaSe ();
```

```
    cout << "\n\nEntre com um numero para ser excluido da lista
original:\n";
    cin  >> N;

    try
    {
        L1.Del (N);
    }
    catch (ExcessaoDeLista E)
    {
        cerr << "\nErro: O numero indicado nao consta da lista\n\n";
        exit (1);
    }

    cout << "\nEm ordem, apos a exclusao, a lista ficou:\n";

    cout << "L1 = "; L1.EscrevaSe ();

    cout << "\n\nEntre com dois numeros separados por um espaco\n"
         << "para que seja verificado se pertencem ou nao a lista:\n";

    cin >> N;

    if (L1.Pertence (N))
        cout << "O primeiro numero pertence a lista L1";
    else
        cout << "O primeiro numero nao pertence a lista L1";

    cin >> N;

    if (L1.Pertence (N))
        cout << "\nO segundo numero pertence a lista L1";
    else
        cout << "\nO segundo numero nao pertence a lista L1";

    cout << "\n\nEntre com dois numeros separados por espacos\n"
         << "(um para ser incluido no inicio "
         << "e outro no final da lista original):\n";

    cin >> N;

    try
    {
        L1.InsInicio (N);
    }
    catch (MemoriaEsgotada E)
    {
        cerr << "\nErro: Nao ha memoria disponivel "
             << "para executar esta operacao\n\n";
        exit (1);
    }

    cin >> N;

    try
    {
        L1.InsFinal (N);
    }
    catch (MemoriaEsgotada E)
    {
        cerr << "\nErro: Nao ha memoria disponivel "
             << "para executar esta operacao\n\n";
        exit (1);
```

```
    }

    cout << "\nApos as inclusoes, a lista ficou:\n";

    cout << "L1 = "; L1.EscrevaSe ();

    try
    {
        Lista L2 = L1;

        cout << "\n\nFoi feita uma copia de sua lista de numeros;";
        cout << "\nVerifique se a copia esta correta:\n";

        cout << "L2 = "; L2.EscrevaSe ();
    }
    catch (MemoriaEsgotada E)
    {
        cerr << "\nErro: Nao ha memoria disponivel "
             << "para executar esta operacao\n\n";
        exit (1);
    }

    Lista L3;

    try
    {
        L3 = L1;
    }
    catch (MemoriaEsgotada E)
    {
        cerr << "\nErro: Nao ha memoria disponivel "
             << "para executar esta operacao\n\n";
        exit (1);
    }

    cout << "\n\nFoi feita uma outra copia de sua lista de numeros;";
    cout << "\nVerifique se esta outra copia esta correta:\n";

    cout << "L3 = "; L3.EscrevaSe ();

    try
    {
        L1.DelInicio ();
    }
    catch (RemocaoDeListaVazia E)
    {
        cerr << "\nErro: Nao se pode remover elementos "
             << "de uma lista vazia!\n\n";
        exit (1);
    }

    try
    {
        L1.DelFinal  ();
    }
    catch (RemocaoDeListaVazia E)
    {
        cerr << "\nErro: Nao se pode remover elementos "
             << "de uma lista vazia!\n\n";
        exit (1);
    }

    cout << "\n\nApos a exclusao do primeiro e do ultimo, "
```

```
          << "a lista ficou:\n";

    cout << "L1 = "; L1.EscrevaSe ();

    cout << "\n\n";
}
```

Anexo I

Exercícios

I. Classes e Objetos

1. Indique (1) Estruturas; ou (2) Classes:

Agrupam variáveis de mesmo tipo;

Um item é chamado membro;

O padrão é que seus membros sejam públicos;

O padrão é que seus membros sejam privados;

Agrupam variáveis de tipos diferentes;

Agrupam variáveis e funções.

2. Responda verdadeiro ou falso: a definição de uma classe reserva espaço de memória para conter todos os seus membros.

3. A finalidade de definir classes é:

Reservar uma quantidade de memória;

Indicar que o programa é orientado a objetos;

Agrupar dados e funções protegendo-os do compilador;

Descrever o formato de novos tipos de dados antes desconhecidos do compilador.

4. A relação entre classes e objetos é a mesma existente entre:

Tipos básicos e variáveis desses tipos;

Variáveis e funções;

Uniões e estruturas;

Estruturas e funções.

5. Qual a diferença entre o uso de estruturas e o de classes?

6. Na definição de uma classe, os membros designados como privados:

São acessados por qualquer função do programa;

Requerem o conhecimento de uma senha;

São protegidos de pessoas não autorizadas;

São acessados por qualquer função-membro da classe;

São acessados pelos membros públicos da classe;

7. Responda verdadeiro ou falso: membros privados definem a atividade interna da classe e membros públicos, a *interface* da classe.

8. Para acessar um membro de um objeto, o operador ponto conecta:

O nome da classe e o nome do membro;

O nome do membro e o nome do objeto;

O nome do objeto e o nome do membro;

O nome da classe e o nome do objeto.

9. Uma função cujo código foi definido dentro de uma classe é sempre:

float;

inline;

Recursiva;

Sobrecarregada.

10. Métodos são:

Classes;

Dados-membro;

Funções-membro;

Chamadas a funções-membro.

11. Mensagens são:

Classes;

Dados-membro;

Funções-membro;

Chamadas a funções-membro.

12. Construtores são funções:

Que constróem classes;

Executadas automaticamente quando um objeto é criado;

Do tipo `int`;

Executadas automaticamente quando um objeto é destruído.

13. O nome de um construtor é sempre _____.

14. Responda verdadeiro ou falso: numa classe, é possível haver mais de um construtor de mesmo nome.

15. Responda verdadeiro ou falso: um construtor é do tipo retornado por meio do comando *return*.

16. Responda verdadeiro ou falso: um construtor não pode ter argumentos.

17. Destrutores são funções:

Que destroem classes;

Executadas automaticamente quando um objeto é declarado;

Do tipo `int`;

Executadas automaticamente quando um objeto é destruído.

18. O nome de um destrutor é sempre _____.

19. Responda verdadeiro ou falso: numa classe pode haver mais de um destrutor de mesmo nome.

20. Responda verdadeiro ou falso: um destrutor é do tipo retornado por meio do comando *return*.

21. Responda verdadeiro ou falso: um destrutor não pode receber argumentos.

22. Assuma a seguinte definição:

```
class Classe
{
     private:
          int A, B;
     public:
          static int S;
          Classe ();
          ~Classe ();
};
```

Como podemos atribuir `5` ao membro `S`?

S = 5;

Classe.S = 5;

Classe C; C.S = 5;

Classe::S = 5;

23. Defina uma classe de nome `Taluno` com dados privativos para armazenar o nome do aluno, a série e o grau. Inclua duas funções públicas: uma para solicitar os dados para o usuário e outra para imprimir os dados.

Escreva uma instrução que declare um objeto chamado `Aluno` da classe `Taluno`.

Escreva uma instrução para executar a função que solicita os dados de entrada para o usuário.

Escreva uma instrução para executar a função que imprime os dados digitados.

Inclua um membro estático privativo para contar o número de alunos cadastrados.

Inclua um construtor que incrementa o contador de alunos cadastrados.

Inclua um destrutor que decremente o contador de alunos cadastrados.

Inclua uma função pública que imprime o número de alunos cadastrados.

Escreva a instrução necessária para declarar uma matriz de `10` objetos da classe `Taluno` e escreva a instrução necessária para preencher o primeiro elemento da matriz acima com os dados digitados pelo usuário.

24. Assuma que `C1`, `C2` e `C3` sejam objetos de uma mesma classe. Quais das seguintes instruções são válidas?

C1 = C2;

C1 = C2 + C3;

C1 = C2 = C3;

C1 = C2 + 7;

25. Uma função-membro pode sempre acessar os dados:

Do objeto do qual é membro;

Da classe da qual é membro;

De qualquer objeto da classe da qual é membro;

da parte pública de sua classe.

26. Se cinco objetos da mesma classe forem declarados, quantas cópias dos itens de dados da classe serão armazenadas na memória? Quantas cópias de suas funções-membro?

27. Escreva uma classe para conter 3 membros inteiros chamados `Horas`, `Minutos` e `Segundos` e a chame de `Thorario`.

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar as `horas`, os `minutos` e os `segundos` para o usuário.

Crie uma função membro para imprimir o horário no formato HH:MM:SS.

Crie uma função membro para adicionar 2 objetos da classe `Thorario` passados como argumentos.

Crie uma função membro que subtraia dois horários e retorne o número de segundos entre elas. A função recebe dois objetos da classe `Thorario` passados como argumentos.

28. Escreva uma classe chamada `Testacionamento` para armazenar dados de um estacionamento. Ela deve ser capaz de armazenar o número da chapa do carro, a marca, o horário de entrada e o horário de saída do estacionamento. Utilize dois membros da classe `Thorario`, definida no exercício anterior para os horários de entrada e saída.

Crie uma função membro para solicitar os dados de um carro para o usuário (utilize as funções da classe tempo para pedir os horários de entrada e saída).

Crie uma função membro para imprimir os dados de um carro.

Admita que o estacionamento cobre R$ 2,00 a hora. Escreva uma função membro que imprima o valor cobrado. Utilize a função que subtrai dois horários da classe `Thorario`.

Escreva um programa que cria uma matriz de 5 objetos desta classe, solicita os dados dos carros para o usuário e imprime um relatório dos dados e do valor cobrado.

29. Escreva uma classe para descrever um mês do ano. A classe deve ser capaz de armazenar o nome do mês, a abreviação em 3 letras, o número de dias e o número do mês.

Crie um construtor para iniciar os dados com zero e outro para iniciar os dados com um valor dado.

Escreva uma função membro que recebe o número do mês como argumento e retorna o total de dias do ano até aquele mês.

Escreva uma função membro sobrecarregada que recebe como argumento o nome do mês em vez do número dele e retorna o mesmo total de dias.

30. Crie a classe `Trestaurante` para descrever restaurantes. Os membros devem armazenar o nome, o endereço, o preço médio e o tipo de comida.

Crie um construtor que inicie os dados com valores nulos e outro que inicie os dados com valores dados.

Crie uma função membro para solicitar os dados para o usuário.

Crie uma função membro para imprimir os dados de um restaurante.

Escreva um programa que cria uma matriz de objetos desta classe e solicite a entrada dos dados pelo usuário. Em seguida, o programa deve perguntar o tipo de comida ao usuário, e lista todos os restaurantes que o oferecem.

II. Sobrecarga de Operadores

1. Sobrecarga de operadores é:

Tornar operadores compatíveis com C++;

Criar novos operadores;

Criar novas operações para os operadores de C++;

Transformar operadores em objetos.

2. A implementação de sobrecargas de operadores é definida por meio de:

Programas pequenos;

Funções-membro de classes;

Dados-membro de classes;

Operações com classe.

3. A sobrecarga de operadores:

Obedece à precedência do operador original;

Define novas precedências para o operador original;

redefine o número de operandos aceitos pelo operador original;

Define um novo símbolo para o operador original.

4. Quantos argumentos são necessários para uma função que sobrecarrega um operador unário?

5. Se `C1`, `C2` e `C3` forem todos objetos de uma mesma classe que contém uma função-membro de nome `operator+`, para que a instrução `C1 = C2 + C3` trabalhe corretamente a referida função deve:

Receber dois argumentos;

Retornar um valor;

Criar um objeto temporário sem nome;

Usar o objeto da qual é membro como operando;

Receber um argumento.

6. Responda verdadeiro ou falso: é possível somar objetos de uma mesma classe mesmo sem utilizar o mecanismo de sobrecarga.

7. Assuma que `X`, `Y`, `Z`, `W`, `V` e `T` sejam objetos de uma mesma classe e que esta classe não contenha nenhuma sobrecarga de operadores. Como escreveríamos a instrução:

```
X = Produto (Produto (Soma (Y, Z), Soma (W, V)), T);
```

se incluirmos na classe funções que sobrecarregam os operadores aritméticos `+` e `*`?

8. Responda verdadeiro ou falso: utilizando sobrecarga de operadores, podemos criar a operação de exponenciação usando como símbolo os caracteres `**`.

9. Qual a diferença entre as funções operadoras do `operador++` quando prefixado e posfixado?

10. Quais das seguintes instruções cria um objeto temporário sem nome da classe A?

B = A (X, Y);

B = Temp.X;

return Temp;

A (X, Y) = B;

Func (A (X, Y));

11. A função que sobrecarrega o `operator--` pós-fixado:

Retorna um valor do tipo `int`;

Recebe um argumento do tipo `int`;

Não recebe argumentos;

Não retorna nada.

12. A função que sobrecarrega o `operator--` pré-fixado:

Retorna um valor do tipo `int`;

Recebe um argumento do tipo `int`;

Não recebe argumentos;

Não retorna nada.

13. A função que sobrecarrega um operador aritmético binário (`+`, `-`, `*` e `/`):

Não recebe argumentos;

Não tem valor de retorno;

O argumento é um objeto do qual a função é membro;

Nenhuma das anteriores.

14. Considere que a função de protótipo:

```
T operator+ (int N);
```

seja membro da classe T e que sejam criados os objetos X e Y desta classe. Quais das seguintes instruções são incorretas?

X = Y + 25;

X = 25 + Y;

X = X + Y;

X = Y + 25 + 5;

15. Quais das seguintes situações são válidas para o uso de conversões de tipos:

Atribuições;

Operações aritméticas;

Passagem de argumentos para uma funções;

Retornando um valor de uma função.

16. Para converter um objeto da classe T para um tipo básico devemos utilizar:

Uma função interna do compilador;

Um construtor de um argumento;

Sobrecarga do operador =;

Uma função conversora membro da classe T.

17. Para converter um tipo básico em um objeto da classe T, devemos utilizar:

Uma função interna do compilador;

Um construtor de um argumento;

Sobrecarga do operador =;

Uma função conversora membro da classe T.

18. Para converter um objeto da classe `T1` para um objeto da classe `T2`, devemos utilizar:

Uma função interna do compilador;

Um construtor de um argumento;

Sobrecarga do operador =;

Uma função conversora membro da classe `T1`.

19. A instrução

```
ObjA = ObjB;
```

onde `Obj1` é um objeto da classe `T1` e `Obj2` é um objeto da classe `B`, será executada corretamente se existir:

Uma função conversora membro da classe `T2`;

Uma função conversora membro da classe `T1`;

Um construtor membro da classe `T2`;

Nenhuma das anteriores.

20. Assuma que `Obj1` é um objeto da classe `T1` e `Obj2` é um objeto da classe `B`. Para utilizar instruções do tipo:

```
Obj1 = Obj2;
```

há vários meios. Podemos:

Incluir um conversor na classe `T1`. Qual seria o conversor?

Incluir um conversor na classe `T2`. Qual seria o conversor?

21. Assuma que `Obj1` é um objeto da classe `T1` e `Obj2` é um objeto da classe `B`. Para utilizar instruções do tipo:

```
Obj1 = Obj2;
```

E também instruções do tipo:

```
Obj2 = Obj1;
```

o que deve ser feito?

22. Escreva uma classe Tcomplexo capaz de armazenar os um número complexo (`a + bi`, onde `a` e `b` são do tipo `float` e `i` é a raiz quadrada de `-1`).

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com valores dados.

Crie uma função-membro para solicitar dados para o usuário.

Crie uma função-membro para imprimir os dados no formato `ai + b`.

Crie os operadores aritméticos menos unário (`-`), menos binário (`-`), multiplicação (`*`), divisão (`/`), resto da divisão inteira (`%`).

Crie os operadores de atribuição com operação embutida `+=`, `-=`, `*=`, `/=` e `%=`.

Crie os operadores relacionais `<`, `<=`, `>`, `>=`, `==` e `!=`.

23. Escreva uma classe `Tempresa` capaz de armazenar os dados de uma empresa (Nome, Endereço, Cidade, Estado, CEP e Fone).

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com valores dados.

Crie uma função-membro para solicitar dados para o usuário.

Crie uma função-membro para imprimir os dados no formato `ai + b`.

24. O objetivo desta questão é implementar uma classe que descreva figuras poligonais convexas sem impor limitações na quantidade de lados. Pede-se:

– Declare a classe Tpolygon. A classe deverá ser capaz de armazenar a quantidade de vértices que o polígono que ela representa deve ter, a quantidade de vértices do polígono que efetivamente foram informados, e as coordenadas de cada um desses vértices. Dica: aloque dinamicamente vetores para armazenar as coordenadas X e Y de tais vértices.

- Escreva um construtor que receba como argumento a quantidade de lados do polígono a ser criado, armazene internamente essa informação, e aloque dinamicamente os vetores para armazenar as coordenadas X e Y dos vértices do polígono.
- Escreva um destrutor para desalocar dinamicamente os vetores que serviram para armazenar as coordenadas X e Y dos vértices do polígono.
- Escreva uma função-membro AddVertex que recebe dois números float que representam respectivamente as coordenadas X e Y de um vértice do polígono e os armazena internamente.
- Escreva uma função-membro Print que imprime as informações do polígono.

25. O objetivo desta questão é implementar uma classe que descreva coordenadas cartesianas planares. Pede-se:

- Declare a classe Tcoord. A classe deverá ser capaz de armazenar os valores X e Y que a caracterizam.
- Escreva um construtor sem parâmetros que inicie com valores nulos as coordenadas X e Y do objeto, e outro que receba como argumento valores representando coordenadas X e Y e armazene internamente essa informação.
- Escreva uma funções-membro PolarR e PolarTeta que retornam respectivamente os valores R e Teta da representação polar daquela coordenada. Saiba que:
  - ✓ $R = \sqrt{X^2 + Y^2}$, e
  - ✓ $\Theta = \begin{cases} 90, \text{ se } X = 0 \text{ e } Y > 0; \\ 270, \text{ se } X = 0 \text{ e } Y < 0; \\ \text{arctg}\left(\frac{X}{Y}\right) + 180, \text{ se } X < 0; \text{ e} \\ \text{arctg}\left(\frac{X}{Y}\right), \text{ cc.} \end{cases}$
- Escreva uma função-membro TamSegto que recebe outro objeto da classe tcoord e retorna o tamanho do segmento de reta que formam. Saiba que:

- ✓ $T = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$

26. objetivo desta questão é implementar duas classes: (1) a classe `Tangle` que descreve ângulos; e (2) a classe `Tcircle` que descreve círculos. Pede-se:

   – Declare a classe Tangle. A classe deverá ser capaz de armazenar um valor angular expresso em graus.

   – Escreva um construtor sem parâmetros que inicie o objeto com o valor angular zero grau, e outro que receba como argumento um valor angular expresso em graus e o armazene internamente.

   – Escreva funções-membro Vgraus, Vgrados e Vradianos que retornem o valor angular do objeto expresso respectivamente em graus, grados e radianos.

   – Declare a classe Tcircle. A classe deverá se capaz de armazenar as coordenadas do centro do círculo, bem como o valor do seu raio.

   – Escreva um construtor sem parâmetros que inicie o objeto com valores nulos, e outro que receba como argumento as coordenadas e o valor do raio do círculo e os armazene internamente.

   – Escreva uma função membro que receba um objeto da classe Tangle e retorne a área do setor com aquela varredura angular.

27. O objetivo desta questão é implementar a classe Lista que descreva listas circulares duplamente ligadas de inteiros. Pede-se:

   - Declare a classe Lista. A classe deverá ser capaz de armazenar um ponteiro para o início de uma lista circular duplamente ligada de inteiros.

   - Escreva um construtor sem argumentos que inicie o referido ponteiro com NULL.

   - Escreva um construtor para iniciar um objeto da classe Lista com outro fornecido como argumento (lembre-se que não basta apontar ponteiro, tem que duplicar).

   - Escreva um destrutor para desalocar dinamicamente a lista apontada por este ponteiro.

- Escreva um operator= membro da classe Lista que receba como argumento um objeto da classe Lista e que atribua a seu objeto chamante (lembre-se que não basta apontar ponteiro, tem que duplicar).
- Escreva uma função membro Add para incluir ordenadamente um dado inteiro em seu objeto chamante.
- Escreva uma função membro Remove para remover todas as ocorrências de um dado inteiro de seu objeto chamante.
- Escreva uma função membro Inverte para inverter seu objeto chamante.
- Escreva uma função membro Concat para concatenar seu objeto chamante com outro da classe Lista fornecido como argumento. O resultado devera ser armazenado em um terceiro objeto desta mesma classe que será retornado.
- Escreva uma função membro Ordena para ordenar seu objeto chamante.
- Escreva uma função membro Merge para fazer a intercalação de seu objeto chamante com outro da classe Lista fornecido como argumento. O resultado devera ser armazenado em um terceiro objeto desta mesma classe que será retornado.
- Escreva um operator== membro (devolve int, lembre-se) para comparar seu objeto chamante com outro objeto da classe Lista fornecido como argumento.

III. Herança

1. Herança é um processo que permite:

A inclusão de um objeto dentro de outro;

Transformar classes genéricas em classes mais específicas;

Adicionar propriedades a uma classe existente sem rescrevê-la;

Relacionar objetos por meio de seus argumentos.

2. As vantagens do uso de herança incluem:

Aumento de funcionalidade de códigos existentes;

Distribuição e uso de bibliotecas;

Concepção de programas dentro de uma hierarquia que mostra quais conceitos compartilham características comuns;

Não rescrita de código.

3. Para que membros de uma classe-base possam ser acessados por membros de uma classe derivada, eles devem ser:

public;

protected;

private;

Todas as anteriores.

4. Qual a diferença entre a derivação pública e a derivação privativa?

5. Assuma que a classe `D` é derivada publicamente da classe-base `B`. Um objeto da classe `D` pode acessar:

Membros públicos da classe `D`;

Membros protegidos da classe `D`;

Membros privativos da classe `D`;

Membros públicos da classe `B`;

Membros protegidos da classe `B`;

Membros privativos da classe `B`.

6. Assuma que a classe `D` é derivada privativamente da classe-base `B`. Um objeto da classe `D` pode acessar:

Membros públicos da classe `D`;

Membros protegidos da classe `D`;

Membros privativos da classe `D`;

Membros públicos da classe `B`;

Membros protegidos da classe `B`;

Membros privativos da classe `B`.

7. Os membros de uma classe-base podem acessar:

Membros públicos da classe derivada;

Membros protegidos da classe derivada;

Membros privativos da classe derivada;

Nenhuma das anteriores.

8. Os membros de uma classe derivada podem acessar:

Membros públicos da classe-base;

Membros protegidos da classe-base;

Membros privativos da classe-base;

Nenhuma das anteriores.

9. Escreva a primeira linha da definição da classe `Thomem` derivada publicamente da classe `Tpessoa`.

10. Para derivar uma classe de outra já existente, deve-se:

Alterar a classe existente;

Usar o código-objeto da classe existente;

Rescrever a classe existente;

Nenhuma das anteriores.

11. Se uma classe-base contém uma função chamada `F ()` e a classe derivada não possui nenhuma função com este nome, responda:

Em que situação um objeto da classe derivada pode acessar a função `F ()`?

Em que situação um objeto da classe derivada não pode acessar a função `F ()`?

12. Se uma classe-base contém uma função chamada `F ()` e a classe derivada também possui uma função com este nome, um objeto da classe derivada:

Pode acessar a função-membro `F ()` da classe-base;

Não pode acessar a função-membro `F ()` da classe-base;

Pode acessar a função-membro `F ()` da classe derivada;

Não pode acessar a função-membro `F ()` da classe derivada.

13. Qual das seguintes instruções executa a função-membro `F ()` da classe-base `B`, a partir de uma função-membro da classe derivada `D`?

F ();

B::F ();

D::F ();

B:F ();

14. Qual das seguintes instruções executa a função-membro `F ()` da classe derivada `D`, a partir de uma função-membro desta mesma classe derivada `D`?

F ();

B::F ();

D::F ();

B:F ();

15. Um objeto de uma classe derivada contém:

Todos os membros da classe-base;

Somente os membros públicos da classe-base;

Somente os membros protegidos da classe-base;

O segundo e o terceiro item são verdadeiros.

16. Escreva a primeira linha de um construtor sem argumentos da classe `D` derivada da classe `B`.

17. Responda verdadeiro ou falso: se nenhum construtor existir na classe derivada, objetos desta classe usarão o construtor sem argumentos da classe-base.

18. Responda verdadeiro ou falso: se nenhum construtor existir na classe derivada, objetos desta classe poderão iniciar os dados herdados de sua classe-base usando o construtor com argumentos da classe-base.

19. Uma classe é dita abstrata quando:

Nenhum objeto dela é declarado;

É representada apenas mentalmente;

Só pode ser usada como base para outras classes;

É definida de forma obscura.

20. A conversão de tipos implícita é usada para:

Converter objetos da classe derivada em objetos da classe-base;

Converter objetos da classe-base em objetos da classe derivada;

Converter objetos da classe derivada em objetos da classe-base e vice-versa;

Não pode ser usada para conversão de objetos.

21. Responda verdadeiro ou falso: uma classe derivada não pode servir de base para outra classe.

22. O que é herança múltipla?

23. Escreva a primeira linha da definição da classe `D` derivada das classes-base `B1` e `B2`.

24. Responda verdadeiro ou falso: se a classe `D` é derivada da classe-base `B`, então a classe `C` não pode ser derivada de `B` e `D`.

25. Responda verdadeiro ou falso: um objeto de uma classe pode ser membro de outra classe.

26. Escreva uma classe `Tempresa` capaz de armazenar os dados de uma empresa (Nome, Endereço, Cidade, Estado, CEP e Telefone).

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar os dados ao usuário.

Crie uma função membro para imprimir os dados.

27. Use a classe Tempresa como base para criar a classe Trestaurante. Inclua o tipo de comida, o preço médio de um prato.

Inclua um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Inclua uma função membro para solicitar os dados ao usuário.

Inclua uma função membro para imprimir os dados.

28. Escreva uma classe `Tmotor` capaz de armazenar dados de um motor (número de cilindros e potência).

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar os dados ao usuário.

Crie uma função membro para imprimir os dados.

29. Escreva a classe Tveiculo capaz de armazenar dados de um veículo (peso em quilos, velocidade máxima em km/h e preço em US$).

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar os dados ao usuário.

Crie uma função membro para imprimir os dados.

30. Crie a classe TcarroDePasseio derivada das classes Tmotor e Tveículo. Inclua os dados Cor e Modelo.

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar os dados ao usuário.

Crie uma função membro para imprimir os dados.

31. Crie a classe Tcaminhao derivada das classes Tmotor e Tveiculo. Inclua os dados Carga Máxima em Toneladas, Altura Máxima e Comprimento.

Crie um construtor que inicie os dados com zero e outro construtor que inicie os dados com um valor dado.

Crie uma função membro para solicitar os dados ao usuário.

Crie uma função membro para imprimir os dados.

32. Crie um programa para testar as classes dos exercícios 28, 29, 30 e 31.

33. O objetivo desta questão é implementar a classe Conjunto, derivada privativamente da classe Lista que descreva conjuntos de números inteiros. Pede-se:
    - Declare a classe Conjunto. A classe não precisará de dados já que herdará tudo de que precisará.
    - Escreva um construtor sem argumentos para iniciar um objeto da classe Conjunto com o conjunto vazio.
    - Escreva um construtor para iniciar um objeto da classe Conjunto com outro fornecido como argumento.
    - Escreva um operator= membro da classe Conjunto que receba como argumento um objeto da classe Conjunto e que atribua a seu objeto chamante (lembre-se que não basta apontar ponteiro, tem que duplicar).

- Escreva uma operator+ membro para incluir um dado inteiro em seu objeto chamante. O operator deverá produzir e retornar um novo objeto da classe Conjunto.
- Escreva uma operator- membro para remover um dado inteiro em seu objeto chamante. O operator deverá produzir e retornar um novo objeto da classe Conjunto.
- Escreva uma operator* membro para fazer a intercessão de seu objeto chamante com outro objeto da classe Conjunto fornecido como argumento. O operator deverá produzir e retornar um novo objeto da classe Conjunto.
- Escreva uma operator+ membro para fazer a união de seu objeto chamante com outro objeto da classe Conjunto fornecido como argumento. O operator deverá produzir e retornar um novo objeto da classe Conjunto.
- Escreva uma operator<< membro para verificar se um dado inteiro ocorre em seu objeto chamante. O operator deverá produzir e retornar um valor booleano (lembre-se, um int).

IV. Ponteiros

1. Um ponteiro é:

O endereço de uma variável;

Uma variável que armazena endereços;

O valor de uma variável;

Um indicador da próxima variável a ser acessada.

2. Escreva uma instrução que imprima o endereço da variável V.

3. Indique com (1) o operador de referenciação e com (2) o operador de derreferenciação.

P = &I;

int &I = J;

cout << &I;

int* P = &I;

int& F (void);

F (&I);

4. A instrução `int* P`:

Cria um ponteiro com valor indefinido;

Cria um ponteiro do tipo `int`;

Cria um ponteiro com valor zero;

Cria um ponteiro que aponta para uma variável `int`.

5. O que significa o operador `*` em cada um dos seguintes casos:

int* P;

cout << *P;

*P = X * 5;

cout << *(P + 1);

6. Quais das seguintes instruções declaram um ponteiro para uma variável `float`?

float *P;

*float P;

float* P;

float* P = &F;

*P;

float& P = Q;

7. Na expressão `int* P`, o que é do tipo `int`?

A variável `P`;

O endereço de P;

A variável apontada por P;

O endereço da variável apontada por P.

8. Se o endereço de V foi atribuído a um ponteiro P, quais das seguintes expressões são verdadeiras?

V == &P;

V == *P;

P == *V;

P == &V;

9. Assuma as declarações abaixo e indique qual é o valor das seguintes expressões:

```
int I = 3, J = 5;
int *P = &I, *Q = &J;
```

P == &I;

*P - *Q;

**&P;

3 * -*P / *Q + 7;

10. Qual é a saída deste programa?

```
#include <iostream.h>

void main ()
{
     int I = 5, *P;
     P = &I;
     cout << P
          << '\t'
          << (*P + 2)
          << '\t'
          << **&P
          << '\t'
          << (3 * *P)
          << '\t'
          << (**&P + 4);
```

```
}
```

11. Se `I` e `J` são variáveis inteiras e `P` e `Q` são ponteiros para inteiros, quais das seguintes expressões de atribuição são incorretas?

P = &I;

*Q = &J;

P = & * &I;

I = (*&)J;

Q = &P;

I = (*P)++ + *Q;

if (P == I) I++;

12. Explique cada uma das seguintes declarações e identifique quais são incorretas.

int * const X = &Y;

const int &X = *Y;

int &const X = *Y;

int const *X = &Y;

13. O seguinte programa é correto?

```
#include <iostream.h>

const int C = 13;

void main ()
{
     int *P = C;
     cout << *P;
}
```

14. O seguinte programa é correto?

```
#include <iostream.h>

const int C = 13;
```

```
void main ()
{
     int  I = C;
     int *P;
     cout << *P;
}
```

15. Qual a diferença entre `V [3]` e `*(V + 3)`?

16. Admitindo a declaração `int V [7]`, por que a instrução `V++` é incorreta?

17. Admitindo a declaração `int V [7]`, quais das seguintes expressões referenciam o valor do terceiro elemento do vetor?

*(V + 2);

*(V + 3);

V + 2;

V + 3;

18. O que faz o programa seguinte:

```
#include <iostream.h>

void main ()
{
     int V [] = {3, 7, 13};
     for (int J = 0; J < 3; J++)
          cout << '\n' << *(V + J);
}
```

19. O que faz o programa seguinte:

```
#include <iostream.h>

void main ()
{
     int V [] = {3, 7, 13};
     for (int J = 0; J < 3; J++)
          cout << '\n' << (V + J);
}
```

20. O que faz o programa seguinte:

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

```
void main ()
{
     int  V [] = {3, 7, 13};
     int* P    = V;

     for (int J = 0; J < 3; J++)
          cout << '\n' << *P++;
}
```

21. Qual é a diferença entre as duas instruções seguintes?

```
char  S [] = "POO";
char *S    = "POO";
```

22. Assumindo a declaração:

```
char *S = "Programacao Orientada a Objetos";
```

O que imprimirão as instruções seguintes:

cout << S;

cout << &S[0];

cout << (S + 11);

cout << S [0];

23. Escreva a expressão M [I][J] em notação de ponteiro.

24. Qual é a diferença entre os seguintes protótipos de funções:

```
void F (char  S []);
void F (char *S   );
```

25. Assumindo a declaração:

```
char *Dsemana [7] = {
                        "Domingo",
                        "Segunda",
                        "Terca",
                        "Quarta",
                        "Quinta",
                        "Sexta",
                        "Sabado"
                    };
```

Para poder escrever a instrução `P = Dsemana`, a variável `P` deve ser declarada como:

char P;

char *P;

char **P;

char ***P;

26. O operador `new`:

Cria uma variável de nome `new`;

Retorna um ponteiro `void`;

Aloca memória para uma nova variável;

Informa a quantidade de memória livre.

27. O operador `delete`:

Apaga um programa;

Devolve memória ao sistema operacional;

Diminui o tamanho do programa;

Cria métodos de otimização.

28. Explique o significado da palavra `void` em cada uma das seguintes instruções:

void* P;

void P ();

void P (void);

void (*P) ();

29. Qual é o erro deste trecho de programa?

```
int   X = 13;
void *P = &X;
```

```
cout << *P;
```

30. O que é o ponteiro `this` e quando é usado?

31. Se `P` é um ponteiro para um objeto da classe `C`, então quais das seguintes instruções executam a função-membro `Impressao ()`?

P.Impressao ();

*P.Impressao ();

P -> Impressao ();

*P -> Impressao ();

32. Se `V` é um vetor de ponteiros para objetos da classe `C`, escreva uma instrução que execute a função-membro `Impressao ()` do objeto apontado pelo terceiro elemento do vetor `V`.

33. Numa lista ligada:

Cada item contém um ponteiro para o próximo item;

Cada item contém dados ou ponteiros para os dados;

Cada item contém um vetor de ponteiros;

Os itens são armazenados num vetor.

34. O que declara cada uma destas instruções?

int (*P) [10];

int *P [10];

int (*P) ();

int *P ();

int (*P [10]) ();

V. Funções Virtuais e Amigas

1. Quais das seguintes instruções são válidas, assumindo as declarações:

```
class B { ... };
class D: public B { ... };
B b, *Pb;
D d, *Pd;
```

   - `Pd = &b;`
   - `Pb = &Pd;`
   - `b = d;`
   - `d = bl`

2. Quais das seguintes características podem ser implementadas por meio de uma função virtual?

   - Permitir que a classe-base estabeleça um protocolo com as suas classes derivadas, de modo que estas últimas obtenham máxima funcionalidade;
   - Funções sem corpo;
   - Assegurar a chamada correta a funções-membro de objetos de diferentes classes, usando uma mesma instrução de chamada à função;
   - Agrupar objetos de diferentes classes de tal forma que possam ser acessados pelo mesmo código de programa;
   - Criar uma matriz de ponteiros para a class-base que pode armazenar ponteiros para classes derivadas;
   - Redefinir funções-membro em classes derivadas.

3. Quais da seguintes afirmações estão corretas quando é executada a chamada a uma função-membro usando um ponteiro para um objeto?

   - Se a função é virtual, a instrução é resolvida levando em conta o tipo do objeto contido no ponteiro;

- Se a função não é virtual, a instrução é resolvida levando em conta o tipo do ponteiro;
- Se a função é virtual, a instrução é resolvida após o início da execução do programa;
- Se a função não é virtual, a instrução é resolvida na compilação do programa.

4. Explique a diferença entre a sobrecarga de funções-membro virtuais e não virtuais.

5. Considerando as seguintes declarações:

```
class B
{
     public:
          void F () { ... }
};

class D: public B
{
     public:
          void F () { ... }
};

D   d;
B *Pb;
```

- A instrução `P -> F ()` executará a versão de `F ()` membro da classe _____.
- Se `F ()` for declarada virtual, então a instrução `P -> F ()` executará a versão de `F ()` membro da classe _____.

6. Escreva a declaração da função virtual `F` de tipo `void` e que recebe um argumento do tipo `int`.

7. Resolução Dinâmica é o processo de:

- Associar chamadas a funções a endereços fixos;
- Associar uma instrução a uma função no momento de sua execução;
- Criação de funções virtuais;

- Criar a tabela “V-Table”.

8. Uma função virtual pura é uma função que:
   - Nao retorna nada;
   - É parte da classe derivada;
   - Não recebe argumentos;
   - Não tem corpo.
9. Escreva a declaração da função virtual pura chamada F () que não retorna nada nem recebe nada como argumento.
10. As classes abstratas:
    - Existem somente para derivação;
    - Contém funções-membro virtuais;
    - Não têm corpo de código;
    - Contém funções virtuais puras.
11. Quais dos seguintes processos são permitidos com classes abstratas?
    - Declarar objetos;
    - Retornar um objeto de uma função;
    - Enviar um objeto como argumento para uma função;
    - Declarar ponteiros.
12. A classe-base virtual é usada quando:
    - Diferentes funções nas classes-base e derivada têm o mesmo nome;
    - Uma classe-base aparece mais de uma vez no processo de herança múltipla;
    - Há múltiplos caminhos de uma classe derivada para outra;
    - A identificação da função da classe-base é ambígua.

13. Uma classe-base é virtual quando:
    - A palavra `virtual` é colocada na sua declaração;
    - Contém uma função-membro virtual;
    - É especificada `virtual` na declaração da classe derivada;
    - Contém uma função virtual pura.
14. Escreva as classes `Tanimal`, `Tvaca`, `Tbufalo` e `Tbezerro`, sendo as classes `Tvaca` e `Tbufalo` derivadas de `Tanimal`, e `Tbezerro` derivada de `Tvaca` e `Tbufalo`.
15. Responda verdadeiro ou falso: toda função-membro pode ser declarada virtual mesmo sendo um construtor ou um destrutor.
16. Responda verdadeiro ou falso: uma função amiga pode acessar dados privativos da classe sem ser membro da classe.
17. Uma função amiga pode ser usada para:
    - Impedir heranças entre classes;
    - Permitir o acesso a classes das quais não temos acesso ao código-fonte;
    - Permitir a uma classe o acesso a classes não documentadas;
    - Aumentar a versatilidade de um operador sobrecarregado.
18. Escreva a declaração da função-amiga chamada `F` de tipo `void` e que recebe um argumento da classe `C`.
19. A palavra-chave `friend` é colocada:
    - Na classe que permite o acesso de outra classe;
    - Na classe que deseja acessar outra classe;
    - Na parte privativa da classe;
    - Na parte pública da classe.

20. Escreva uma declaração que, na classe onde ela aparece, torna todos os membros da classe `C` funções-amigas.

21. Considerando os exercícios anteriores que envolviam as classe `Tmotor`, `Tveiculo`, `TcarroDePasseio` e `Tcaminhao`, faça um programa que cria um vetor de ponteiros para `Tveiculo`. Inclua um laço que pergunta ao usuário sobre o tipo de veículo e use o operador `new` para criar objetos do tipo escolhido (`TcarroDePasseio` ou `Tcaminhao`). Quando o usuário terminar a entrada dos dados de todos os veículos, imprima os resultados usando outro laço.

VI. Streams

1. O que é um `stream`?

   Arquivo;

   Classe;

   Um fluxo de dados de um lugar para outro;

   Um lugar de origem ou destino de dados.

2. A classe-base para todas as classes `stream` é:

   fstream;

   ios;

   iostream;

   ifstream.

3. Indique (1) para disco; (2) para entrada e saída padrão; (3) para memória; e (4) para nenhuma das anteriores:

   istream_withassign e ostream_withassign;

   istrstream, ostrstream e strstream;

   ios, istream e ostream;

ifstream, ofstream e fstream.

4. Os operadores >> e << podem ser usados para ler ou gravar dados em arquivos em disco pois:

   Trabalham com todas as classes;

   São sobrecarregados nas classes istream e ostream;

Anexo II

Referências

Dershem, H.L.; and Jipping, M.J., "Programming Languages: Structures and Models", Wadsworth, Inc., 1990.

Takahashi, T., "Introdução à Programação Orientada a Objetos", III EBAI - Escola Brasileiro-Argentina de Informática, 1988.

Smith, D.C., "Pygmalion: An Executable Eletronic Blackboard" *in* Watch What I Do: Programming by Demostration. Cypher, A.; Halbert, D.C. ... [et al.] eds., The MIT Press, Cambridge, Massachussets; London, England, 1993.