AEL

Advanced Eletromagnetics Laboratory

Introdução a Orientação a Objetos utilizando a Linguagem C++

Mini Curso: Introdução a Orientação a Objetos utilizando a Linguagem C++
duração: 30 horas
Material editado por Tecgº. Carlos Henrique da Silva Santos e
distribuído pelo Ramo Estudantil do IEEE na Unicamp

Colaboração: Advanced Eletromagnetics Laboratory (Unicamp), coordenado pelo Prof. Dr. Hugo Enrique Hernandéz Figueroa

Ramo Estudantil do IEEE na Unicamp
sb.unicamp@ieee.org
Gestão 2004

Alexandre Castro dos Santos
Presidente

Comissão Organizadora do Curso

Tecgº Carlos Henrique da Silva Santos
Professor

Fernanda Spada Villar
Diretora de Eventos

Júlio Cesar de Assis Novais
Diretor de Treinamento

Rafael Palma Ribeiro
Membro Voluntário

M.Sc. Ricardo Cotrin Teixeira
Diretor do Capítulo EDS

Índice

Introdução 6
Capítulo 1 8
Orientação a Objetos - OO 8
1.1. Estrutura de um programa em C/C++ 10
1.2. Conjunto de caracteres 10
Comentários 11
Diretivas de Compilação 11
Declaração de variáveis 12
Entrada e saída de dados 12
Estruturas de Controle 12
até condição 13
Capítulo 2 14
Constantes, variáveis e operadores 14
2.1 Constantes 14
2.1.1 Constantes inteiras 14
2.1.1 Constantes de ponto flutuante 15
2.1.1 Constantes caracteres 16
2.1.1 Constantes *strings* 17
2.2 Identificadores 17
2.2.1 Regras de sintaxe 17
2.2.1 Palavras reservadas 18
2.3 Tipos de dados 18
2.3.1 Tipos básicos 19
2.3.2 Declaração de variáveis 19
2.3.3 Tipos modificados 20
2.4 Constantes Simbólicas 22
2.4.1 Constantes definidas pelo programador 23
2.4.2 Constantes pré-definidas 24
2.5 Operadores 25
2.5.1 Operador de Atribuição 25
2.5.1 Atribuição múltipla. 26
2.6 Operadores Aritméticos 27
2.7 Operadores de Atribuição Aritmética 27
2.8 Operadores Incrementais 29

2.9 Operadores Relacionais e Lógicos 30
2.10 Operador Condicional 30
Capítulo 3 32
Comandos de Controle 32
3.1 Comandos de Seleção 32
3.1.1 IF 32
3.1.2 ELSE 33
3.1.3 IF-ELSE-IF 34
3.1.4 IFs aninhados 35
3.1.5 Operador ? 35
3.1.6 SWTICH 36
3.2 Comandos de Iteração 38
3.2.1 FOR 38
3.2.2 WHILE 39
Capítulo 4 41
Funções 41
4.1 Chamada por Valor e Chamada por Referência 44
4.2 Argumentos argc e argv para a main() 45
4.3 Recursão 45
Capítulo 5 47
Ponteiros 47
5.1 Manipulando memória em tempo real 48
Capítulo 6 49
Classes 49
6.1 Definindo Classes 49
6.2 Ponteiro *this* 51
Capítulo 7 52
Sobrecarga de Operadores 52
7.1 Função *Friend* 54
7.2 Sobrecarga de Operadores Unários 55
7.3 Sobrecarga de Operadores Binários 56
Capítulo 8 57
Herança 57
Herança Múltipla 59
Capítulo 9 61
Polimorfismo e Funções Virtuais 61
9.1 Polimorfismo 61
9.2 Funções Virtuais 63

Capítulo 10 69
Manuseamento de Arquivos 69
10.1 Abrindo Arquivos 69
10.2 Manuseamento de dados de um arquivo 71

Introdução

A linguagem C++ foi desenvolvida inicialmente por Bjarne Stroustrup na AT&T, de 1979 a 1983, a partir da linguagem C, tendo como idéia principal o agregamento do conceito de classes, de orientação a objetos, a linguagem. Razão pela qual inicialmente chamava-se de “C com classes”. Stroustrup procurou possibilitar que os programas desenvolvidos em C pudessem ser compilados por um compilador C++ com um mínimo de alterações.
Entretanto, é errado definir C++ como ser um super conjunto de C++, pois ela realmente é uma outra linguagem, podendo arguir-se de diversas formas [JOHANN 2004], sendo:

- Nem todo o programa escrito em C é compilado em C++ sem erros, e pode nem mesmo gerar o mesmo resultado, já que a sintaxe e a semântica de algumas construções diferem.
- Ao ligar-se partes de um programa escrito em C++ com partes em C, estas devem ser bem especificadas, pois as convenções de funcionamento do código compilado também diferem.
- C++ oferece um conjunto de mecanismos básicos que não estão presentes em C, e estes devem ser usados para produzir softwares mais modulares e confiáveis explorando-se as verificações disponíveis no compilador.
- Finalmente, os mecanismos de C++ devem inspirar a programação segundo o paradigma de orientação a objetos e, portanto, não se deve programar em C++ como se faz em C, de forma estruturada.

A partir da primeira versão de 1983, a linguagem foi sendo revisada e evoluindo, tornou-se disponível fora da AT&T em 1985, e após um longo processo foi padronizado pela ISO no final de 1997, pelo padrão ISO/IEC 14882. Você pode obter mais informações sobre o desenvolvimento da linguagem na página do próprio autor em [STR 2004].
STL é uma parte do padrão C++, e consiste em uma biblioteca de funções e estruturas de dados que todo compilador C++ deve oferecer, provê as implementações mais comuns em um programa, e pode-se utilizá-la com diferentes tipos de dados.

Para compilar os códigos desenvolvidos em C++, não necessariamente é preciso pagar, pois existem compiladores gratuitos disponibilizados pela GNU na Internet. Como exemplo os compiladores gcc e g++ para Windows, que podem ser acessados pelo ambiente de programação BloodShed Dev-C++ [BSD-C++].

Percebendo a necessidade de grupos de pesquisas e estudantes em geral, foi proposto esse curso com o intuito de apresentar conceitos sobre orientação a objetos através da linguagem de programação, para contribuir na construção do conhecimento e de alguma forma colaborar profissionalmente. Essa apostila é resultado de pesquisas em livros e, especialmente de materiais da Internet.

Capítulo 1

Orientação a Objetos - OO

A programação orientada a objeto é um paradigma de programação avançado. Paradigma é definido como um "conjunto de teorias, padrões e métodos que, juntos, representam um modo de organizar o conhecimento". Desenvolvedores tem utilizado outros paradigmas tais como a programação imperativa (C, Pascal, Cobol, Ada), programação lógica (Prolog, C5) e programação funcional (FP, ML).
Atualmente, programamos utilizando C++, Java, Smaltalk Objective C seguindo os paradigmas de orientação a objetos para modelar a realidade [LEE & TEPFENHART 2002]. A seguir são apresentados diversos conceitos utilizados na Orientação a Objetos, que refletem na análise e implementação de programas.

- Classe: é um tipo de abstração de dados (TAD), que possui uma *interface* para ocultar detalhes de implementação, possibilitando que tenham especificações, em diferentes tempos, sem afetar as aplicações que utilizam o TAD. As classes podem ser vistas como uma "fábrica de objetos idênticos" no diz respeito à sua *interface* e implementação.
- Objeto: O conceito/mecanismo fundamental da programação orientada objeto é o próprio *objeto*. O objeto pode ser definido como uma unidade de software constituído de atributos (dados) e de métodos (códigos de funções) que atuam sobre os dados.
- Abstração: é a seleção que um desenvolvedor faz de alguns aspectos, suprimindo outros.
- Encapsulamento: O objeto contém tanto os dados como os métodos (códigos) que manipularão ou transformarão esses dados, tendo por objetivo restringir o escopo ou visibilidade da informação para obter melhor legibilidade, manutenibilidade e

principalmente reutilização no desenvolvimento de um novo sistema **[UFSCAR-OOP]**.

- Atributo: qualquer propriedade, qualidade ou característica que possa ser atribuída a uma pessoa ou objeto pode ser acessada apenas através de um serviço.
- Serviços: atividade executada para permitir o acesso a alguns recursos, em OO é um comportamento específico que um objeto deve exibir.
- Passagem de mensagem: Possibilita a comunicação entre objetos.
- Herança: mecanismo para expressar a similaridade entre classes, simplificando a definição de classes iguais a outras que já foram definidas. Este princípio permite representar membros comuns, serviços e atributos uma só vez, assim como especializar estes membros em casos específicos. Desta forma, sendo vantajoso por possibilitar o reuso de código.
- Subclasse: herda atributos e métodos de sua superclasse como, por exemplo, Homem herdará todos os atributos e métodos de Mamíferos.
- Superclasse: Classe base para as subclasses.
- Superclasse Abstrata: Classe utilizada somente para criar subclasses, como no caso dos mamíferos.
- Escala: princípio que permite ao desenvolvedor considerar algo muito grande através do enfoque *Todo-parte.*
- Todo-parte: também conhecido como agregação, que é um mecanismo que permite a construção de uma classe agregada a partir de outras classes componentes. Costuma-se dizer que um objeto da classe agregada (Todo) tem objetos das classes componentes (Parte).
- Associação: vem do relacionamento de entidades do mundo real, e é usada para agrupar certos objetos que ocorrem algum ponto no tempo ou sob circunstâncias similares. Portanto, associação é uma união ou conexão de idéias.
- Polimorfismo: princípio relacionado com as diferentes formas de um objeto. Operacionalmente, o polimorfismo pode ser visto conforme mostra a Figura 2, onde se pode instanciar um objeto Janela de várias formas.

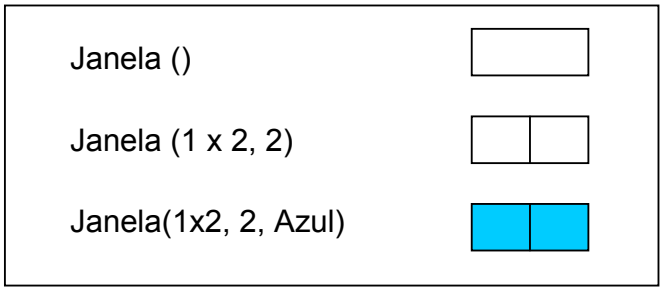


Figura 2 – Esquema de Polimorfismo.

1.1. Estrutura de um programa em C/C++

Um programa em *C/C++* é constituído de:

- um cabeçalho contendo as diretivas de compilador onde se definem os valores de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, declaração de rotinas, etc.
- um bloco de instruções principal e outros blocos de rotinas.
- documentação do programa: comentários.

Programa Exemplo: Um interessante exemplo a ser mostrado é o “Olá Mundo”, que tem a única finalidade de imprimir na tela a mensagem “Olá Mundo!”.

```
#include <iostream.h>

class hello {
        public:
                void primeiro()
                {
                        cout << "Olá Mundo!!";
                }
};

int main()
{
        hello objeto;
        objeto.primeiro();
        return 0;
}
```

1.2. Conjunto de caracteres

Um programa fonte em C/C++ é um texto não formatado escrito em um editor de textos usando um o conjunto padrão de caracteres ASCII. A seguir estão os caracteres utilizados em C/C++:

Caracteres válidos:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

1 2 3 4 5 6 7 8 9 0

+ - * / \ = | & ! ? # % ( ) { } [ ] _ ‘ “ . , : < >

Caracteres não válidos:

@ $ ¨ á é õ ç

Os caracteres acima são válidos apenas em *strings*. Veja seção 2.1.4.

Comentários

Em C/C++, os comentários podem ser escritos em qualquer lugar do texto para facilitar a interpretação do algoritmo e contribuindo para a documentação do código. Para que o comentário seja identificado como tal, ele deve ter um /* antes e um */ depois. Observe que no exemplo olamundo.cpp todo o cabeçalho está dentro de um comentário.

Exemplo:

/* esta é uma linha de comentário em C /C++*/

Observação: O C/C++ permite que comentários sejam escritos de outra forma: colocando um // em uma linha, o compilador entenderá que tudo que estiver à direita do símbolo é um comentário.Verifique essa afirmação no exemplo olamundo.cpp

Diretivas de Compilação

Em C/C++, existem comandos que são processados durante a compilação do programa. Estes comandos são genericamente chamados de *diretivas de compilação*. Estes comandos informam ao compilador do C/C++ basicamente quais são as constantes simbólicas usadas no programa e quais bibliotecas devem ser anexadas ao programa executável. A diretiva `#include` diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente, estes arquivos contêm bibliotecas de funções ou rotinas do usuário. No capítulo 5

novamente será comentado o assunto. A diretiva `#define` diz ao compilador quais são as constantes simbólicas usadas no programa. Esta diretiva será melhor especificada no capítulo 2.

Declaração de variáveis

Em C/C++, como na maioria das linguagens, as variáveis devem ser declaradas no início do programa, por questão de organização. Estas variáveis podem ser de vários tipos: int (inteiro), float (real de precisão simples) e outras que apresentadas no capítulo 2.

Entrada e saída de dados

Em C/C++ existem várias maneiras de fazer a leitura e escrita de informações. Estas operações são chamadas de operações de entrada e saída (I/O). No capitulo 3 são apresentadas algumas funções de entrada e saída de informações via teclado e tela. Outras funções de leitura e escrita em arquivos, saída gráfica, funções de manipulação de *mouse,* entrada e saída de informações via portas serial e paralela serão vistas em capítulos posteriores. Como exemplo de comando de saída de dados pode-se ter o `cout`, que é específico de C++ e serve para imprimir na tela as mensagens desejadas, seguindo das variáveis que possam vir a acompanhar (ver exemplo olamundo.cpp).

Estruturas de Controle

A linguagem C/C++ permite uma ampla variedade de estruturas de controle de fluxo de processamento. Estas estruturas serão vistas em detalhes nos capítulos 4 e 5. Duas estruturas das estruturas básicas (decisão e repetição) são muito semelhantes as estruturas usadas nas Pseudo-linguagem algorítmicas:

Estrutura de Decisão: Permite direcionar o fluxo lógico para dois blocos distintos de instruções conforme uma condição de controle.

Pseudo-linguagem	*Linguagem C*
se condição	*if(condição){*
então bloco 1	*bloco 1;*
senão bloco 2	*}else{*
fim se	*bloco 2;*
	};

Estrutura de Repetição: Permite executar repetidamente um bloco de instruções ate que uma condição de controle seja satisfeita.

Pseudo-linguagem	*Linguagem C*
faça	*do{*
bloco	*bloco;*
até	condição

Capítulo 2

Constantes, variáveis e operadores

Este capítulo aborda como os dados constantes e variáveis são manipulados pela linguagem C/C++. Quais os tipos de dados disponibilizados, e especificar os que estão incorporados apenas C++ como, por exemplo, *strings*. Também são apresentadas as formas de declaração das variáveis e constantes simbólicas usadas no programas.

2.1 Constantes

O C/C++ possui quatro tipos básicos de tipos de constantes: inteiras, de ponto flutuante, caracteres e *strings*. Constantes inteiras e de ponto flutuante representam números de um modo geral. Caracteres e strings representam letras e agrupamentos de letras (palavras).

2.1.1 Constantes inteiras

Uma constante inteira é um número de valor inteiro. Números inteiros podem ser escritos no formato decimal (base 10), hexadecimal (base 16) ou octal (base 8).

Uma constante inteira decimal é formada por uma seqüência de dígitos decimais: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Se a constante tiver dois ou mais dígitos, o primeiro não pode ser 0. Na verdade, pode ser 0 mas o compilador considerará esta constante como octal e não decimal.

Exemplo: A seguir são mostradas algumas constantes inteiras decimais válidas.

0 3 -45 26338 -7575 1010

Exemplo: Algumas constantes inteiras decimais inválidas.

1. (ponto)

1,2 (vírgula)

045 (primeiro dígito é 0: não é constante decimal)

212-22-33 (caracter ilegal: -)

Uma constante inteira hexadecimal é formada por uma seqüência de dígitos decimais: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (ou a, b, c, d, e). Uma constante hexadecimal deve começar por 0x. Neste caso, os dígitos hexadecimais podem ser minúsculos ou maiúsculos.

Exemplo: A seguir são mostradas algumas constantes inteiras hexadecimais válidas.

0x0 0x3 0x4f5a 0x2FFE OxABCD 0xAaFf

Exemplo: Algumas constantes inteiras hexadecimais inválidas.

0x3. (ponto)

0x1,e (vírgula)

0x ff (espaço)

FFEE (não começa com 0x: não é constante hexadecimal)

0Xfg34 (caracter ilegal: g)

Uma constante inteira octal é formada por uma seqüência de dígitos octais: 0, 1, 2, 3, 4, 5, 6, 7. A constante octal deve ter o primeiro digito 0 para que o compilador a identifique como tal

Exemplo: A seguir são mostradas algumas constantes octais válidas.

00 -03 045 02633 07575 -0101

Exemplo: Algumas constantes inteiras octais inválidas.

010. (ponto)

01,2 (vírgula)

0 4 (espaço)

45 (primeiro digito não é 0: não é constante octal)

01784 (caracter ilegal: 8)

2.1.1 Constantes de ponto flutuante

Números reais (não inteiros) são representados em base 10, por números com um ponto decimal e (opcionalmente) um expoente (mantissa). Um número ponto flutuante deve ter um ponto decimal que não pode ser substituído por uma vírgula. Um número de ponto flutuante pode ser escrito em notação científica. Neste caso o x10 é substituído por e ou E. O número 1.23e4 representa 1.23×10^4 ou 12300.

Exemplo: Números de ponto flutuante válidos.

0.234 125.65 .93 1.23e-9 -1.e2 10.6e18 -.853E+67

A forma de representação de um número real em C é bastante flexível.

Exemplo: O número 314 pode ser representado por qualquer uma das seguintes formas:

314. 3.14e2 +3.14e+2 31.4e1 .314E+3 314e0

2.1.1 Constantes caracteres

Uma constante caracter é uma letra ou símbolo colocado entre aspas simples.

Exemplo: Abaixo estão representados alguns constantes caracteres.

'a' 'b' 'X' '&' '{' ' '

Embora sejam visualizados como letras e símbolos as constantes caracteres são armazenadas internamente pelo computador como um número inteiro entre 0 e 255. O caracter 'A' por exemplo, tem valor 65. Os valores numéricos dos caracteres estão padronizados em uma tabela chamada de *American Standard Code for Information Interchange Table* ou simplesmente tabela ASCII.

Certos códigos de controle da tabela ASCII (como o *line feed*) ou caracteres especiais (como ') possuem representação especial no C/C++. Esta representação chama-se seqüência de escape representada por uma *barra invertida* (\) e um caracter. Seqüências de escape são interpretadas como caracteres simples. Na Tabela 1 é mostrada a principal seqüência de escape usada em C/C++.

Controle/Caracter	Seqüência de Escape	Valor ASCII
Nulo (*null*)	\0	00
Campainha (*bell*)	\a	07
Retrocesso (*backspace*)	\b	08
Tabulação	\t	09
Nova Linha	\n	10
Tabulação Vertical	\v	11
Alimentação de Folha (*form feed*)	\f	12
Retorno de carro (*carriage return)*	\r	13
Aspas (“)	\”	34

Apóstrofo	\’	39
Interrogação	\?	63
Barra Invertida (\)	\\	92

Tabela 1 – Principais seqüências de escape

2.1.1 Constantes *strings*

Uma constante do tipo *string* consiste de um conjunto de caracteres colocados entre aspas duplas. Embora as instruções do C/C++ usem apenas os caracteres do conjunto padrão ASCII, as constantes caracter e *string* podem conter caracteres do conjunto estendido ASCII: é, ã, ç, ü, ...

Exemplo: Abaixo seguem algumas constantes *strings* válidas.

”Oba!”

”Caxias do Sul”

”A resposta é: ”

”João Carlos da Silveira”

”a”

”isto é uma string”

2.2 Identificadores

Identificadores são os nomes das variáveis e funções usadas no programa.

2.2.1 Regras de sintaxe

Os identificadores devem seguir as seguintes regras de construção:

- Os identificadores devem começar por uma letra (a - z, A - Z) ou um *underscore* (_).
- O resto do identificador deve conter apenas letras, *underscores* ou dígitos (0 - 9). Não pode conter outros caracteres. Em C/C++, os identificadores podem ter até 32 caracteres.
- Em C/C++, letras maiúsculas são diferentes de letras minúsculas: Por exemplo: MAX, max, Max são nomes diferentes para o compilador. Esta propriedade é chamada de casos sensíveis (*case sensibility)*.

Exemplo: os nomes abaixo são válidos:

abc, y24, VetorPontosMovimentoRobo, nota_1, TAM_MAX.

Exemplo: os nomes abaixo não são válidos:

3dia, vetor-1, pao&leite, iteração.

2.2.1 Palavras reservadas

Existem certos nomes que não podem ser usados como identificadores. São chamadas as *palavras reservadas* e são de uso restrito da linguagem C/C++ (comandos, estruturas, declarações, etc.). Na Tabela 2 são listados algumas palavras reservadas.

asm	auto	break	case	cdecl	char
class	const	continue	_cs	default	delete
do	double	_ds	else	enum	_es
extern	_export	far	_fastcall	float	for
friend	goto	huge	if	inline	int
interrupt	_loadds	long	near	new	operator
pascal	private	protected	public	register	return
_saveregs	_seg	short	signed	sizeof	_ss
static	struct	switch	template	this	typedef
union	unsigned	virtual	void	volatile	while

Tabela 2 – Palavras reservadas C/C++

Exemplo: Não é possível declarar o conjunto de variáveis abaixo, pois *do* e *new* são palavras reservadas.

do, re, mi, fa, sol, la, si

old, new

2.3 Tipos de dados

Em C/C++, como na maioria das linguagens, os dados são divididos tipos: inteiro, real, caracter, etc. Esta divisão se deve basicamente ao número de *bytes* reservados para cada dado. Cada tipo de dado possui um intervalo de valores permitidos.

2.3.1 Tipos básicos

Abaixo segue uma tabela com os tipos básicos de dados permitidos em C/C++. Os tipos *char* e *int* são inteiros e os tipos *float* e *double* são de ponto flutuante.

Tipo	Tamanho (bytes)	Intervalo	Uso
Char	1	-128 a 127	Número muito pequeno e caracter ASCII
Int	2	-32768 a 32767	Contador, controle de laço
Float	4	3.4e-38 a 3.4e38	Real (precisão de 7 dígitos)
Double	8	1.7e-308 a 1.7e308	Cientifico (precisão 15 dígitos)

2.3.2 Declaração de variáveis

Para que se possa usar uma variável em um programa, é necessário fazer uma *declaração de variável* antes. A declaração de variáveis simplesmente informa ao processador quais são os nomes utilizados para armazenar dados variáveis e quais são os tipos usados. Deste modo, o processador pode *alocar* (reservar) o espaço necessário na memória para a manipulação destas variáveis. É possível declarar mais de uma variável ao mesmo tempo, basta separá-las por vírgulas (,).

Sintaxe: A sintaxe para declaração de variáveis é a seguinte:

tipo variavel_1 [, variavel_2, ...] **;**

Onde *tipo* é o tipo de dado e *variavel_1* é o nome da variável a ser declarada. Se houver mais de uma variável, seus nomes são separados por vírgulas.

Exemplo: Declaração das variáveis:

```
int i;
int x,y,z;
char letra;
float nota_1,nota_2,media;
double num;
```

No exemplo acima, i, x, y e z foram declaradas variáveis inteiras. Assim elas podem armazenar valores inteiros de -32768 até 32767. Do mesmo modo letra foi declarada como variável caracter podendo receber valores de -128 até 127 ou caracteres

do conjunto padrão ASCII. As variáveis *nota_1, nota_2* e *media* foram declaradas como ponto flutuante tipo *float* e num como ponto flutuante tipo *double*.

A declaração de variáveis é feita, em geral, dentro de uma rotina. Por exemplo, a rotina principal *main()*. Deste modo se diz que está se fazendo uma declaração de variáveis *locais*. Variáveis locais podem ser referenciadas apenas dentro da rotina dentro da qual foi declarada, neste caso a rotina *main()*.

Exemplo: Observe o uso da declaração de variáveis no trecho de programa abaixo:

```
void main()
{
      float raio, area;     // declaracao de variaveis
      raio = 2.5;
      área = 3.14 * raio * raio;
}
```

No exemplo acima, as variáveis *area* e *raio* foram declaradas como sendo locais tipo *float*. Assim o processador faz a alocação de dois espaços (endereços) de 4 bytes cada para armazenar as informações, um para cada variável. Na terceira linha, o processador coloca no endereço alocado para raio o valor 2.5. Depois, na quarta linha, o processador coloca o resultado da conta (19.625) no endereço de área.

É possível fazer a declaração de variáveis fora de uma rotina. Neste caso diz-se que se fez a declaração de variáveis *globais*.

2.3.3 Tipos modificados

Além dos tipos de dados citados acima existem outros tipos de dados ditos modificados (Ver Tabela 3). Em C/C++ existem dois modificadores: o modificador *long* e o modificador *unsigned*. Tipicamente o modificador *long* aumenta o número de bytes usados para o registro do número. Por conseqüência o intervalo de validade do número fica aumentado significativamente. O modificador *unsigned*, usado somente em inteiros, permite que um bit usado para armazenar o sinal do número seja usado para guardar o valor do número. Em conseqüência disto o intervalo do número fica dobrado, porém somente permite o uso de números positivos.

Tipo	Tamanho (bytes)	Intervalo
Unsigned char	1	0 a 255
Unsigned int	2	0 a 65535
Long int	4	-2 147 483 648 a 2 147 483 647
Unsigned long int	4	0 a 4 294 967 295
Long double	10	3.4e-4932 a 1.1e4932

Tabela 3 – Tipos de modificados de dados

2.3.4 Strings.

Uma string é um conjunto ordenado de caracteres que pode ser armazenado sob forma de um vetor de ponteiros. Nos capítulos seguintes serão melhor detalhadas essas estruturas. Por enquanto, basta saber como declarar e armazenar um conjunto caracter em uma variável.

Sintaxe: Para declarar uma variável para receber um conjunto caracter deve-se escrever:

```
char* var;
```

Exemplo: No exemplo seguinte a variável *nome* foi declarada como conjunto caracter e em seguida recebe uma constante string.

```
char* nome;
nome = "João Carlos de Oliveira Andrade";
```

2.3.5 Inicialização de variáveis

Quando se faz a declaração de uma variável está se determinando que tipo de dado ela vai receber. É possível, em C/C++, declarar uma variável e já armazenar nela um valor inicial. Chamamos este procedimento de inicialização de uma variável.

Sintaxe: A sintaxe para a inicialização de variáveis é:

tipo var_1 = valor_1 [, var_2 = valor_2, ...] ;

Onde *tipo* é o tipo de dado, *var_1* é o nome da variável a ser inicializada e *valor_1* é o valor inicial da variável.

Exemplo: Inicialização de variáveis:

```
int i = 0, j = 100;
float num = 13.5;
char* titulo = " Programa Teste ";
```

No exemplo acima, *i* e *j* foram declaradas variáveis tipo *int*. O valor inicial de *i* é 0 e o de *j* é 100. Do mesmo modo num foi declarada como variável *float* com valor inicial de 13.5. Também a variável *titulo* foi declarada como um conjunto caracteres e recebeu como conteúdo inicial a string " Programa Teste ".

2.3.6 Conversão de tipo (*Casting*)

Algumas vezes é necessário, momentaneamente, modificar o tipo de dado representado por uma variável, isto é, pretende-se que o dado seja apresentado em um tipo diferente do qual a variável foi inicialmente declarada. Por exemplo: é declarada uma variável como *int* (inteiro) e em certa linha, espera-se que seu conteúdo seja apresentado como ponto flutuante (float). Este procedimento é chamado de conversão de tipo ou *casting* (moldagem, em inglês).

Sintaxe: A sintaxe da instrução de conversão de tipo é:

(*tipo*) *variável*

onde tipo é o nome do tipo ao qual queremos converter o dado armazenado em variável.

Exemplo: observe a conversão de tipo feita no exemplo abaixo:

```
int num;
float valor = 13.0;
num = (int)valor % 2;
```

No exemplo acima a variável valor foi declarada inicialmente como sendo do tipo *float* recebendo o valor inicial 13.0. Logo em seguida o conteúdo de valor é convertido para o tipo *int* para realizar a operação módulo (%) com o inteiro 2. Aqui a conversão é necessária pois a operação módulo somente pode ser feita com inteiros. É importante salientar que a conversão de tipo é feita com o dado armazenado em uma variável, mas a variável continua tendo o seu tipo original. No exemplo acima, a variável *valor* e os dados nela armazenados continuam sendo do tipo *float* após a conversão.

2.4 Constantes Simbólicas

Muitas vezes são identificadas constantes numéricas por um símbolo como, por exemplo, Pi = 3,14159. Desta forma, pode-se definir um nome simbólico para esta constante, isto é, há permissão para definir uma constante simbólica que represente valor.

2.4.1 Constantes definidas pelo programador

O programador pode definir constantes simbólicas em qualquer programa.

Sintaxe: A sintaxe da instrução de definição de uma constante simbólica é:

#define nome valor

Onde #define é uma diretiva de compilação que diz ao compilador para trocar as ocorrências do texto nome por valor. Observe que não há ; no final da instrução pois trata-se de um comando para o compilador e não para o processador. A instrução *#define* deve ser escrita antes da instrução de declaração da rotina principal.

Exemplo: a seguir definimos algumas constantes simbólicas.

```
#define PI  3.14159
#define ON  1
#define OFF 0
#define ENDERECO 0x378
void main(){
...
}
```

No exemplo acima, definimos PI como 3.14159. Isto significa que todas as ocorrências do texto PI será trocado por 3.14159. Assim se escrevemos uma instrução:

```
area = PI * raio * raio;
```

o compilador vai interpretar esta instrução como se fosse escrita assim:

```
area = 3.14159 * raio * raio;
```

Poderíamos escrever estas instruções assim:

```
float pi = 3.14159;
área = pi * área * área;
```

Porém este tipo de instrução tem duas desvantagens: Primeiro, reserva 4 bytes de memória desnecessariamente. Segundo, esta instrução é executada mais lentamente pois o processador precisa acessar a memória para verificar qual é o valor de pi.

Observe também que no exemplo foram definidos os nomes simbólicos com letras maiúsculas. Isto não é necessário, é perfeitamente permitido definir nomes simbólicos usando letras minúsculas, porém faz parte do jargão dos programadores C/C++ usar letras maiúsculas para definir constantes simbólicas.

O uso da diretiva *#define* não se restringe apenas ao apresentado acima, podemos usá-la para definir macro instruções. Não veremos o uso de macros neste texto, procure mais detalhes na bibliografia recomendada.

2.4.2 Constantes pré-definidas

Em alguns compiladores C/C++, algumas constantes simbólicas já estão pré-definidas. Estas constantes em geral definam alguns valores matemáticos (π, $\pi/2$, e, etc.), limites de tipos etc. Na Tabela 4 estão contidas algumas (existem muitas outras) constantes simbólicas pré-definidas no compilador Turbo C++ da Borland.

Biblioteca	**Constante**	**Valor**	**Significado**
math.h	M_PI	3.14159...	π
math.h	M_PI_2	1.57079...	$\pi/2$
math.h	M_PI_4	0,78539...	$\pi/4$
math.h	M_1_PI	0,31830...	$1/\pi$
math.h	M_SQRT2	1,41421...	$\sqrt{2}$
conio.h	BLACK	0	valor da cor (preto)
conio.h	BLUE	1	valor da cor (azul)
conio.h	GREEN	2	valor da cor (verde)
conio.h	CYAN	3	valor da cor (cyan)
conio.h	RED	4	valor da cor (vermelho)
conio.h	MAGENTA	5	valor da cor (magenta)
limits.h	INT_MAX	32767	limite superior do tipo int
limits.h	INT_MIN	-32768	limite inferior do tipo int

Tabela 4 – Constantes simbólicas pré-definidas

Cada uma das constantes acima está definida em uma biblioteca. Uma biblioteca, em C/C++, é um arquivo pré-compilado chamado arquivo *header* (cabeçalho, em inglês). Em cada biblioteca estão agrupadas constantes e funções semelhantes. Por exemplo,

constantes e funções matemáticas estão guardadas na biblioteca math.h (*mathematical functions*), constantes e funções de manipulação teclado e monitor estão guardadas na biblioteca conio.h (*console input and output*). Para que se possa usar a constante simbólica em um programa é preciso incluir a biblioteca na compilação do programa.

Sintaxe: A sintaxe de inclusão de bibliotecas é a seguinte:

#include <*nome_bib*>

onde *nome_bib* é o nome da biblioteca que se deseja incluir. Esta instrução deve ser escrita antes do programa principal.

Exemplo: O programa abaixo usa a constante predefinida M_PI para calcular a área de um disco circular.

```
#include <iostream.h>
#include <math.h>
void main(){
  float area, raio = 5.0;
  área = M_PI * raio * raio;
}
```

2.5 Operadores

Os operadores podem ser de atribuição, aritméticos, de atribuição aritmética, incrementais, relacionais, lógicos e condicionais.

Exemplo: o símbolo + é um *operador* que representa a operação aritmética de adição. O identificador sqrt() é uma *função* que representa a operação de extrair a raiz quadrada de um número.

Uma expressão é um arranjo de operadores e operandos. A cada expressão válida é atribuído um valor numérico.

Exemplo: 4+6 é uma expressão cujo valor é 10. A expressão sqrt(9.0) tem valor 3.0.

2.5.1 Operador de Atribuição

A operação de atribuição é a operação mais simples do C. Consiste de atribuir valor de uma expressão a uma variável.

Sintaxe: A sintaxe da operação de atribuição é a seguinte:

identificador = *expressão*;

onde *identificador* é o nome de uma variável e *expressão* é uma expressão válida (ou outro identificador).

Exemplo: A seguir são mostradas algumas atribuições válidas:

```
a = 1;
delta = b * b - 4. * a * c;
i = j;
```

Observe que o símbolo de atribuição (=) não tem o mesmo significado que o usual da matemática que representa a igualdade de valores. Este símbolo, em C/C++, representa a atribuição do valor calculado em *expressão* a variável *identificador*. Também não se pode confundir o operador de atribuição (=) com o operador relacional de igualdade (==).

Observa-se também que o operando esquerdo deve ser um identificador de variável, isto é, não pode ser uma constante ou expressão.

Exemplo: A seguir são mostradas algumas atribuições inválidas:

```
1 = a;          // constante!
b + 1 = a;      // expressão!
```

2.5.1 Atribuição múltipla.

E possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de atribuição múltipla.

Sintaxe: A sintaxe da atribuição múltipla é seguinte:

var_1 = [*var_2* = ...] *expressão*;

onde *var_1*, *var_2*, ... são os identificadores de variáveis e *expressão* é uma expressão válida.

Observe que na atribuição múltipla as operações ocorrem da direita para a esquerda, isto é, inicialmente o valor de *expressão* é atribuído a *var_2* e depois o valor de *var_2* é atribuído a *var_1*. Deve-se tomar cuidado com as conversões de tipo e limites de intervalo para atribuições de tipos diferentes.

Exemplo: Observe a instrução de atribuição múltipla abaixo: as variáveis inteiras i, j e k são todas inicializadas com o valor 1. E as variáveis de dupla precisão max e min são inicializadas com o valor 0.0:

```
int i, j, k;
double max, min;
```

i = j = k = 1;

max = min = 0.0;

2.6 Operadores Aritméticos

Existem cinco operadores aritméticos em C/C++. Cada operador aritmético está relacionado ao uma operação aritmética elementar: adição, subtração, multiplicação e divisão. Existe ainda um operador (%) chamado operador de módulo cujo significado é o resto da divisão inteira. Note que na Tabela 5 estão os operadores Aritméticos.

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão inteira)

Tabela 5 – Operadores Aritméticos

Sintaxe: A sintaxe de uma expressão aritmética é:

operando operador operando

onde *operador* é um dos caracteres mostrados acima e *operando* é uma constante ou um identificador de variável.

Exemplo: Algumas expressões aritméticas:

1+2 a-4.0 b*c valor_1/taxa num%2

Não existe em C/C++, como existe em outras linguagens, um operador específico para a operação de potenciação (a^b). Existe, porém, uma função de biblioteca (pow()) que realiza esta operação. Embora as operações do C sejam semelhantes as operações aritméticas usuais da matemática, alguns detalhes são específicos da linguagem, e devem ser observados.

2.7 Operadores de Atribuição Aritmética

Muitas vezes queremos alterar o valor de uma variável realizando alguma operação aritmética com ela. Como por exemplo: i = i + 1 ou val = val * 2. Embora seja perfeitamente possível escrever estas instruções, foi desenvolvido na linguagem C/C++ algumas instruções otimizadas com o uso de operadores ditos operadores de atribuição aritmética. Os símbolos usado são (+=, -=, *=, /= , %=). Deste modo as instruções acima podem ser rescritas como: i += 1 e val *= 2, respectivamente.

Sintaxe: A sintaxe da atribuição aritmética é a seguinte:

var += *exp*;

var -= *exp*;

var *= *exp*;

var /= *exp*;

var %= *exp*;

onde *var* é o identificador da variável e *exp* é uma expressão válida. Estas instruções são equivalentes as seguintes:

var = var + *exp*;

var = var - *exp*;

var = var * *exp*;

var = var / *exp*;

var = var % *exp*;

Exemplo: Observe as atribuições aritméticas na Tabela 6 e suas instruções equivalentes:

Atribuição aritmética	Instrução equivalente
i += 1;	i = i + 1;
j -= val;	j = j - val;
num *= 1 + k;	num = num * (1 + k);
troco /= 10;	troco = troco / 10;
resto %= 2;	resto = resto % 2;

Tabela 6 – Atribuições aritméticas

O operador de atribuição aritmética tem precedência menor que os outros operadores até aqui discutidos.

2.8 Operadores Incrementais

Em programação existem instruções muito comuns chamadas de incremento e decremento. Uma instrução de incremento adiciona uma unidade ao conteúdo de uma variável, enquanto que a de decremento subtrai uma unidade do conteúdo de uma variável.

Existem, em C/C++, operadores específicos para realizar as operações de incremento (++) e decremento (--). Eles são genericamente chamados de operadores incrementais.

Sintaxe: A sintaxe dos operadores incrementais é apresentada na Tabela 7, mostrando sua linha de comando similar.

Instrução	Equivalente
++var	var=var+1
var++	var=var+1
--var	var=var-1
var--	var=var-1

Tabela 7 – Incremento e Decremento da variável *var*

Observe que existem duas sintaxes possíveis para os operadores: pode-se colocar o operador à esquerda ou á direita da variável. Nos dois casos o valor da variável será incrementado (ou decrementado) de uma unidade. Porém se o operador for colocado à esquerda da variável, o valor da variável será incrementado (ou decrementado) antes que a variável seja usada em alguma outra operação. Caso o operador seja colocado à direita da variável, o valor da variável será incrementado (ou decrementado) depois que a variável for usada em alguma outra operação.

Exemplo: Observe o fragmento de código abaixo e note o valor que as variáveis recebem após a execução da instrução:

```
int a, b, c, i = 3;        // a: ?   b: ?   c: ?   i: 3
a = i++;                   // a: 3   b: ?   c: ?   i: 4
b = ++i;                   // a: 3   b: 5   c: ?   i: 5
c = --i;                   // a: 3   b: 5   c: 4   i: 4
```

Os operadores incrementais são bastante usados para o controle de laços de repetição e tem a mais alta precedência entre todos, sendo superados apenas pelos parênteses. É importante que se conheça exatamente o efeito sutil da colocação do operador, pois isto pode enganar o programador inexperiente.

2.9 Operadores Relacionais e Lógicos

Operador	Significado
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
==	Igual a
!=	Diferente de

Tabela 8 – Operadores Relacionais

Os operadores relacionais de igualdade (== e !=) tem precedência **menor** que os de magnitude (>, <, >= e <=). Estes, por sua vez, tem precedência **menor** que os operadores aritméticos. Operadores relacionais de mesma precedência são avaliados da esquerda para a direita.

Operador	Significado
&&	E(AND)
||	Ou(OR)
!	Não(NOT)

Tabela 9 – Operadores Lógicos

São três os operadores lógicos de C/C++: &&, || e !. Estes operadores tem o mesmo significado dos operadores lógicos Booleanos AND, OR e NOT.

O Operador && tem precedência sobre o operador ||. Estes dois têm precedência menor que os operadores relacionais. O operador ! tem a mesma precedência que os operadores incrementais.

2.10 Operador Condicional

O operador condicional (?:) é usado em expressões condicionais. Uma expressão condicional pode ter dois valores diferentes dependendo de uma condição de controle.

Sintaxe: A sintaxe de uma expressão condicional é:

condição ? *expressão_1* : *expressão_2*

onde *expressão_1* e *expressão_2* são duas expressões quaisquer, e *condição* é uma expressão lógica que será avaliada primeiro. Se o valor de *condição* for 1, isto é, verdadeiro, então a expressão condicional assumirá o valor de *expressão_1*. Caso contrario assumirá o valor de *expressão_2*. Uma expressão condicional é equivalente a uma estrutura de decisão simples:

se condição
então expressao_1
senão expressao_2
fim se

Exemplo: Observe as expressões condicionais abaixo e verifique o resultado de sua avaliação. Admita que i, j e k são variáveis tipo int com valores 1, 2 e 3, respectivamente.

Expressão	Valor
i ? j : k	2
j > i ? ++k : --k	4
k == i && k != j ? i + j : i - j	-1
i > k ? i : k	3

Capítulo 3

Comandos de Controle

Os comandos de controle dos programas em C/C++ são divididos pelo padrão ANSI nos seguintes grupos:

- Seleção (if e switch)
- Iteração (while, for e do-while)
- Desvio (break, continue, goto e return)
- Rótulo (case, default e label)
- Expressão (expressões válidas em C/C++)
- Bloco (blocos de códigos)

Muitos comandos em C/C++ contam com testes de condicionais que determina o curso da ação. Uma expressão condicional chega a um valor verdadeiro ou falso. No C/C++, ocorre de estar organizado em valores diferente de zero caso a condição seja verdadeira, e a expressão retorna o valor zero quando for falso. Importante conceito para o entendimento dos comandos de controle.

3.1 Comandos de Seleção

3.1.1 IF

if (condição) declaração;

A expressão, na condição, será avaliada. Se ela for zero, a declaração não será executada. Se a condição for diferente de zero a declaração será executada. Abaixo segue um exemplo do uso de IF.

```
#include <iostream.h>

int main()
{
        int nota;
        cout << "Informe a nota do Aluno:";
        cin >> nota;
        if ((nota <=10) && (nota >= 9))
                printf ("Nota A\n");
        if ((nota <9) && (nota >= 7))
                printf ("Nota B\n");
        if ((nota <7) && (nota >= 6))
                printf ("Nota C\n");
        if ((nota <6) && (nota > 0))
                printf ("Nota D\n");
        if (nota==0)
                printf ("Nota E\n");
        if (nota >=6)
                printf ("Aluno Aprovado\n");
        if (nota < 6)
                printf ("Aluno Reprovado\n");
return 0;

}
```

3.1.2 **ELSE**

Podemos pensar no comando else como sendo um complemento do comando if. O comando if completo tem a seguinte forma geral:

if (condição) declaração_1;
else declaração_2;

A expressão da condição será avaliada. Se ela for diferente de zero a declaração 1 será executada. Se for zero a declaração 2 será executada. É importante nunca esquecer que, quando usamos a estrutura if-else, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Abaixo está um exemplo do uso do if-else.

```
#include <iostream.h>
int main ()
{
int num;
cout << "Digite um numero: ";
cin >> num;
if (num==10)
      {
      cout << "\n\nVocê acertou!\n";
      cout << "O numero e igual a 10.\n";
      }
else
```

```
        {
        cout << "\n\nVocê errou!\n";
        cout << "O numero e diferente de 10.\n";
        }
return 0;
}
```

3.1.3 **IF-ELSE-IF**

A estrutura if-else-if é apenas uma extensão da estrutura if-else. Sua forma geral pode ser escrita como sendo:

if (condição_1) declaração_1;
else if (condição_2) declaração_2;
else if (condição_3) declaração_3;
.
.
.
else if (condição_n) declaração_n;
else declaração_default;

A estrutura acima funciona da seguinte maneira: o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero. Neste caso ele executa a declaração correspondente. Só uma declaração será executada, ou seja, só será executada a declaração equivalente à *primeira* condição que der diferente de zero. A última declaração (default) é a que será executada no caso de todas as condições darem zero e é opcional. Um exemplo da estrutura acima:

```
#include <iostream.h>
int main()
{
        int nota;
        cout << "Informe a nota do Aluno:";
        cin >> nota;
        if ((nota <=10) && (nota >= 9))
                printf ("Nota A\n");
        else if (nota >=7)
                printf ("Nota B\n");
        else if (nota >=6)
                printf ("Nota C\n");
        else if (nota >=5)
                printf ("Nota D\n");
        else
                printf ("Nota E\n");
        if (nota >=6)
                printf ("Aluno Aprovado\n");
        else
```

```
                    printf ("Aluno Reprovado\n");
    return 0;
    }
```

3.1.4 IFs aninhados

O if aninhado é simplesmente um if dentro da declaração de um outro if externo. O único cuidado que devemos ter é o de saber exatamente a qual if um determinado else está ligado. Abaixo segue um exemplo.

```
#include <iostream.h>
int main ()
{
int num;
cout << "Digite um numero: ";
cin  >> num;
if (num==10)
        {
        cout << "\n\nVoce acertou!\n";
        cout << "O numero e igual a 10.\n";
        }
else
        {
        if (num>10)
                {
                cout << "O numero e maior que 10.";
                }
        else
                {
                cout << "O numero e menor que 10.";
                }
        }
        cin >> num;
        return 0;
}
```

3.1.5 Operador ?

Uma expressão como:

```
if (a>0)
    b=-150;
else
    b=150;
```

pode ser simplificada usando-se o operador **?** da seguinte maneira:

```
b=a>0?-150:150;
```

De uma maneira geral expressões do tipo:

if (condição)

expressão_1;

else

expressão_2;

podem ser substituídas por:

condição?expressão_1:expressão_2;

O operador ? é limitado (não atende a uma gama muito grande de casos) mas pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular. Veja o exemplo:

```
#include <iostream.h>
int main()
{
     int index = 0, contador;
     char letras[5] = "Joao";
     for (contador=0; contador<1000; contador++)
        {
        printf("%c\n", letras[index]);
        index = (index == 4)? index=0: ++index;
        }
     cin >> index;
     return 0;
}
```

O nome Joao é escrito na tela verticalmente até a variável contador determinar o término do programa. Enquanto isto a variável index assume os valores 0, 1, 2, 3, 4, 0, 1, ... progressivamente.

3.1.6 **SWTICH**

O comando if-else e o comando switch são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o if, mas o comando switch tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando switch é próprio para se testar uma variável em relação a diversos valores preestabelecidos. Sua forma geral é:

switch (variável)

```
    {
case constante_1:
declaração_1;
break;
case constante_2:
declaração_2;
break;
.
.
.
case constante_n:
decalração_n;
break;
default
declaração_default;
}
```

Podemos fazer uma analogia entre o switch e a estrutura if-else-if apresentada anteriormente. A diferença fundamental é que a estrutura switch *não* aceita expressões. Aceita apenas constantes. O switch testa a variável e executa a declaração cujo case corresponda ao valor atual da variável. A declaração default é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes.

O comando break, faz com que o switch seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando switch. Se após a execução da declaração não houver um break, o programa continuará executando. Isto pode ser útil em algumas situações, mas eu recomendo cuidado. Veremos agora um exemplo do comando switch:

```
#include <iostream.h>
int main()
{
        int total_alunos=7, num_aluno=1;
        while (num_aluno <= total_alunos)
        {
                switch(num_aluno)
                {
                case 5:
                        printf ("Carlos\n"); break;
                case 4:
                        printf ("Ana\n"); break;
                case 3:
                        printf ("Fabiano\n"); break;
                case 2:
                        printf ("Luciano\n"); break;
                case 1:
                        printf ("Eduardo\n"); break;
                default:
```

```
                        printf ("Nenhum aluno cadastrado!\n");
                }
                num_aluno++; // Incremento
        }
        return 0;
}
```

3.2 Comandos de Iteração

3.2.1 FOR

O for é o primeiro de uma série de três comandos para se trabalhar com loops. Os outros são o while e o do. Os três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a dos comandos de repetição controlada.

Como já foi dito, o loop for é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

for (inicialização;condição;incremento) declaração;

O melhor modo de se entender o loop for é ver como ele funciona "por dentro". O loop for é equivalente a se fazer o seguinte:

inicialização;

if (condição)

{
declaração;
incremento;
"Volte para o comando if"
}

Podemos ver, então, que o for executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Um ponto importante é que podemos omitir qualquer um dos elementos do for, isto é, se não quisermos uma inicialização poderemos omiti-la. Segue um exemplo.

```
#include <iostream.h>

int main()
{
        int valor=10,
        final =100,
        contador;
        for (contador=0; contador < 100; contador++)
                valor = valor + 10;
        cout << "O valor final é "<< valor << "\n";
        return 0;
}
```

Esta é a forma usual de se fazer o incremento (ou decremento) em um loop for. Todas as linguagens de programação possuem algo parecido com o for do C/C++. Mas, o for na linguagem C/C++ é muito mais flexível e poderoso. Temos acesso à inicialização, à condição e ao incremento. Isto permite manipular a estrutura da forma desejada.

3.2.2 WHILE

O comando while tem a seguinte forma geral:

while (condição) declaração;

Assim como fizemos para o comando anterior, vamos tentar mostrar como o while funciona fazendo uma analogia. Então o while seria:

if (condição)

{

declaração;

"Volte para o comando if"

}

Podemos ver que a estrutura while testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do for, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Veja o exemplo do uso do while.

```
#include <iostream.h>
int main()
{
        int total_alunos=10, num_aluno=1, nota;
        while (num_aluno <= total_alunos)
        {
```

```
                        cout << "Forneça a Nota do " << num_aluno << "°:
";
                        cin >> nota;
                        if ((nota <=10) && (nota >= 9))
                                printf ("Nota A\n");
                        else if (nota >=7)
                                printf ("Nota B\n");
                        else if (nota >=6)
                                printf ("Nota C\n");
                        else if (nota >0)
                                printf ("Nota D\n");
                        else
                                printf ("Nota E\n");
                        if (nota >=6)
                                printf ("Aluno Aprovado\n");
                        else
                                printf ("Aluno Reprovado\n");
                        num_aluno++; // Incremento
                }
                return 0;
        }
```

Capítulo 4

Funções

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se elas não existissem, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos é necessário construi-los bloco a bloco.

Uma função no C/C++ tem a seguinte forma geral:

tipo_de_retorno nome_da_função (declaração_de_parâmetros)
{
corpo_da_função
}

O tipo de retorno é o tipo de variável que a função vai retornar, por padrão o tipo int, ou seja, uma função para qual não declarada o tipo de retorno é considerado como retorno um valor inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1,tipo nome2, ... ,tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).

Além disso, as funções que necessitam de retorno de valores utilizam o comando *return* tem a seguinte forma geral:

return valor_de_retorno; ou *return;*

Exemplo, imaginar que uma função está sendo executada. Quando se chega a uma declaração return a função é encerrada imediatamente e, se o valor de retorno é

informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser, pelo menos, compatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração *return*. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração *return*. Veja o exemplo.

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

double qual_seno(float valor)
{
   if (valor > 30)
         return sin(valor);
   else
          return cos(valor);
}

int main()
{
         double valor=0,
         seno;
         while (valor <= 360)
         {
                  seno = qual_seno(valor);
                  cout << "O Seno é " << seno << "\n";
                  valor += 30;
         }
         return 0;
}
```

No geral, as funções não obrigatoriamente são escritas antes da função *main.* Para possibilitar que as funções sejam escritas após a *main* é necessário que sejam "*prototipadas*" antes.

Os Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que usará a seguir. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

Onde o tipo_de_retorno, o nome_da_função e a declaração_de_parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Veja um exemplo de código.

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

double qual_seno(float);  <-- Protótipo

int main()
{
        double valor=0,
        seno;
        while (valor <= 360)
        {
                seno = qual_seno(valor);
                cout << "O Seno é " << seno << "\n";
                valor += 30;
        }
        return 0;
}

double qual_seno(float valor)
{
        return sin(valor);
}
```

A importância de entendimento das funções também está relacionado com a compreensão das definições de escopo das variáveis. Todas as variáveis declaradas fora de qualquer função é considerada uma variável global. No caso de uma variável declarada dentro de alguma função, tem-se a variável de escopo local, sendo possível utiliza-la apenas dentro desta função.

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

double valor;  <-- Variável Global
double qual_seno();

int main()
{
         double seno;  <-- Variável Local
        while (valor <= 360)
        {
                seno = qual_seno(valor);
                cout << "O Seno é " << seno << "\n";
                valor += 30;
        }
        return 0;
}

double qual_seno()
{
        return sin(valor);
}
```

4.1 Chamada por Valor e Chamada por Referência

Já foi visto que quando uma função é chamada os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios. Veja o exemplo abaixo.

```
#include <iostream.h>
float sqr (float num);
void main (void)
{
 float num,sq;
 printf ("Entre com um numero: ");
 scanf ("%f",&num); sq=sqr(num);
 printf ("\n\nO numero original e: %f\n",num);
 printf ("O seu quadrado vale: %f\n",sq);
}
float sqr (float num)
{
 num=num*num;
 return num;
}
```

No exemplo acima o parâmetro formal num da função sqr() sofre alterações dentro da função, mas a variável num da função main() permanece inalterada: é uma chamada por valor.

Outro tipo de chamada de função é quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um & (Será melhor detalhado no Capítulo 6) na frente das variáveis que estivermos passando para a função. Veja o exemplo.

```
#include <stdio.h>
void Swap (int *a,int *b);
void main (void)
{
      int num1,num2;
      num1=100;
      num2=200;
      Swap (&num1,&num2);
      printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}
void Swap (int *a,int *b)
{
      nt temp;
      temp=*a;
      *a=*b;
      *b=temp;
}
```

4.2 Argumentos argc e argv para a main()

Algumas vezes é importante que sejam passadas informações para um programa ao invocar sua execução. Geralmente, esses valores são passados para a função *main()* através da linha de comando. Um argumento da linha de comando é informação que segue o nome do programa na linha de comando do sistema operacional. Um exemplo interessante, suponha que tem um programa de cálculo de matriz que onde pela linha de comando é possível especificar as dimensões das matrizes, tornando o aplicativo mais dinâmico para a realização dos testes.

A linha de comando para a passagem dos argumentos pode ser:

*Int main(int argc, char ** argv)*

4.3 Recursão

A recursão pode ser definida como sendo a função que chama a si mesma direta ou indiretamente, como forma de iteração, veja o exemplo de recursão para a resolução de Fatorial. Esse recurso é complexo de ser compreendido e de ser implementado corretamente, principalmente quando é possível a resolução através das estruturas de repetição.

```
#include <iostream.h>
unsigned long fatorial(unsigned long);
int main()
{
        int i;
        for (i=0; i <= 10; i++)
                cout << i << "! = " << fatorial (i) << endl;
```

```
      return 0;
}
// Funcao recursiva da Fatoracao
unsigned long fatorial (unsigned long valor)
{
        if(valor <=1)
                return valor;
        else
                return valor * fatorial(valor-1);
}
```

Outro interessante exemplo é a resolução da série de Fibonacci que é mostrada abaixo.

```
#include <iostream.h>
unsigned long fibonacci(unsigned long);
int main()
{
        unsigned long resultado, numero;
        cout << "Digite um número inteiro: ";
        cin >> numero;
        resultado = fibonacci(numero);
        cout << "Fibonacci (" << numero << ") = " <<
resultado << endl;
        return 0;
}
// Funcao de Fibonacci
unsigned long fibonacci (unsigned long valor)
{
        if(valor==0 || valor==1) // base
                return valor;
        else
                return fibonacci(valor-1) + fibonacci(valor-
2);
}
```

Capítulo 5

Ponteiros

O gerenciamentos dos endereços de memória é importante para a realização de diversas tarefas como, por exemplo, nos sistemas de gerenciamento de banco de dados (SGBD) para agilizar os procedimentos de inserção, alteração e exclusão dos dados.
No C/C++ existe um operador especial para a obtenção de elementos na memória, o operador é chamado de operador de *endereço*, *endereço-de* ou *referência* (o operador & endereço, que toma um operando, que não deve ser confundido com o operador & bit-a-bit, que toma dois operandos). No exemplo a seguir é apresentado um exemplo para verificação do endereço da variável *carro*.

```
#include <iostream.h>

int main()
{
        int carro = 10;
        cout << "O código do carro no endereço " << &carro << " é: " <<
carro;
        return 0;
}
```

A declaração dos ponteiros utiliza o * após o tipo de dado definido, como no esquema a seguir:

*<tipo de dado> * <ponteiro>;*

O uso de ponteiros muitas vezes pode causar problemas, por não serem inicializados corretamente, apontarem para endereços de memória inválidos entre outros problemas graves que podem comprometer o equipamento. É importante compreender o emprego dos operadores dos ponteiros, pois a interpretação incorreta, causam resultados imprevistos.

```
#include <iostream.h>

int main()
{
        int carro, *caminhao;
        carro = 10;
        caminhao = &carro;
        cout << "O código do carro é " << *caminhao << ", sendo o valor
do ponteiro.";
        cin >> carro;
        return 0;
}
```

5.1 Manipulando memória em tempo real

Para o C++, a memória está dividida em três áreas: *armazenamento automático*, onde as variáveis de uma função são alocadas e armazenadas quando o programa entra na função, *armazenamento estático* onde os dados são declarados como static e o *armazenamento livre* que no C++ é tido cp,p pilha ou *heap*.

A alocação dinâmica significa permitir alocar espaço para o número de elementos efetivamente necessário em tempo de execução, o que pode ser muito mais eficiente em vários casos.

O alocamento dinâmico pode ser realizado com o operador *new* e liberá-la com o operador *delete*.

No exemplo abaixo, é apresentado um esquema de alocação e liberação dinâmica de memória.

```
#include <iostream.h>

int main()
{
        double * pi = new double;
        *pi = 3.14159;
        cout << "O valor de PI é " << *pi << endl;
        delete pi;
        return 0;
}
```

Capítulo 6

Classes

A imersão na orientação a objetos começa a partir deste capítulo, no qual é iniciado a definição e os tratamentos que as classes podem receber na orientação a objetos utilizadanesta linguagem de programação.

É importante que alguns conceitos chaves e terminologias sejam destacadas na programação orientada a objetos (OOP), como o encapsulamento dos dados (atributos) e funções (comportamento) em pacotes chamados *classes,* sendo que os dados e as funções das classes estão interligados.

Uma classe pode ser reutilizada muitas vezes para criar diversos objetos. Os objetos de uma determinada classe podem comunicar-se com objetos de outras classes através de *interfaces,* que devem ser bem definidas. Segundo Deitel e Deitel (2004), as classes C++ são a evolução natural da noção de *struct* de C.

6.1 Definindo Classes

As funções membros de uma classe também podem ser chamadas de métodos, e são invocadas em reposta as *mensagens* enviadas aos objetos. A mensagem corresponde a uma chamada da função membro enviada de um objeto ao outro, ou enviada de uma função a um objeto. Função membro, também conhecida como método, é aquela definida em uma classe, sendo que esta pode ser definida dentro e fora da classe, porém o protótipo sempre deve permanecer dentro da classe.

A seguir é apresentado um simples exemplo de classe:

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

Construtor

Destrutor

```
            void DadosPessoais(int indice);
            void TestesFisicos(int indice);
      private:
            int aprovado;
            int altura;
};
```

Atenção: Comumente ocorre o erro de se esquecer o ponto-e-vírgula (;) ao final da definição de uma classe (ou estrutura), sendo um erro de sintaxe.

Os rótulos *public, protected* e *private* são chamados de especificadores de acesso a membros. O especificador *public* permite que todos os membros da classe sejam acessados de qualquer local do programa. No caso do especificador *private*, os dados podem ser acessados apenas de dentro da mesma classe onde ocorre a especificação, sendo um recurso de segurança das rotinas, importante para delimitar os recursos que podem ser acessados e uma ferramenta importante na Engenharia de Software.

Um exemplo muito interessante do emprego do rótulo *private* é em aplicações que possuem dados a serem ocultados fora da classe, como na especificação de um relógio, onde vários métodos são disponibilizados aos usuários para que as horas sejam manipuladas, porém, como uma hora sempre são sessenta minutos, e as checagens e transformações sempre levam esse parâmetro em consideração, pode optar por torná-los privados, não permitindo aos usuários que as alterem.

Esse simples exemplo apresenta a idéia fundamental da ocultação dos dados para o resto do código, porém no caso da intentar disponibilizar certos dados fora da classe, é importante que se tenha bem definido quais serão os tipos de acesso, pois podem ser utilizados os rótulos public e protected.

No caso protected os dados de uma classe podem ser acessados por classes derivadas a esta, ou seja, classes que possuem ligações explicitadas no código utilizando os recursos de herança (ver Capítulo 8). Enquanto que no rótulo public o acesso aos dados é indiscriminado, portanto, qualquer dado rotulado desta forma pode ser utilizado por todo o programa.

```
class Test {
      public:
            Test (int=0);
            void print() const;
      protected:
```

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

6.2 Ponteiro *this*

O ponteiro *this* é empregado para possibilitar a passagem por referência de objetos como no exemplo abaixo.

```
#include <iostream.h>

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

class Test {
            public:
                Test (int=0);
                void print() const;
            private:
                int x;
            };

Test::Test(int a) {x=a;};

void Test::print() const
{

            cout << x << endl;
            cout << this->x << endl;
            cout << (*this).x << endl;
};

int main()
{
            Test test(12);
            test.print();
            return 0;
}
```

Neste exemplo notamos a impressão do apontamento dos valores utilizando *this* que é o ponteiro para membros da classe.

Capítulo 7

Sobrecarga de Operadores

Os tipos abstratos de dados (TAD) são recursos presentes em todas as linguagens de programação. Em C++ não poderia ser diferente, valendo-se de processos sensíveis e enfocados nos tipos.

Além dos tipos de dados, em C++ existem os operadores que realizam operações entre os dados, podendo ser eles de qualquer tipo como, por exemplo, uma variável de valores numéricos inteiros ou uma classe. Essa vastidão no uso dos operadores vale-se do uso da sobrecarga dos operadores, ou seja, um operador pode ter diversas aplicações.

Para exemplificar a sobrecarga primitiva da linguagem é interessante comentar sobre o operador + (soma). A adição entre dois valores numéricos inteiros pode ser realizada, assim como a soma de um valor numérico inteiro (*int*) com um valor numérico de ponto flutuante com precisão dupla (*double*).

Essa possibilidade, e muitas outras, se dão pela sobrecarga dos operadores nos padrões da linguagem. Porém existem casos em que os programadores intentam por diferentes aplicações para determinado operador. Para isso, em C++ é permitido aos usuários criarem as sobrecargas necessárias, seguindo algumas regras que serão comentadas a seguir.

A principal ressalva para a sobrecarga de operadores é que somente os operadores já implementados podem ser sobrecarregados. No caso de operadores inexistentes na linguagem não há possibilidade de sobrecarga como, por exemplo, para simular um operador muito utilizado em BASIC o ** para cálculos exponenciais.

Na tabela a seguir são mostrados todos os operadores possíveis de serem sobrecarregados.

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

Tabela – Operadores que podem ser Sobrecarregados (Deitel e Deitel2004)

Além desses operadores, existem alguns operadores já definidos que não podem ser sobrecarregados.

.	.*	::	?:	sizeof

O recurso de sobrecarga é muito importante em C++, porém deve ser trabalhado com muito cuidado para não causar problemas de interpretação e questionamentos com o usuário final como, por exemplo, mudar o significado do operador +, tornando-o um operador de decremento ao invés de incremento.

A sobrecarga é realizada escrevendo-se uma definição de uma função com um cabeçalho e corpo, porém o nome da função se torna a palavra chave *operator* seguido pelo símbolo operador que está sendo sobrecarregado, como segue na especificação abaixo de uma sobrecarga de um operador unário.

Int Operator+() const

Outra importante regra da sobrecarga é manter o tipo do operador a ser sobrecarregado como, por exemplo, o operador unário ! (não=not) não pode ser sobrecarregado para se tornar um operador binário como o +=. Portanto, é importante que se tenha idéia do tipo de operador que se está utilizando para se evitar erros de interpretação na compilação.

A seguir é apresentado um exemplo simples tirado da referência [http://www.ic.unicamp.br/~cmrubira/aacesta/cpp/cpp15.html] do uso da sobrecarga de

```
#include <iostream.h>
class vetor2 {
      public:
            float f[2]; // array com 2 float
            vetor2(){}; // construtor sem parâmetros
```

```
            // sobrecarrega operator+ - ret = a+b - ret = operator+(a,b)
            friend vetor2 operator+(vetor2 & a, vetor2 & b)
            {
                  vetor2 ret;
                  ret.f[0] = a.f[0] + b.f[0];
                  ret.f[1] = a.f[1] + b.f[1];
                  return ret;
            };
            // sobrecarrega insersor (operator<<)
            friend ostream & operator<<(ostream & stream, const vetor2
      &obj)
            {
                  stream << "f[0]=" << obj.f[0] << endl << "f[1]=" <<
            obj.f[1] << endl;
                  return stream;
            };
}; // end of class vetor2

void main()
{
      vetor2 a, b, ret;
      a.f[0]=1.1; a.f[1]=1.2; b.f[0]=2.5; b.f[1]=2.6; // carrega dados
      ret = a+b; // ret = a.operator+(b), ou
ret.operator=(a.operator+(b))
      cout << "a" << endl << a; // imprime a
      cout << "b" << endl << b; // imprime b
      cout << "ret" << endl << ret; // imprime ret
}
```

Neste ponto é importante ter compreendido sobre as definições dos tipos e características das funções. As funções dos operadores podem ser membro ou não-membro, no caso das funções membro elas são freqüentemente definidas como *friends* por razões de desempenho. Porém, no caso das funções membros são utilizados os ponteiros *this* implicitamente para obter um dos argumentos do objeto da classe (o argumento à esquerda para operadores binários).

7.1 Função *Friend*

Quando uma função é declarada como *friend* (amiga), esta terá permissão para acessar os membros públicos, protegidos e privados da classe. Esse recurso é útil quando tem-se um número grande de classes e não pretende-se que sejam definidos um mesmo método em todas as classes.

Esse recurso ao ser utilizado pode causar dúvidas com relação ao encapsulamento dos dados, pois permite o acesso aos dados protegidos, porém ele apenas torna no membro uma extensão da classe, não comprometendo o encapsulamento.

```
#include <iostream.h>

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

class CL
{
      public:
              int count;
              CL operator=(int i);
              friend CL operator+(CL ob, int i);
              friend CL operator+(int i, CL ob);
};

CL CL::operator=(int i)
{
      count = i;
      return *this;
}

CL operator+(CL ob, int i)
{

            CL temp;
      temp.count = ob.count + i;
      return temp;
}

CL operator+(int i, CL ob)
{
      CL temp;
      temp.count = ob.count + i;
      return temp;
}

int main (void)
{
      CL obj;
      obj = 10;
      cout << obj.count << " ";
      obj = 10 + obj;
      cout << obj.count << " ";
      obj = obj + 12;
      cout << obj.count;
      return 0;
}
```

7.2 Sobrecarga de Operadores Unários

A sobrecarga de operadores unários que tomam um operando basicamente procede-se da seguinte maneira:

```
<Tipo> operator<operador> ()
{
...
}
```

O formato da definição é similar ao das funções mudando apenas que ao invés do nome da função, tem-se o indicador *operator* seguido do operador desejado. Sendo que o tipo de retorno pode ser um objeto ou um tipo de dado simples.

7.3 Sobrecarga de Operadores Binários

A sobrecarga de operadores binários utiliza dois operandos e é esquematizado de forma diferente da sobrecarga unária. Basicamente, a representação desse recurso pode ser dada por:

```
Tipo operator<operador> (nome_da_classe,segundo_operando)
{
...
}
```

Um importante e muito utilizado recurso, é o uso de const para indicar que o objeto não será alterado, sendo indicado da seguinte forma:

```
Tipo operator<operador> (const nome_da_classe,segundo_operando)
{
...
}
```

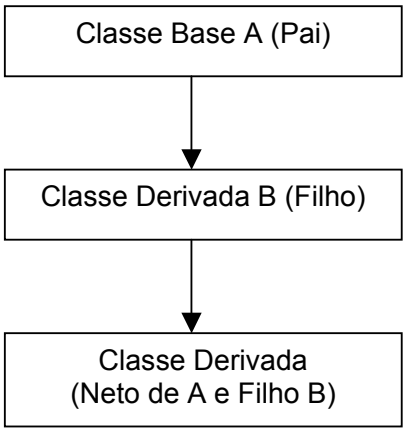
Capítulo 8

Herança

A herança pode ser definida computacionalmente como sendo a forma de reutilização *software* na qual novas classes são criadas a partir de algumas já existentes, ou seja, são criadas classes derivadas por fazerem uso de recursos disponibilizados por alguma classe base. Para isso utiliza-se dos atributos e comportamentos redefinindo-os ou "embelezando-os" com recursos que as novas classes requerem.

Juntamente com o polimorfismo, a herança é uma das principais técnicas disponibilizadas na linguagem C++, sendo principalmente importante para as representações/modelagens orientadas a objetos disponíveis na Engenharia de Software.

Abaixo é apresentado um esquema bem simples para demonstrar qual é a idéia básica de como trabalhar e pensar na hereditariedade.



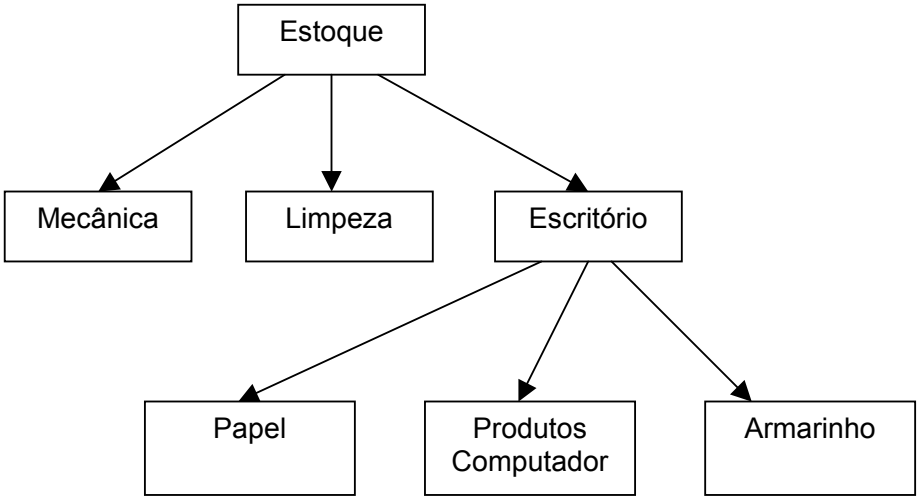
Seguindo o esquema acima, nota-se a hereditariedade seguindo do avô ao neto, porém em orientação a objetos costuma-se apenas mencionar quem é o pai e quem é o filho, para definir que é a classe base ou a derivada.

A herança varia de acordo com os rótulos das classes e métodos, valendo-se das propriedades da classe base. No algoritmo do exemplo abaixo, é apresentada uma herança pública.

```
#include <iostream.h>

class coordenadaX {
        public:
           void recebendo_x(int i);
        protected:
           int x;
};

class coordenadaZ: public coordenadaX
{
        public:
           int resultado_z(void);
         };

void coordenadaX::recebendo_x(int i)
{
  x=i;
}

int coordenadaZ::resultado_z(void)
{
   int z;
   z=3;
   return x*z;
}

main(void)
{
  coordenadaZ objeto;
  objeto.recebendo_x(20);
  cout << "A coordenada Z= " << objeto.resultado_z();
  int i;
  cin >> i;
  return 0;
}
```

Derivando Propriedades da classe coordenadaX

Na orientação a objetos, costuma-se tratar a classe base como sendo a superclasse e a classe derivada como a subclasse. A seguir é apresentado um esquema que será trabalhado ao longo deste capítulo para facilitar a seqüência de trabalho e entendimento.



No esquema acima se nota a importância da organização dos dado, para que não ocorram replicações desnecessárias. Portanto, é importante que haja um planejamento prévio de do funcionamento do sistema.
A sintaxe para utilizar a herança é a seguinte:

```
class mecanica : public Estoque
{
...
}
```

Após conceituar o desenvolvimento de rotinas utilizando herança, é importante enumerar algumas especificidades, como seguem nos tópicos a seguir.

Herança Múltipla

A herança múltipla ocorre quando um classe é derivada de mais de uma classe base. Este é um importante recurso da linguagem, porém pode causar sérios problemas de ambigüidade. O exemplo a seguir foi retirado do endereço [http://www.cttmar.univali.br/~ademir/progs_ex/clas13.txt], porém foram feitas algumas modificações para melhorar a compreensão.

```
#include <iostream.h>

class coordenadaX {
```

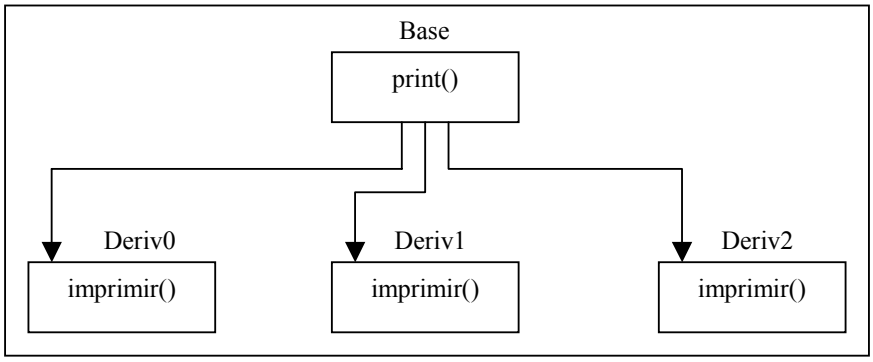
```
        public:
           void recebendo_x(int i);
        protected:
           int x;
};

class coordenadaY {
        protected:
                int y;
        public:
                void recebendo_y(int i);
};

class coordenadaZ: public coordenadaX, public coordenadaY {
        public:
           int resultado_z(void);
        };

void coordenadaX::recebendo_x(int i)
{
  x=i;
}
void coordenadaY::recebendo_y(int i)
{
   y=i;
}
int coordenadaZ::resultado_z(void)
{
   return x*y;
}

main(void)
{
  coordenadaZ objeto;
  objeto.recebendo_x(20);
  objeto.recebendo_y(12);
  cout << "A coordenada Z= " << objeto.resultado_z();
  return 0;
}
```

Capítulo 9

Polimorfismo e Funções Virtuais

9.1 Polimorfismo

O polimorfismo é o recurso que possibilita que uma única instrução possua várias formas. Para ilustrar é apresentado um esquema abaixo onde a função base e suas derivadas preocupam-se em atribuir diversas funcionalidades para a impressão de textos na tela utilizando a função *imprimir*.



A seguir é apresentado um programa simples para demontrar o esquema acima. Na linguagem C as funções têm sempre identificadores diferentes ao contrário do C++, em que as funções podem ter identificadores iguais, desde que os parâmetros sejam diferentes (em número ou tipo), ou seja, não podem existir ambiguidades na chamada às funções.

```
#include <iostream.h>
class base
{
        public:
                virtual void print( ) {cout << "\nBase";}
```

Especificador para o acesso as classes derivadas

```
};
class deriv0 : public base
{
        public:
                void print() {cout << "\nDeriv0";}
};
class deriv1 : public base
{
        public:
                void print() {cout << "\nDeriv1";}
};
class deriv2 : public base
{
        public:
                void print() {cout << "\nDeriv2" ; }
};
void main()
{
        base *p[3]; // Matriz de ponteiros para a classe base
        deriv0 dv0; // Objeto da classe derivo
        deriv1 dv1; // Objeto da classe deriv1
        deriv2 dv2; // Objeto da classe deriv2
        p[0]=&dv0; // Preenche a matriz
        p[1]=&dv1; // com endereço dos objetos
        p[2]=&dv2; // das classes derivadas
        for(int i=0; i<3; i++)
            p[i]->print(); // chama print( )


        int i;
        cin >> i;
}
```

No exemplo acima é destacado a palavra *virtual* da função *print()* da classe base, esta tem a finalidade de permitir que sejam acessadas as funções membros polimórficas das classes derivadas, dependendo da necessidade. O resultado a ser impresso sem o termo reservado *virtual* o resultado seria:

base

base

base

Porém, quando é inserida a *virtual* o resultado retornado é:

Deriv0

Deriv1

Deriv2

Compreendendo que o polimorfismo são as diferentes resoluções de um mesmo problema através de funções com o mesmo nome, porém com diferentes atributos

ou parâmetros. Procurando esclarecer melhor essa confusão que pode ter sido criada, abaixo segue um outro exemplo sem utilizar classes, porém mais fácil de compreender o polimorfismo.

```
#include <iostream.h>
  int func(int i)
  {
     i++;
     return i;
  }
  float func(float f)
  {
     f++;
     return f;
  }
  void main()
  {
     int k = 7;
     float w = 3.56;
     cout <<"f(k) = " << func(k);      // chama a função func(int)
     cout << "\nf(w) = "<<func(w); // chama a função func(float)
     cin >>k;
  }
```

9.2 Funções Virtuais

O mecanismo de função virtual permite que classes derivadas substituam a implementação realizada na classe base. O compilador certifica-se que a substituição seja sempre invocada quando o objeto em questão é realmente da classe derivada, seja esse objeto acessado por um pointeiro para classe base, ao invés de por um ponteiro para a classe derivada.

Isso permite que algoritmos na classe base sejam substituídos por outros nas classes derivadas, mesmo que as classes clientes não conheçam as classes derivadas que contém as novas versões do algoritmo.

A classe derivada pode tanto substituir totalmente (sobrescrever) a função membro da classe base, quanto substituí-la parcialmente (ampliar). A substituição parcial é realizada quando a função membro na classe derivada invoca a função membro na classe base.

Uma hierarquia é mostrada a seguir, usando uma classe base denominada `List` e uma classe derivada denominada `TotalingList`:

```
#include <iostream.h>

class List
{
        int array[100];
        int count;
public:
        List(): count(0) {}
        void Insert(int n) { array[count++]=n; }
        int Get(int i) { return array[i]; }
        int Size() { return count; }
};

void ManipList(List list)
{
        // do things to the list
        list.Insert(100);
        list.Insert(200);
        // do things to the list
}

class TotalingList: public List
{
        int total;
public:
        TotalingList(): List(), total(0) {}
        void Insert(int n)
        {
                total += n;
                List::Insert(n);
        }
        int GetTotal() { return total; }
};

void main()
{
        TotalingList list;
        int x;

        list.Insert(10);
        list.Insert(5);
        cout << list.GetTotal() << endl;
        ManipList(list);
        cout << list.GetTotal() << endl;
        for (x=0; x < list.Size(); x++)
                cout << list.Get(x) << ' ';
        cout << endl;
}
```

Nesse código, a classe *List* implementa a lista mais simples possível com as três funções membro *Insert*, *Get* e *Size* e o construtor. A função *ManipList* é um exemplo de uma função

qualquer que usa a classe *List*, e que chama a função *Insert* duas vezes apenas como ilustração do que pretendemos demonstrar.

A classe *TotalingList* herda a classe *List* e acrescenta um dado membro denominado *total*. Esse dado membro armazena o total corrente de todos os números contidos na lista. A função *Insert* é sobreposta para que *total* seja atualizado a cada inserção.

A função *main* declara uma instância da classe *TotalingList*. Insere os valores *10* e *5* e imprime o total. Em seguida chama *ManipList*. Pode ser uma surpresa para você que essa construção não resulte em erro de compilação: se você olhar o protótipo de *ManipList* vai ver que ela espera uma parâmetro do tipo *List*, e não do tipo *TotalingList*. Acontece que C++ entende certas peculiaridades sobre herança, uma delas é que um parâmetro do tipo classe base deve aceitar também qualquer tipo classe derivada daquela classe base. Portanto, desde que *TotalingList* seja derivada da classe *List*, *ManipList* vai aceitar um parâmetro do tipo *TotalingList*. Esse é um dos recursos do C++ que torna o mecanismo de herança tão poderoso: é possível criar classes derivadas e passá-las para funções existentes que conhecem apenas a classe base.

Quando o código mostrado for executado, ele não vai produzir o resultado correto. Ele vai produzir a seguinte saída:

```
15
15
10 5
```

Essa saída indica que não apenas a totalização não funcionou, mas que os valores *100* e *200* não foram inseridos na lista quando da chamada a *ManipList*. Uma parte desse erro se deve a um erro no código: o parâmetro aceito por *ManipList* deve ser um pointer ou uma referência, do contrário nenhum valor será retornado. Modificando o protótipo de *ManipList* para corrigir parcialmente o problema:

```
void ManipList(List& list)
```

Agora teremos a seguinte saída:

15
15
10 5 100 200

É didático seguir a execução de *ManipList* passo-a-passo e observar o que acontece. Quando ocorre a chamada para a função *Insert*, a função invocada é *List:Insert* ao invés de *TotalingList:Insert*.

Esse problema também pode ser resolvido. É possível, em C++, criar uma função com um prefixo *virtual*, *e isso leva o C++ a sempre chamar a função na classe derivada*. Ou seja, quando uma função é declarada como *virtual*, o compilador pode chamar versões da função que sequer existiam quando o código da função chamada foi escrito. Para verificar isso, é acrescentada a palavra *virtual* a frente das funções *Insert* tanto na classe *List* quanto na classe *TotalingList*, como mostrado a seguir:

```
class List
{
	int array[100];
	int count;
public:
	List(): count(0) {}
	virtual void Insert(int n) { array[count++]=n; }
	int Get(int i) { return array[i]; }
	int Size() { return count; }
};

void ManipList(List&  list)
{
	list.Insert(100);
	list.Insert(200);
}

class TotalingList: public List
{
	int total;
public:
	TotalingList(): List(), total(0) {}
	virtual void Insert(int n)
	{
		total += n;
		List::Insert(n);
	}
```

```
        int GetTotal() { return total; }
};
```

De fato é necessário colocar a palavra *virtual* apenas a frente do nome da função na classe base, mas é um bom hábito repetí-la nas funções das classes derivadas, como uma indicação explícita do que está ocorrendo.

Agora você pode executar o programa e obter as saídas corretas:

15
315
10 5 100 200

O que está acontecendo? A palavra *virtual* a frente da função determina para o C++ que você planeja criar novas versões dessa mesma função em classes derivadas. Ou seja, *virtual* permite que sejam declaradas suas futuras intenções em relação à classe em definição. Quando uma função *virtual* é chamada, o C++ examina a classe que chamou a função, e busca a versão da função para esta classe, mesmo que a classe derivada ainda não existisse quando a chamada da função foi escrita.

Isso significa que em certos casos é necessário que pense no futuro quando escreve o código. Devem ocorrer reflexões do tipo "*eu, ou qualquer outro programador, poderiamos no futuro precisar modificar o comportamento dessa função?"* Se a resposta for sim, então a função deve ser declarada como virtual.

Deve-se ter alguns cuidados para que uma função *virtual* funcione corretamente. Por exemplo, a previsão da necessidade de se modificar o comportamento da função e lembrar-se de declará-la como *virtual* na classe base. Um outro ponto importante pode ser visto no exemplo acima: experimentar remover o & da lista de parâmetros em *ManipList* e siga o código passo-a-passo. Mesmo a função *Insert* estando marcada como *virtual* na classe base, a função *List::Insert* é chamada, ao invés da função *TotalingList::Insert*.

Isso acontece porque, quando o *&* não está presente, a lista de tipos de parâmetros em List está atuando como se fosse uma definição de conversão de tipos. Qualquer classe passada é convertida para a classe base *List*. Quando o *&* está presente na lista de parâmetros, tal conversão não acontece.

Pode-se ver funções virtuais em praticamente toda hierarquia de classes em C++. Uma

hierarquia de classes típica espera que, no futuro, se altere o comportamento das funções para adaptar a biblioteca de classes às necessidades específicas de uma ou de outra aplicação.

Funções virtuais são usadas com freqüência quando o projetista da classe não pode realmente saber o que será feito com a classe no futuro. Suponha que você está usando a interface de uma classe que implementa botões na tela. Quando uma instância do botão é criada, ele se desenha na tela e se comporta de modo padrão, ou seja, ilumina-se quando é pressionado pelo usuário. Entretanto, o programador que projetou essa classe não tinha idéia do que os programadores usuários da classe poderiam querer fazer quando o botão fosse pressionado. Para esses casos, o projetista da classe poderia ter definido uma função virtual denominada, por exemplo, *handleEvent* que é chamada sempre que o botão é clicado. Feito isso, é possível sobrescrever a função virtual com uma função própria que maneja a situação do modo mais adequado à sua aplicação.

Capítulo 10

Manuseamento de Arquivos

O trabalho com variáveis comuns e vetores são armazenados temporariamente no computador. Existem casos em que são necessárias armazenar os dados por tempo indeterminado, variando de acordo com as necessidades dos usuários, e não com liga/desliga dos computadores. Esse processo de reiniciar o computador recria a alocação de endereços da memória temporária automaticamente, independente de ações do sistema operacional.

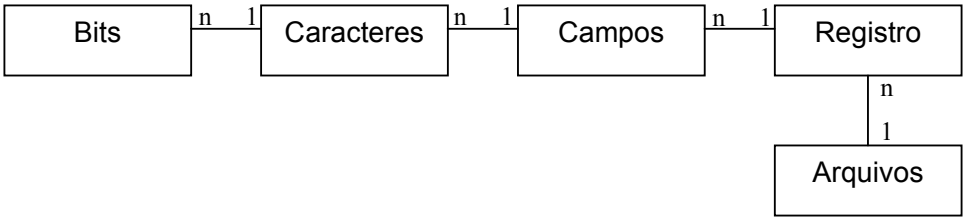
Para contribuir com essas necessidades existe a possibilidade de criação de arquivos para a armazenagem dos dados por tempo indeterminado. A seguir são apresentadas rotinas e comandos que possibilitam o manuseamento de arquivos utilizando os recursos do C++.

10.1 Abrindo Arquivos

A linguagem C++, assim como a maioria das outras linguagens de programação, compreendem os arquivos como sendo seqüências de bytes. Cada arquivo possui um delimitador ou um byte específico para indicar o seu final, sendo no caso byte o seu número equivalente é armazenado em estrutura administrativa dos dados mantidos pelo sistema.

Quando um arquivo é aberto um objeto é criado e um *stream* (seqüência de bytes) é associado a ele. A criação de arquivos pode ser realizada de diversas maneiras, porém a orientada a objetos deve ter os cabeçalhos <*iostream*> e <*fstream*> para a abertura, manuseamento e fechamento dos arquivos.

Esses cabeçalhos organizam os dados seguindo os conceitos de que um arquivo é um conjunto de registros. Os registros são conjuntos de campos, que são conjuntos de caracteres, que por sua vez, são conjuntos de bits, como segue o esquema abaixo.



A seguir é apresentado um algoritmo tomado do livro do Deitel & Deitel [2004], com comentários sobre as linhas referentes aos arquivos. Esse algoritmo cria um arquivo seqüencial.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::endl;

using std::ofstream;
int main()
{
   int indice;
   char produto[ 30 ];
   double balance;

   ofstream outClientFile( "produtos.txt", ios::out );
   if ( !outClientFile )
   {
      cerr << "Erro: Arquivo não pode ser aberto/criado" << endl;
      exit( 1 );
   }
   cout << "Insira o Índice e o nome do Produto.\n"
        << "Tecle <ENTER> para confirmar o cadastro do produto.\n";
   while ( cin >> indice >> produto) {
      outClientFile << indice << ' ' << produto << '\n';
      cout << "? ";
   }
   return 0;  // Aciona os destrutores dos objetos.
}
```

Cabeçalho para tratamento de Arquivos

Conversões de números em textos e vice-versa

Originário da biblioteca fstream

Criando objeto do arquivo que está sendo aberto

Checando se o arquivo foi aberto/criado

Inserindo os dados no arquivo

A abertura/criação de arquivos pode ser realizando seguindo as propriedades descritas na tabela abaixo.

Modo	Descrição
ios::app	Grava toda a saída no fim do arquivo
ios::ate	Abre um arquivo para saída e avança até o dele (normalmente utilizado para acrescentar dados a um arquivo)
ios::in	Abre um arquivo para entrada
ios::out	Abre um arquivo para saída
ios::trunc	Elimina o conteúdo do arquivo se ele existe
ios::binary	Abre um arquivo para entrada ou saída binária (não texto)

Tabela de tipos de manuseamento de arquivo [Deitel e Deitel 2004]

10.2 **Manuseamento de dados de um arquivo**

O manuseamento de arquivos pode ocorrer de maneira seqüencial, aleatória ou a combinação dos tipos. Essas disponibilidades resultam em muitos recursos que possibilitam diferentes tipos de trabalhos possíveis de serem realizados. Esse procedimento é complexo, e será detalhado na sala de aula para não causar “vício” de programação.

Referências Bibliográficas

[DEITEL & DEITEL 2004] Deitel, H. M. Deitel, P. J. **C++ Como Programar.** Bookman, 2004

[HOLZNER 2002] Holzner, S. **C++ Black Book – Guia Completo para Resolução de Problemas.** Makron Books, 2002

[JOHANN 2004] Johann, M. O. **Curso de Introdução à Programação em C++.** 2004,

[LEE & TEPFENHART 2002].Lee, R. C. Tepfenhart, William M. **UML e C++ Guia Prático de Desenvolvimento Orientado a Objeto**, 2002

[STR 2004] Stroustrup, B. **The C++ Programming Language**. 2004. Disponível em: "http://www.research.att.com/~bs/C++.html" (20/08/2004)

[UFSCAR-OOP] **Orientação a Objetos.**

[BSD-C++] http://www.bloodshed.net/devcpp.html (acessado: 27/09/2004)

[RUBIRA-C++] http://www.ic.unicamp.br/~cmrubira/aacesta/cpp/cpp15.html (acessado: 21/09/2004)

[CCTMA-C++] http://www.cttmar.univali.br/~ademir/progs_ex/clas13.txt (acessado:24/09/2004)