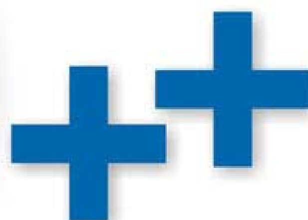
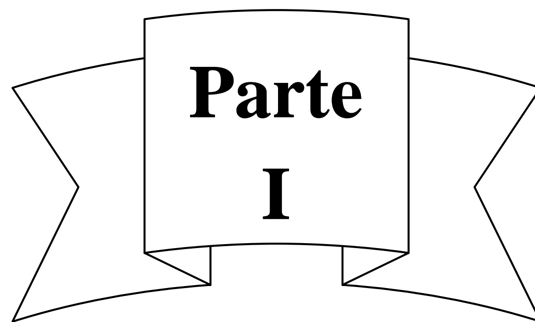
Alfredo Boente

APRENDENDO A PROGRAMAR EM



Usando Classes e Objetos

Noções de UML
Visão da linguagem C
Visão de C#
Visão de Java
Tabela ASCII



Conhecendo o C++ Orientado a Objetos

1

CONCEITOS BÁSICOS DA LINGUAGEM C++

Antes mesmo de falarmos na linguagem de programação C++, gostaria de apresentar-lhes o Ronald, um "*galinho*" bem simpático que tem por objetivo chamar sua atenção para os momentos de maior importância deste livro.

> *Olá*!
>
> Meu nome é ***Ronald***. Espero que vocês gostem desse livro assim como eu.

A linguagem de programação C++ é uma linguagem de programação que apresenta um foco na visão orientada a objetos. Para saber mais como tudo verdadeiramente começou vamos fazer uma leitura necessária do histórico apresentado a seguir:

Muitas linguagens de programação surgiram na década de 60 e 70, como por exemplo, ALGOL, CPL, BCPL, B, BASIC, C, Pascal, dentre outras. Nestas duas décadas, a linguagem ALGOL teve duas versões: ALGOL 60 (1960) e ALGOL 68 (1968). A ALGOL 60 obteve da FORTRAN os conceitos da programação estruturada. É interessante, de um ponto de vista histórico, salientar que FORTRAN não foi a primeira linguagem de programação, como afirma a maioria dos historiadores de informática. Na verdade, a linguagem ADA foi a precursora de todas elas. Só a título de curiosidade, esse nome foi dado em homenagem a uma matemática inglesa, famosa na sua época, chamada Ada Augusta. Voltando ao ALGOL, era uma linguagem cuja abstração tornava-a pouco prática para tarefas habituais de programação da época.

Em 1963 nasceu a CPL com a intenção de ser mais específica para as tarefas habituais de programação da época e foi esta mesma especificidade que dificultou o aprendizado e a implementação.

Em 1967, Martin Richards desenvolveu a BCPL que era, na verdade, uma simplificação da CPL no sentido de tomar da mesma as melhores coisas e num escopo mais básico. Mesmo assim, a BCPL continuou com característica muito abstratas que contribuiu para sua pouca adaptabilidade às máquinas reais existents naquela época.

Em 1970, Ken Thompson criou a linguagem B, que era a porta de entrada da BCPL para o computador DEC PDP-7 sob o sistema operacional Unix da Bell Labs. A linguagem B tinha suas limitações como, por exemplo, gerava um código de execução lenta e, assim, inadequada para o desenvolvimento de um sistema operacional, que por parâmetros lógicos, deveria ter uma execução bem mais rápida.

Em 1971, Denis Ritchie, também da Bell Labs, começou o desenvolvimento de um compilador B que era capaz de gerar código executável, suprindo assim, uma característica que a linguagem B original não possuia e que era o foco da lentidão dos seus executáveis. Esta "nova B" foi renomeada então, e passou a ser conhecida como linguagem C. A linguagem C adicionou alguns novos conceitos como, por exemplo, os tipos de dados, biblioteca padrão e etc.

Em 1973, Denis Ritchie já tinha desenvolvido as bases da linguagem C, com inclusão de tipos e seus manipuladores, arranjos e ponteiros e com uma

capacidade de portabilidade que a tornou uma linguagem de alto nível. Neste mesmo ano foi editado o livro clássico da linguagem C, posso até afirmar que se fosse um pouco mais grosso, seria chamado da bíblia da Linguagem C, "The C Programming Language" de Brian Kernigham e Denis Ritchie que ditou as normas de mercado para a linguagem C até a publicação formal das normas ANSI (American National Standards Institute) em 1989. Tentando ser bem direto e objetivo, faço uma exploração do conteúdo da linguagem de programação C, de forma fácil e eficiente, no meu livro "*Aprendendo a Programar em Linguagem C: do básico ao avançado*", também publicado pela editora Brasport. Vale apena conferir.

Em 1980, Bjarne Stroustrup, também da Bell Labs, iniciou o desenvolvimento de uma nova linguagem com bases na linguagem C. Esta incrementava, em relação à linguagem C, conceitos de orientação a objetos com o uso de classes. Em 1983 recebeu o nome formal de C++ (++ é chamado de pós-incremento na linguagem C), quando da publicação do seu primeiro manual.

Em 1985 nasceu a primeira versão comercial da C++ e a primeira edição do livro "The C++ Programming Language" de Bjarne Stroustrup, seguindo a mesma visão adotada no livro da linguagem C. É importante mensionar que C++ não tem nada haver com C avançado. Logo, trata-se de uma linguagem de programação totalmente independente da linguagem de programação C.

Em 1989, o ANSI publicou as normas para a linguagem C. Já as normas para a linguagem C++, só foram publicadas em 1997. O C++ serviu de base para o desenvolvimento de outras linguagens de programação orientadas a objetos, como é o caso da linguagem Java, abordada por mim com mais detelhadamente no meu livro "*Aprendendo a Programar em Java2: orientado a objetos*", também publicado pela editora Brasport.

Já no século XXI, escuta-se falar muito sobre linguagem de programação baseada em componentes. De acordo com a estruturação da linguagem C++, surgiu o C#, uma linguagem baseada em componentes, dita tecnicamente como uma linguagem .NET.

IMPORTANTE:

O C++ é uma linguagem de programação baseada em classes e orientada a objetos, pois ela permite a reutilização de objetos criados através de heranças ou instâncias das classes, que contêm o código fonte em questão, em qualquer outro programa de mesma natureza.

A ESTRUTURA BÁSICA DE UM PROGRAMA C++

A linguagem de programação C++ trabalha basicamente com funções pré-definidas que, a partir delas, o programador pode produzir muitas outras. Assim como a linguagem C, o C++ apresenta a função main( ) como a função principal de todos os programas escritos. Observe atentamente o nosso primeiro programa abaixo:

```
#include <iostream.h>
void main( )
{
   cout << "Primeiro Programa";
}
```

Note que utilizamos a cláusula de pré-processamento #include cujo objetivo é permitir ao programador C++ atribuir funções de uma biblioteca pré-definida (funciona da mesma forma que a linguagem de programação C). Neste caso a biblioteca utilizada foi a <iostream.h>, específica para ações básicas de entrada e saída.

Antecedendo a função principal main( ), foi utilizada a palavra reservada void que indica que uma função não retornará valor algum. Assim como na linguagem C, caso seja de interesse do programador a função principal do programa pode retornar um valor de qualquer tipo. Por exemplo: int main( ) e float main( ).

O início e fim da função é determinado pelo { e }, assim como na linguagem de programação C. Na verdade, eles apresentam a mesma funcionalidade do Begin e End da linguagem Pascal.

Quando utilizamos a instrução cout << “Primeiro Programa”, o compilador C++ interpreta que tudo aquilo que vem entre aspas aparecerá na tela do seu computador. Na verdade, funciona como um comando de saída de informações.

Veja que a instrução tem um ponto-e-vírgula (;) no seu final para indicar que a mesma foi concluída. Fique atento que o ponto-e-vírgula serve como separador de instruções da linguagem C++. Logo, se você assim desejar, pode utilizar dois ou mais instruções escritas na mesma linha de comando. No próximo capítulo voltaremos a falar instrução cout.

IMPORTANTE:

Todo programa que é escrito e gravado através do compilador C++ originalmente recebe a extensão CPP. Na verdade, CPP é a forma utilizada para representar C++ em inglês pois, CPP são as letras iniciais de ***C Plus Plus***, ou seja, C++.

Exercícios

1- Quem foi o criador da Linguagem C++?
2- Explique a origem da Linguagem C++.
3- Quem foi o autor do livro clássico The C++ Programming Language?
4- Qual a extensão utilizada para um programa C++?
5- Para que serve a biblioteca iostream.h?

6- Por que devemos utilizar a palavra-reservada void na frente da função principal main( )?

7- O que significa CPP, utilizada como extensão de um programa escrito em C++?

8- O que acontece se usarmos a função main( ) sem o especificador void?

9- Escreva um programa em C++ para imprimir o nome do seu time de futebol predileto.

10- Defina suscintamente biblioteca padrão.

2

SAÍDA DE DADOS

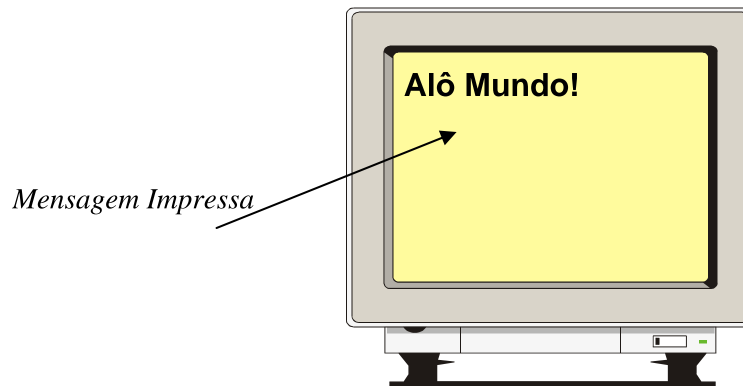
Quando falamos em saída de dados, queremos dizer realmente, saída de informações, pois, segundo os conceitos básicos de processamento de dados, tudo aquilo que é inserido no computador, através de um INPUT, é dito dado e, conseqüentemente, tudo aquilo que sai, é dito informação.

A INSTRUÇÃO COUT

Observe um exemplo em C++ do clássico programa Alô Mundo usando **cout**:

```
/* Representa uma linha de comentário */
#include <iostream.h>

/* Início do corpo principal do programa */
void main (   )
{

/* Instrução utilizada para saída de dados */
   cout << "Alô Mundo!";
}
```

Aparecerá então na tela do seu computador a mensagem “Alô Mundo!”, como você pode observar a seguir:



A instrução cout expressa C++ Out, ou seja, saída de C++. Nela precisamos utilizar o símbolo << chamado de operador de inserção para conectar a mensagem a ser impressa à instrução cout.

No nosso programa exemplo utilizamos, também, os símbolos /* e */ que, assim como na linguagem de programação C, são utilizados para inserir uma linha de comentário ao seu programa.

OUTROS PROGRAMAS COM COUT

```
#include "iostream.h"
void main (  )
{
   cout << "Tenho” << 35 << “anos de idade";
}
```

Note que no programa exemplo a cima utilizamos a constante numérica 35 para determinar a idade de uma certa pessoa. Como não faz parte integrante da mensagem (String) a ser exibida, teve que aparecer separadamente da mensagem.

```
#include <iostream.h>
void main (   )
{  cout << "A letra " << 'j' << " pronuncia-se ", "jota";  }
```

Observe atentamente que ao referenciarmos a letra jota, a colocamos entre plicas (aspas simples) e o restante das mensagens, entre aspas duplas. Isso porque quando referenciamos uma única letra, não usamos aspas duplas, como é o caso das strings.

CARACTERES ESPECIAIS

A tabela a seguir mostra os códigos da linguagem C++ para caracteres que não podem ser inseridos diretamente pelo teclado.

CÓDIGOS ESPECIAIS	SIGNIFICADO
\n	NOVA LINHA
\t	TAB
\b	RETROCESSO
\"	ASPAS
****	BARRA INVERSA
\f	SALTA PÁGINA DE FORMULÁRIO
\0	NULO
\x	MOSTRA CARACTER HEXADECIMAL
\a	Beep do auto-falante
\r	Leva o cursor para o início da linha

Verdadeiramente, conseguimos observar que os caracteres especiais existem com o objetivo de escrever caracteres que não possam ser descritos de forma direta por estarem servindo como parâmetros chave para a programação em C++. Veja o programa abaixo que apresenta o uso de caracteres especiais da linguagem C++:

```
#include “iostream.h”


void main(   )
{
   cout << “São Pedro da Aldeia está a aproximadamente “ <<
            200 << “ km do” << ‘\n’ << “centro do Rio de Janeiro” << “\n”;
}
```

Neste programa exemplo utilizamos o caracter especial ‘\n’ que expressa uma nova linha (new line).

Exercícios

1- Qual a instrução utilizada para saída de dados?
2- Qual a diferença do uso das aspas simples e aspas duplas?
3- Para que servem os caracteres especiais?
4- Como podemos escrever uma linha de comentário em C++?
5- O que representa o símbolo << na instrução cout?
6- O que acontece ao usarmos a seguinte linha de comandos cout << “\n”?

3

TIPOS DE DADOS

Tudo que processamos, na verdade representa as informações que teremos acesso através do computador. Estas informações são caracterizadas por dados numéricos, caracteres ou lógicos.

Os chamados dados numéricos, podem ser inteiros, número positivo ou negativo, sem o uso de casas decimais ou reais, número positivo ou negativo, com o uso de casas decimais. Como exemplo tem-se: 56, 0, -8, entre outros.

Os dados caracteres, podem ser representado por um único caracter ou por um conjunto de caracteres, o qual nomeamos de string. Como exemplo tem-se: “GIULLIA”, “Juan Gabriel”, “Rua Alfa, nº 24”, “171”, “C”, entre outros.

Os dados que são conhecidos como lógicos são também chamados de dados boleano por apresentarem apenas dois valores possíveis:

Verdadeiro (true) ou **Falso** (false).

VARIÁVEIS

Uma variável nada mais é do que um endereço de memória que é reservado para armazenar dados do seu programa. Procure utilizar, sempre que possível, as chamadas variáveis significativas, pois seu nome significa, na íntegra, o que elas armazenam (referem-se).

Exemplos:

nome, idade, altura, endereço, data_nasc, salário, cargo etc.

TIPOS DE VARIÁVEIS EM C++

Com exceção do tipo ***void***, os demais tipos podem vir acompanhados por modificadores do tipo ***short***, ***long*** e ***unsigned***, no momento de sua declaração.

TIPO	BIT	BYTES	ESCALA
char	8	1	-128 a 127
int	16	2	-32768 a 32767
float	32	4	3.4e-38 a 3.4e+38
double	64	8	1.7e-308 a 1.7e+308
void	0	0	sem valor

Cada tipo de dado é associado a uma determinada variável a fim de suprir a necessidade real do programa de computador a ser desenvolvido. Logo, você deve estar bastante atento a este simples porém valioso detalhe.

Ao usar os modificadores de variáveis, a escala de valores sofre uma significante alteração.

Observe a tabela abaixo:

TIPO	BIT	BYTES	ESCALA
unsigned char	8	1	0 a 255
unsigned	16	2	0 a 65535
short	16	2	-32768 a 32767
long	32	4	-2147483648 a 2147483647
unsigned long	32	4	0 a 4294967295
Long double	80	10	3.4E-4932 a 1.1E+4932

VARIÁVEL INTEIRA

No exemplo a seguir, iremos demonstrar como usar variáveis do tipo INTEIRA. Um pouco mais tarde teremos acesso a outros tipos de variáveis que serão declaradas como externas.

```
/* Exemplo Prático */
#include <iostream.h>

void main (   )
{
    int num;
    num = 2;
    cout << "Este é o numero dois: " <<  num;
}
```

Outro exemplo:

```
#include <iostream.h>

void main (   )
{
    int num1,num2=4;
    num1 = num2;
    cout << num1 << '\n' << num2;
}
```

Em ambos os exemplos anteriores utilizamos o símbolo = que para a linguagem C++ representa o símbolo de atribuição de valores, assim como na linguagem C. Em C++, o símbolo = = representada igualdade e != representa diferença. Tais símbolos, serão mostrados mais tarde no capítulo que fala de operadores.

Quando estivermos nomeando variáveis para nossos programas, devemos tomar bastante cuidado com os nomes criados para que não venhamos a usar as chamadas **PALAVRAS RESERVADAS** da linguagem C++.

Observe algumas dessas palavras reservadas:

asm	_far	public
auto	far	register
break	float	return
case	for	_saveregs
catch	friend	_seg

_cdecl	goto	short
cdecl	huge	signed
char	if	sizeof
class	inline	_ss
const	int	static
continue	interrupt	struct
_cs	_loadds	switch
default	long	template
do	_near	this
double	near	typedef
_ds	new	union
else	operator	unsigned
enum	_pascal	virtual
_es	pascal	void
_export	private	volatile
extern	protected	while

VARIÁVEL REAL

As variáveis reais são aquelas que apresentam o uso de casas decimais (valores fracionários). Em C++, podemos utilizar duas categorias de variáveis reais: as variáveis reais de simples precisão ou precisão simples, e as variáveis reais de dupla precisão ou precisão dupla.

REAL DE PRECISÃO SIMPLES

No C++, utiliza-se float para representar uma variável do tipo REAL de precisão simples (ocupa 4 bytes).

```
/* Real de precisão simples */
#include <iostream.h>

void main (   )
{
   float n1, n2;
   n1=6;
   n2=5.5;
   cout << "A soma de " << n1 << " com " << n2 << " é igual a "
         << n1+n2;
}
```

Aqui, utilizamos o símbolo de soma (+) para realizar uma operação aritmética de soma. Mais tarde iremos realizar um estudo mais aprofundado sobre o assunto.

REAL DE PRECISÃO DUPLA

Caso você queira usar uma variável real de precisão dupla, ou seja, aquela que ocupa 8 bytes ao invés de 4, utiliza-se em C++ a palavra reservada double no lugar de float.

```
/* Real de precisão dupla */
#include "iostream.h"
```

```
void main(   )
{
     double result;
     int num;

    num = 59;
    result = 3.1415 * num;

    cout << "O resultado é " << result;
}
```

VARIÁVEL CARACTER

No exemplo a seguir, será usada a palavra reservada char para representar o tipo de variável caracter (ocupa 1 byte, conforme foi mostrado anteriormente).

```
/* Exemplo Prático */
#include "iostream.h"

void main (   )
{
     char letra='a';

     cout << letra << " é a primeira letra do alfabeto" << '\n';
}
```

Já falamos nisso anteriomente mas, é importante lembrar que o símbolo de igualdade (=) para atribuição de valores do tipo caracter.

VARIÁVEL CADEIA DE CARACTERES

É importante saber que no C++, assim como na linguagem C, não existe o tipo de variável STRING, encontrado, por exemplo, na linguagem de programação Pascal. Caso você queira representar uma cadeia de caracteres (STRING), que ocupa n bytes na memória, use o seguinte formato:

char <nome da variável>[<tamanho>];

```
char nomedoaluno[35];
```

Mais a diante iremos trabalhar com esse tipo de variável.

IMPORTANTE:

É verdadeiro afirmar que uma variável "STRING" é visto como um vetor de caracteres, pela linguagem C++.

VARIÁVEIS LOCAIS x VARIÁVEIS GLOBAIS

Uma variável local é aquela declarada dentro do corpo de uma certa função e somente pode ser utilizada por aquela função e nenhuma outra mais. Já uma variável global, que poderá ser utilizada por todas as funções existentes em seu programa, é declarada fora, antes, do início do corpo da função principal do programa, main( ).

Num programa, podem ser apresentadas tanto variáveis locais quanto variáveis globais. Mais tarde iremos fazer uso desses dois tipos distintos de declaração de variáveis.

CONSTANTES

Uma constante representa um valor fixo, constante, durante todo o processamento de um certo programa. Em C++, existem duas formas de declaração de uma constante. A primeira delas é através do uso da cláusula de pré-processamento #define. A outra é através da palavra reservada const. Observe abaixo as diferentes formas de declaração:

```
/* Usando constantes com #define */
#include <iostream.h>
#include <conio.h>

/* Definição da constante DOIS com o valor numérico 2 */
#define DOIS 2

void main( )
{
        float num1=5, num2=3, result;

        result = (num1 + num2) / DOIS;

        clrscr( );
        gotoxy(10,10);
        cout << "Resultado = " << result;
}
```

Aqui foi criada uma constante chamada DOIS para representar o numeral 2. Na verdade, a criação de constantes não é feita para ser usada de forma ao acaso pois, tudo aquilo que você cria dentro do seu programa, a final de contas, ocupa espaço físico na memória do computador.

Ainda, utilizamos a biblioteca conio.h que permite a inserção de instruções de entrada e saída ligadas ao console. Por meio dessa biblioteca, neste programa exemplo, utilizamos duas instruções a serem consideradas. A primeira clrscr( ) serve para limpar a tela do computador. A segunda, gotoxy(COL, LIN) determina em que ponto da tela, dado por um número de coluna e linha, o cursor será posicionado.

Outro Exemplo:

```
#include <iostream.h>

void main( )
{
    const float PI = 3.1415;

    cout << "O valor de PI é " << PI;
}
```

No exemplo anterior usamos a palavra reservada const para especificar a criação de uma constante. Funciona da mesma forma que uma constante declarada pela cláusula de pré-processamento #define.

Nos programas citados anteriormente não citamos nenhuma constante do tipo literal. Para termos essa visão acompanhe atentamente as linhas de código do programa apresentado a seguir:

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

#define EU "Kilmer Boente"

void main( )
{
```

```
    clrscr( );
    cout << "Comunico a quem interessar que os laboratórios de informática"
           << '\n' << "ncontram-se em manutenção." << '\n'
     cout << " Assinado: " << EU;
}
```

Neste exemplo, foi definido EU como constante cujo valor atribuído é "Kilmer Boente". Logo, toda vez que no programa for referenciada a constante EU, na verdade, será referenciado como conteúdo "Kilmer Boente".

MANIPULADORES DE IMPRESSÃO

A instrução **cout** permite definir o tamanho de um determinado campo para o processo de impressão. Na maioria das vezes, essa definição permite uma melhor estética nos relatórios de dados, por exemplo. Estes manipuladores são objetos que estão embutidos na biblioteca <**iomanip.h**>, como podemos confirmar a seguir:

- ***setfill*** - Seleciona o caracter que será utilizado para preenchimento das colunas do campo que estiverem em branco;
- ***setiosflags*** – Permite definir o tipo de dado a ser manipulado (ponto decimal, notação científica etc.);
- ***setprecision*** - Permite definir o número de casas decimais para impressão de números reais;
- ***setw*** - Seleciona o tamanho de impressão do próprio campo a ser impresso;

Vejamos então alguns programas exemplos com o uso de manipuladores de impressão:

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

```
void main( )
{
   int qtd1, qtd2, qtd3, qtd4;

    qtd1=98;
    qtd2=178;
    qtd3=1028;
    qtd4=214;

    cout << “\n\n\n”;
    cout << ‘\n’ << “Bananas” << setw(12) << qtd1;
    cout << ‘\n’ << “Uvas     ” << setw(12) << qtd2;
    cout << ‘\n’ << “Pêras    ” << setw(12) << qtd3;
    cout << ‘\n’ << “Figos    ” << setw(12) << qtd4;
}
```

Neste programa utilizamos os manipuladores de tamanho de campos para variáveis do tipo inteira. Naturalmente este recurso também pode ser utilizado com números do tipo real, conforme podemos verificar a seguir:

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

void main( )
{

   float val1, val2, val3, val4, val5;

    val1=1.50;
```

```
val2=3.55;
val3=2.20;
val4=2.85;
val5=0.99;

clrscr( );

cout << "\n\n\n";
cout << '\n' << "Bananas " << setw(12) << val1;
cout << '\n' << "Uvas     " << setw(12) << val2;
cout << '\n' << "Pêras    " << setw(12) << val3;
cout << '\n' << "Figos    " << setw(12) << val4;
cout << '\n' << "Laranjas" << setw(12) << val5;

getch( );
}
```

Exercícios

1- Para que utilizamos uma variável?
2- Cite três nomes de variáveis válidas.
3- Diferencie Variável Local e Variável Global.
4- Como declarar uma variável do tipo real de precisão dupla?
5- Quais as formas de declaração de uma constante num programa em C++?
6- Quais são os tipos primitivos de variáveis?
7- Diferencie variável real de precisão simples e real de precisão dupla.
8- Uma variável String em C++ é interpretada por um conjunto de caracteres. Verdadeiro ou Falso?

9- Quais os respectivos tamanhos em bytes da variável real de precisão simples e da variável real de precisão dupla?

10- Qual a função da biblioteca conio.h?

11- O que faz a instrução clrscr( )?

12- Explique com suas palavras a linha de comando gotoxy(25, 6).

13- Qual a função dos modificadores de variáveis?

14- O que são os manipuladores de impressão?

15- Onde estão armazenados os manipuladores de impressão?

4

ENTRADA DE DADOS

Em C++ a entrada de dados é efetuada através da instrução **cin**. Ela expressa C++ In, ou seja, entrada de C++. Nela precisamos utilizar o símbolo >> chamado de operador de extração para conectar o valor que será recebido e armazenado na variável especificada.

A INSTRUÇÃO CIN

É uma instrução própria para entrada de dados na linguagem C++. Embora não seja a única, é a mais utilizada. Observe abaixo, o programa exemplo com o uso desta instrução:

```
#include "iostream.h"

void main (  )
{

   int a, b, soma;

   cout << "Entre com dois numeros inteiros:";
```

```
    cin >> a;
    cin >> b;
    soma = a + b;
    cout << "Soma = " << soma;
}
```

Neste nosso exemplo, tivemos que utilizar os operadores de atribuição (=) e soma (+) devido o objetivo do programa apresentado. Inicialmente devemos saber que o símbolo de igualdade, na verdade, representa atribuição de valores para a linguagem C++. Assim como na linguagem C, a representação de igualdade é feita através do símbolo (==) e, conseqüentemente, a representação de diferença, é feita pelo símbolo (!=). Já o símbolo de soma (+) quer realmente expressar um processo de adição. No próximo capítulo iremos estudar mais detalhadamente os operadores da linguagem C++.

```
/* Outro Programa Exemplo */
#include "iostream.h"

void main (   )
{

    float base, altura, areatri;

    cout << "Entre com a base do triangulo:";
    cin >> base;

    cout << "Entre com a altura do triangulo:";
    cin >> altura;

    area = base * altura / 2;
```

```
    cout << "A Área do Triangulo e... " << area;
}
```

Neste programa exemplo apresentado, utilizamos uma entrada de dados do tipo real onde, através da entrada da base e altura de um certo triângulo, calcula-se e imprime-se sua respectiva área.

Observe o próximo programa exemplo:

```
#include <iostream.h>

void main (   )
{
    char nome[30];

    cout << "Digite o seu nome:";
    cin >> nome;

    cout << "Como vai voce " << '\n' << nome;
}
```

Você precisa saber que existem outras funções de entrada de dados que podem ser utilizadas na linguagem C++, conforme podemos ver a seguir.

OUTRAS FUNÇÕES DE ENTRADA

Como já foi comentado anteriormente, o cin não é a única instrução utilizadas para realizar as entradas de dados de um certo programa. Iremos estudar as entrada de dados feitas através das funções gets(   ), getchar(   ), getche(   ) e getch( ).

FUNÇÃO gets ( )

Essa função processa tudo que foi digitado até que a tecla ENTER seja pressionada. O caracter ENTER não é acrescentado à string mas sim identificada como término da mesma.

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

void main ( )
{
    char nome[30];

    clrscr( );

    cout << "Digite o seu nome:";
    gets( nome);

    cout << "\nComo vai voce " << '\n' << nome;
}
```

FUNÇÃO getchar( )

Toda vez que desejarmos efetuar uma ação de entrada de dados para apenas um caracter, devemos utilizar a função getchar( ). Sempre que uma letra for digitada através dessa função, há necessidade do pressionamento da tecla ENTER para concluir tal leitura.

A biblioteca C++ dispõe de funções que lêem um caracter no instante em que ele é digitado sem a necessidade do pressionamento da tecla ENTER, veremos mais adiante.

```
/* Aproveito a oportunidade para mostrar
   a forma de representar a utilização de
   comentários com múltiplas linha       */

#include <iostream.h>
#include <conio.h>

void main (   )
{
    char letra;

    clrscr(   );

    cout << "Digite uma letra qualquer:";
    letra = getchar(  );

    cout << "\n\nVocê digitou a letra " <<  letra;
}
```

FUNÇÃO getche( )

A função getche(   ) recebe um caracter digitado e permite que ele seja mostrado na tela do computador. É dispensável o pressionamento da tecla ENTER por parte do usuário. Isso já ocorre de forma automática.

```
#include <iostream.h>
#include <conio.h>
    char letra;
```

```
void main (   )
{

    clrscr(   );

    cout << "Digite uma letra qualquer:";
    letra = getche(  );

    cout << "\n\nVocê digitou a letra " <<  letra;
}
```

Note que a variável **letra** foi declarada como global por estar fisicamente fora do corpo principal do programa.

FUNÇÃO getch( )

A função getch(   ) permite que o usuário forneça um caracter através do teclado. Este caracter não será mostrado na tela do computador. Assim como a função getche(   ), ela dispensa o pressionamento da tecla ENTER por parte do usuário, pois a passagem para a próxima linha já ocorre automaticamente.

```
/*  Programa exemplo */
#include <iostream.h>
#include <conio.h>

void main (   )
{
    char ch;
```

```
    clrscr(   );
    cout << "Digite uma tecla:";
    ch = getch(   );

    clrscr(  );
    cout << '\n' << "A tecla que você pressionou foi " << ch;

    getch(  );  /* Realiza uma pausa temporária até
                   que o usuário pressione ume tecla*/
}
```

Lembre-se que a escolha da melhor função a ser utilizada no seu programa irá depender exclusivamente da sua decisão, pois quem sabe a real necessidade do seu programa é você mesmo.

Exercícios

1- Para que utilizamos a instrução cin?

2- Para que serve a função getchar( )?

3- Qual a diferença da função getche( ) e getch( )?

4- Para que serve a função gets( )?

5- Explique detalhadamente o que irá fazer cada uma das funções apresentadas abaixo:

   variavel = getchar( ); variavel = getche( ); variavel = getch( );

5

OPERADORES

Os operadores são utilizados para a formação das expressões numéricas que utilizaremos na linguagem C++. Na verdade, existem diversos tipos diferentes de operadores a saber. Façamos então um estudo mais detalhado sobre eles.

ARITMÉTICOS

Símbolos	Operadores
*	Multiplicação
/	Divisão
%	Módulo (resto)
+	Adição
-	Subtração
++	Incremento
--	Decremento

A precedência matemática quanto à utilização de sinais é mantida da mesma forma que em algoritmos computacionais, ou seja, (, *, /, +, -. Do decorrer

desse livro teremos acesso a programas escritos em C++ que utilizam algumas dessas funções.

OPERADORES DE INCREMENTO

Parece ser bem complicado, à primeira vista, mas não é nenhum bicho de sete cabeças. Incrementar uma variável significa na íntegra que estamos adicionando um valor a ela, ou seja, somar valores a uma variável em questão.

Veja:

```
i = i + 1;          i ++;          ++ i;
```

Ambas as expressões significam a mesma coisa, ou seja, incrementar um à variável i.

PÓS-INCREMENTO E PRÉ-INCREMENTO

```
#include <iostream.h>

void main (   )
{
   int a, b;
   a = 2;
   b = a ++;
   cout << a << " " << b;
}
```

```
#include <iostream.h>

void main (   )
{
   int a, b;
   a = 2;
   b = ++ a;
   cout << a << " " << b;
}
```

No primeiro caso serão impressos os valores 3 para a variável a e 2 para a variável b. Já no segundo caso, será impresso o valor 3 tanto para a variável a quanto para a variável b.

OPERADORES DE DECREMENTO

Decrementar uma variável é justamente ao contrário do incremento, ou seja, o valor da variável é depreciado progressivamente. Da mesma forma que as expressões abaixo têm a mesma função.

Veja:

i = i – 1; i --; -- i;

Aqui também, as expressões significam a mesma coisa, ou seja, decrementar um à variável i.

PÓS-DECREMENTO E PRÉ-DECREMENTO

```
#include <iostream.h>

void main (   )
{
    int a, b;
    a = 2;
    b = a --;
    cout << a << " " << b;
}
```

```
#include <iostream.h>

void main (   )
{
    int a, b;
    a = 2;
    b = -- a;
    cout << a << " " << b;
}
```

Assim como no incremento, no primeiro caso serão impressos os valores 1 para a variável a e 2 para a variável b. Já no segundo caso, será impresso o valor 1 tanto para a variável a quanto para a variável b.

LÓGICOS

São também conhecidos como conectivos lógicos de operação, pois objetivam conectar expressões lógicas que geralmente, são apresentadas através de comandos de decisão.

São eles:

<table>
<tr><th>Operador</th><th>Função</th></tr>
<tr><td>&&</td><td>E lógico</td></tr>
<tr><td>||</td><td>Ou lógico</td></tr>
<tr><td>!</td><td>Não lógico</td></tr>
</table>

RELACIONAIS

Símbolos	Significado
>	Maior que
<	Menor que
>=	Maior ou igual
<=	Menor ou igual
==	Igualdade
!=	Diferença
=	Atribuição simples

*Também, assim como nas linguagm C, o C++ permite manipular os chamados operadores de atribuição composta. Veja o exemplo: a += b; que verdadeiramente significa a mesma coisa que a = a + b. Veja outro exemplo, c *= a; que significa c = c * a.*

Entendeu?

Modelos:

```
if ( (estado_civil==´S´) && (idade>17))
        cout << "To dentro...";

if ((uf==´R´) || (uf==´S´) || (uf==´M´) || (uf==´E´))
        cout << "Região Sudeste";

if(Not(sexo==´F´))
        cout << "MASCULINO";

if((ano<1990) && ((idade=20) || (idade=30)))
        cout << "Mensagem Enviada...";
    else
        cout << "Mensagem interrompida...";
```

Os operadores lógicos &&, || e !, também são conhecidos como conectivos de operação. São usados para permitir o uso de mais de uma condição numa mesma expressão. Veremos sua aplicação no próximo capítulo desse livro.

OPERADOR CONDICIONAL TERNÁRIO

Funciona muito bem para situações de decisões do tipo IF... THEN... ELSE, ou seja, quando em uma determinada condição tem-se que obter duas alternativas possíveis, uma verdadeira e outra falsa. O próximo capítulo trata detalhadamente das estruturas condicionais.

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

void main( )
{
    int a,
        b,
       max;

    clrscr( );
    cout << "Digite dois numeros:";
    cin >> a.;
    cin >> b;
    max = ( a > b ) ? a : b;
    cout << "O maior deles e " << max;
}
```

Observe a seguir que ainda, podemos fazer uma representação bem melhor desse comando:

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

```
void main (   )
{
    int a,
        b;

    clrscr (  );
    cout << "Digite dois numeros:";
    cin >> a >> b;
    cout << "O maior deles e " << ((a > b) ? a : b);
}
```

A forma pela qual você irá trabalhar com o operador condicional ternário, na verdade, não importa muito pois, ao utilizá-lo, você estará eliminando, por exemplo, o uso da função condicional if(   ), que iremos abordar no próximo capítulo.

```
if( a > b )
  Cout << "O maior deles e " << a;
  else
    cout << "O maior deles e " << b;
```

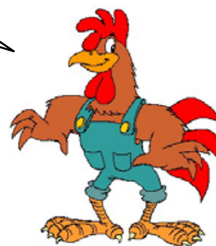
Você poderá ainda ter uma estrutura ternária encadeada a outra estrutura ternária assim como também pode acontecer com a estrutura condicional ninho de if´s. Veja a seguir:

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

```
void main (   )
{
    int a,
        b,
       val;

    clrscr(  );
    cout << "Digite dois numeros:";
    cin >> a;
    cin >> b;
    val = ( a == b ) ? a+b : ( a > b) ? a : b;
    cout << "O maior deles e " << val;
}
```

Derrepente você perguntaria: “Quando devo utilizar essa estrutura?” Na verdade, não existe uma regra que determine onde e como você deva utilizar o operador condicional ternário ou a função if( ). A escolha será exclusivamente sua como programador.

Logo, para a solução de um determinado problema que envolva um tipo de teste lógico condicional cabe especificamente a você decidir “quem” irá servi-lo no momento.



PRIORIDADE DOS OPERADORES

1º	Parênteses e Funções
2º	Sinais unários
3º	Exponenciação
4º	Divisão e Multiplicação
5º	Adição e Subtração
6º	Operadores Relacionais
7º	NÃO
8º	E
9º	OU

Exercícios

1- Para que servem os operadores?

2- Qual o símbolo que representa o operador de módulo da linguagem de programação C++?

3- O que significa dizer quando falamos que uma variável está sendo decrementada?

4- Diferencie incremento e decremento.

5- Na expressão A + B * C – F, qual a precedência de operações?

6- Qual a diferença do uso do Pré-incremento para o Pós-incremento?

7- O que são os chamados operadores de atribuição composta? Exemplifique.

6

ESTRUTURAS CONDICIONAIS

Essas estruturas permitem com que o programa execute diferentes tipos de procedimentos baseados em uma determinada decisão. Basicamente, existem dois tipos de estruturas condicionais: Alternativa simples e alternativa composta.

ALTERNATIVA SIMPLES – if...

É uma estrutura que através de uma determinada condição, retorna um valor possível, somente se essa condição for verdadeira (debidamente atendida).

```
#include <iostream.h>
void main( )
{
    cout << “Entre com um número:”;
    int num;
        cin >> num;
       if(num > 0)
           cout << “\nNum é maior que ZERO”;
 }
```

No exemplo, se o valor da variável num for maior do que zero, será impressa a mensagem “*Num é maior que ZERO*”.

ALTERNATIVA COMPOSTA – if... else...

Esta estrutura, diferente da primeira, permite ao usuário retornar dois valores possíveis. O primeiro verdadeiro verdadeiro, se a condição estipulada for satisfeita e o segundo falso, caso a condição não seja devidamente atendida.

```
#include <iostream.h>
void main( )
{
    cout << “Entre com um número:”;
    int num;
        cin >> num;
       if(num > 0)
            cout << “\nNum é maior que ZERO”;
           else
                    cout << “\nNum não é maior que ZERO”;
 }
```

ENCADEAMENTO DE if´s

Trata-se de um recurso que permite ao usuário utilizar várias estruturas if, uma dentro de outra obtendo assim, diversas possibilidades de respostas. Contudo, a última resposta será sempre a negativa da última alternativa levantada.

```
#include <iostream.h>
void main( )
{
```

```
cout << “Entre com um número:”;
int num;
    cin >> num;
   if(num == 0)
        cout << “\nNúmero igual a ZERO”;
       else
       if(num > 0)
                cout << “\nNúmero Positivo”;
                else
                        cout << “\nNúmero Negativo!”;
}
```

Aqui podemos realmente constatar que o comando If tem ordem matemática N-1 que significa dizer...

Para cada N respostas que eu precise obter utilizarei N-1 comando If.

MÚLTIPLA ESCOLHA – switch ...

É conhecido como estrutura condicional de múltiplas escolhas ou estudo de casos. Torna-se vantajoso ao ninho de if's quanto a utilização de uma expressão bastante longa pois, facilita na escrita do programa, quanto ao comando de tomada de decisão, se o mesmo apresentar muitas alternativas.

Observe atentamente o exemplo apresentado a seguir:

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

```
void main( )
{
    int num;

   /*  Pede que o usuário forneça um número pelo teclado  */

    clrscr( );
    cout << "Digite ou numero 1, 2 ou 3:";
    cin >> num;

    switch(num)
    {
       case 1 :
              cout << "Numero Um";
              break;
       case 2 :
              cout << "Numero Dois";
              break;
       case 3 :
              cout << "Numero Tres";
              break;
       default :
              cout << "Numero Incorreto...";
     }
  }
```

Observação:

Na estrutura condicional de múltipla escolha ou estudo de casos, é opcional a utilização do comando default para representar uma alternativa que significa "EM NENHUM DOS CASOS ANTERIORES". Também houve a necessidade de utilizar o comando break para cada uma das opções do case, assim como ocorre na linguagem de programação C. Sua função é fazer com que o C++ execute apenas a instrução referente ao estudo de caso em questão, sem que ele procure executar as demais instruções, referentes aos outros estudos de casos do comando switch.

Também, é importante saber que a execução do comando switch segue os seguintes passos:

1- A expressão é avaliada.

2- Se o resultado da expressão não for igual a nenhuma das constantes e já estiver sido incluída no comando switch a opção default, o comando associado ao default será executado. Caso contrário, isto é, se a opção default não estiver presente, o processamento continuará a partir do comando seguinte ao comando switch.

IMPORTANTE:

Pode haver uma ou mais instruções seguindo cada case. Se isso ocorrer precisaremos especificar para cada uma delas um bloco de comandos.

A expressão em case (<expressão>) deve ter um valor compatível com um inteiro, isto é, podem ser usadas expressões do tipo char e integer com todas as suas variações. Você não pode usar reais, ponteiros, strings ou estruturas de dados.

UTILIZANDO CONECTIVOS DE OPERAÇÃO

Já sabemos que os conectivos de operação são utilizados para permitir ao usuário tomar uma decisão baseado em duas ou mais condições em uma mesma expressão. São eles: && (e lógico), || (ou lógico) e ! (não lógico). No capítulo 5, estão dispostos os operadores lóogicos. Abaixo, podemos observar um exemplo prático do uso de conectivos lógicos de operação:

```
if((n1 > n2) && ((n1 > n3) || (n4 > n2)))
    cout << “A L E R T A ! ! !”;
    else
        cout << “Z O N A  S E G U R A !”;
```

ATIVIDADES PRÁTICAS

Atividade Prática 1

Faça um programa em C++ que permita cadastrar dois números interiros distintos através do teclado. Ao final do processamento, imprima qual o maior e o menor desses números.

Atividade Prática 2

Escreva um programa na linguagem de programação C++ que permita ao usuário ler três números distintos pelo teclado listando, ao final do programa, qual o maior, menor e o mediano deles.

Atividade Prática 3

Faça um programa em C++ que permita ao usuário ler uma letra qualquer através do teclado. No final do processamento, o programa deverá informar se a letra digitada é uma vogal ou uma consoante.

Atividade Prática 4

Escreva um programa em C++ que leia um número informando ao final do processamento se esse número é primo ou não.

Atividade Prática 5

Em C++, desenvolva um programa de computador que leia dois números inteiros. Realize o seguinte processamento: Calcular o quadrado do primeiro número pelo segundo número e também, o cubo do segundo número pelo primeiro. Isso só será feito apenas se os dois números fornecidos derem suporte para realização desse tipo de operação.

Exercícios

1- Qual a diferença da múltipla escolha para o ninho de if´s?

2- O que significa dizer que o comando if apresenta ordem matemática N-1?

3- Explique como funciona a chamada alternativa simples. Exemplifique.

4- Explique como funciona a chamada alternativa composta. Exemplifique.

5- Qual a função do “comando” default na estrutura de múltipla escolha?

6- Para que utilizamos, em nossos programas de computador, as chamadas estruturas condicionais?

7- O que quer dizer a seguinte linha de comando abaixo:

```
if( a > b )
    cout << a;
    else
      cout << b;
```

8- Qual a função dos chamados conectivos lógicos de operação?

9- Qual o conectivo de operação que expressa o contrário do que estamos querendo referir?

10- O que faz a seguinte linha de comando:

```
if( ! (numero == 0))
        cout << "Numero diferente de ZERO";
```

7

ESTRUTURAS DE ITERAÇÃO

São utilizadas para que uma parte de seu programa possa ser repetida n vezes sem a necessidade de reescrevê-lo. Essas estruturas também são conhecidas como LOOP ou laços.

Iremos estudar as três estruturas possíveis conhecidas em C++: for (para/variando), while (enquanto/faça) e do… while… (repita/até). Iremos, a seguir, analisar cada uma delas cuidadosamente nessa ordem.

LOOP for

É encontrado na maioria das linguagens de programação, incluindo C++. A idéia básica do comando for é que você execute um conjunto de comandos, um número fixo de vezes, enquanto uma variável de controle, é incrementada ou decrementada a cada passagem pelo laço.

Vejamos o exemplo a seguir:

```
// Programa Exemplo Laço For
#include <iostream.h>
void main( )
        {
```

```
        int x;

        for(x=0; x<10; x++)
            cout << “X = ” << x;
}
```

Outro Exemplo:

```
#include <iostream.h>
void main( )
{
        int x;

        for(x=1; x<10; x++)
        {
            cout << “O valor de X e…” << ‘\n’;
            cout << x;
        }
}
```

Veja a seguir, outro exemplo da estrutura for tendo como resultado um outro comando for resultando assim, em um for dentro de outro for.

```
#include <iostream.h>
void main( )
{
        int x, y;
```

```
        for(x=1; x<=3; x++)
            for(y=1; y<3; y++)
                cout << (x + y);
}
```

Outro Exemplo:

```
#include <iostream.h>
void main( )
{
        int x;
        double a, b;

          cout << “Entre com valor de A e B”;
          cin >> a;
          cin >> b;

        for(x=1; x<=3; x++)
        {
            a = a + b;

            if((a > 0) && (a < 10))
                b = a + 3;
                else
                    b = a + 1;

          cout << “A =  ” << a << ‘\n’;
          cout << “B = “ << b;
```

```
                    }
}
```

```
// Exemplo de Loop Infinito
#include <iostream.h>
void main( )
{
       for( ; ; )
            cout << “Loop Infinito”;
}
```

LOOP while

É o mais genérico dos três e pode ser usado para substituir os outros dois; em outras palavras, o laço while supre todas as necessidades. Já os outros dois, são usados por uma questão de comodidade. Vamos analisar o exemplo a seguir:

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

void main( )
{
                    int x=0;

                   while(x<10)
                   {

                        cout << “X = “ << x;
```

```
            x++;
        }
    getch( );
}
```

Note que aqui, inicialmente, a condição estipulada no laço é testada e só depois, a instrução do loop será executada. No próximo laço a ser estudado, do.. while, a diferença básica está no teste realizado. Vejamos então a seguir.

LOOP do... while

A instrução do ... while é semelhante ao comando while. A diferença está no momento da avaliação da expressão, o que sempre ocorre sempre após a execução do comando. Isto faz com que o comando do laço do ... while sempre execute pelo menos uma vez antes de realizar tal teste. Observe atentamente o exemplo abaixo:

```
#include <iostream.h>
void main( )
{
        int x=0;

        do
        {

            cout << “X = “ << x;
            x++;
        } while(x != 10);
}
```

Observe que no laço do… while, como já foi dito anteriormente, primeiramente a instrução é executada, ao menos uma vez, e logo em seguida a condição será verificada.

Exercícios

1- Qual o uso prático das estruturas de iteração?

2- Qual a diferença básica em utilizar a estrutura de repetição repita...até e enquanto...faça?

3- Exemplifique o uso da estrutura para...variando com um contador incrementado.

4- Escreva um programa em C++ que escreva todos os números pares compreendidos na seguinte seqüência: 20 até 41. Para tal, utilize a estrutura de repetição...

   a. Para...variando
   b. Repita...até
   c. Enquanto...faça

5- Qual a vantagem de elaborarmos um programa cujo teste seja feito no início?

6- Qual a vantagem de elaborarmos um programa cujo teste seja feito no final?

8

FUNÇÕES

As funções representam um conjunto de instruções descritas com a finalidade de atender o cumprimento de uma dada tarefa particular. Elas são agrupadas numa unidade com um nome ao qual é usado para referenciá-la.

Tecnicamente falando, a principal razão do uso de funções está na aplicação da técnica de programação denominada MODULARIZAÇÃO, que consiste em dividir o seu programa em diversas partes objetivando uma melhor organização e manutenção do mesmo.

O código referente a uma função é carregado ao programa principal apenas uma vez embora possa ser executado diversas vezes. As funções em C++ apresentam o mesmo propósito daquelas descritas na linguagem C, ou seja, criação de subprogramas.

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

int celsius (int fahrenheit);      // Variáveis Globais – dito protótipo da função

/* Definição do programa principal */
void main( )
{
                int c, f;

        clrscr( );
        cout << “Entre com a temperatura em graus Fahrenheit:”;
        cin >> f;

        c = celsius( f );      // Chamada da função Celcius

        cout << “\nTemperatura em Celsius = “ << c;
}

/* Definição da função Celsius( ) */
int celsius (int fahrenheit)
{
        int c = (fahrenheit – 32) * 5 / 9;
        return c;         // Retorna a função que fez a chamada o valor de c
}
```

Observe que ao fazer a chamada da função *celsius(f)*, foi colocada a variável f entre parênteses. Quando isso ocorre, dizemos que está acontecendo uma passagem de parâmetros, ou seja, é passada de uma função para outra valores de variáveis que interessam ao bom andamento do processamento da função.

Note que, tecnicamente dizendo, a forma de escrita de um programa que tenha funções em C++ é dita Top-Down (de cima para baixo- primeiramente é descrita a função principal e em seguida, as demais funções pertencentes ao programa em questão.

Assim como na linguagem de programação C, podemos escrever uma função sem a necessidade da descrição do dito protótipo da função. Para isto, basta inverter o tipo de estruturação do programa (de Top-Down para Bottom-Up). Bottom-Up é a forma de apresentação contrária da Top-Down, ou seja, inicialmente vem descritas todas as funções necessárias ao programa e só depois disso, é descrita a função principal. Veja exemplo a seguir:

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

/* Definição da função Celsius( ) */
int celsius (int fahrenheit)
{
        int c = (fahrenheit – 32) * 5 / 9;
        return c;       // Retorna a função que fez a chamada o valor de c
}

/* Definição do programa principal */
void main( )
{
              int c, f;

        clrscr( );
        cout << “Entre com a temperatura em graus Fahrenheit:”;
        cin >> f;

        c = celsius( f );     // Chamada da função Celcius
```

```
	cout << “\nTemperatura em Celsius = “ << c;

	getch( );
}
```

Não podemos deixar de mensionar que sempre que falarmos em passagem de parâmetros, tecnicamente, existem dois tipos distintos: passagem de parâmetros por valor e passagem de parâmetro por referência. A seguir, iremos estudar cada um desses dois casos separadamente.

Passagem por Valor

Aqui, literalmente, a função cria cópia dos valores que são passados pelos parâmetros transmitidos originalmente. Assim, não altera os valores originais das variáveis (são mantidos íntegros). Observe a seguir:

```
#include <iostream.h>

/* Definição da função Absoluto( ) */
int absoluto (int numero)
{
	return ( numero > 0 ) ? numero : -numero;
}

/* Definição do programa principal */
void main( )
{
	int numero;
```

```
        cout << "Entre com um numero:";
        cin >> numero;

        cout << absoluto( numero );     // Chamada da função Celcius
}
```

Passagem por Referência

Nós podemos observar que na utilização de parâmetros por valor, os valores originários dos variáveis não serão alterados em hipótese alguma, pois são criadas cópias dessas variáveis para que isto seja efetivamente evitado. Já na passagem de parâmetros por referência, os valores originários são modificados. Porém, apresenta como vantagem o uso de variáveis existentes na função que fez a chamada desta. Além desse benefício, este recurso permite que sejam retornados mais de um valor para a função chamadora. Vejamos então no nosso próximo programa exemplo:

```
#include <iostream.h>

/* Definição da função Reajuste( ) */
void reajuste (float& preco, float& reajusta)
{
        reajuata = preço * 0.2;
        preço = preço * 1.2;
}

/* Definição do programa principal */
void main( )
{
        float valorproduto, valorreajuste;
```

```
    do
    {
        cout << "Entre com preço atual do produto:";
        cin >> valorproduto;

        reajuste (valorproduto, valorreajuste);    // Chamada da função

        cout << "\nNovo Preço ..........” << valorproduto << "\n\n
            Aumento........” << valorreajuste;

        }  while ( valorproduto != 0.0);
}
```

Notem que a mesma variável, *valorproduto* e *valorreajuste*, passa a ter neste programa dois valores diferentes (um valor antes da chamada da função reajuste( ) e outro após o retorno desta.

Usando Funções Recursivas

Inicialmente preciso esclarecer o verdadeiro significado de função recursiva. Verdadeiramente, uma função recursiva é aquela que faz referência à ela mesma. Observe abaixo o clássico exemplo de Fatorial:

```
#include <iostream.h>

/* Definição da função Reajuste(  ) */
long fatorial (int num)
{
    return ( (num==0) ? long( 1 ) : long( num ) * fatorial( num – 1 ));
}
```

```
/* Definição do programa principal */
void main( )
{
        int num;

        while ( 1 )
        {
                cout << "Entre com um número:";
                cin >> num;

                if ( num > 0 )
                        break;    // Ocorre Quando Número Negativo

                cout << "\nFatorial = " << fatorial( num );

        }
}
```

É necessário lembrar que o código gerado pela função recursiva sempre exigirá mais memória disponível, tornando assim a execução cada vez mais lenta.

Exercícios

1- Para que servem as funções?
2- O que é uma função recursiva?
3- Escreva um programa em C++ que permita ao usuário ler dados suficientes pelo teclado para calcular e imprimir a área de um triângulo qualquer. Sabe-se que o cálculo da área do triangulo deverá ser feito através da função chamada *calcula( )*.
4- Qual a diferença existente entre um programa com função que utilize passagem de parâmetro por valor e outro que utilize passagem de parâmetro por referência?

5- Faça uma pesquisa e escreva um programa em C++ que utilize a técnica de função recursiva.

6- Recursividade é o mesmo que função recursiva? Justifique sua resposta.

9

VETORES E MATRIZES

Na verdade, um vetor representa uma posição de memória que é redimensionada para *n* pedaços possíveis, que armazenarão dados do mesmo tipo. Uma matriz representa um vetor de *n* dimensões. Verdadeiramente, posso afirmar que uma matriz uni-dimensional trata-se de um vetor de dados do mesmo tipo.

Devemos sempre estar atentos para lembrar de que a definição de um vetor ou matriz é feita por tipodistinto, salvo casos especiais no uso efetivo de estruturas de dados (será mostrado mais tarde na segunda parte deste livro).

Vejamos a seguir alguns programas exemplos escritos na linguagem C++.

Vetores Unidimensionais

```
/* Programa Exemplo 1 – Fornecendo dados de  5 pessoas */
#include <iostream.h>
#include <conio.h>

void main( )
{
          int contador,
            idade[5];
          float altura[5];
          char nome[5][30];       // Declaração de uma String
                                  // dúvidas rever capítulo 3
          char sexo[5];

     for (contador=0; contador<5; contador++)
       {

          clrscr( );
                                                    Índice do Vetor
          cout << "\nEntre com o nome:";
          cin >> nome[contador];   <----------------

          cout << "\nEntre com sua idade:";
          cin >> idade[contador];

          cout << "\nEntre com sua altura:";
          cin >> altura[contador];
```

```
                cout << "\nEntre com o Sexo:";
                sexo[contador] = getche( );
          }
}
```

Neste primeiro exemplo, a variável nome é dimensionada da seguinte forma:

```
Nome[ 0 ] ← Primeiro Nome
Nome[ 1 ] ← Segundo Nome
Nome[ 2 ] ← Terceiro Nome
Nome[ 3 ] ← Quarto Nome
Nome[ 4 ] ← Quinto Nome
```

Logo, podemos deduzir que a primeira posição de um vetor em C++ será sempre a posição ZERO e as demais consecutivas a esta, obviamente. Isto também, ocorre para as demais variáveis utilizadas no programa (idade[contador], altura[contador] e sexo[contador]).

Note que no programa exemplo, ao invés de utilizarmos os números pertinentes as posições do vetor, utilizamos a variável contador, pois ela é uma variável cujo valor será variado de ZERO até QUATRO (os mesmos números a serem referenciados pelo vetor). Ele é tecnicamente conhecido como *índice do vetor*.

```
/* Programa Exemplo 2 – Fornecendo e mostrando média de 40 alunos */
#include <iostream.h>
#include <conio.h>
```

```
void main( )
{
            int cont,

            char nome[40][35];
            float media[40];

      for (cont=0; cont<40; cont++)
         {
            cout << “\nEntre com o ” << cont << “o Aluno:”;
            cin >> nome[cont];

            cout << “\nEntre com a ” << cont << “a Média:”;
            cin >> media[cont];
        }

      cont = 0;

      while ( cont < 40 )
      {
            cout << “\nAluno = “ << nome[cont];
            cout << “\nMédia = “ << media[cont] << ‘\n’;

            cont++;
      }

      getch(   );      // Pausa temporária na execução do programa
}
```

Exercícios

1- Escreva um programa em C++ que permita ao usuário ler nome, endereço e telefone de 10 amigos. No final do processamento, imprima todos ao dados armazenados no vetor.

2- Faça um programa na linguagem C++ que permita ao usuário fornecer via teclado 100 nomes de pessoas, acompanhados do sexo e telefone. Após o processamento, imprima todos os nomes e telefones das pessoas do sexo F (feminino).

3- Escreva um programa em C++ que permita criar um vetor de 40 posições para cada uma das variáveis definidas abaixo:

Aluno...........

Nota1...........

Nota2...........

Nota3...........

Nota4...........

Para cada entrada de dado, calcular a média aritmética de cada aluno armazenando-as no vetor Média. No final do processamento, imprima todos os nomes de alunos e suas respectivas médias para todos os alunos que estiverem na situação de APROVADOS, segundo o seguinte critério:

Média >= 7 .... APROVADO

Média < 5 .... REPROVADO

Média >=5 e Média <7 RECUPERAÇÃO

Usando Matrizes Multidimensionais

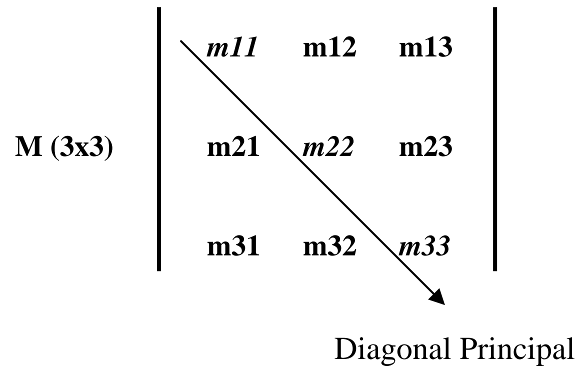
Agora iremos fazer um estudo de matrizes multidimensionais. Nos próximos exemplos a seguir, trabalharei com matrizes bi-dimensionais, num estudo totalmente voltado para a álgebra linear (matemática). Vejamos então a seguir os seguintes estudos de casos:

Estudo de Caso Proposto:

Fazer um programa em C++ que permita ao usuário criar uma matriz quadrada de ordem três, preenchendo-a com valores inteiros, através do teclado. Após o processamento, imprima todos os elementos da diagonal principal.

Então vejamos: uma matriz quadrada é aquela que apresenta o mesmo número de linhas e colunas. Logo, uma matriz quadrada de ordem três, significa dizer que esta matriz terá três linhas e três colunas, ou seja, M (3x3). Uma das características da matriz quadrada é apresentar uma diagonal principal e uma diagonal secundária. É bem simples. Na diagonal principal todos os seus elementos apresentam o mesmo número de linha e coluna. Portanto, numa matriz quadrada de

ordem três os elementos serão os seguintes: m11, m22 e m33, conforme podemos visualizar a seguir:



Diagonal Principal

Naturalmente, os elementos m13, m22 e m31 estão fisicamente fazendo parte da chamada diagonal secundária (aquela contrária a diagonal principal).

```
/* Solução do Estudo de Caso Nº 1 */
#include <iostream.h>
#include <conio.h>

void main( )
{
        int lin, col,
            mat[3][3];

    for (lin=1; lin<=3; lin++)
        for(col=1; col<=3; col++)
            {
                cout << "\nEntre com o elemento da matriz:";
                cin >> mat[lin][col];
            }
```

```
        for (lin=1; lin<=3; lin++)
                for(col=1; col<=3; col++)
                        if (lin==col)
                                cout << '\n' << mat[lin][col];

        getch(   );
}
```

Exercícios

1- Escreva um programa em C++ que permita ao usuário preencher uma matriz quadrada de ordem 4 com elementos do tipo inteiro. Após o processamento, imprima todos os elementos pares que façam parte da diagonal principal.

2- Faça um programa em C++ que permita ao usuário preencher uma matriz quadrada de ordem 5 com elementos do tipo caracter (uma letra). Após o processamento, imprima todos os elementos que sejam vogal.

3- Escreva um programa em C++ que permita ao usuário preencher uma matriz quadrada de ordem 6 com elementos do tipo caracter (uma letra). Após o processamento, imprima todos os elementos que sejam consoante e pertençam a diagonal principal.

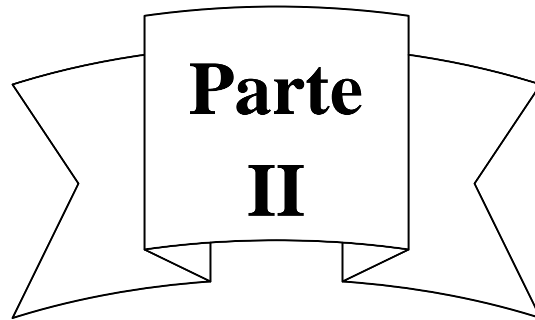
4- Faça um programa em C++ que permita ao usuário preencher uma matriz quadrada de ordem 4 com elementos do tipo inteiro. Após o processamento, imprima todos os elementos ímpares.

Observe que para trabalhar com uma matriz não-quadrada basta especificar o número de linhas e colunas que ela apresente. Assim: **M (2x6)**, ou seja, duas linhas e seis colunas; **M (4x2)**, ou seja, quatro linhas e duas colunas; dentre outros exemplos que poderíamos aqui citar.

Atividade

Só pra não perder a prática, escreva um programa em C++ que permita ao usuário preencher uma matriz **M (4x6)** com elementos do tipo real ou inteiro. No final do processamento, apenas os elementos que forem inteiros deverão ser impressos.

Na verdade, você poderá fazer um estudo mais aprofundado sobre Matrizes e Determinantes em Álgebra Linear e, logicamente, poderá modelar todos os modelos computacionais em C++.


Parte
II

Tópicos Avançados
em C++

10

ESTRUTURAS E UNIÕES

Uma estrutura em C++ é criada através da palavra reservada struct, assim como na linguagem C. Criar uma estrutura de dados nada mais é do que definir um tipo de dado não existente como padrão (PRIMITIVO) para uma linguagem de programação.

ESTRUTURA SIMPLES

Trata-se do uso de mais de uma estrutura cujos elementos apresentados podem ser de tipos diferentes. Vejamos então um programa exemplo a seguir:

```
/* Exemplo de Estruturas Simples em C++ */
#include <iostream.h>
#include <conio.h>
void main( )
{
        struct livro {
                char titulo[30];
                int regnum;
              };
struct livro livro1 = { "Juanito", 4100};
```

```
struct livro livro2 = { "Giullica", 3850};

    cout << "\n Lista de livros:\n";
    cout << "\n    Titulo: " <<  livro1.titulo;
    cout << "\n     Numero do registro:" << livro1.regnum;

    cout << "\n     Titulo: " << livro2.titulo;
    cout << "\n     Numero do registro:" << livro2.regnum;

    getch(  );
}
```

O tratamento de uma estrutura em C++, funciona da mesma forma da linguagem C. Para mencionar uma variável da estrutura criada, basta colocar o nome da variável estrutura, um ponto (para concatenação) e o campo o qual vai ser trabalhado (livro1.titulo).

ESTRUTURA COMPOSTA

Trata-se do uso de mais de uma estrutura encadeada a outras estruturas. Os itens (campos) dessas estruturas são trabalhados de forma semelhante aos da estrutura simples. Observe o programa exemplo abaixo:

```
/* Exemplo Prático Estrutura Composta em C++ */
#include <iostream.h>
#include <conio.h>
void main(  )
{
        struct cliente {
                        int codcli;
```

```
                    char nome[30],
                    char tel[15];
                    struct dtnasc {
                                        int dia,
                                            mes,
                                            ano;
                                        };   struct dtnasc data;
                    };    struct cliente cli;

          clrscr(  );

          cout << "\n Código do Cliente:";
          cin >> cli.codcli;

          cout << "\n Nome:";
          cin >> cli.nome;

          cout << "\n Telefone:";
          cin >> cli.tel;

          cout << "\n Data de nascimento:";
          cin >> cli.data.dia >> cli.data.mês >> cli.data.ano;

          cout << "\n\nPressione uma tecla para continuar...");
          getch(  );
}
```

Vejamos a seguir um outro programa exemplo utilizando o recurso de estruturas em C++:

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
#include <stdlib.h>
#include <stdio.h>

struct data {
        int dia;
        char mes[10];
        int ano;
      };

struct venda {
        data diavenda;
        int pecas;
        float preco;
       };

void novavenda (void);
void listavenda (void);

venda vendas[50];
venda total = {{0, " ", 0}, 0, 0.0};

int n=0;
```

```
/* Função Principal do Programa */
void main( )
{
	const char ESC=27;
	char ch;

	while ( 1 )
	{
		clrscr( );

		cout << "( A ) dicionar uma venda";
		cout << "\n( L ) istar as vendas";
		cout << "\nESC para terminar\n";

		ch = getch( );

		switch (ch)
		{
			case 'A' : novavenda( );
				break;
			case 'L' : listavenda( );
				break;
			case ESC : exit ( 0 );
			default   :  cout << "\nOpção Inválida";
		}
	}
}
```

```
/* Função Nova Venda */
void novavenda( )
{
        char aux[80];

        cout << “Dia: “;
        gets (aux);
        vendas[n].diavenda.dia = atoi (aux);

        cout << “Mês: “;
        gets (vendas[n].diavenda.mes);

        cout << “Ano: “;
        gets (aux);
        vendas[n].diavenda.ano = atoi (aux);

        cout << “Peças: “;
        gets (aux);
        vendas[n].pecas = atoi (aux);

        cout << “Preço: “;
        gets (aux);
        vendas[n].preco = atof (aux);

        total.pecas += vendas[n].pecas;
        total.preco += vendas[n++].preço;
}
```

```
void listavenda( )
{
        int i=0;

        if (!n)
        {
                cout << "\nNão há peças vendidas";
                return;
        }

        cout << setprecision (2);

        for ( ; i<n ; i++ )
        {
                cout << "\n" << setw (2) << vendas[i].diavenda.dia
                     << " de " << setw (10) << vendas[i].diavenda.mes
                     << " de " << setw (4) << vendas[i].diavenda.ano;
                cout << setw (10) << vendas[i].pecas;
                cout << setw (20) << vendas[i].preco;
                cout << "\n\n\n";
}
}
```

Neste nosso programa exemplo utilizamos as funções de conversão de tipos, ***atoi( )*** e ***atof( )***. A função ***atoi( )*** converte um dado string (alfabético) em inteiro (int). Já a função ***atof( )*** converte um dado string (alfabético) em real (float). Se você for conhecedor da linguagem C consegue perceber que são as mesmas funções existentes na biblioteca padrão do compilador

turbo C. É indispensável a inserção da biblioteca padrão **stdlib.h**
para que estas funções possam efetivamente ser utilizadas.

USANDO UNIÕES

Assim como na linguagem de programação C, as uniões (unions) são utilizadas de forma semelhante as estruturas. Notavelmente, não posso deixar de destacar que o espaço ocupado em memória por uma união é bem menor que o espaço ocupado por uma estrutura, pois na estrutura cada membro que faz parte dela ocupa uma posição de memória diferente e, aqui, o espaço é o mesmo devido a união ter todos os membros agrupados como se fossem um só.

A palavra-reservada utilizada para representar uma união é ***union***. Quando você declara uma variável como união, automaticamente é alocado um espaço de memória suficiente para conter o seu maior membro, independente do tamanho dos demais membros dessa união.

UNIÃO SIMPLES

Assim como no uso de estruturas, as uniões seguem o mesmo estilo de definição e sintaxe. Observe atentamente o programa exemplo abaixo:

```
/* Exemplo Prático */
#include <iostream.h>
#include <conio.h>

union
{
        int num1, num2;
        float num3, num4;
}       exemp;      // aqui é definida a variável união

void main( )
```

```
{

        exemp.num1 = 2;
        exemp.num2 = 3;

        cout << "num1 = " << exemp.num1 << "num2 = " << exemp.num2;
        cout << "\n\n";

        exemp.num3 = 5.5;
        exemp.num4 = 9.8;

        cout << "num3 = " << exemp.num3 << " num4 = " << exemp.num4;
        cout "\n\n";

        getch( );
}
```

UNIÃO COMPOSTA

Da mesma forma que as estruturas podem associar membros de outras estruturas, as uniões também podem fazê-lo. No entanto, também, podem ser referenciados membros de estruturas e uniões e vice-versa. Observe nosso próximo exemplo:

```
/* Exemplo Prático */
#include <iostream.h>
#include <conio.h>

struct alfa
{
```

```
        int n1, n2;
};
struct gama
{
        float n3, n4;
};

union
{
        struct alfa
                a;
        struct gama
                g;
}       omega;      // variável união que representa o conjunto das
                    // estruturas aqui apresentadas

void main( )
{
        clrscr( );

        omega.a.n1 = 2;
        omega.a.n2 = 3;
        omega.g.n3 = 1.5;
        omega.g.n4 = 2.5;

        cout << "n1 = " << omega.a.n1 << "\n\n" << " n2 = " << omega.a.n2;
        cout << "\n\n" << "n3 = " << omega.g.n3 << "\n\n"
             << " n4 = " << omega.g.n4;
```

```
    getch( );
}
```

Exercícios

1- Escreva um programa em C++ que permita ao usuário entrar com dados para preenchimento da seguinte estrutura:

FUNCIONARIO

- **MATRÍCULA**
- **NOME**
- **DATA_DE_NASCIMENTO**
  - **DIA**
  - **MÊS**
  - **ANO**
- **CARGO**
- **SALÁRIO**

2- Faça um rpograma em C++ que permita ao usuário criar e preencher a seguinte estrutura abaixo:

CONTRATADO

- **IDENTIDADE**
- **CPF**
- **NOME**
- **DATA_DE_NASCIMENTO**
  - **DIA**
  - **MÊS**
  - **ANO**

PROFISSÃO

REMUNERAÇÃO

3- Escreva um programa em C++ que crie uma união capaz de juntar as duas estruturas desenvolvidas nos exercícios 1 e 2 anteriores.

4- Imagine uma situação em que você tenha que desenvolver um programa para controle de uma academia de ginástica. Então, escreva em linguagem C++ um programa que seja capaz de criar uma estrutura para tal manipulação de dados. (Os dados cadastrais referentes a definição da estrutura, fica a cargo do programador).

11

PONTEIROS OU APONTADORES

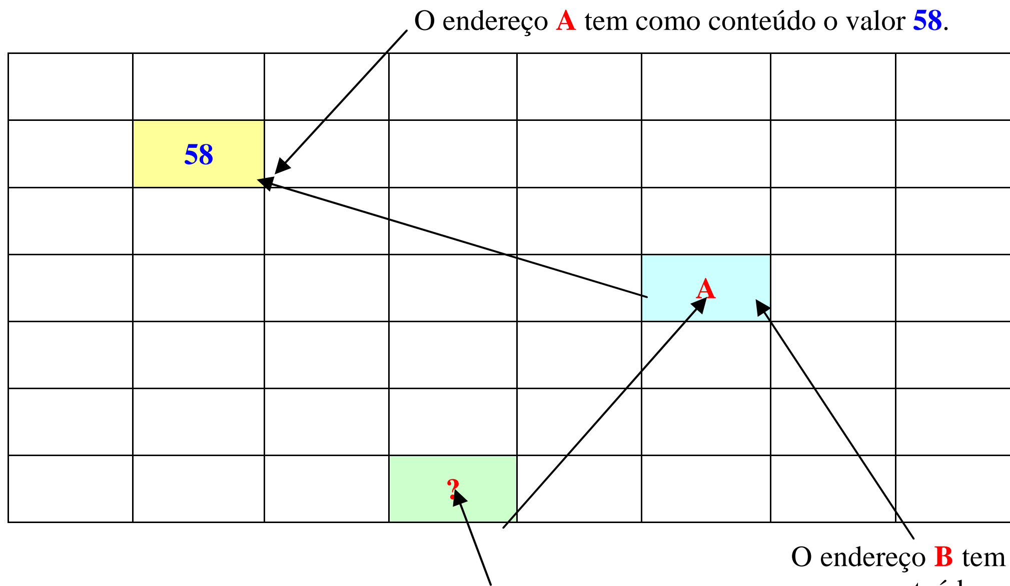
Inicialmente precisamos saber que a manipulação de ponteiros em C++ ocorre de forma idêntica da linguagem C. Vejamos então que um ponteiro ou apontador nada mais é que uma variável que, no seu espaço de memória, armazena o endereço de uma segunda variável, essa sim, normalmente contendo o dado a ser manipulado pelo programa.

Na verdade, os ponteiros ou apontadores são utilizados em situações onde o uso de uma variável é considerado difícil ou indesejável por parte do programa ou até do próprio programador. Abaixo seguem algumas razões para você optar no uso de ponteiros:

- Manipulação de elementos de matrizes;
- Alocação ou desalocação de memória do sistema;
- Receber valores de argumentos de funções que necessitam alterar o valor do argumento passado originalmente;
- Passagem de strings entre funções;
- Criação de estruturas de dados complexas como listas, filas, pilhas e árvores.

Não podemos esquecer que ao manipularmos ponteiros ou apontadores precisaremos fazer uso do operador de endereços (**&**).

Vejamos a seguir como é feita a manipulação e distribuição de dados de um certo ponteiro ou apontador:



O endereço **B** tem como conteúdo o endereço de **A**

Como o endereço de **C** aponta para o endereço de **B** que referencia o conteúdo do endereço de **A**, portanto o conteúdo armazenado no endereço **C** será igual ao endereço da variável **A** cujo valor armazenado em seu conteúdo é 58;

Portanto, o valor apontado pela variável **C** será o conteúdo da variável **A**, ou seja,

58

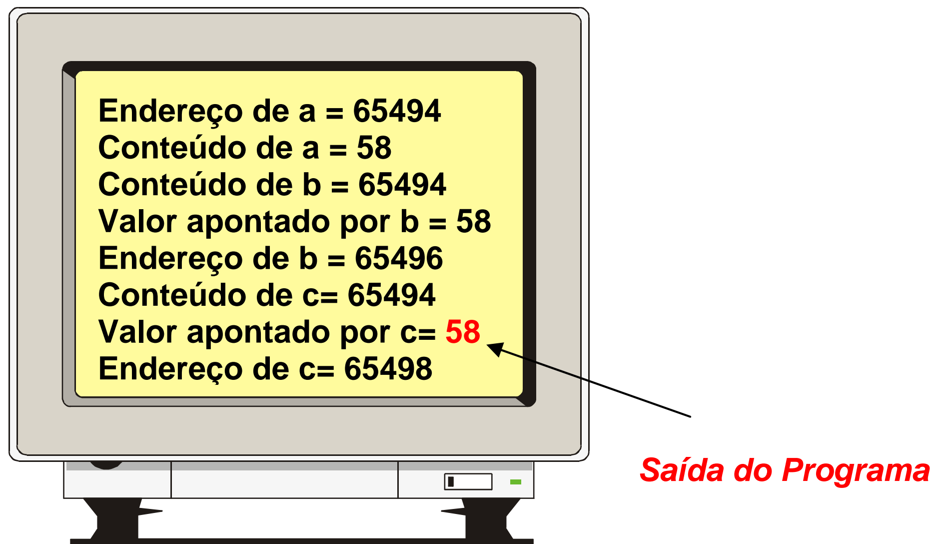
Como o endereço de C aponta para o endereço da variável B que referencia o conteúdo do endereço de A, é verdadeiro afirmar que o conteúdo armazenado no

endereço de C será igual ao endereço da variável A, cujo valor armazenado é igual a 58. Portanto, o valor apontado pela variável C será igual a 58, pois, o conteúdo da variável A é igual a **58**.

Observe atentamente o programa exemplo escrito em C++ abaixo:

```
/* Exemplo – Uso de Ponteiro ou Apontadores */
#include <iostream.h>
#include <conio.h>

void main (    )
{
    int
        a,
        *b, *c;      // representação de uma variável ponteiro

    b = &a;
    *b = 58;
     c = b;

    clrscr(  );
    cout << "Endereco de a: " << &a << "\n";
    cout << "Conteudo de a: " << a << "\n";
    cout << "Conteudo de b: " << b << "\n";
    cout << "Valor apontado por b: " << *b << "\n";
    cout << "Endereco de b: " << &b << "\n";
    cout << "Conteudo de c: " << c << "\n";
    cout << "Valor apontado por c: " << *c << "\n";
    cout << "Endereco de c: " << &c << "\n";

    getch(  );
```

```
}
```



ALOCAÇÃO DINÂMICA

Com o uso efetivo de ponteiros ou apontadores você opoderá tratar a alocação de endereços de memória que tecnicamente é denominada alocação dinâmica.

```
/* Alocação Dinâmica */
#include <iostream.h>
#include <alloc.h>

void main (  )
{
   int
```

```
    *ptr;

  ptr = ( int * ) malloc( sizeof( int ));
  *ptr = 3;
  cout << *ptr;
}
```

Neste programa atribuímos à ptr um valor retornado por uma função chamada malloc( ), a qual é declarada em alloc.h.

Para que possamos entender a instrução ptr = (int *) malloc(sizeof(int)) vamos dividi-la em partes:

1º) O operador sizeof( ) devolve o tamanho, em bytes, do tipo ou da expressão entre parênteses.

2º) A função malloc( ) tem o objetivo de retornar um ponteiro para uma área livre de memória a ser identificada por ela.

Assim, a instrução ptr = (int *) malloc(sizeof(int)) cria dinamicamente uma variável inteira referenciada por *ptr. Podemos trabalhar com várias variáveis do tipo ponteiro na memória do computador.

A determinação referente a quantidade necessária para utilização em um determinado programa depende exclusivamente do que o programa se porpõe. Sejamos a seguir, outro programa em C++ que trabalha o tratamento de alocação de memória dinâmica:

```
/* Outro Programa de Alocação Dinâmica */
#include <iostream.h>
#include <conio.h>
```

```
void main( )
{
        int *pt;

        pt = new int;

        clrscr( );

        cout << "Digite um número: ";
        cin >> *pt;

        cout << "\n\nVocê digitou o número " << *pt;

        delete pt;
}
```

Note que neste programa utilizamos os operadores especiais de alocação e desalocação de memória ***new*** e ***delete***. O operador especial **new** permite alocar um espaço de memória para uma certa variável ponteiro em tempo de execução do programa. Ele retorna sempre para o primeiro byte do novo bloco de memória que tenha sido alocado.

Sempre que alocada (**new**), a memória continuará ocupada até que seja efetivamente desalocada. Para isto, utilizaremos o operador especial **delete**, ou seja, uma variável criada pelo operador **new** continuará existindo até que ela seja destruída pelo operador **delete**.

Mais tarde continuaremos falando nesses dois operadores especiais quando falarmos de manipulação de objetos em C++.

ARITMÉTICA COM PONTEIROS

Inicialmente, vejamos o programa exemplo a seguir:

```
/* Programa Exemplo */
#include <iostream.h>
#include <conio.h>

void main (  )
{
    int vetor[3],
        *p1, *p2;

    p1 = vetor;
    p2 = &vetor[2];
    *p1 = 0;
    *(p1 + 1) = 1;
    *( p1 + 2) = 2;

    if (p2 > p1)
        cout << "Posicoes: " << p2 - p1 << "\n\n";

    getch(  );
}
```

Observe no exemplo anterior que podemos usar com variáveis ponteiros os mesmos operadores aritméticos que usamos com variáveis

comuns. Isto porque, embora seja uma variável dita ponteiro ou apontadora, simplesmente, não deixa de ser uma variável, a qual tratamos com os já conhecidos operadores.

A única diferença é que em determinados momentos estaremos trabalhando com endereços de memória e não somente conteúdos, que é o que acontece com as variáveis comuns.

```
/* Outro Exemplo com Ponteiros ou Apontadores*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define MAX 5

void main (   )
{
    int d;
    int entra = 0;
    char nome[40];
    static char *list[MAX] =
                              { "Karine",
                                "Richard",
                                "Ronaldo",
                                "Juan",
                                "Giullia"       };

    clrscr(  );

    cout << "Digite seu nome:";
```

```
    gets(nome);
    for ( d = 0; d < MAX; d++ )
        if (strcmp(list[d], nome) == 0)
            entra = 1;
    if ( entra == 1 )
        cout << "Voce foi identificado em nosso cadastro";
    else
        cout << "Guardas! Prendam este sujeito!";
          getch( );
}
```

Neste programa utilizamos a biblioteca ***stdlib.h*** para que pudéssemos ter acesso a função de comparação de strings, **strcmp( )** que significa String Compare, ou seja, comparação entre strings. Se fosse o caso, poderíamos ter utilizado também a função de cópia de strings, **strcpy( )** que significa String Copy, ou seja, cópia entre strings.

Veja as possíveis variações na utilização da função strcmp( ):

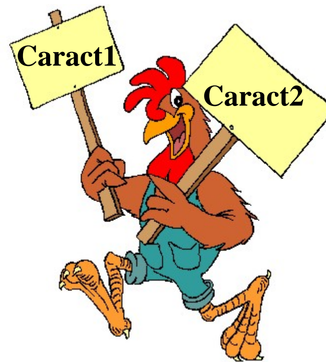
VARIAÇÕES	SIGNIFICADOS
strcmp(STRING1, STRING2) == 0	STRING1 é igual STRING2
strcmp(STRING1, STRING2) != 0	STRING1 é diferente de STRING2
strcmp(STRING1, STRING2) > 0	STRING1 é maior que STRING2
strcmp(STRING1, STRING2) < 0	STRING1 é menor que STRING2

Observe a sintaxe de utilização da função de cópia de strings, strcpy( ) – STRING COPY:

```
strcpy(STRING1, STRING2);
```

strcpy(endereço, “Rua Alfa 32”);

```
/* Programa Exemplo para Tratamento de Strings */
#include <iostream.h>
#include <conio.h>
#include <string.h>

char string1[80];

void main( )
{
  char string2[80],
        caract1,
        caract2;

  clrscr( );

  cout << “Entre com a string origem:”;
  cin >> string1;
  cout << “\n\n” << “Entre com a string resultante:”;
  cin >> string2;
  cout << “\n\n” << “Caracter a ser trocado (velho):”;
  caract1=getche( );
  cout << “\n\n” << “Caracter para mudar (novo):”;
  caract2=getche( );

  cout << “\n\n” << “Nº de ocorrências = ” << substitui(string1, string2,
                                                         caract1, caract2);
 getch( );
```



```
}
/* Declaração da função substitui( ) */
int substitui(char s1[80], char s2[80], char c1, char c2)
{
  int i,
      n=0;

  if(strcmp(s1, s2)==0)
    return(0);
    else{
           for(i=0; i<sizeof(string1); i++)
             if(s1[i]==c1)
             {
                s1[i]=c2;
                 n++;
              }
           return(n);
    }
}
```

Note que usamos a função especial sizeof( ) para determinar o tamanho em bytes de uma certa variável. Funciona da mesma forma que na linguagem C. Observe o próximo programa exemplo:

```
/* Exemplo Prático */
#include <iostream.h>
#include <conio.h>
#include <string.h>

void main( )
```

```
{
   char string1[80],
          string2[80],
          velha[80],
          nova[80];
   int idade1,
        idade2;

   clrscr(  );

   cout << “Nome da primeira pessoa:”;
   cin >> string1;
   cout << “\n\n” << “Idade da primeira pessoa:”;
   cin >> idade1;
   cout << “\n\n” << “Nome da segunda pessoa:”;
   cin >> string2;
   cout << “\n\n” << “Idade da Segunda pessoa:”;
   cin >> idade2;

   if(idade1==idade2)
      cout << “\n\n” << “As pessoas têm a mesma idade...”;
      else{
             if(idade1>idade2)
             {
                strcpy(maior, string1);
                strcpy(menor, string2);
              }
              else{
                      strcpy(maior, string2);
```

```
                    strcpy(menor, string1);
                  }
            }
  clrscr( );

  cout << "O nome da pessoa mais velha é " << maior;
  cout << " e o nome da pessoa mais nova é " << menor;
  getch( );
}
```

Exercícios

1- O que você entende por alocação dinâmica?

2- Defina variáveis apontadoras.

3- Qual o objetivo da função strcmp( )?

4- Diferencie strcmp( ) e strcpy( ).

5- Na aritmética com ponteiros posso efetuar operações com tipos de variáveis diferentes (que não tenham o mesmo tipo)? Justifique sua resposta.

6- Numa alocação dinâmica posso usar vetor de diferentes tipos? Justifique sua resposta.

7- Escreva um programa em linguagem C++ que permita ao usuário armazenar em uma variável ponteiro valores de forma indireta, ou seja, o conteúdo armazenado deserá ser feito a partir de outra variável, também ponteiro do mesmo tipo.

8- Explique com suas palavras para que serve a função malloc( )?

12

Programando C++ Orientado a Objetos

Para que você entenda melhor algumas aplicações feitas em C++ é necessário que lhe sejam esclarecidos alguns conceitos fundamentais sobre orientação a objetos. Tornar-se-ia meio nebuloso um tipo de estudo em C++ sem falar inicialmente em objeto, classe, método, herança, dentre outros conceitos importantes da orientação a objetos. Na verdade, todos os conceitos e entendimentos explorados neste capítulo apresentam uma visão geral da orientação a objetos podendo ser aplicada no Projeto Orientado a Objetos, Análise Orientada a Objetos e na Programação Orientada a Objetos efetivamente.

Olá!
Olha só, essa placa que estou segurando. Em C++, ela é chamada tecnicamente de objeto, pois ela representa uma entidade que existe no chamado mundo real (nosso mundo).
Entendeu??

Talvez você não tenha entendido muito bem o que o nosso amiguinho ***Ronald*** quis dizer ao referenciar a placa que está segurando como um objeto. Mas não se preocupe com isso agora, pois para uma melhor compreensão de tudo aquilo que iremos aprender nesse livro, através da linguagem de programação C++, vamos estudar os conceitos básicos da orientação a objetos.

CONCEITOS BÁSICOS

- ***Objetos*** ⇨ Elementos (componentes) de um sistema que apresenta atributos (propriedades/características) próprios;

- ***Classes*** ⇨ Expressam descrições estáticas de dados e métodos (comportamento) de um dado objeto.

- ***Abstração*** ⇨ Princípio de ignorar os aspectos de um assunto não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais;

- ***Relacionamento entre Objetos*** ⇨ É de extrema importância que os Objetos que estão sendo modelados por uma OOA apresentem um forte relacionamento entre si, para que o princípio básico da OOA conhecido como Herança, possa prevalecer neste tipo de abordagem; É claro, que este é aplicado na programação orientada a objetos.

- ***Encapsulamento*** ⇨ As propriedades e Métodos de um objeto estão contidos na sua própria definição (nele mesmo);

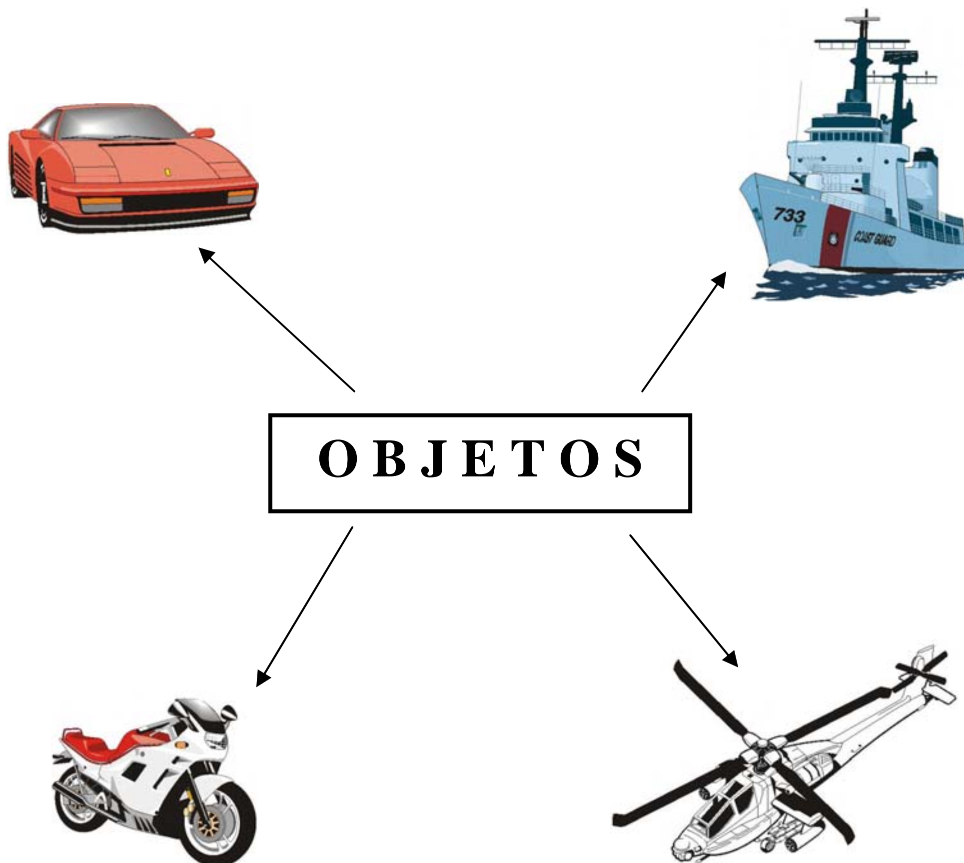
- ***Herança*** ⇨ Mecanismo para representar a similaridade entre classes através do embasamento de um novo objeto à outro já existente, herdando este assim, suas características e métodos, minimizando o trabalho de descrição das mesmas para um novo objeto similar a outro que já tenha sido definido anteriormente;

- ***Polimorfismo*** ⇨ É quando dois ou mais objetos querem deter o mesmo método, ao mesmo tempo, sendo que o domínio (controle) do método pertence ao objeto que o detenha;

- ***Atributos*** ⇨ Representam as características dos objetos;

- ***Mensagens*** ⇨ Representa a troca de dados existentes entre os objetos de um dado sistema;

- ***Interfaces*** ⇨ Representa um conjunto definido de comportamentos que as classes podem efetivamente implementar.



Analise que cada um dos objetos citados ateriormente apresenta suas próprias características (propriedades) e seu próprio comportamento (método) que é diferente dos demais objetos apresentados, pois cada objeto é um objeto é dito unívoco.

ENCAPSULAMENTO, HERANÇA E POLIMORFISMO

Dos princípios básicos da Orientação a Objetos, a Herança e o Polimorfismo podem ser aplicados através de métodos, por meio da linguagem de programação C++. Mas antes disso tudo, devemos compreender como funciona a criação de classes e objetos em C++. Observe atentamente os exemplos a seguir:

```
#include <iostream.h>
class Retangulo                 // Define uma classe de nome Retangulo
{
        private:
                int base, altura;

        public:
                void init (int bas, int alt)
                {
                        base = bas;
                        altura = alt;
                }

                void mostrarea( )
                {
                        cout << "\nBase = " << base << " Altura = " << altura;
                        cout << "\nÁrea = " << (base * altura);
                }
};
```

```
/* Corpo Principal do Programa */
void main( )
{
        Retangulo x, y;

        x.init(5,3);
        y.init(10,6);

        x.mostrarea( );

        y.mostrarea( );
}
```

No programa exemplo acima a classe Retangulo é composta de duas funções que vem acompanhadas de dois itens de dados. É na função mostrarea( ) onde os dados são efetivamente calculados antes de serem exibidos.

Esse agrupamento de dados e funções numa mesma entidade é característica e fundamento exclusivo da programação orientada a objetos. Sendo assim, dizemos que toda vez que temos uma instância de um certo tipo de classe, na verdade temos o que chamamos tecnicamente de objeto.

Mais adiante iremos abordar um programa exemplo com o uso efetivo de ***Herança*** através da Orientação a Objetos.

Em Orientação a Objetos, o que é, realmente, a chamada UML?

INTRODUÇÃO À UML

A **UML** (Unified Modeling Language) foi desenvolvida pela empresa Rational, e é uma linguagem padrão para a elaboração de estrutura de modelagem de projetos de software. A **UML** poderá ser empregada para visualização, a especificação, a construção e documentação de artefatos que façam uso de sistemas complexos de software.

A **UML** é adequada para a modelagem de sistema, cuja abrangência poderá incluir sistemas de informação, corporativos a serem distribuídos a aplicações baseadas em *WEB* e até sistemas complexos embutidos de tempo real. É uma linguagem muito expressiva, abrangendo todas as visões necessárias ao desenvolvimento e implantação desses sistemas. Ela é apenas uma linguagem, e, portanto somente uma parte de um método para desenvolvimento de software. A **UML** é independente do processo, apesar de ser perfeitamente utilizada em processo orientado a casos de usos, centrada na arquitetura, interativa e incremental.

Imagine só o casamento idel: ter um ***Sistema de Informação*** modelado através da **UML** (Orientado a Objetos) e, depois da modelagem de dados, realizar o processo de implementação através da linguagem de programação Java (também Orientada a Objetos). Simplesmente Perfeito!!!

PRINCÍPIOS DA UML

A **UML** é mais do que um mero punhado de símbolos gráficos. Por trás de cada símbolo empregado na notação da **UML** existe uma semântica bem definida. Dessa maneira, um desenvolvedor, poderá usar a **UML** para escrever o seu modelo e qualquer outro desenvolvedor, ou até outra ferramenta, será capaz de interpretá-lo sem ambigüidades.

Para compreender a **UML**, você precisará formar um modelo conceitual da linguagem e isso pressupõe aprender três elementos principais: os blocos de construção básicos da **UML**, as regras que determinam como esses blocos poderão ser combinados e alguns mecanismos comuns aplicados na **UML**. Após entender

essas idéias, você será capaz de ler modelos da UML e criar outros modelos básicos. À medida que acumular experiência na aplicação da **UML**, construa novos modelos a partir desse modelo conceitual, usando características mais avançadas da linguagem.

NOTAÇÃO UML

A **UML** utiliza em sua notação diagramas diversos. Um diagrama é a apresentação gráfica de um conjunto de elementos, geralmente representado como gráfico de vértice (*itens*) e arcos (*relacionamentos*). São desenhados para permitir a visualização de um sistema sob diferentes perspectivas. Nesse sentido, um diagrama constitui uma projeção de um determinado sistema de informação.

Em todos os tipos de sistemas, com exceção dos mais triviais, um diagrama representa uma visão parcial dos elementos que compõe o sistema.

A **UML** se destina principalmente a sistemas de informação complexos de software. Tem sido empregada de maneira efetiva em domínios como: Sistemas de Informação Corporativos, Serviços Bancários e Financeiros, Telecomunicações, Transportes, Defesa do Espaço Aéreo, Vendas de Varejo, Eletrônica Médica, Científicos, Serviços Distribuídos Baseados na Web etc.

O ***Rational Rose*** é uma ferramenta Case padrão utilizada para modelagem de dados Orientado a Objetos, embora existam outras disponíveis no mercado como é o caso do *Play Ground*, *Fast Case* e *Poseidon for UML.*

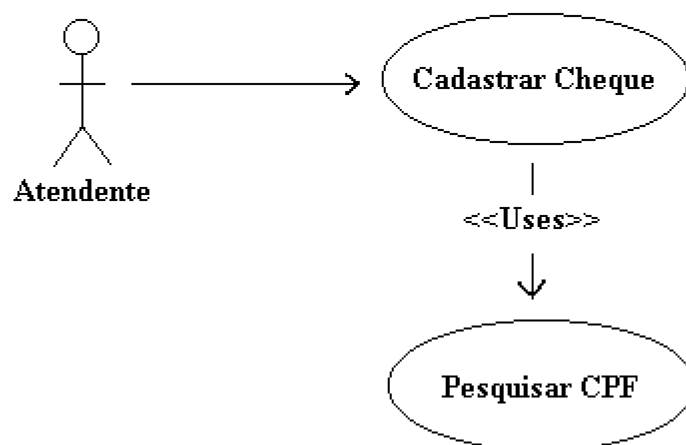
Em particular, o Fast Case, defende o uso da chamada metodologia rápida, que nada mais é que a forma de representação da modelagem de dados numa visão orientada a objetos com um número de diagramas bem reduzido em relação ao Rational Rose (por isso o nome metodologia rápida). Para obter maiores informações a respeito da metodologia rápida você deverá ler o livro "***Desenvolvimento de Software Orientado a Objetos***", publicado pela editora Brasport Livros e Multimídia.

Embora algumas ferramentas CASE (Computer Aided Software Engineering) possam usar apenas alguns dos diagramas disponíveis numa notação

UML, e linguagem de modelagem unificada utiliza como base um conjunto total de ***nove*** (**09**) diagramas, cujos tipos iremos citar a seguir:

- Diagrama de ***Casos de Uso***
- Diagramas de ***Classes***
- Diagrama de ***Objetos***
- Diagramas de ***Seqüência***
- Diagrama de ***Colaboração***
- Diagrama de ***Estado***
- Diagrama de ***Atividade***
- Diagrama de ***Componentes***
- Diagrama de ***Implantação***

Como exemplo, podemos tomar o diagrama de casos de uso. Ele mostra um conjunto de casos de uso, atores e relacionamentos. Pode-se dizer que um Caso de Uso é a representação de um comportamento necessário ao sistema sem detalhar sua estrutura interna. Observe o modelo abaixo de um diagrama de Casos de Uso (*Uses Case*):



Se você quiser mergulhar a fundo nesse emocionante mundo da Análise Orientada a Objetos, você deverá ler o livro "***Desenvolvendo Aplicações com UML***", também publicado pela Brasport.

Qualquer modelagem de sistemas com uma visão orientada a objetos pode ser codificada na linguagem C++. Na verdade, a princípio, baseado em um certo modelo computacional é que surge a necessidade de implementá-lo transformando uma verdade do mundo real em uma realidade do mundo computacional. Vejamos a seguir novos conceitos de orientação a objetos em C++.

CONSTRUTORES EM C++

Construtores são métodos especiais chamados pelo sistema em questão, no momento da criação de um objeto. Eles não possuem um valor de retorno, porque você não pode chamar um construtor para um objeto, você só usa o construtor no momento da inicialização do objeto.

```
/* Programa Exemplo de Manipulação de Classes */
#include <iostream.h>
#include <conio.h>

class data
{
        private :
                int dia,
                    mes,
                    ano;
        public :
                int bissexto( )
                {
                  return ((ano % 4 == 0) && (ano % 100) || (ano % 400) ==0);
                }
```

Objeto **data**

```
        data (int d, int m, int a);     // Construtor em C++

        void initdata(int d, int m, int a);
        void imptimedata(  );
        void imprimesigno(  );
        void imprimebissexto(  );
};

/* Definição do Construtor */
data ::data(int d, int m, int a)     // Construtor sem especificação de tipo
{
        initdata(d, m, a);
}

/* Função de Cálculo de Data */
void data::initdata(int d, int m, int a)
{
        int dmes[  ] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        ano = a > 0 ? a : 1;
        dmes[2] = dmes[2] + bissexto(  );
        mes = m >= 1 && m <=12 ? m : 1;
        dia = d >= 1 && d <= dmes[mes] ? d : 1;
}

/* Função que Imprime a Data */
void data::imprimedata(  )
{
        char nome[13][10] =
                { “zero”, “Janeiro”, “Fevereiro”, “Março”,  “Abril”, “Maio”,
```

```
                "Junho", "Julho", "Agosto", "Setembro", "Outubro",
               "Novembro", "Dezembro" };

        cout << "\n\n" << dia << " de " << nome[mes] << " de " << ano;
}

/* Função que Imprime o Nome do Signo */
void data::imprimesigno( )
{
        char nsigno[14][12]=
               { "zero", "Aquário", "Peixes", "Áries", "Touro", "Gêmeos",
                 "Câncer", "Leão", "Virgem", "Libra", "Escorpião",
                "Sargitário", "Capricórnio" };

        int sig[ ] = { 0, 20, 19, 20, 20, 20, 20, 21, 22, 22, 22, 21, 21 };

        if (dia<sig[mes])
               cout << "n\Signo: " << nsigno[mes];
               else
                  cout << "\nSigno: " << nsigno[mes+1];
}

/* Função que Imprime Bissexto */
void data::imprimebissexto( )
{
        if(bissexto( ))
               cout << "\nAno Bissexto";
               else
                  cout << "\nAno Não-Bissexto";
```

```
}

/* Programa Principal */
void main( )
{
        data x(12, 1, 1976),
             y(30, 7, 1978),
             z(14, 6, 1992);

        clrscr( );

        /* Imprime valores referentes a variável x */
        x.imprimedata( );
        x. imprimesigno( );
        x. imprimebissexto( );

        /* Imprime valores referentes a variável y */
        y. imprimedata( );
        y. imprimesigno( );
        y. imprimebissexto( );

        /* Imprime valores referentes a variável z */
        z. imprimedata( );
        z. imprimesigno( );
        z. imprimebissexto( );

        getch( );
}
```

Note que no programa exemplo apresentado, foi criada uma classe de nome ***data*** que serviu de base para todo e qualquer processamento decorrido nele. A cláusula **private** é utilizada com o objetivo de definir as variáveis que são assim ditas próprias da classe. Já a cláusula **public** é utilizada para fazer definições de variáveis, funções que serão usadas em todo decorrer do programa. Note que sempre que um objeto é criado seus dados-membros são inicializados automaticamente e simultaneamente com os dados que são colocados entre parênteses e separados por vírgula (aqueles que aparecem ao lado do nome do objeto), no momento da declaração.

Observe atentamente no programa que o construtor **data** é executado toda vez que um objeto é efetivamente criado, cujos dados são passados em forma de argumentos. Neste caso o construtor efetuou chamadas de funções vinculadas a ele no programa. A estas funções dá-se tecnicamente o nome de função-membro. As linhas de código da função-membro são igual a qualquer outra função, porém essa função-membro não apresenta um tipo definido (não tem tipo – sem tipo).

Vejamos então um próximo exemplo de construtor que aparecerá sem argumento algum, ou seja, vazio:

```
/* Programa Exemplo com uso de Construtor Vazio */
#include <iostream>
#include <conio.h>
#include <iomanip.h>

/* Definição da Classe Venda */
class venda
{
        private :
                int numpeca;
                float precopeca;

        public :
                venda( );      // Definição do construtor vazio
```

```
                {       //Vazio
                }

        venda(int quant, float preco)
        {
                numpeca = quant;
                precopeca = preco;
        }

/* Função Realiza Venda */
void fazvenda(  )
{
        clrscr(  );

        cout << "Quantidade de Peças: ";
        cin >> numpeca;
        cout << "Valor da Peça: ";
        cin >> precopeca;
}

/* Função Mostra Venda */
void imprimevenda(  ) const;     //Identifica que o objeto criado não poderá ser
                                 //modificado pela função chamadora

/* Função Adiciona Nova Venda */
void somavenda( venda vend1, venda vend2 )
{
        numpreca = vend1.numpeca + vend2.numpeca;
        precopeca = vend1.precopeca + vend2.precopeca;
```

Não altera o objeto

```
}
};

/* Função Imprime Venda */
void venda::imprimevenda(  ) const
{
        cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint)
              << setprecision(2) << setw(10) << numpeca << "\n";
        cout << setw(10) << precopeca << "\n";
}

/* Corpo Principal do Programa*/
void main(  )
{
        venda A(50, 876.55),      //Atribuição direta de valores
              B,
              Total;

        B.fazvendas(  );     //Atribuição indireta de valores
        Total.somavenda(A, B);

        cout << "\nVenda A      ";
        A.imprimevenda(  );
        cout << "\nVenda B      ";
        B.imprimevenda(  );
        cout << "\nTotal          ";
        Total.imprimevenda(  );
}
```

Neste programa exemplo utilizamos um construtor vazio (sem parâmetros) e também, usamos a palavra-reservada **const** no final da declaração da função para indicar que esta função descrita não alterará em hipótese alguma o objeto do programa.

Vejamos então a seguir um próximo programa exemplo simulando um objeto chamado aluno:

```
/* Outro Programa Exemplo */
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#include <iomanip.h>

/* Definição da Classe Aluno */
class aluno
{
        private :
                int matr;
                char nome[35];
                float grau;

        public :
                aluno(int m, char n[35], float g)
                {
                        matr = m;
                        strcpy(nome, n);
                        grau = g;
                }
        /* Função Entrada */
```

```
        void entraaluno( )
        {
                clrscr( );

                cout << "\nMatrícula: ";
                cin >> matr;
                cout << "\nNome:";
                gets(nome);
                cout << "Grau: ";
                cin >> grau;
        }

        /* Função Saída */
        void saialuno( ) const;
};
```

```
/* Função Imprime Aluno */
void aluno::saialuno( ) const
{
        cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint)
             << setprecision(2) << setw(10) << matr << "\n";
        cout << nome << "\n";
        cout << setw(10) << grau << "\n";
}

/* Corpo Principal do Programa*/
void main( )
{
        aluno Al1,
```

```
        Al2,
        Al3;

    Al1.entraaluno( );
    Al2.entraaluno( );
    Al3.entraaluno( );

    clrscr( );

    cout << "\nAluno1: ";
    Al1.saialuno( );
    cout << "\nAluno2: ";
    Al2.saialuno( );
    cout << "\nAluno3: ";
    Al3.saialuno( );

    getch( );
}
```

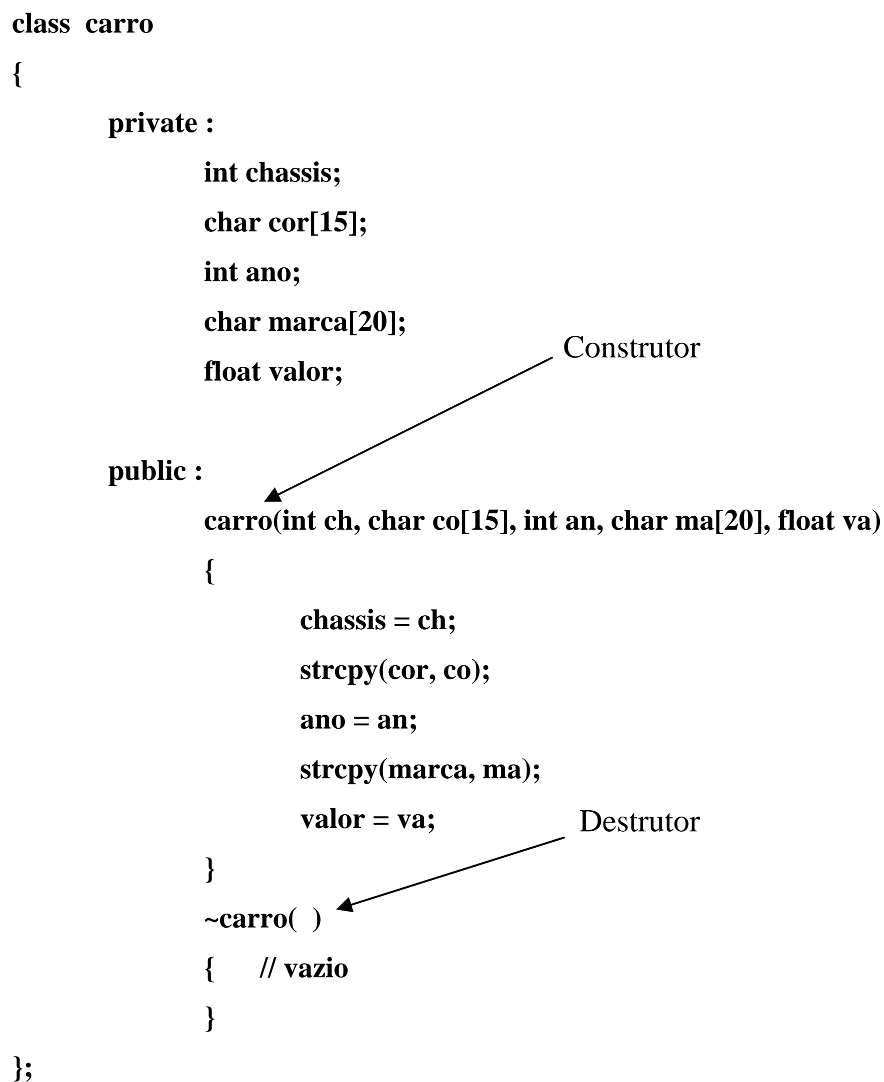
Neste caso o objeto criado **aluno** foi utilizado como referência para as três variáveis existentes no programa Al1, Al2 e Al3 (representativas de três alunos de uma certa turma).

DESTRUTORES EM C++

Bem, na verdade já sabemos que um construtor é automaticamente chamado sempre que um certo objeto é declarado. Mas imagine só a seguinte situação: Você vai criando novos objetos, e mais novos objetos, e mais novos objetos... Ufa! Acho que ta bom por demais não? Veja bem, cada objeto criado fica armazenado na memória do computador ocupando assim aquela quantidade em bytes necessários para sua subsistência. Porém veja bem: Se você não mais

utilizará aqueles objetos criados na memória do computador para que deixá-los lá? Neste caso existem os destrutores cuja função é liberar o espaço em memória ocupado por um certo objeto que não mais está sendo utilizado pelo programa.

Na linguagem C++ um destrutor é identificado colocando-se como prefixo do nome da classe criada o símbolo de til (~). Observe o exemplo apresentado abaixo:

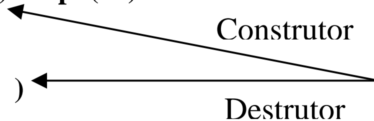



```
class  carro
{
        private :
                int chassis;
                char cor[15];
                int ano;
                char marca[20];
                float valor;

        public :
                carro(int ch, char co[15], int an, char ma[20], float va)
                {
                        chassis = ch;
                        strcpy(cor, co);
                        ano = an;
                        strcpy(marca, ma);
                        valor = va;
                }
                ~carro(  )
                {     // vazio
                }
};
```

Note que na definição do objeto **carro(PARÂMETROS) { Definição }** tivemos a descrição do construtor carro e logo em seguida, tivemos a descrição do destrutor **~carro( ) { Vazio }**. Assim, o destrutor no exemplo apresentado é chamado automaticamente quando se encerra o escopo dentro do qual a instância da classe foi declarada, que é então o momento em que ela será verdadeiramente excluída (destruída).

Vamos observar atentamente o programa exemplo abaixo com o uso efetivo de construtores e destrutores em C++:

```
/* Criação e Destruição de Objeto */
/* com Estrutura de Dados Complexa */
/* Uso de Pilha */
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>

class pilha
{
        private:
                int pil[100];
                int topo;
        public:
                pilha(  ) : topo( 0 )
                {  }
                ~pilha(  )
                {  }
```



```
                /* Função Limpa Pilha */
                void limpa(  )
                {  topo=0;  }
```

```
        /* Função Coloca na Pilha */
        void poe( int i )
        {
                if(topo<100)
                        pil[topo++] = I;
        }

        /* Função Retira da Pilha */
        int tira(  )
        {
                if(topo>0)
                        return pil[-topo];
                        else
                          return 0;
        }

        /* Função Tamanho da Pilha */
        int tamanho(  )
        {  return topo;  }
};

/* Corpo Principal do Programa */
int main(  )
{
        pilha pilha1,
              pilha2;

        clrscr(  );
```

```
        pilha1.poe(10);
        pilha1.poe(20);
        pilha1.poe(30);

        cout << pilha1.tira(  ) << endl;

        pilha2 = pilha1;

        cout << pilha1.tira(  ) << endl;
        cout << pilha2.tira(  ) << endl;
        cout << pilha1.tamanho(  ) << endl;
        cout << pilha2.tamanho(  ) << endl;

        return 0;
}
```

Note que uma classe utiliza tanto a quantidade em bytes na memória do computador quanto uma estrutura usaria para tratamento dos mesmos dados membro.

No nosso programa exemplo utilizamos o especificador especial **endl** (final de linha) que serve para identificar a finalização de uma certa linha em C++.

Na linguagem C existe o caracter especial '\0' para indicar o término de uma string. Isso pode ser feito em C++ através do especificador especial **ends** (final de string). Para utilizar tais especificadores é necessário a presença da biblioteca **iomanip.h**.

Nosso próximo programa exemplo também trata uma estrutura de dados complexa. Observe então na próxima página, como faremos o tratamento de dados em uma certa lista através do objeto **lista**.

```
/* Programa Exemplo com Estrutura de Dados Complexa */
/* Utilizando Lista */
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>

typedef struct      // Faz a definição do tipo estrutura
{
        char nome[40];
        char rua[15];
        int numero;
        char cidade[15];
        char estado[15];
}  enderStruct;

const int MAX = 100;

class List
{
        private:
                enderStruct list[MAX];
                int numList;
        public:
                List( ) : numList( 0 )
                { }

                ~List( )
                { }
```

```
int Cheio(  )
{
        if( numList >= MAX )
                return 1;
                else
                    return 0;
}

int Vazio(  )
{
        if( numList == 0 )
                return 1;
                else
                    return 0;
}

int Tamanho(  )
{
        return numList;
}

int Inseri( enderStruct ender )
{
        if( !Cheio(  ) )
        {
                list[numList++] = ender;
                return 0;
        }
        return 1;
```

```
		}
		int Ler( enderStruct& ender, int i )
		{
			if( i < Tamanho(  ) )
			{
				ender = list[i];
				return 0;
			}
			return 1;
		}
};

List list;

void inseriDados(  )
{
	enderStruct e;

	if( !list.Cheio(  ) )
	{
		clrscr(  );

		cout << "\nNome: ";
		cin >> e.nome;

		cout << "\nRua: ";
		cin >> e.rua;

		cout << "\nNúmero: ";
```

```
                cin >> e.numero;
                cout << "\nCidade: ";
                cin >> e.cidade;

                cout << "\nEstado: ";
                cin >> e.estado;

                list.Inseri(e);
        }
        else
            cout "\nA lista está cheia...\n\n";
}

void imprimeUmRegistro( enderStruct e )
{
        clrscr(  );

        cout << endl;
        cout << e.nome << endl;
        cout << e.rua << endl;
        cout << e.numero << endl;
        cout << e.cidade << endl;
        cout << e.estado << endl;
}

void imprimeDados(  )
{
        int i;
        enderStruct e;
```

```
        for( i=0; i< list.Tamanho; i++ )
        {
                list.Ler(e, i);
                imprimeUmRegistro(e);
        }

        cout << endl;
}

void procuraRegistro(  )
{
        char str[40];
        int i;
        int achei=0;
        enderStruct e;

        if( list.Tamanho(  ) == 0 )
                cout << "\nLista Vazia\n\n";
                else
                {
                        cout << "\nNome a Pesquisar: ";
                        cin >> str;

                        for( i=0; i<list.Tamanho(  ); i++ )
                        {
                                list.Ler(e, i);
                                if( strcmp(str, e.nome ) == 0 )
                                {
```

```
                        imprimeUmRegistro( e );
                        achei=1;
                    }
                }

                if( !achei )
                    cout << “\nRegistro Não Encontrado\n\n”;
            }
}

void mostraMenu(  )
{
    clrscr(  );

    cout << “* * *  M E N U  * * *\n\n”
    cout << “1-  Incluir\n”;
    cout << “2- Consultar\n”;
    cout << “3- Procurar Registro\n”;
    cout << “4- Finalizar\n\n”;
    cout << “Escolha sua opção: ”;
}

int main(  )
{
    char opção[10];
    int aux=0;

    while( !aux )
    {
```

```
        mostraMenu( );
        cin >> opção;

        switch( opção[0] )
        {
            case '1' :
                inserirDados( );
                break;
            case '2' :
                imprimeDados( );
                break;
            case '3' :
                procuraRegistro( );
                break;
            case '4' :
                aux=1;
                break;
            default :
                cout << "Opção Inválida.\n\n";
        }
    }

    return 0;
}
```

No nosso programa exemplo fazemos a manipulação de uma lista de dados (programação com estrutura de dados complexa). Nela fazemos o tratamento completo da estrutura de dados lista (inserir elemento na lista, ler elemento da lista, procurar elemento na lista, verificar tamanho da lista e etc.). Aqui criamos um construtor de nome **List( )** e, ao mesmo tempo, tivemos a necessidade de definirmos um destrutor de nome **~List( )**.

Utilizamos também o especificador **typedef** que permite ao compilador C++ fazer a definição de um tipo inexistente. Neste caso, a definição do tipo estrutura para o endereço de pessoas. A forma de interpretação e estruturação funciona como na linguagem de programação C.

HERANÇA EM C++

```
/* Programa Exemplo para tratamento de Herança */
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

class conta
{
        private:
                char nome[40];
```

```
            int numconta;
            float saldo;
      public:
            void entrada( )
            {
                  clrscr( );

                  cout << “\nNome do Cliente:”;
                  gets(nome);
                  cout << “\nNúmero da Conta:”;
                  cin >> numconta;
                  cout << “\nSaldo:”;
                  cin >> saldo;
            }

            void saida( )
            {
                  clrscr( );

                  cout << “\nNome do Cliente:” << nome << endl;
                  cout << “\nNúmero da Conta:” << numconta << endl;
                  cout << “\nSaldo:” << setiosflags(ios::fixed)
                      << setprecision(2) << saldo << endl;
            }

            float Saldo( )
            {  return saldo;  }
};
```

```
class contaSimples : public conta
{ };

class contaEspecial : public conta
{
	private:
		float limite;
	public:
		void entrada( )
		{
			conta::entrada( );
			cout << "\nLimite de Cheque Especial:";
			cin >> limite;
		}

		void saida( );
		{
			conta::saida( );
			cout << "\nLimite de Cheque Especial:" << limite;
			cout << "\nSaldo Total:" <<  << setiosflags(ios::fixed)
			     << setprecision(2) << (Saldo( ) + limite);
		}
};

class contaPoupanca : public conta
{
	private :
		float taxa;
```

```
        public:
                void entrada( )
                {
                        conta::entrada( );
                        cout << "\nValor da Taxa:";
                        cin >> taxa;
                }

                void saida( )
                {
                        conta::saida( );
                        cout << "\nValor da Taxa:" << taxa;
                        cout << "\nSaldo Total:" << setiosflags(ios::fixed)
                             << setprecision(2) << (Saldo( ) * taxa);
                }
};

void main( )
{
        int x;
        contaSimples cli1,
                     cli2;

        contaEspecial cli3,
                      cli4,
                      cli5;

        contaPoupanca cli6;
```

```
    /* Fornece com Dados dos Clientes */
    for( x=0; x<=6; x++ )
    {
    cout << "\nForneça os dados do cliente nº " << x+1;
    clix+1.entrada( );
    }

    /* Consulta Dados dos Clientes */
    x=1;
    while( x<=6 )
    {
        cout << "\nDados da Conta nº " << x;
        clix.saida( );
    }
}
```

No programa exemplo apresentado são declarados seis objetos que são baseados nas propriedades descritas pelas diferentes classes apresentadas: ***cli1***, ***cli2***, ***cli3***, ***cli4***, ***cli5*** e ***cli6***.

A biblioteca **<stdio.h>** foi utilizada para permitir o uso da função **gets( )** no programa, assim como ocorre na linguagem de programação C.

Note que a classe **conta** foi criada como uma classe padrão para manipulação de conta de clientes de um certo banco. Com base nos parâmetros definidos na classe **conta**, foram criadas as classes ***contaSimples***, contaEspecial e contaPoupanca. Isso só foi possível devido o princípio da Orientação a Objetos conhecido como **Herança** onde, uma classe herda características, ***métodos*** e ***propriedades***, de uma determinada classe classificada como classe pai. Logo, na verdade, a classe **conta** é considerada a classe **Pai**, aquela que dá origem a outras classes, e as classes **contaSimples**, **contaEspecial** e **contaPoupanca** são consideradas as classes **Filhas**, que só existem porque foram geradas com base em outra classes qualquer.

POLIMORFISMO EM C++

```
/* Programa Exemplo para tratamento de Polimorfismo */
#include <iostream.h>
#include <conio.h>

class pai      // Criação da classe Pai
{

        protected:      // Define como parâmetro protegido as variáveis j e k
                int j, k;

        public:
                void setJ(int new j);
                void setK(int new k);
                int getJ( );
                int getK( );
};
```

```
class filha:      // Criação da classe Filha
        public pai
        {
                protected:
                        int m, n;
                public:
                        void setM(int new m);
                        void setN(int new n);

                        int getM( );
                        int getN( );
        };

void pai::setJ(int new j)
{
        j = new j;
}

void pai::setK(int new k)
{
        k = new k;
}

int pai::getJ( )
{
        return j;
}

int pai::getK( )
{
        return k;
}

void filha::setM(int new m)
{
        m = newm;
}


void filha::setN(int newn)
{
        n = newn;
}
```

```
int filha::getM( )
{
        return m;
}

int filha::getN( )
{
        return n;
}

// Descrição da Função Principal
int main( )
{
        pai prnt, prntl;
        filha chld, chldl;

        prnt = chld;
        // chld = prnt; // error
        pai* pp = &chld;     // Pode acessar todas as 4 variáveis
        pai& rr = chld;
        // child* cc = &prnt; // error
        pp = &prntl;
        pp = &chldl;

        // Lista Heterogênea
        pai* list[4];
        list[0] = &prnt;
        list[1] = &chld;
        list[2] = &prntl;
        list[3] = &chldl;
}
```

No programa exemplo apresentado são definidos objetos, variáveis e funções que irão interagir diretamente com uma lista heterogênea que é criada para tratamento das variáveis existentes no programa.

Cuidado ao declarar a função ***main( )***, pois ela é especificada como inteira. A não atenção no momento da escrita do programa exemplo, pode ocasionar em erro na execução do programa proposto.

O prâmetro ***protect*** na definição da classe **pai** foi utilizado para determinar que as variáveis j e k ligadas a esta classe estarão provativas e ao mesmo tempo, protegidas de qualquer variação de valor.

Note que no momento da criação da classe **filha**, por meio de **Herança**, as ***propriedades*** e os ***métodos*** pertencentes a classe **pai** passam a ser assumidos pela classe **filha**.

SOBRECARGA DE OPERADOR

Bem, a linguagem C++ permite ao programador um tipo de operação especial onde através do uso de novos tipos de dados haja efetivamente a sobrecarga de operador. Mas afinal de contas o que realmente é sobrecarga de oprrador? Verdadeiramente, trata-se da possibilidade de transformação de expressões ditas como complexas (não muito claras para o programador) em expressões bem mais simplificadas e entendíveis. Isso já não ocorre na maioria das linguagens de programação tradicionais (podemos até considerar uam deficiência existente nelas).

É verdadeiro afirmar que a sobrecarga de operador é um conceito que, no nosso caso, pode ser aplicável para as classes e estruturas de um programa C++. Esse será o foco principal de nossos estudos. Vejamos então o comportamento do clássico programa exemplo de sobrecarga de operador:

```
/* Exemplo Prático do Uso de Sobrecarga de Operador */
#include <iostream.h>
#include <conio.h>

class ref
{
        private:
                int a,
                    b;
```

```
        public:
                ref(int x=0, int y=0)
                {
                        a=x;
                        b=y;
                }

                ref operador++( )
                {
                        ++a;
                        ++b;

                        ref aux;      // Cria um objeto auxiliar para apoio de
                                      // Sobrecarga do operador de incremento
                        aux.a = a;
                        aux.b = b;

                        return aux;
                }

                void imprimeref( ) const
                {
                        clrscr( );
                        cout << '(' << a << ',' << b << ')';
                }
};

/* Corpo Principal */
void main( )
```

```
{
        ref ref1,
            ref2(2, 3),
            ref3;

        cout << "\n ref1 = ";
        ref1.imprimeref( );
        cout << "\n ref2 = ";
        ref2.imprimeref( );

        cout << "\n++ref1 = ";
        (++ref1).imprimeref( );

        cout << "\nref2 = ";
        (++ref2).imprimeref( );

        ref3=++ref1;
        cout << "\n ref3 = ";
        ref3.imprimeref( );
}
```

Como podemos verificar foi criado um objeto auxiliar baseado na classe **ref** para ser utilizado como valor de retorno. Isto ocorre especificamente na função **operador**++( ) descrita no programa.

Exercícios

1- Defina Classe.
2- O que é ou representa um objeto?
3- Defina UML.

4- Diferencie objeto e classe.

5- Qual o objetivo de utilizar a técnica de sobrecarga de operador?

6- Escreva um programa em C++, baseado em uma Vídeo Locadora, cujo objetivo é trabalhar classes simples e classes derivadas a partir do princípio da orientação a objetos denominado Herança.

7- O que você entende por Programação Orientada a Objetos?

8- Podemos escrever um programa em C++ baseado na orientação a objetos para manipulação e tratamento de estruturas de dados complexas como Lista e Pilha, por exemplo? Justifique sua resposta.

9- Defina Construtor. Exemplifique.

10- Defina Destrutor. Exemplifique.

11- Quais os princípios da Linguagem de Modelagem Unificada?

12- Escreva um programa em C++ para efetuar uma sobrecarga de operador baseado no operador de decremento (--).

13- O que é sobre carga de operador?

14- Defina encapsulamento.

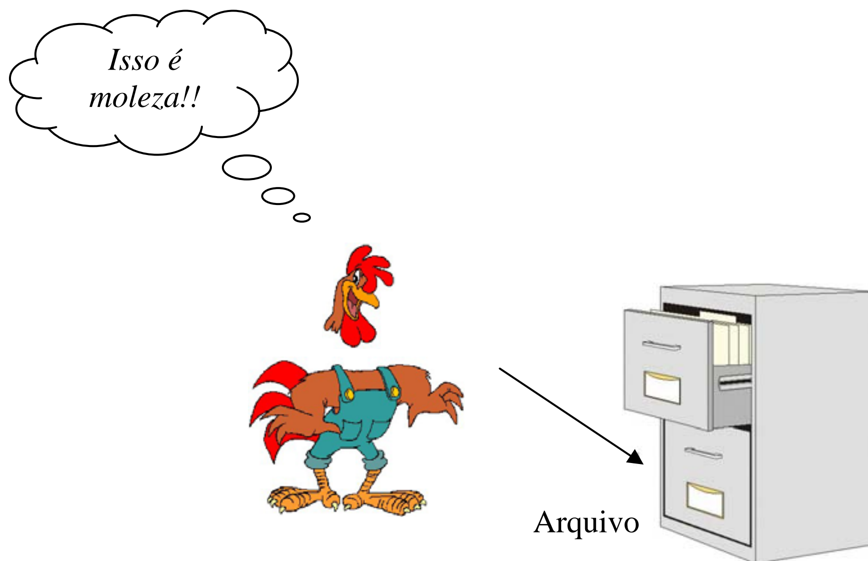
15- Exemplifique polimorfismo.

13

MANIPULANDO ARQUIVOS

Bem, nosso objetivo agora é estudar o armazenamento de dados em dispositivos de armazenamento permanente. Até então, aprendemos a guardar dados em dispositivos de memória temporários (arrays ou vetores). Contudo, essa prática não é perfeita para o armazenamento de dados por um longo período de tempo. A linguagem de programação C++ utiliza o conceito de ***stream*** para esta prática.

Na realidade um ***stream*** que nada mais é que a via de comunicação que é estabelecida entre o remetente e o destinatário da mensagem.



OS OBJETOS STREAM

Na linguagem C++ os chamados objetos ***stream*** são componentes integrantes da classe **iostream**, utilizada nos programas exemplos abordados neste livro. Esses objetos servem de ponto referencial para envio e recebimento de dados (bytes). Na verdade, um arquivo em disco armazena um conjunto de caracteres por meio do objeto ***stream***. Observe atentamente o diagrama hierárquico de classes ***stream*** apresentado a seguir:

ios

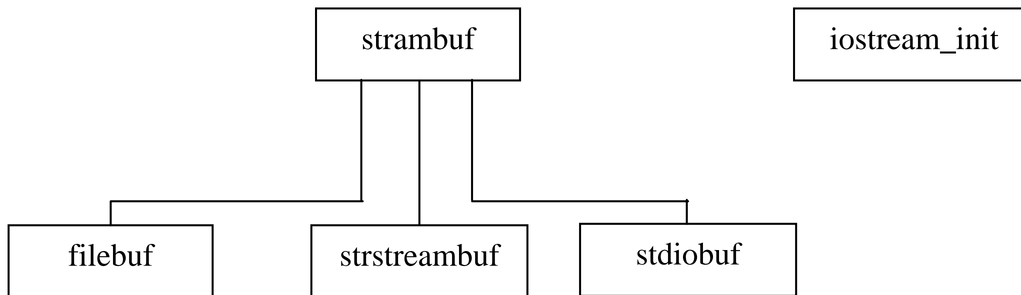
istream
ostream

istrstream
ifstream
ofstream
ostrstream

istream_withassign
Ostream_withassign

iostream

fstream
strstream
stdiostream



Veja a seguir a tabela de relação das classes do diagrama hierárquico com as bibliotecas do C++ (***iostream.h***, ***fstream.h***, ***strstream.h*** e ***stdiostream.h***):

iostream.h	**fstream.h**	**strstream.h**	**stdiostream.h**
ios	filebuf	istrstream	stdiobuf
streambuf	ifstream	ostrstream	stdiostream
istream	fstream	strstream	
ostream	ofstream	strstreambuf	
iostream			
istream_withassign			
ostream_withassign			
iostream_init			

TRABALHANDO COM ARQUIVOS TEXTO

Dizemos que um arquivo aberto de modo texto é interpretado pelo C++ como uma sequencia de caracteres agrupados em forma de linha. Essas linhas são separadas por um único caracter chamado tecnicamente de caracter de nova linha ou simplesmente LF.

Utilizamos arquivos do tipo texto principalmente quando queremos armazenar uma carta redigida pelo processo de gravação que, tecnicamente pode ocorre de duas formas distintas: caracter por caracter ou linha a linha. Como foi dito, são processos distintos que permitem a leitura dos dados desse arquivo da mesma forma, ou seja, caracter por caracter e linha a linha. A seguir, iremos estudar detalhadamente cada um desses processos separadamente.

GRAVANDO CARACTER A CARACTER NO ARQUIVO

```
/* Programa Exemplo de Gravação Caracter a Caracter no Arquivo */
#include <stream.h>
void main( )
{
        ofstream fout("TESTE.TXT");
        char c;

        while(cin.get(c) != '\x1b')      // Pressionamento da tecla ESC
                fout.put(c));
}
```

Usamos o objeto chamado **fout** da classe ***ofstream***. Com isso nós tivemos a possibilidade de gravar caracter a caracter no arquivo denominado TESTE.TXT. Verifique que no loop while( ) foi utilizado o caracter especial **'\x1b'** que representa a tecla ESC (escape). A instrução ***cin.get( )*** realiza a leitura do caracter digitado que será gravado através da instrução **fout.put**( ). Vejamos a seguir como efetuar a leitura dos dados gravados no arquivo linha a linha.

LENDO CARACTER POR CARACTER DO ARQUIVO

```
/* Programa Exemplo de Leitura Caracter a Caracter do Arquivo */
#include <stream.h>
void main( )
```

```
{
        ifstream fin("TESTE.TXT");
        char c;

        while(fin.get(c))
                cout << c;
}
```

Usamos o objeto chamado **fin** da classe ***ofstrem***. Com isso nós tivemos a possibilidade de ler caracter a caracter no arquivo denominado TESTE.TXT. Verifique que no loop while( ) foi utilizada a instrução ***fin.get( )*** realiza a leitura do caracter gravado no arquivo.

Muitoooooo Fácilllllll!!

Vejamos a seguir como graver e ler linha a linha.

GRAVANDO LINHA A LINHA NO ARQUIVO

```
/* Programa Exemplo de Gravação de Dados Linha a Linha no Arquivo */
#include <fstream.h>

void main( )
{
        ofstrem fout("TESTE.TXT");
```

```
        fout << “Nome: Juan Gabriel\n”;
        fout << “Nascimento: 23/Jan/2004\n”;
        fout << “Signo: Aquário\n”;
}
```

Usamos o objeto chamado **fout** da classe ***ofstrem***. Com isso nós tivemos a possibilidade de gravar linha a linha no arquivo denominado TESTE.TXT. Vejamos a seguir como efetuar a leitura dos dados gravados no arquivo linha a linha.

LENDO LINHA A LINHA DO ARQUIVO

```
/* Programa Exemplo de Leitura de Dados Linha a Linha no Arquivo */
#include <fstream.h>

void main( )
{
        ofstrem fout(“TESTE.TXT”);

        char buff[80];

        ifstream fin(“TESTE.TXT”);

                while(fin)      // Enquanto houverem dados a serem lidos
                {
                        fin.getline(buff, 80);      // Lê uma linha do texto
                        cout << buff << ‘\n’;
                }
}
```

Usamos o objeto chamado **fin** da classe ***ofstrem***. Com isso nós tivemos a possibilidade de ler caracter a caracter no arquivo denominado TESTE.TXT. Verifique que no loop while( ) foi utilizada a instrução ***fin*** realiza a leitura da linha gravada no arquivo. A função **getline( )** lê caracteres gravados linha a linha e aceita um terceiro argumento que representa o caracter terminador da leitura. Quando esse argumento não é especificado, por padrão, a linguagem C++ assume o caracter '\n'.

TRABALHANDO COM ARQUIVOS BINÁRIOS

A manipulação feita com arquivos binários é conhecida como arquivos não bufferizados ou baixo nível, pois obriga o programador a criar e manter o buffer de dados para as manipulações de leitura e gravação.

GRAVAÇÃO E LEITURA DE OBJETOS NO ARQUIVO

```
/* Programa Exemplo de Gravação de Dados em Arquivo Binário */
#include <fstream.h>
#include <stdio.h>
#include <conio.h>

class Livro
{
        private:
                char titulo[40];
                char autor[30];
                char editora[20];
                int isbn;

        public:
                void entrada( );
```

```
            void saida( );
};

void Livro::entrada( )
{
      clrscr( );

      cout << "\nTítulo do Livro:";
      gets(titulo);

      cout << "\nNome do Autor:";
      gets(autor);

      cout << "\nEditora:";
      gets(editora);

      cout << "\nNº ISBN:";
      cin >> isbn;
}

void Livros::saída( )
{
      clrscr( );

      cout << "\nTitulo do Livro:" << titulo;
      cout << "\nNome do Autor:" << autor;
      cout << "\nEditora:" << editora;
      cout << "\nNº ISBN:" << isbn;
}
```

```
void main( )
{
        fstream arqes;
        Livro book;

        arqes.open("livro.dat", ios::ate | ios::out | ios::in);     // a | significa OU

        do{
                book.entrada( );
                arqes.write((char *)&book, sizeof(Livros));
                cout << "\n\nCadastra outro livro? (s/n)";
        }while(getche( )!='n');

        arqes.seekg(0);     //Coloca o ponteiro no topo do arquivo

        clrscr( );

        cout << "\nRelação de Livros";

        while(arqes.read((char *)&book, sizeof(Livro)))
                book.saida( );

        getch( );     // Pausa temporária até que uma tecla seja pressionada
}
```

A instrução **open( )** um membro da classe fstream, numa instrução de criação de objeto. Na verdade, tanto o construtor quanto a função open( ) aceitam a inclusão de um possível segundo argumento que é indicado através do modo de abertura do arquivo. Utilizamos **ate** para acrescentar registros novos ao final do arquivo sem a necessidade de apagar aqueles já existentes. Também, usamos **out** e

in, pois na verdade, desejamos executar duas operações possíveis: ***gravação*** e ***leitura*** de registros.

A instrução **arqeswrite( )** foi utilizada para efetuar a gravação dos dados do livros que foram fornecidos na função de entrada por meio do usuário.

O uso da função **seekg( )** permite movimentar o ponteiro de registros do arquivo para a posição de leitura desejada, neste caso, ***seekg(0)*** para o topo do arquivo. Existe também a função **seekp( )** que executa basicamente a mesma tarefa que a função ***seekg( )***, sendo que esta é utilizada para gravação. Ainda, se você quisesse poderia ter acesso aos recursos das funções ***tellg( )*** e ***tellp( )***. A função **tellg( )** permite retornar a opsição corrente de leitura, em bytes, sempre a partir do início do arquivo. Já a função **tellp( )** faz a mesma tarefa sendo ela utilizada para o processo de gravação.

A instrução **arqesread( )** foi utilizada para realizar a leitura dos dados gravados no arquivo ***livro.dat*** para serem exibidos através da função saída( ).

A especificação do modo de abertura **ios::binary** identifica que o arquivio que está sendo tratado é do tipo binário.

ARGUMENTOS VÁLIDOS DA FUNÇÃO MAIN( )

Na linguagem C++, da mesma forma interpretada na linguagem C, os únicos dois argumentos que são aceitos pela função main( ) são os **argc** e **argv**. O argumento ***argc*** é dado a partir de uma variável inteira. Ele serve para referenciar o número de entradas ocorridas através da linha de comandos quando um certo programa é carregado e, através dele são digitados argumentos.

O ***argv*** é a partir de uma variável vetor de ponteiro de caracter. Serve para armazenar os valores que são passados como argumentos na execução de um certo programa.

Vejamos como fica representado através do programa exemplo a seguir:

```
/* Programa Exemplo de Gravação de Dados em Arquivo Binário */
#include <fstream.h>
#include <stdio.h>
#include <conio.h>

void main(int argc, char **argv)
{
	ifstrem fin;
	char c;
	int x=0;

	if(argc != 2)
	{
		cout << "\nSintaxe: C:\>FILEX nomearq";
		exit(1);
	}
	fin.open(argv[1]);

	while(fin.get(c))
		x++;

	cout << "\nX= " << x;
}
```

Não se esqueça de que ao chamar a execução do arquivo, através da linha de comandos, você deverá fornecer os parâmetros necessários para a execução correta do programa.

Exercícios

1- Defina objeto stream.

2- Diferencie arquivo texto de arquivo binário.

3- Quais os arqgumentos válidos da função main( )?

4- Diferencie argc e argv.

5- Escreva um programa em C++ que permita ao usuário criar um diário que seja capaz de gravar e ler as informações nele armazenadas. Utilize para isto um Menu de Opções onde, [1] Grava Texto, [2] Lê Mensagem e [3] Finalização.

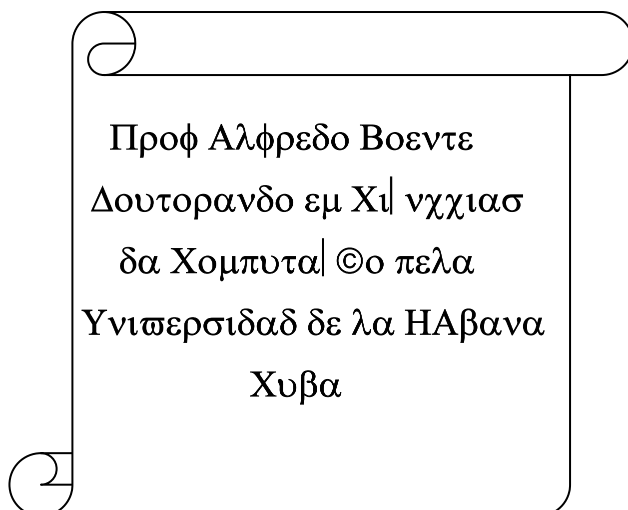
6- Crie um programa que manipule um arquivo chamado ALUNOS.DAT onde, deverão ser manipuladas as operações básicas de arquivamento: Inclusão, Consulta, Alteração e Remoção.

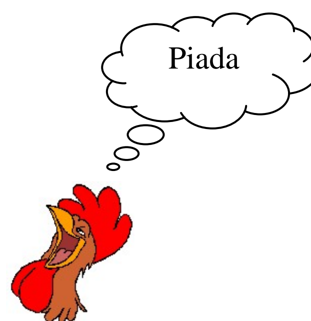
14

CRIPTOGRAFIA EM C++

Embora ninguém saiba realmente quando iniciou o processo de escrita secreta, é afirmado por [*Schildt, 89*] que um dos exemplos mais antigos, o tablete cuneiforme feito por volta de 1500 a.C.

Esse tablete cuneiforme contém uma fórmula codificada para fazer cobertura vitrificada em cerâmica. Os gregos e espartanos usavam códigos em 475 a.C., e a classe alta romana freqüentemente usava cifras simples durante o reino de Julio César.





Durante a Idade Média, o interesse pela criptografia (bem como muitas outras áreas intelectuais) diminuiu, exceto entre os monges, que as usavam ocasionalmente. Com a vinda do renascimento italiano, a arte da criptografia novamente floresceu.

Na época de Luis XIV da França, um código baseado em 587 chaves randomicamente selecionadas era usado em mensagens governamentais. Por volta de 1800, dois fatores ajudaram no desenvolvimento da criptografia. Primeiro eram as estórias de Edgard Allan Poe, tais como *THE GOLD BUG*, que apresentava mensagens em código e excitava a imaginação dos leitores. O segundo foi a invenção do telégrafo e do código Morse. O código Morse foi a primeira representação binária (pontos e traços) do alfabeto que teve grande aplicação.

Ao chegar a primeira guerra mundial, várias nações construíram "máquinas de codificação" mecânicas que permitam facilmente codificar e decodificar textos usando cifragens sofisticadas e complexas.

Aqui a história da criptografia dá uma leve virada para a história da decifragem de códigos. Antes da utilização de dispositivos mecânicos para codificar e decodificar mensagens, cifragens complexas eram raramente usadas por causa do tempo e do trabalho necessário para codificar e decodificar. Assim, a maioria dos códigos eram quebrados num período relativamente curto de tempo.

Logo, é possível a partir de qualquer linguagem de computador criar um processo de cifragem de códigos. O programa em linguagem C abaixo, permite que

você seja capaz de codificar qualquer texto usando qualquer descolamento após especificar com qual letra começará o alfabeto.

Segue abaixo um programa exemplo da utilização prática do processo de criptografia:

```
/* Exemplo Prático de Criptografia em C++ */

#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
   FILE *arch;

   char encriptado[100],
        desencriptado[100],
        r,
        pas[11];

   int longitud,i;

   arch=fopen("Dados.txt","w");

   clrscr( );

   cout << "\nEscreva o texto que deseja criptografar:";
```

```
cin.getline(encriptado,100);

longitud = strlen(encriptado);

for (i=0;i<longitud;i++)
{
  encriptado[i] = char(toascii(encriptado[i])+122);
}

fprintf(arch,"%s",encriptado);

cout << "\n\nSeu texto criptografado fica assim: " << encriptado;

cout << "\n\nDeseja descriptografar seu texto? (s/n) ";
cin >> r;

if (toupper(r)=='S')
{
  cout<<"\n\nEntre com a senha para descriptografar seu texto: ";
  cin >> pas;

  if (strcmp(pas,"piresboente")==0)
  {
    cout << "\n\nSeu texto descriptografado fica assim: ";


    for(i=0;i<longitud;i++)
    {
      desencriptado[i]=char(encriptado[i])-122;
```

```
      cout << desencriptado[i];
    }
  }

  else
  {
    cout << "\n\n\nLamento acesso negado...";
    getch( );
  }
}

else
  cout << "\n\nObrigado por utilizar esse programa...";
  getchar( );

fclose(arch);

return(0);
}
```

Note que através desse programa exemplo podemos realizar dois processos de simples manipulação: A criptografia e a descriptografia (função inversa da criptografia). Uma questão interessante que ocorre é o fato de que o usuário deve validar uma senha para que o texto criptografado seja então, efetivamente, descriptografado.

Você pode aplicar essa teoria na prática quando precisar criar um programa que execute esse tipo de função.

Exercícios

1- O que é criptografar?

2- O que é descriptografar?

3- Onde e quando iniciou a criptografia?

4- Escreva um programa em C++ capaz de criptografar uma certa mensagem que será fornecida pelo usuário através do teclado.

5- Escreva um programa em C++ que seja capaz de descriptografar uma mensagem criptografada anteriormente, mediante a validação de uma certa senha de permissão.