UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha solteira

Trabalho de Formatura

Apostila da Linguagem de Programação C++

Autor: Enrique Camargo Trevelin

Orientador: Carlos Antônio Alves

Ilha Solteira – Julho de 2007

Sumário

Módulo 1 – A Linguagem C++

1.1 – História da Linguagem C/C++

O C++ foi inicialmente desenvolvido por Bjarne Stroustrup durante a década de 1980 com o objetivo de melhorar a linguagem de programação C, mantendo a compatibilidade com esta linguagem. Stroustrup percebeu que a linguagem Simula possuía características bastante úteis para o desenvolvimento de software, mas era muito lenta para uso prático. Por outro lado o BCPL era rápido, mas possuía baixo nível, dificultando sua utilização em desenvolvimento de aplicações. Durante seu período na Bell Labs, ele enfrentou o problema de analisar o kernel UNIX com respeito à computação distribuída. A partir de sua experiência de doutorado, começou a acrescentar elementos do Simula no C.

C foi escolhido pois possuía uma proposta de uso genérico, era rápido e também portável para diversas plataformas. Algumas outras linguagens que também serviram de inspiração para o informático foram ALGOL 68, Ada, CLU e ML. Novas características foram adicionadas, como funções virtuais, sobrecarga de operadores e funções, referências, constantes, controle de memória pelo usuário, melhorias na checagem de tipo e estilo de comentário de uma linha (//). A primeira versão comercial da linguagem C++ foi lançada em outubro de 1985.

1.2 – Características da Linguagem C++

O principal desenvolvedor da linguagem C++, Bjarne Stroustrup, descreve no livro "In The Design and Evolution of C++" quais seus principais objetivos ao desenvolver e expandir esta linguagem:

- Em proposta geral, C++ deve ser tão eficiente e portável quanto C, sendo desenvolvida para ser uma linguagem com tipos de dados estáticos.
- C++ é desenvolvido para ser o quanto mais compatível com C possível, fornecendo transições simples para código C.
- C++ é desenvolvido para suportar múltiplos paradigmas de programação, principalmente a programação estruturada e a programação orientada a objetos, possibilitando múltiplas maneiras de resolver um mesmo problema.
- C++ é desenvolvido para fornecer ao programador múltiplas escolhas, mesmo que seja possível ao programador escolher a opção errada.

1.3 – Exemplos de Aplicações Escritas em C++

Abaixo temos alguns exemplos de aplicações e programas comerciais desenvolvidos totalmente ou parcialmente em C++.

- Grande parte dos programas da Microsoft, incluindo Windows XP, Windows NT, Windows 9x, Pacote Office, Internet Explorer, Visual Studio e outros.
- Sistemas Operacionais como o já citado Windows, Apple OS X, BeOS, Solaris e Symbian (sistema operacional para celulares).
- Bancos de dados como SQL e MySQL.
- Aplicações Web, como a máquina de busca Google e o sistema de comércio virtual da Amazon.
- Aplicações gráficas como os programas da Adobe (Photoshop, Illustrator), Maya e AutoCAD.
- Jogos em geral, como o Doom III.

A lista é enorme e poderia se estender por muitas e muitas páginas. Atualmente C++ é, juntamente com Java, a linguagem de programação comercial mais difundida no mundo.

1.4 – Comparação de C++ com outras linguagens

Podemos dividir as linguagens de programação conforme o "dialeto" utilizado por elas. Quanto mais próximo da linguagem humana for a maneira com que passamos instruções para a máquina, mais alto será seu nível: por exemplo, Fortran e Basic são consideradas linguagens de alto nível pois seus comandos parecem-se com frases humanas (em inglês, claro): "Se x = y então faça x = x+1 e imprima y". De maneira análoga, quanto mais próximo da linguagem da máquina for a linguagem de programação, mais baixo será seu nível: por exemplo, o Assembly é considerada uma linguagem de nível baixo, pois seus comandos são escritos em hexadecimal. Ambos os tipos possuem vantagens e desvantagens, mas de maneira geral podemos dizer que a vantagem das linguagens de nível alto é a simplicidade de programação, enquanto que a vantagem das linguagens de nível baixo é a alta velocidade que seus programas podem ter.

Tanto C como C++ podem ser consideradas linguagens de nível intermediário, pois utilizam-se de um dialeto de nível alto mas possibilita ao programador facilidades para se trabalhar em nível baixo, como manipulação de bits, bytes e endereços de memória de maneira direta, sem recorrer a abstrações apresentadas por outras linguagens de alto nível.

A filosofia que existe por trás da linguagens C e C++ é que o programador sabe realmente o que está fazendo. Estas linguagens quase nunca colocam-se no caminho do programador, deixando-o livre para usá-la de qualquer forma que queira, mas arcando com

as consequências de seu mau ou incorreto uso. O motivo para essa "liberdade na programação" é permitir ao compilador criar códigos muito rápidos e eficientes, deixando a responsabilidade da verificação de erros para o programador. O próprio criador de C++, Bjorne Stroustrup afirma que "C faz com que dar um tiro no pé seja fácil; C++ torna isso mais difícil, mas quando nós o fazemos arrebenta com a perna toda". A citação de Stroutrup trata com humor o fato de o C++, ao possibilitar a programação de alto nível, ter facilitado a codificação de algoritmos e organização de projetos em relação ao C, uma linguagem que requer constante atenção contra erros lógicos de programação devido à sua alta flexibidade. Por outro lado, o C++ possui nuances da sintaxe e semântica da linguagem muito sutis, difíceis de serem identificados, e que quando não percebidos podem levar a comportamentos indesejados no código.

As principais vantagens e desvantagens do C++ são listadas a seguir:

Vantagens

- Possibilidade em programação de alto e baixo nível.
- Alta flexibilidade, portabilidade e consistência.
- Compatilidade com C, resultando em vasta base de códigos.
- Adequado para grandes projetos.
- Ampla disponibilidade e suporte, devido principalmente à grande base de desenvolvedores.
- Não está sob o domínio de uma empresa (em contraste do Java - Sun ou Visual Basic – Microsoft).
- Padronização pela ISO.
- Grandes possibilidades para a metaprogramação e programação genérica.

Desvantagens

- Compatilidade com o C herdou os problemas de entendimento de sintaxe do mesmo.
- Os compiladores atuais nem sempre produzem o código mais otimizado, tanto em velocidade quando tamanho do código.
- Grande período para o aprendizado.
- A biblioteca padrão ainda não cobre áreas importantes da programação, como threads, conexões TCP/IP e manipulação de sistemas de arquivos, o que implica na necessidade de criação de bibliotecas próprias para tal, que pecam em portabilidade.
- Devido à grande flexibilidade no desenvolvimento, é recomendado o uso de padrões de programação mais amplamente que em outras linguagens.

1.5 – Paradigmas de Programação: Programação Estruturada e Orientada a Objetos

Um paradigma de programação é um conjunto de idéias que fornecem ao programador uma visão sobre a estruturação e execução de um programa. Assim como ao resolver um problema podemos adotar uma entre variadas metodologias para resolvê-lo, ao criar um programa podemos adotar um determinado paradigma de programação para desenvolvê-lo. Certas linguagens de programação são escritas especificamente para trabalhar com um tipo de paradigma: este é o caso de Smalltalk e Java que suportam a

programação orientada a objetos. Outras linguagens suportam vários paradigmas, ou seja, o programador pode escolher qual paradigma se adapta melhor ao problema que ele precisa resolver e trabalhar com ele, e até mesmo alternar entre paradigmas (desde que ele saiba o que está fazendo).

A linguagem C utiliza o paradigma da programação estruturada. A base da programação estruturada é trabalhar a lógica do programa como uma estrutura composta de similares sub-estruturas, reduzindo a compreensão do programa à compreensão de cada sub-estrutura individualmente. Na prática, este método envolve a criação de várias funções dentro de um programa, pequenas e simples o suficiente para serem entendidas individualmente, sendo o programa a sequência de todas estas funções trabalhando em conjunto. A programação estruturada se opõe ao uso de comandos de pulo como "GOTO", preferindo a criação de estruturas e condições lógicas que substituam ou mesmo eliminem a necessidade de um comando de pulo. Este paradigma é o mais utilizado no ensino e aprendizado de linguagens de programação, por ser mais facilmente entendido por estudantes e por criar hábitos de programação saudáveis e úteis mesmo em outros paradigmas.

A linguagem C++ é uma das linguagens que suportam vários paradigmas. Inicialmente, sendo uma "evolução" de C, ela suporta inteiramente o paradigma da programação estruturada. Além disso, ela suporta outros paradigmas como a programação procedural, a programação genérica, abstração de dados e a programação orientada a objetos. Dentre estes paradigmas, o mais utilizado atualmente é a Programação Orientada a Objetos, ou mais comumente chamado de OOP (Object-Oriented Programming). Apesar de ter sido criada nos anos 60, este paradigma só começou a ganhar aceitação maior após os anos 90,com a explosão das linguagens C++, Java e Visual Basic. A idéia básica por trás da OOP é criar um conjunto de "objetos" (unidades de software) para modelar um sistema. Estes objetos são independentes entre si, possuindo responsabilidades e funções distintas no programa como um todo, mas que se comunicam entre si através do envio e recebimento de mensagens. A OOP é especialmente útil para grandes programas que se beneficiam mais com a modularidade oferecida por este paradigma: dividindo o programa em vários módulos independentes, aumenta-se a flexibilidade e a facilidade para manutenção do programa como um todo.

Nesta apostila, enfocaremos os aspectos de linguagem de programação estruturada da linguagem C++, deixando os aspectos de linguagem orientada a objetos para os últimos capítulos a título de introdução ao assunto. Isto se deve a maior dificuldade de aprendizado e entendimento do paradigma da programação orientada a objetos, principalmente tratando-se de estudantes com pouco contato com linguagens de programação. Por isso, é preferível estabelecer uma base para o estudante com os conceitos da programação estruturada, que são mais facilmente compreendidos e trabalhados, para que depois este estudante possa progredir para o paradigma da OOP com maior facilidade.

Módulo 2 – Compiladores

2.1 – O que é um Compilador

Toda linguagem de programação possui um tradutor de código. Este tradutor pode ser um compilador ou um interpretador, dependendo da linguagem. Interpretadores são programas que leêm o código-fonte e executam ele diretamente, sem a criação de um arquivo executável. Chamamos de compilador o programa que traduz um arquivo escrito em código de linguagem de programação (arquivo-fonte) para a linguagem do microprocessador, criando um arquivo capaz de executar as instruções pedidas (arquivo executável).

O primeiro passo de um compilador é analisar o código presente no arquivo-fonte, verificando se existem erros de sintaxe. Caso algum erro de sintaxe seja encontrado, a compilação é interrompida para que o programador possa corrijir estes erros. Caso o código não possua erros o próximo passo do compilador é criar um arquivo de código-objeto, que possui as instruções do programa já traduzidas para a linguagem da máquina e informações sobre alocação de memória, símbolos do programa (variáveis e funções) e informações de debug. A partir deste arquivo de código-objeto, o compilador finalmente cria um arquivo executável com o programa compilado, que funciona independente do compilador e realiza as instruções criadas pelo programador.

2.2 – Compiladores de C++

Existem muitos compiladores de C++ no mercado. Os mais famosos são os softwares da Borland e da Microsoft, que são realmente muito bons e oferecem muitos recursos. O problema é que estes compiladores são caros e voltados principalmente para programadores experientes, que podem fazer uso dos recursos avançados destes programas. Para quem não está ainda aprendendo a linguagem e não quer ainda gastar dinheiro com compiladores, existem várias opções de compiladores freeware (software livre, “de graça”). Nesta seção descreveremos a instalação e o uso do DevC++, um compilador freeware muito utilizado.

2.3 – DevC++

O Dev-C++ é um compilador freeware das linguagens C, C++ e C#. É uma opção muito interessante, pois é de fácil utilização e aprendizado para usuários novos e possui muitos recursos avançados para usuários experientes. Além de, claro, seu download ser gratuito.

2.3.1 - Instalação

A versão mais recente do DevC++ pode ser baixada através da página http://www.bloodshed.net/dev/devcpp.html, no link "Download". Utilizou-se, na elaboração desta apostila, a versão do DevC++ beta 9.2, disponível diretamente através do link http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2_setup.exe. O arquivo de instalação possui aproximadamente 9 megas. Após o fim do download, é preciso clicar duas vezes neste arquivo para começar a instalação.

A instalação do DevC++ é bem simples. Utilizaremos nesta apostila a instalação completa, escolhendo o item full durante a etapa da instalação mostrada na figura 2.1.

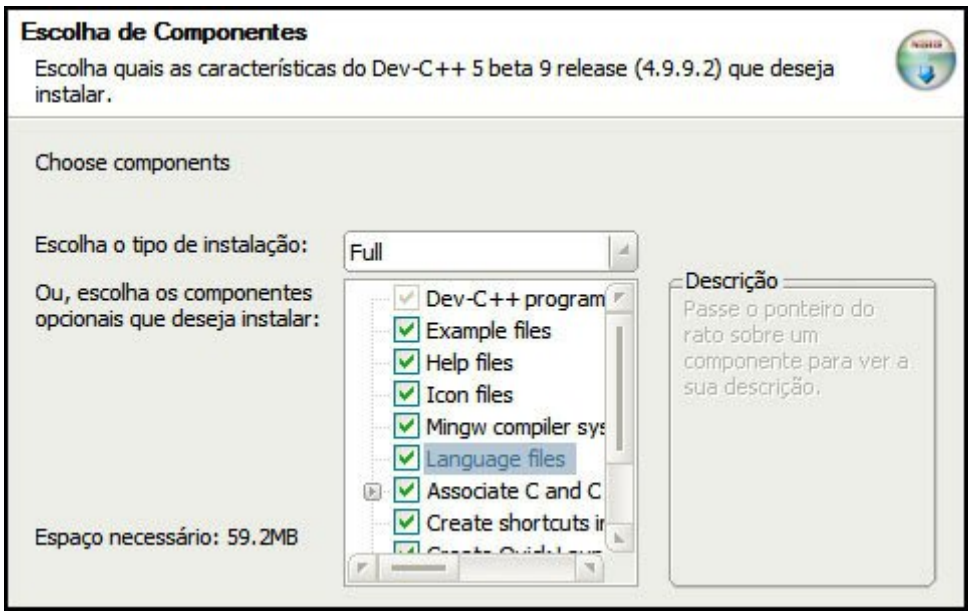

Figura 2.1 – Escolha o tipo de instalação "Full"

O próximo passo é escolher o diretório onde o programa será instalado. Neste diretório estarão todos os arquivos relacionados ao compilador, como bibliotecas, executáveis, arquivos de ajuda e muitos outros. Além disso, é neste diretório que o compilador salvará os códigos e programas que fizermos. Normalmente os programas são instalados por definição em sua pasta própria, geralmente "c:/arquivos de programas/dev-cpp", mas podemos escolher outro diretório qualquer que satisfaça nossas necessidades. Escolhemos para esta apostila instalar o compilador em "C:/dev-cpp" , como mostra a figura 1.b, por maior facilidade de acesso e referência. São necessários aproximadamente 60 megas de espaço livre em disco.

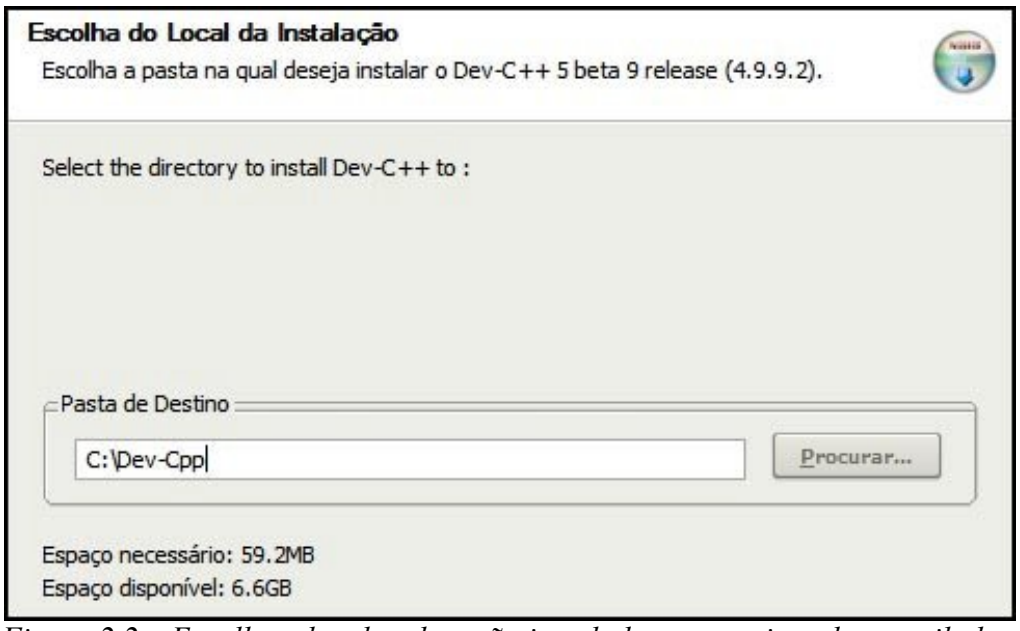

Figura 2.2 – Escolha o local onde serão instalados os arquivos do compilador.

Após isto, a instalação será concluída com sucesso. Para acessar o programa, basta encontrar o atalho ao programa no menu iniciar sob o nome "Bloodsheed Dev-C++" e clicar para executá-lo.

2.3.2 - Interface

Importante: Na primeira vez que iniciamos o Dev-C++, todos os seus menus estão em inglês. Porém, o programa é traduzido para várias línguas, inclusive português. Para mudar os menus do programa para a nossa língua (ou qualquer outra língua que desejar), basta acessar o menu "Tools -> Enviroment Options". Uma janela se abrirá, com várias opções referentes ao funcionamento do ambiente de trabalho. Na aba "Interface" encontra-se a opção "Language" com as várias linguas em que o programa está disponível. Basta procurar a opção "portuguese (Brazil)" e clicar OK, e o programa já estará traduzido para nossa língua.

A tela principal do programa é mostrada na figura abaixo.

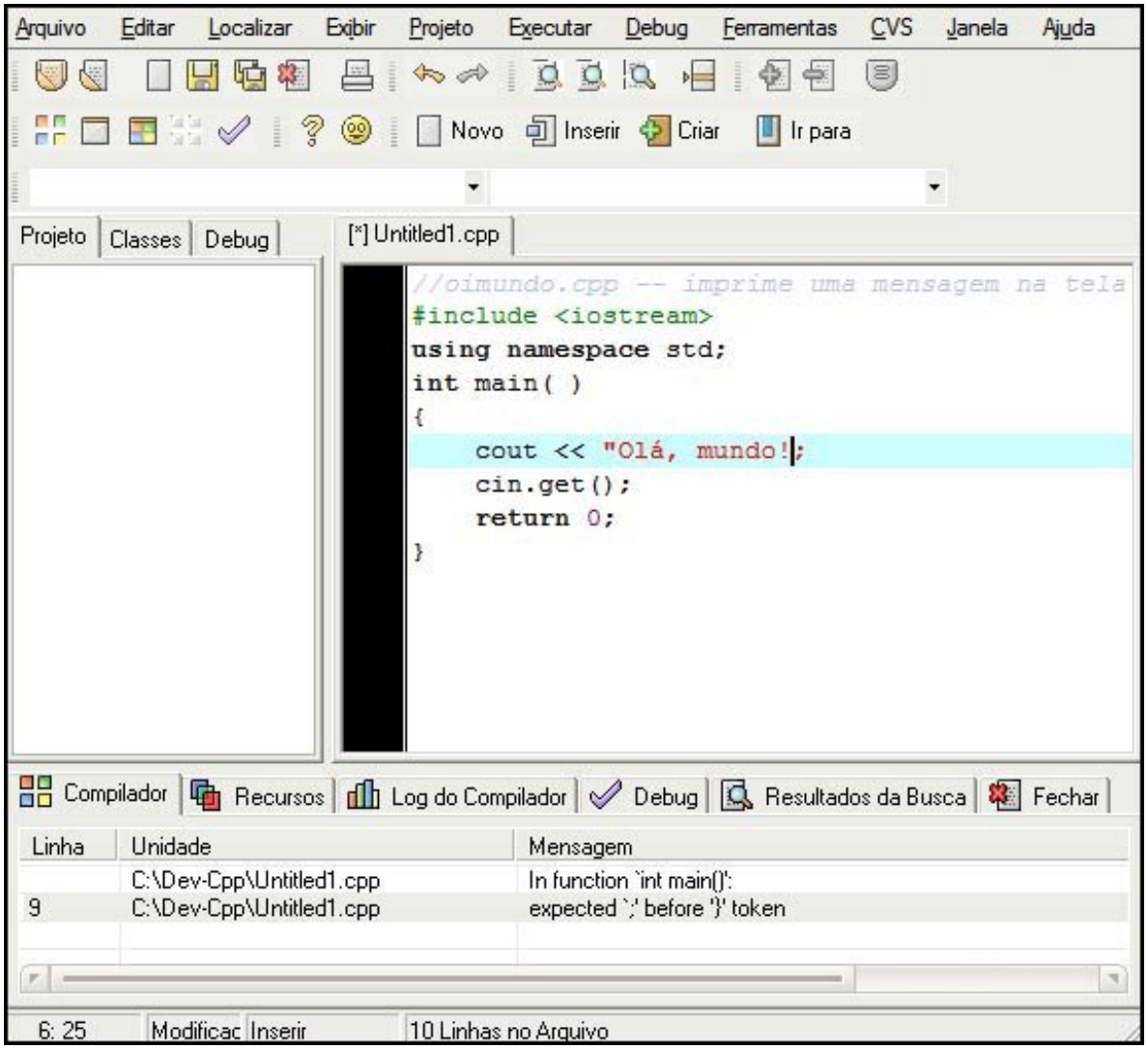

Figura 2.3 – Interface do DevC++

Os menus principais do programa são muito parecidos com os programas windows padrão. Temos os menus:

- Arquivo: possui as funções básicas de manuseio de arquivos (criar novo arquivo, abrir arquivo, fechar, imprimir, ver propriedades)
- Editar: aonde estão localizadas as funções de edição básicas de edição (copiar, recortar, colar) e algumas funções úteis para programação (como comentar e descomentar trechos do programa, e criar e acessar "bookmarks", que são marcas de acesso rápido para partes do programa, especialmente úteis para programas extensos)
- Localizar: possui os comandos de procurar e substituir partes do código; o menu Exibir permite o controle de quais componentes da tela são exibidos
- Projeto: refere-se a projetos de programas que possuem vários componentes e arquivos de códigos separados e é utilizado para adicionar e retirar componentes do projeto
- Executa: é talvez o mais importante para nós, e nele estão localizadas as funções básicas do compilador (como os comandos Compilar, Executar) e algumas funções

úteis como procurar por erros de sintaxe
- Debug: serve para controlar o debug de um programa, que é a sua execução passo-a-passo para melhor análise e busca por erros
- Ferramentas: refere-se a várias opções do compilador, do ambiente de trabalho e de edição, além de configurações diversas
- CVS: é uma função extra do compilador, e não nos tem serventia
- Janela: possui comandos úteis para os casos em que temos vários arquivos ou projetos abertos ao mesmo tempo e precisamos alternar entre eles.
- Ajuda: dá acesso à ajuda do programa, que possui uma listagem dos principais comandos do compilador e um breve tutorial da linguagem C.

Logo abaixo dos menus, temos as barras de ferramenta com as principais funções e comandos do programa representados por ícones para acesso rápido. Basta posicionar o mouse sobre qualquer um dos ícones para saber sua função.

Abaixo das barras de ferramentas, estão as duas principais janelas do programa. A janela da esquerda é chamada de Navegador de Classes e Projetos, e serve para acessar rapidamente os vários arquivos de código pertencentes à um projeto ou então acessar rapidamente as várias classes existentes em um programa. A janela da direita é nossa tela de trabalho, onde digitamos nossos códigos. Note que caso exista mais de um arquivo sendo trabalhado ao mesmo tempo, podemos alternar entre eles através das pequenas abas que existem diretamente acima da tela de trabalho, cada uma identificada pelo nome de seu arquivo.

Finalmente, a janela inferior do programa possui várias informações sobre o processo de compilação e debugagem de um programa. Ela é particularmente útil para encontrar erros de compilação, como veremos mais adiante.

2.3.3 - Utilização

Para iniciarmos um novo arquivo de código, é preciso acessar o menu "Arquivo -> Novo -> Arquivo Fonte" (como mostra a figura 2.4) ou então utilizar o atalho CTRL + N. O novo arquivo será criado imediatamente e poderemos começar a trabalhar nele.

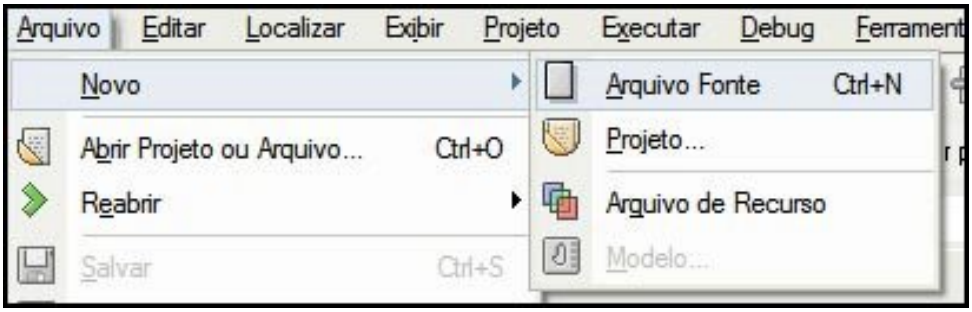

Figura 2.4 – Utilize o comando Arquivo Fonte para criar um novo arquivo em branco.

As funções básicas do compilador podem ser encontradas no menu Executar, como mostra a figura 2.5. Os comandos que utilizaremos são: Compilar (atalho: CTRL + F9), Executar (CTRL + F10) e Compilar & Executar (atalho: F9). Utilizamos o comando Compilar para compilar o arquivo código do programa em que estamos trabalhando e gerar

um arquivo executável deste programa. Em seguida, utilizamos o comando Executar para automaticamente executar o arquivo criado pela compilação. O comando Compilar & Executar é a união dos dois comandos: compila e executa o programa logo em seguida. Como já indicado antes, estes três comandos possuem ícones de acesso rápido na barra de ferramentas (veja a figura 2.6).

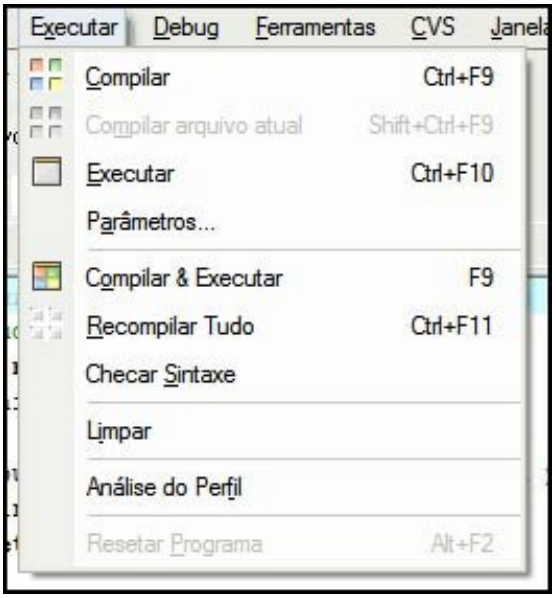

Figura 2.5 – O menu Executar possui todas os comandos necessários para compilar e executar os programas que criaremos.

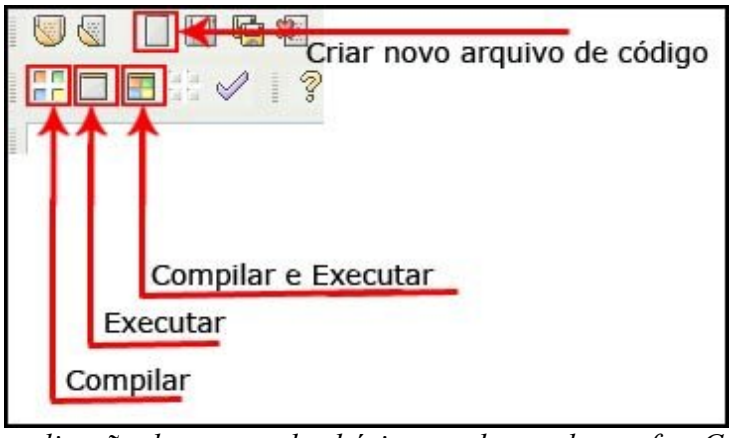

Figura 2.6 – Localização dos comandos básicos na barra de tarefas. Coloque o mouse sobre qualquer um dos ícones para saber qual é sua função.

2.3.4 – Erros

Quando compilamos um arquivo de código no Dev-C++, a janela indicadora do progresso da compilação é automaticamente aberta. Caso o arquivo de código não contenha nenhum erro, a compilação terminará e a janela de progresso permanecerá aberta para indicar que tudo correu bem (verifique o quadrado da janela chamado "status": ele deverá indicar Done após o fim da compilação). Desta maneira, após o fim da compilação

basta fechar a janela e executar o programa executável que foi gerado.

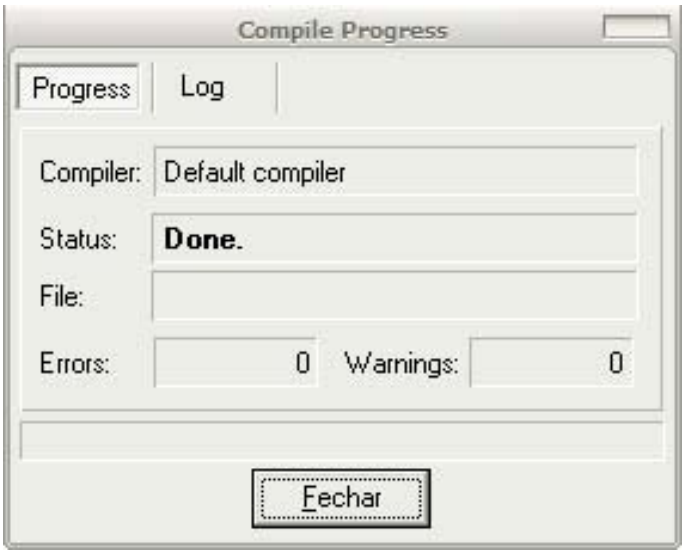

Figura 2.7 – Janela que indica o progresso da compilação do arquivo de código.

Caso nosso arquivo de código contenha uma ou mais linhas de códigos com erro, a compilação é interrompida para que estes erros (ou advertências) sejam verificados pelo programador. A janela de progresso da compilação é fechada, e a janela inferior do programa é maximizada mostrando todos os erros que foram encontrados durante a compilação do programa.

Linha	Unidade	Mensagem
	C:\Dev-Cpp\Untitled1.cpp	In function `int main()':
7	C:\Dev-Cpp\Untitled1.cpp	expected `;' before "cout"
9	C:\Dev-Cpp\Untitled1.cpp	expected `;' before "return"

Figura 2.8 – A janela que indica a posição e o tipo de erros encontrados durante a compilação do programa.

A figura acima mostra que a janela possui três colunas: linha, unidade e mensagem. A coluna linha indica a linha de código onde o erro foi encontrado; a coluna unidade indica o arquivo onde foi encontrado o erro e a coluna mensagem relata o tipo de erro encontrado. Um duplo clique em qualquer uma das indicações de erro nesta janela faz com que a linha de código onde o erro foi encontrado seja sublinhada em vermelho na janela de edição de código.

Geralmente, os erros encontrados são erros de digitação do código. Quando erramos o nome de uma variável ou mesmo um comando, o Dev-C++ indica que o nome errado não foi declarado anteriormente ("variável_x undeclared(first use in this function)"), pois ele age como se este nome desconhecido fosse uma variável não declaradada e tenta continuar a compilação.

Outro erro bastante comum é a falta de ponto-e-vírgula no fim de uma linha de comando. Neste caso, a mensagem de erro geralmente é "; expected before algum_comando", indicando que o compilador esperava o ponto-e-vírgula antes do próximo comando ou variável. A mensagem de erro indica a próxima linha de código, mas o ponto-e-vírgula ausente está na linha anterior. O compilador também indica quando utiliza-se o ponto-e-vírgula antes da hora, ou seja, quando o compilador espera por uma

expressão ou comando e encontra somente o ponto-e-vírgula. Por exemplo, uma declaração de variável sem declaração de valor: "variável = ;". Neste caso, a mensagem de erro dada pelo programa é "expected primary-expression before ';' token".

2.4 – Estrutura Básica de um Programa em C++

Temos abaixo a estrutura de um programa escrito na linguagem C++:

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

int main()
{

//comandos do programa

system("PAUSE > null");
return 0;
}
```

As duas primeiras linhas são o **cabeçalho** do programa. Todo programa deve ter um cabeçalho desse tipo para definir quais as bibliotecas ele utilizará. "Bibliotecas" são arquivos que normalmente são instalados juntos com o compilador e que possuem os comandos e funções pertencentes à linguagem.

O cabeçalho *#include<>* serve para indicar ao compilador todas as bibliotecas que este programa utilizará. Na maioria dos programas que escreveremos durante esta apostila, só utilizaremos o *#include <iostream>,* que serve para incluir a biblioteca iostream em nossos programas. Esta biblioteca contém as principais funções, comandos e classes de entrada e saída de C++, necessárias para realizar programas que, por exemplo, recebam dados via teclado e enviem dados via monitor.

A segundal linha do cabeçalho, using namespace std; , é um aviso ao compilador que estaremos utilizando os comandos e funções padrão de C++. Ele é necessário porque em C++ podemos criar várias bibliotecas para serem utilizáveis em vários programas. Cada uma dessas bibliotecas contém comandos, classes e funções próprias, e para evitar confusões e problemas com os nomes destes comandos, utilizamos o cabeçalho "using namespace ...;" para definir qual o campo de nomes que estamos utilizando. Num programa normal, que não utiliza outras bibliotecas além da padrão de C++, utilizamos o namespace std como nosso campo de nomes de comandos e funções. Assim, sempre que utilizamos um comando próprio de C++, o compilador reconhecerá automaticamente este comando como sendo pertencente à biblioteca padrão de C++.

Assim como em C, tudo o que acontece durante a execução do programa está contido dentro de uma função principal, chamada main. Declaramos a função main com:

int main ( )

Todos os comandos executados pelo programa estão contidos entre as chaves "{ }" da função main. No módulo 4 estudaremos as funções à fundo e veremos que um programa pode ter mais de uma função, mas é indispensável que todos os programas possuam a

função main.

Cada programa terá seus próprios comandos, logicamente. Entretanto, o encerramento de um programa geralmente é feito da mesma maneira para todos eles. As duas últimas linhas antes do fecha-chaves são dois comandos normalmente utilizados ao fim de um programa.

A linha "system("PAUSE > null")" é uma chamada de função própria de C++. A função system( ) recebe argumentos como o PAUSE que na verdade são comandos para o sistema operacional. Neste caso, ela recebe o comando "PAUSE > null" para pausar a execução do programa até que o usuário aperte uma tecla qualquer. Utilizamos este recurso para que a tela do programa não seja terminada automaticamente pelo sistema, impedindo que vejamos os resultados do programa.

Finalmente, o comando "return 0" é a "resposta" da função main para o sistema. Quase toda função retorna um valor para o sistema ou programa que a chamou, por exemplo, uma função pode retornar o resultado de uma operação matemática executada por ela. No caso da função main, ela retorna um valor para o sistema operacional que executou o programa. Esse valor é interpretado pelo sistema como uma mensagem indicando se o programa foi executado corretamente ou não. Um valor de retorno 0 indica que o programa foi executado sem problemas; qualquer outro valor de retorno indica problemas. Quando o programa é executado até o fim, ele retorna 0 ao sistema operacional, indicando que ele foi executado e terminado corretamente. Quando o programa encontra algum erro ou é terminado antes da hora, ele retorna um valor qualquer ao sistema, indicando erro durante a execução.

Módulo 3 – Características e Definições Gerais da Linguagem C++

3.1 – Nomes e Identificadores Usados na Linguagem C++

Existem algumas regras para a escolha dos nomes (ou identificadores) de variáveis em C++:

- Nomes de variáveis só podem conter letras do alfabeto, números e o caracter underscore "_".
- Não podem começar com um número.
- Nomes que comecem com um ou dois caracteres underscore ("_" e "__") são reservados para a implementação interna do programa e seu uso é extremamente desaconselhado. O compilador não acusa erro quando criamos variáveis desse jeito, mas o programa criado se comportará de forma inesperada.
- Não é possível utilizar palavras reservadas da linguagem C++ (para mais detalhes, veja o item 2.2). Também não é possível criar uma variável que tenha o mesmo nome de um função, mesmo que essa função tenha sido criada pelo programador ou seja uma função de biblioteca.
- C++ diferencia letras maiúsculas e minúsculas em nomes de variáveis. Ou seja, count, Count e COUNT são três nomes de variáveis distintos.
- C++ não estabelece limites para o número de caracteres em um nome de variável, e todos os caracteres são significantes.

3.2 – Palavras Reservadas na Linguagem C++

Na linguagem C++ existem palavras que são de uso reservado, ou seja, que possuem funções específicas na linguagem de programação e não podem ser utilizadas para outro fim, como por exemplo, ser usada como nome de variável. Por exemplo, a palavra reservada "**for**" serve para chamar um laço de repetição, e não pode ser utilizada como nome de uma variável.

A lista abaixo relaciona as palavras reservadas da linguagem C++:

asm	auto	bool	break	case
catch	char	class	const	const_cast
Continue	default	delete	do	double
Dynamic_cast	else	enum	explicit	export

extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
Volatile	wchar_t	while		

É importante notar que a linguagem C++ diferencia letras maiúsculas e minúsculas, ou seja, ***char*** é uma palavra reservada de C++ mas ***CHAR*** ou ***ChAr*** não é (entretanto, normalmente desaconselha-se o uso dessa diferenciação por atrapalhar a legibilidade do código). Reforçando o que já foi mencionado, as palavras reservadas só irão executar os comandos que lhes foram designados.

3.3 – Tipos e Dados

Quando um programa é escrito em qualquer linguagem de programação é necessário a definição de algumas variáveis. Variáveis são instâncias em que serão armazenados valores utilizados durante a execução de programas. Estas variáveis podem ser modificadas para suportar diferentes tipos de dados. Os principais tipos de dados utilizados em C++ podem ser divididos em variáveis inteiras e reais.

Variáveis inteiras servem para armazenar números inteiros, sem partes fracionárias. O principal tipo de variável inteira em C++ é o **int**. Além dele, existem os tipos **char**, **short** e **long**, cada um deles caracterizado por um tamanho em bits diferente. Estes tipos podem ser modificados pelo prefixo "**unsigned**", que determina que a variável em questão só terá valores positivos, liberando o bit de sinal e aumentando a capacidade de armazenamento da variável (por default, todas as variáveis inteiras e reais declaradas em C++ são "**signed**", ou seja, possuem um bit de sinal e podem ser tanto positivas como negativas). A tabela abaixo mostra os principais tipos de inteiros, seus tamanhos em bits e seu intervalo de armazenamento.

Tabela 3.1 – Tipos de variáveis inteiras em C++

Tipo	**Tamanho (em bits)**	**Intervalo**
Char	8	-128 a 127
unsigned char	**8**	0 a 255
Int	16	-32768 a 32767
unsigned int	**16**	0 a 65535
Short	**16**	-32768 a 32767
unsigned short	**16**	0 a 65535
Long	**32**	-2147483648 a 2147483647
unsigned long	**32**	0 a 4294967295

Variáveis reais servem para armazenar números que possuem partes fracionárias.

Existem duas maneiras de representar números fracionários em C++. A primeira, a mais simples, é utilizar o ponto para separar as partes inteiras e fracionárias. Por exemplo:

0.00098
1.2145
3.1461
8.0

(Mesmo no caso de um número com parte fracionária igual a zero, a utilização do ponto assegura que este número seja considerado um número de ponto flutuante por C++).

A segunda maneira é utilizar a notação científica E. Por exemplo : 3.45E7 significa "3.45 multiplicado por 10 elevado à sétima potência (10.000.000)". Essa notação é bastante útil para representar números realmente grandes ou realmente pequenos. A notação E assegura que o número seja armazenado em formato de ponto flutuante. Alguns exemplos:

2.52E8 = 2.52 x 100.000.000 = 252.000.000
-3.2E3 = -3.2 x 1000 = -3200
23E-4 = 23 x 0.0001 = 0.0023

Assim como os inteiros, os números reais em C++ podem ser representados por 3 tipos de variáveis com diferentes intervalos. São elas: **float**, **double** e **long double**. **Float** é o tipo de variável real natural, aquela com a qual o sistema trabalha com maior naturalidade. **Double** e **long double** são úteis quando queremos trabalhar com intervalos de números reais realmente grandes. Utilizamos números reais geralmente para expressar precisão através do número de casas decimais, então podemos dizer que uma variável **float** é menos precisa que uma variável **double**, assim como uma variável **double** é menos precisa que **long double**. A tabela abaixo mostra os tipos de variáveis reais, seu tamanho em bits e o intervalo de armazenagem.

Tabela 3.2 – Tipos de variáveis reais em C++

Tipo	Tamanho (em bits)	Intervalo
Float	32	3,4E-38 a 3,4E+38
Double	64	1,7E-308 a 1,7E+308
long double	80	3,4E-4932 a 1,1E+4932

3.4 – Definição de Variáveis

As variáveis devem ser declaradas, ou seja, devem ser definidos nome, tipo e algumas vezes seu valor inicial. As variáveis são classificadas em variáveis locais e globais.

Variáveis ***globais*** são aquelas declaradas fora do escopo das funções.

Variáveis ***locais*** são aquelas declaradas no início de um bloco e seus escopos estão restritos aos blocos em que foram declaradas. A declaração de variáveis locais deve obrigatoriamente ser a primeira parte de um bloco, ou seja, deve vir logo após um caractere de "abre chaves", **'{'**; e não deve ser intercalada com instruções ou comandos.

Para declarar uma variável somente é obrigatório declarar seu tipo e nome:

<tipo> <nome>;

Por exemplo:

int exemplo;

Além disso, caso seja necessário, podemos declarar um valor a esta variável no momento de sua declaração, e também adicionar um prefixo a ela, da seguinte forma:

<prefixo> <tipo> <nome> = <valor>;

Por exemplo:

unsigned int exemplo = 12;

3.5 – Definição de Constantes

O conceito de constantes em linguagens de programação é atribuir um certo valor constante a um nome, e quando este nome for referenciado dentro do código do programa, será utilizado nas operações o valor atribuído a este nome. Ou seja, se for definida a constante ***PI*** com o valor "3,1415926536", quando for encontrado no código o nome ***PI***, será utilizado em seu lugar o valor "3,1415926536".

Em C++ , utilizamos o prefixo const associado a um tipo, um nome e um valor para definir uma constante. Assim:

const <tipo> <nome> = <valor>;

Por exemplo:

const int eterna = 256;

No exemplo acima, definimos uma constante inteira de nome "eterna" que possui o valor numérico 256. É importante notar que devemos declarar a constante e lhe atribuir um valor na mesma linha de comando. Não podemos criar uma constante e lhe atribuir um valor posteriormente, ou seja, as seguintes linhas de comando são inválidas:

const int eterna;

eterna = 256;

A partir da primeira linha, "eterna" passa a ser uma constante e seu valor não pode ser mais mudado durante a execução do programa. Como seu valor não foi declarado, esta constante pode ter qualquer valor que esteja na memória do computador naquele momento da declaração da variável.

3.6 – Números Hexadecimais e Octais

Em programação algumas vezes é comum usar um sistema de numeração baseado em 8 ou 16 em vez de 10. O sistema numérico baseado em 8 é chamado **octal** e usa os dígitos de 0 a 7. Em octal, o número 10 é o mesmo que 8 em decimal. O sistema numérico de base 16 é chamado **hexadecimal** e usa os dígitos de 0 a 9 mais as letras de A até F, que equivalem a 10, 11, 12, 13, 14 e 15. Por exemplo, o número hexadecimal 10 é 16 em decimal. Por causa da freqüência com que estes dois sistemas numéricos são usados, a linguagem C++ permite que se especifique valores inteiros em hexadecimal ou octal para uma variável ou constante em vez de decimal. Um valor hexadecimal deve começar com

"0x" (um zero seguido de um x), seguido pelo valor em formato hexadecimal. Um valor octal começa com um zero. Aqui estão alguns exemplos:

hex = 0xFF; / 255 em decimal */*

oct = 011; / 9 em decimal */*

Outra base numérica muito utilizada na programação é a base binária. Apesar de C++ não possuir uma forma específica de se expressar valores de base binária, podemos utilizar a notação hexadecimal para esta função. A tabela abaixo mostra como pode ser feita a conversão de um valor binário para um valor hexadecimal.

Tabela 3.3 – Conversão de valor binário para hexadecimal

Dígito Hexadecimal	**Equivalente Binário**	**Dígito Hexadecimal**	**Equivalente Binário**
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

3.7 – Valores Strings

Outro tipo de valor suportado pela Linguagem C++ é o tipo ***string***. Uma string é um conjunto de caracteres entre aspas. Por exemplo, **"você é um vencedor"** é uma string, composta pelas várias letras que formam a frase. Não confunda strings com caractere. Uma constante caractere simples fica entre dois apóstrofos, por exemplo **'a**'. Entretanto **"a"** é uma string que contém somente uma letra.

3.8 – Códigos de Barra Invertida

A linguagem C++ fornece constantes caractere mais barra invertida especiais, úteis para caracteres que não são facilmente inseridos através do teclado ou de strings (como por exemplo, o retorno de carro). Estes códigos são mostrados na tabela a seguir:

Tabela 3.4 – Códigos de barra invertida em C++

Código	**Significado**	**Código**	**Significado**
\b	Retrocesso	\f	Alimentação de formulário
\n	Nova linha	\r	Retorno de carro
\t	Tabulação horizontal	\"	Aspas
\'	Apóstrofo	\0	Nulo
\\	Barra invertida	\v	Tabulação vertical
\a	Sinal sonoro	\N	Constante octal
\xN	Constante hexadecimal		

Usa-se um código de barra invertida exatamente da mesma maneira como usa qualquer outro caractere. Por exemplo:

```
ch = '\t';
printf("Este é um teste\n");
```

Esse fragmento de código primeiro atribui uma tabulação a ***ch*** e, então, imprime "este é um teste" na tela, seguido de uma nova linha.

3.9 – Operadores

Um operador é um símbolo que diz ao compilador para realizar manipulações matemáticas e lógicas específicas. A linguagem C++ possui três classes gerais de operadores: ***aritméticos***, ***relacionais e lógicos*** e ***bit-a-bit***.

3.9.1 – Operador de atribuição

O operador "=" atribui um valor ou resultado de uma expressão contida a sua direita para a variável especificada a sua esquerda. Exemplos:

```
a = 10;

b = c * valor + getval(x);

a = b = c = 1;
```

O último exemplo é interessante por mostrar que é possível associar vários operadores de atribuição em sequência, fazendo com que todas as variáveis envolvidas tenham o mesmo valor especificado.

3.9.2 – Operadores Aritméticos

São aqueles que operam sobre números e expressões, resultando valores numéricos. São eles:

Tabela 3.5 – Operadores Aritméticos em C++

Operador	*Ação*
+	Soma
-	subtração
*	multiplicação
/	divisão
%	módulo da divisão (resto da divisão inteira)
-	sinal negativo (operador unário)

3.9.3 – Operadores Relacionais

Operam sobre expressões, resultando valores lógicos de TRUE (verdadeiro) ou FALSE (falso). são eles:

Tabela 3.6 – Operadores Relacionais em C++

Operador	*Ação*
>	Maior
>=	maior ou igual
<	Menor
<=	menor ou igual
==	Igual
!=	não igual (diferente)

Atenção!

Não existem os operadores relacionais: "=<", "=>" e "<>".

Não confunda a atribuição ("=") com a comparação ("==").

3.9.4 – Operadores Lógicos

Operam sobre expressões, resultando valores lógicos de TRUE (verdadeiro) ou FALSE (falso). Possuem a característica de **"short circui**t", ou seja, sua execução é curta e só é executada até o ponto necessário. São eles:

Tabela 3.7 – Operadores Lógicos em C++

Operador	*Ação*
&&	operação AND
||	operação OR
!	operador de negação NOT (operador unário)

Exemplos de "short circuit":

```
(a == b) && (b == c)   /*  Se a != b não avalia o resto da expressão */

(a == b) || (b == c)   /*  Se a == b não avalia o resto da expressão */
```

3.9.5 – Manipulação de bits

A manipulação é feita em todos os bits da variável. É importante notar que a variável manipulada não pode ser do tipo float ou double. Os operadores que manipulam bits estão relacionados abaixo:

Tabela 3.8 – Operadores para manipulação de bits em C++

Operador	*Ação*
&	bit and
|	bit or
^	bit xor - exclusive or
<<	Rotação a esquerda
>>	Rotação a direita
~	bit not (complemento)

Observação: x << n irá rotacionar n vezes a variável x à esquerda.

3.9.6 – Operadores de assinalamento

É expresso da seguinte forma: (operadores combinados)

*var = var **op** expr* -> *var **op** = expr*

Onde tempos **op** como um dos seguintes operadores:

Tabela 3.9 – Operadores de Assinalamento em C++

Operador	*Ação*
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	módulo (resto da divisão)
>>	Rotação a direita
<<	Rotação a esquerda
&	And
^	xor - exclusive or
|	Or

Exemplo de aplicação:

```
i+= 2; /* É equivalente a: i = i + 2 */

j-= 3; /* É equivalente a: j = j – 3 */

k >>= 3; /* É equivalente a: k = k >> 3;*/

z &= flag; /* É equivalente a: z = z & flag;*/
```

3.9.7 – Operadores de Pré e Pós-Incremento

Operadores de pré e pós-incremento são aqueles usados quando é necessário incrementar ou decrementar um determinado valor.

As operações abaixo podem ser representadas assim:

i = i + 1; → *i = ++i;* → *++i;*

i = i – 1; → *i = --i;* → *--i;*

z = a; a = a + 1; → *z = a++;*

z = a; a = a – 1; → *z = a--;*

a = a + 1; z = a; → *z = ++a;*

a = a - 1; z = a; → *z = --a;*

3.9.8 - Operadores de Endereço

São operadores usados com ponteiros, para acesso a endereços de memória.

Tabela 3.10 – Operadores de endereçamento em C++

Operador	*Significado*
&	endereço de uma variável
*	conteúdo do endereço especificado

Exemplos:

*int var, *x;*

x = &var;

*var = *x;*

3.10 – Tabela de Operadores da Linguagem C

A tabela abaixo mostra todos os operadores apresentados anteriormente:

Tabela 3.11 – Resumo dos operadores em C++

Operador	Função	Exemplo "C"
-	menos unário	a = -b;
+	mais unário	a = +b ;
!	negação lógica	! flag
~	bitwise not	a = ~b ;
&	endereço de	a = &b ;
*	referência a ptr	a = *ptr ;
sizeof	tamanho de var	a = sizeof(b) ;
++	incremento	++a; ou a++;
--	decremento	--a; ou a--;
*	multiplicação	a = b * c;
/	divisão inteira	a = b / c;
/	divisão real	a = b / c;

%	resto da divisão	a = b % c;
+	soma	a = b + c;
-	subtração	a = b – c;
>>	shift right	a = b >> n;
<<	shift left	a = b << n;
>	maior que	a > b
>=	maior ou igual a	a >= b
<	menor que	a < b
<=	menor ou igual a	a <= b
= =	igual a	a == b
!=	diferente de	a != b
&	bitwise AND	a = b & c;
|	bitwise OR	a = b | c;
^	bitwise XOR	a = b ^ c;
&&	logical AND	flag1 && flag2
||	logical OR	flag1 || flag2
=	assinalamento	a = b;
OP=	assinalamento	a OP= b;

3.11 – Expressões

Operadores, constantes e variáveis constituem ***expressões***. Uma expressão em C++ é qualquer combinação válida dessas partes. Uma vez que muitas expressões tendem a seguir as regras gerais da álgebra, estas regras são freqüentemente consideradas. Entretanto, existem alguns aspectos das expressões que estão especificamente relacionadas com a linguagem C e serão discutidas agora.

3.12 – Precedência e Associatividade de Operadores

C++ possui uma série de regras de precedência de operadores, para que o compilador saiba decidir corretamente qual operador será executado primeiro, em uma expressão com vários operadores. Essas regras seguem basicamente as regras algébricas. Além disso, é possível o uso de parênteses para forçar o compilador a executar uma parte de uma expressão antes das outras.

Além das regras de precedência, existem também certas regras de associatividade para determinados operadores que possuem o mesmo nível de precedência, como por exemplo os operadores de divisão e multiplicação. Quando C++ encontra dois operadores com o mesmo nível de precedência em uma expressão, ele verifica se estes operadores possuem associatividade esquerda-para-direita ou direita-para-esquerda. Associatividade esquerda-para-direita significa que se estes dois operadores agirem sobre um mesmo operando, o operador da esquerda será aplicado primeiro. Da mesma forma, associatividade direita-para-esquerda significa que na mesma situação, o operador da direita será aplicado primeiro. Por exemplo, os operadores divisão e multiplicação possuem associatividade esquerda-para-direita.

A tabela abaixo mostra as regras de precedência e associativade para os operadores

de C++. O número na coluna de precedência indica qual o nível de precedência dos operadores em questão, sendo 1 o nível mais alto (operador a ser executado primeiro). Na coluna de precedência, é feito o uso dos termos unário (para operadores que usam um só operando) e binário (para operadores que necessitam de dois operandos) serve para diferenciar operadores que possuem símbolos iguais. Na coluna de associatividade, as siglas E-D e D-E indicam o tipo de associatividade dos operadores daquele nível de precedência: E-D significa associatividade esquerda-para-direita, e D-E significa associatividade direita-para-esquerda.

Tabela 3.10 – Precedência e Associatividade de operadores em C++

Precedência	**Operador**	**Associatividade**	**Explicação**
1	`(expressão)`		Agrupamento
2	`()`	E-D	Chamada de função
	`++`		Operador de incremento, pósfixado
	`--`		Operador de decremento, pósfixado
3 (todos unários)	`!`	D-E	Negação lógica
	`~`		Negação Bitwise
	`+`		Mais unário (sinal positivo)
	`-`		Menos unário (sinal negativo)
	`++`		Operador de incremento, pré-fixado
	`--`		Operador de decremento, pré-fixado
	`&`		Endereço
	`*`		Conteúdo de endereço
	`()`		Modelador de tipo “(tipo) nome”
	`sizeof`		Tamanho em bytes
	`new`		Dynamically allocate storage
	`new []`		Dynamically allocate array
	`delete`		Dynamically free storage
	`delete []`		Dynamically free array
4 (todos binários)	`*`	E-D	Multiplicação
	`/`		Divisão
	`%`		Resto de divisão
5 (todos binários)	`+`	E-D	Adição
	`-`		Subtração
6	`<<`	E-D	Shift para esquerda
	`>>`		Shift para direita
7	`<`	E-D	Menor que
	`<=`		Menor que ou igual a
	`>=`		Maior que ou igual a
	`>`		Maior que
8	`==`	E-D	Igual a
	`!=`		Não igual a
9 (binário)	`&`	E-D	Bitwise AND

10	^	E-D	Bitwise XOR (OR exclusivo)
11	|	E-D	Bitwise OR
12	&&	E-D	AND Lógico
13	||	E-D	OR Lógico
14	=	D-E	Atribuição simples
	*=		Multiplicação e atribuição
	/=		Dividir e atribuição
	%=		Achar resto da divisão e atribuição
	+=		Adição e atribuição
	-=		Subtração e atribuição
	&=		Bitwise AND e atribuição
	^=		Bitwise XOR e atribuição
	|=		Bitwise OR e atribuição
	<<=		Shift para esquerda e atribuição
	>>=		Shift para direita e atribuição
15	:?	D-E	Condicional
16	,	E-D	Combine two expressions into one

3.13 – Conversões de Tipos

3.13.1 - Conversões durante a Atribuição

C++ permite que o programador atribua um valor de um certo tipo para uma variável designada para armazenar outro tipo de valor, mas quando isto é feito o valor é automaticamente convertido para o tipo da variável.

Quando atribuímos um certo valor para uma variável cujo tamanho é maior do que o valor atribuído, não há problemas. Porém, quando o tamanho da variável é menor do que o valor a ser atribuído à ela, haverá truncamento deste valor e consequente perda de informação e/ou precisão.

Outro caso complicado é a atribuição de valores reais em variáveis inteiras. Quando armazenamos valores reais em variáveis inteiras, a parte fracionária do valor é descartada, resultando em truncamento do valor. Pode ocorrer também incompatibilidades de tamanho, por exemplo, um valor de tamanho float (64 bits) tentando ser armazenado em uma variável int (16 bits), causando invariavelmente perda de informações.

Por isso, é preciso tomar cuidado com os tipos e tamanhos de variáveis envolvidos durante uma atribuição de valores, para termos certeza que o valor inserido pode ser armazenado por aquela variável.

3.13.2 – Conversões durante Expressões

Quando constantes e variáveis de tipos diferentes são misturadas em uma expressão, elas são convertidas para o mesmo tipo. O compilador C++ converterá todos os operandos para o tipo do operando maior. Isso é feito na base de operação a operação, como descrito nestas regras de conversão de tipos:

1. Se algum dos operandos for do tipo long double, o outro operando é convertido para long double também.
2. Senão, se algum dos operando for double, o outro operando sera convertido para double.
3. Senão, se algum dos operandos for float, o outro operando será convertido para float.
4. Senão, os operandos são inteiros e promoções inteiras serão feitas.
5. Nesse caso, se algum dos operandos for do tipo unsigned long, o outro operando sera convertido para unsigned long.
6. Senão, caso um dos operandos seja long int e o outro seja um unsigned int, a conversão dependerá do tamanho relative entre os dois tipos. Se long conseguir expressar valores possíveis de unsigned int, a variável unsigne int será convertida para long.
7. Senão, ambos os operadores são convertidos para unsigned long.
8. Senão, se algum dos operadores for long, o outro será convertido para long.
9. Senão, se algum dos operadores for um unsigned int, o outro será convertido para unsigned int.
10. Caso o compilador chegue a este ponto na lista, ambos operadores serão int.

3.14 – Modeladores de Tipos

É possível também forçar a conversão de uma variável ou expressão para um determinado tipo, usando o mecanismo dos modeladores de tipos, também chamado de "type cast". É possível utilizar duas sintaxes diferentes para obter o mesmo resultado de modelamento. São elas:

(tipo) variável_ou_expressão; por exemplo*:*
(int) modelos; (int) 19.99;

e

tipo (variável_ou_expressão); por exemplo*:*
int (modelos); int (19.99);

Ambos os comandos produzem o mesmo resultado: forçar o valor presente na variável (ou expressão) à ser convertido no tipo desejado. Note que o valor da variável não é alterado; somente é feita uma conversão momentânea, cujo valor resultante deve ser armazenada em outra variável ou trabalhado imediatamente em uma expressão.

Módulo 4 – Funções na Linguagem C

4.1 – Funções

Funções são os blocos de construção da linguagem C++, com os quais podemos construir programas melhores e mais facilmente compreensíveis. Quando os programas tornam-se maiores e mais complexos, pode-se melhorar a clareza e compreensão do trabalho dividindo-o em partes menores, que chamamos de ***funções***. Todo programa possui ao menos uma função: a função **main**, a qual podemos chamar de "corpo" do programa, onde estão localizados todos os comandos e chamadas de outras funções que são executadas pelo programa. Além da função main, podemos utilizar e também criar várias outras funções dentro do mesmo programa. Por exemplo, imagine um programa que organize o funcionamento de uma loja. Poderia haver uma função para organizar o estoque, outra para relacionar os preços dos produtos, outra para acessar um banco de dados com os cadastros dos clientes, entre outras. Se fossem colocados todos os comandos do programa dentro da função ***main***, o programa ficaria muito grande e provavelmente incompreensível. Dividindo o programa em funções separadas, deixando para a função main somente a tarefa de organizar as chamadas das demais funções, podemos trabalhar mais facilmente com o programa, modificando e corrigindo funções individualmente. À medida que o tamanho e a complexidade do programa aumentam, aumenta também a possibilidade de erros, já se o mesmo for dividido em blocos menores e organizados, fica mais fácil encontrar e evitar erros. Basicamente, uma função funciona da seguinte forma: é feita uma chamada para esta função durante a execução do programa; o programa é interrompido temporariamente, e "pula" para esta função executando seus comandos; quando a função termina, ou seja, quando seus comandos acabam (ou quando o comando return é encontrado), o programa volta ao ponto onde foi interrompido para continuar sua execução normal. Uma função pode receber dados do programa para executar seus comandos (estes dados são chamados de **parâmetros** ou **argumentos**), e pode também retornar dados para o programa (o que chamamos de **retorno de função**).

Por exemplo, vamos observar a função sqrt( ), que retorna o valor da raiz quadrada de um número. Esta função foi criada e definida na biblioteca padrão da linguagem C/C++, de modo que podemos nos preocupar somente com sua execução neste momento. Você pode usar a seguinte linha de comando em um programa qualquer:

```
x = sqrt(6.25);
```

A expressão sqrt(6.25) chama a função sqrt( ). O número entre parênteses é o valor que estamos enviando para a função sqrt( ), e o chamamos de parâmetro ou argumento. A função calculará a raiz quadrada de 6.25, e enviará o valor calculado para o programa, o chamado retorno da função. Nesta linha de comando, especificamos que o retorno da função, ou seja, seu resultado, será armazenado na variável x. Resumindo, um valor é enviado para a função, e a função retorna o valor resultante para o programa.

Entretanto, antes de podermos utilizar a função sqrt( ) ou qualquer outra função em um programa é preciso declará-la. Veremos isso em detalhes no próximo item.

4.2 – Declarando uma Função

Antes de utilizar uma função em um programa é preciso declará-la, ou seja, especificar para o compilador exatamente o que faz esta função e que tipo de dados ela recebe e retorna. Podemos dividir o processo de declarar uma função em duas etapas distintas: o protótipo da função e a definição da função.

4.2.1 – Protótipo de uma Função

O protótipo de uma função tem a mesma função de uma declaração de variáveis: dizer ao compilador quais tipos de variáveis estarão envolvidos na função. Isso permite que o compilador saiba trabalhar corretamente com os valores que entram e saem da função, fazendo conversões de tipos sempre que necessário. A sintaxe da declaração de um protótipo de função é a seguinte:

<tipo_da_função> <nome> (<tipo_de_parâmetro> <nome_da_variável>);

O tipo da função denominará qual será o tipo de valor que esta função retorna. Uma função pode retornar qualquer tipo de valor, seja inteiro, fracionário ou nenhum valor. Quando uma função não retorna nenhum valor para o programa, seu tipo é void. Esse tipo de função executará seus comandos normalmente, mas não retornará nenhum valor para o programa quando terminar.

Já o tipo de parâmetro serve para determinar o tipo de variáveis que a função receberá como parâmetros. Assim como o tipo da função, qualquer tipo de variável pode ser utilizado como parâmetro. Uma função que não recebe parâmetros de entrada deve ter a palavra chave void entre parênteses. É possível ter vários parâmetros na mesma função, separando-os com vírgulas. Alguns exemplos de protótipo de função:

int livro (unsigned int paginas);

float divide (float dividendo, float divisor);

void imprime (void);

float divide (float, float);

O último exemplo mostra que a escolha do nome do parâmetro no protótipo é opcional: basta inserir o tipo de parâmetros que serão utilizados pela função e o compilador alocará a memória necessária para eles. Porém a atribuição de nomes para parâmetros é aconselhável, pois melhora a legibilidade do programa.

Quando declaramos e definimos uma função no início do programa, isto é, antes da função main, podemos omitir o protótipo da função, fazendo somente a definição da função como veremos abaixo.

4.2.2 – Definição de uma Função

O protótipo de uma função diz para o compilador quais os tipos de variáveis ela usará. Já a definição de uma função diz ao compilador exatamente o que a função faz. A sintaxe da definição de uma função é a seguinte:

```
<tipo da função> <nome> ( <tipo do parâmetro> <nome do parâmetro>)
{
        comandos da função;
}
```

A primeira linha da definição de uma função é idêntica ao protótipo, com a exceção de que somos obrigados a declarar nomes para as variáveis de parâmetro e que a linha não termina em um ponto-e-vírgula, e sim em uma abre-chaves. Dentro das chaves estarão todos os comandos pertencentes a função, inclusive declaração de variáveis locais, chamadas para outras funções e chamadas de retorno. Por exemplo:

```
int cubo ( int valor) {
resultado = valor*valor*valor;
return resultado;
}
```

A função acima calcula o cubo de um valor inteiro. Note que podemos declarar variáveis dentro de uma função; entretanto, estas variáveis só poderão ser utilizadas dentro desta mesma função. O comando return termina a função e retorna um valor para o programa.

4.2.3 – Retorno de uma função

O comando return é utilizado para terminar a execução de uma função e retornar um valor para o programa. Sua sintaxe é a seguinte:

return <variável ou expressão>;

O comando return aceita qualquer constante, variável ou expressão geral que o programador precise retornar para o programa principal, desde que este valor seja igual ou convertível para o tipo da função (já estabelecido no protótipo da função). É importante notar que o valor retornado não pode ser uma matriz; porém, é possível retornar uma matriz indiretamente, desde que ela faça parte de uma estrutura ou objeto (tipos de dados que serão estudados mais adiante).

É possível também criar funções que contênham múltiplos comandos ***return***, cada um dos quais retornando um valor para uma condição específica. Por exemplo, considere a função ***compara_valores***, mostrada a seguir:

```
int compara_valores(int primeiro, int segundo)
{
        if (primeiro == segundo)
        return (0);
        else if (primeiro > segundo)
        return (1);
        else if (primeiro < segundo)
        return (2);
}
```

A função ***compara_valores*** examina dois valores listados na tabela abaixo:

Resultado	**Significado**
0	Os valores são iguais
1	O primeiro valor é maior que o segundo
2	O segundo valor é maior que o primeiro

Como regra, deve-se tentar limitar as funções a usar somente um comando ***return***. À medida que as funções se tornarem maiores e mais complexas, ter muitos comandos ***return*** normalmente tornará as funções mais difíceis de compreender. Na maioria dos casos, pode-se reescrever a função para que ela use somente um comando ***return***

4.3 – Main como uma Função

Como já dissemos anteriormente, todo programa possui uma função principal que contém todos os comandos e chamadas para outras funções presentes no programa. A função main funciona como uma função normal: possui um protótipo e uma definição. Geralmente omitimos o protótipo, fazendo apenas a definição da função main da seguinte forma:

```
int main (void) {
//corpo do programa
return 0;
}
```

Note que a função main é do tipo int, e retorna 0. Entretanto, não existe outra função acima de main que a tenha chamado, para que ela possa retornar um valor de resposta. Para que serve este retorno então? Simples: consideramos que a "função chamadora" de main é o próprio sistema operacional. Assim, utilizamos o retorno para indicar o funcionamento do programa. Caso o programa termine e retorne o valor 0 para o sistema operacional, sabemos que tudo correu bem e que o programa terminou normalmente. Um valor retornado diferente de 0 indica que o programa não rodou até o final (ou seja, até o ponto "return 0;") e que aconteceu algum erro. Muitos sistemas operacionais e programas utilizam esse sistema simples para detectar erros durante a execução de seus aplicativos.

4.4 – Variáveis dentro das Funções

À medida que as funções vão se tornando mais úteis nos programas, muitas delas requerem que as variáveis gerem resultados valiosos. Para usar uma variável dentro de uma função, precisa-se primeiro declarar a variável, exatamente como feito na função principal ***main***. Quando se declara variáveis dentro de uma função, os nomes usados para essas variáveis são exclusivos para a função.

Portanto, se o programa usa dez funções diferentes e cada função usa uma variável chamada ***contador***, o compilador considerará a variável de cada função como distinta. Se uma função requer muitas variáveis, elas deverão ser declaradas no início da função, exatamente como se faria dentro de ***main***.

4.4.1 – Variáveis Locais

A Linguagem C++ permite declarar variáveis dentro de suas funções. Essas variáveis são chamadas de variáveis locais, pois seus nomes e valores somente têm significado dentro da função que contém a declaração da variável.

O programa a seguir ilustra o conceito de uma variável local. A função ***valores_locais*** declara 3 variáveis a, b e c, e atribui às variáveis os valores 1, 2 e 3, respectivamente. A função main tenta imprimir o valor de cada variável. No entanto, como os nomes dos valores são locais à função, o compilador gera erros, dizendo que os símbolos a, b, e c estão indefinidos.

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

void valores_locais(void);
void valores_locais(void)
     {
       int a=1, b=2, c=3;
     }
int main (void)
     {
      cout<<"A vale "<< a <<". B vale "<< b <<". C vale "<< c
<<.\n";
     system("PAUSE > null");
     return 0;
     }
```

O programa abaixo é a versão corrigida do programa acima. Veja que a função main chama a função valores_locais, que por sua vez declara as variáveis junto com seus valores e as imprime na tela corretamente.

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

void valores_locais(void);
```

```
void valores_locais(void)
     {
       int a=1, b=2, c=3;
      cout<<"A vale "<< a <<". B vale "<< b <<". C vale "<< c
<<".\n";
     }

int main (void)
     {
      valores_locais();
     system("PAUSE > null");
     return 0;
     }
```

4.4.2 –Variáveis Globais

Além das variáveis locais, a Linguagem C permite que os programas usem variáveis globais, cujos nomes, valores e existência são conhecidos em todo o programa. Em outras palavras, todos os programas em Linguagem C podem usar variáveis globais. O programa a seguir ilustra o uso de três variáveis globais a, b e c:

```
#include <iostream>
using namespace std;
int a = 1, b = 2, c = 3;  // Variaveis globais
void valores_globais(void)
     {
      cout<<"A vale "<< a <<". B vale "<< b <<". C vale "<< c
<<".\n";
     }
int main(void)
     {
        valores_globais();
     cout<<"A vale "<< a <<". B vale "<< b <<". C vale "<< c
<<".\n";
         system("PAUSE > null");
        return 0;
     }
```

Quando este programa é compilado e executado, as funções ***variaveis_globais*** e ***main*** exibem os valores das variáveis globais. Observe que as variáveis globais são declaradas fora de todas as funções. Declarando variáveis globais deste modo, todas as funções do programa podem usar e alterar os valores da variável global simplesmente referenciando o nome dela.

Embora as variáveis globais possam parecer convenientes, o uso incorreto delas podem causar erros que são difíceis de depurar. Se um programa usa variáveis globais, algumas vezes o nome de uma variável global é o mesmo que aquele de uma variável local que seu programa declara dentro de uma função. Por isso, é preciso estar atento com o uso de variáveis globais e sua nomeação. Quando nomes de variáveis globais e locais

estiverem em conflito, a linguagem C++ usará sempre a variável local.

4.5.1 – Chamada por Valor

Os programas passam informações para funções usando parâmetros. Quando um parâmetro é passado a uma função, a Linguagem C++ usa uma técnica conhecida como **chamada por valor** para fornecer à função uma cópia dos valores dos parâmetros. Usando a chamada por valor, quaisquer modificações que a função fizer nos parâmetros existem apenas dentro da própria função. Quando a função termina, o valor das variáveis que a função chamadora passou para a função não é modificada dentro da função chamadora.

Por exemplo, o programa a seguir passa três parâmetros (as variáveis a, b e c) para a função ***exibe_e_altera***. A função, por sua vez, exibirá os valores, somará 100 aos valores e depois exibirá o resultado. Quando a função terminar, o programa exibirá os valores das variáveis. Como a Linguagem C usa chamada por valor, a função não altera os valores das variáveis dentro do chamador, como mostrado a seguir:

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

void exibe_e_altera(int primeiro, int segundo, int terceiro)
     {

        cout<<"Valores originais da funcao: "<<primeiro<<"
"<<segundo<<"  "<<terceiro<<"\n";
        primeiro = primeiro +100;
        segundo = segundo + 100;
        terceiro = terceiro + 100;
        cout<<"Valores originais da funcao: "<<primeiro<<"
"<<segundo<<"  "<<terceiro<<"\n";
     }
int main(void)
     {
        int a = 1, b = 2, c = 3;
        exibe_e_altera(a, b, c);
        cout<<"Valores finais em main: "<<a<<"  "<<b<<"
"<<c<<"\n";
        system("PAUSE > null");
     }
```

Como pode ser visto, as alterações que a função faz nas variáveis somente são visíveis dentro da própria função. Quando a função termina, as variáveis dentro de ***main*** estão inalteradas.

4.5.2 - Chamada por Referência

Usando a chamada por valor, as funções não podem modificar o valor de uma variável passada para uma função. No entanto, na maioria dos programas, as funções modificarão as variáveis de um modo ou de outro. Por exemplo, uma função que lê informações de um arquivo precisa colocar as informações em uma matriz de string de caracteres. Da mesma forma, uma função tal como ***strupr*** precisa converter as letras em uma string de caractere para maiúsculas. Quando as funções alteram o valor de um parâmetro, os programas precisam passar o parâmetro para a função usando chamada por referência.

A diferença entre chamada por valor e chamada por referência é que, usando a chamada por valor, as funções recebem uma cópia do valor de um parâmetro. Por outro lado, com a chamada por referência, as funções recebem o **endereço de memória** da variável. Portanto, as funções podem alterar o valor armazenado na posição de memória específica (em outras palavras, o valor da variável); alterações essas que permanecem após a função terminar.

Para usar a chamada por referência, seu programa precisar usar ponteiros. Por ora tudo o que precisamos saber é que um ponteiro armazena um endereço de memória, assim como uma variável armazena um valor. O módulo 8 é dedicado totalmente à explicação de ponteiros e suas aplicações em C++, incluindo o funcionamento de uma chamada de função por referência.

4.6 – Biblioteca de Execução

Muitas vezes, uma função criada para um determinado programa atende as necessidades de um segundo programa. A capacidade de reutilizar as funções em mais de um programa pode poupar um tempo considerável de programação e teste. Para isto é só copiar a função de um para outro programa.

A linguagem C++ possui uma biblioteca padrão com muitas funções úteis previamente implementadas para uso do programador, inclusive incluindo todas as funções da biblioteca padrão de C. A biblioteca padrão se divide em várias bibliotecas ou arquivos menores, divididos pelos tipos de função que cada um contém. Por exemplo, a biblioteca "cmath" contém as funções matemáticas padrão da linguagem C e a biblioteca "string" contém funções da biblioteca padrão de C++ que tratam de strings. Para utilizar uma destas bibliotecas, é preciso "avisar" ao compilador através da diretiva

#include <nome_da_biblioteca>

O comando *"#include"* copia o conteúdo da biblioteca para o programa que estamos compilando, para que as funções pertencentes à ela possam ser utilizadas. Algumas das bibliotecas padrão de C e seus cabeçalhos são mostrados abaixo:

Funções matemáticas: #include <cmath>

Funções de string: #include <cstring>

Funções de string do C++: #include <string>

Funções de I/O: #include <cstdio>

Funções de tempo: #include <ctime>

Uma descrição detalhada das funções presentes na biblioteca padrão de C++ não cabe nesta apostila. Entretanto, muitos livros e websites contém listagens e explicações das funções padrões. Um bom exemplo é o website http://www.cppreference.com/index.html , que contém uma vasta referência sobre vários aspectos da linguagem C++.

Não deixe de examinar as funções que seu compilador fornece. Muitos compiladores referenciam essas funções internas como biblioteca de execução. A maioria dos compiladores fornece centenas de funções de biblioteca de execução com propósito que vão de abertura e trabalho com arquivos para acessar informações do disco ou de diretório para determinar o tamanho de uma string de caracteres. As duas ou três horas que serão gastas para ler a documentação da biblioteca de execução pouparão muitas horas de programação.

4.7 – Funções Recursivas

Em C++, as funções podem chamar a si próprias, com exceção da função principal *main*. Uma função é ***recursiva*** se um comando no corpo da função chama ela mesma. Algumas vezes chamada de ***definição circular***, a recursividade é o processo de definição de algo em termos de si mesmo.

Exemplos de recursividade existem em grande número. Uma maneira de definir um número inteiro sem sinal por meio de recursividade é utilizando-se os dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 mais ou menos outro número inteiro. Por exemplo, o número 15 é o número 7 mais o número 8; 21 é 9 mais 12 e 12 é 9 mais 3.

Para uma linguagem ser recursiva, uma função deve estar apta a chamar a si própria. O exemplo clássico de recursividade é mostrado na função ***fatorial_recursivo()***, que calcula o fatorial de um número inteiro. O fatorial de um número ***N*** é o produto de todos os números inteiros entre 1 e *N*. Por exemplo, o fatorial de 3 é 1 x 2 x 3, ou 6.

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

unsigned long fatorial_recursivo (int n){
   unsigned long resposta;
   if ((n == 1) || (n == 0))return(1);
   resposta = n * fatorial_recursivo(n - 1);
   return(resposta);
```

```
}
int main()
{
   unsigned long f;
   int n;
   cout<<"Digite um número: \n";
   cin >> n;
   f = fatorial_recursivo(n);
   cout<<"O fatorial de "<<n<<" e "<<f<<" \n";
   system("PAUSE > null");
   return 0;
}
```

Quando uma função chama a si própria, as novas variáveis locais e os argumentos são alocados na pilha, e o código da função é executado com esses novos valores a partir do início. Uma chamada recursiva não faz uma nova cópia da função. Somente os argumentos e as variáveis são novos. Quando cada chamada recursiva retorna, as antigas variáveis locais e os parâmetros são removidos da pilha e a execução recomeça no ponto de chamada da função dentro da função.

Tendo em vista que o local para os argumentos de funções e para as variáveis locais é a pilha e que a cada nova chamada é criado uma cópia destas variáveis na pilha, é possível ocorrer overflow da pilha (stack overflow) e o programa terminar com um erro.

4.8 - Sobrecarga da Função

Quando um programa usa uma função, a Linguagem C armazena o endereço de retorno, os parâmetros e as variáveis locais na pilha. Quando a função termina, a Linguagem C descarta o espaço da pilha que continha as variáveis locais e parâmetros, e, depois, usa o valor de retorno para retornar a execução do programa para a posição correta.

Embora o uso da pilha seja poderoso porque permite que o programa chame e passe as informações para as funções, também consome tempo de processamento. Os programadores chamam a quantidade de tempo que o computador requer para colocar e retirar informações da pilha de sobrecarga da função. Na maioria dos sistemas, os cálculos baseados em funções podem requerer quase o dobro do tempo de processamento.

Entretanto, C++ introduz um conceito que pode reduzir bastante a sobrecarga da função: são as funções inline.

4.9 – Funções Inline

As funções inline são funções que são compiladas “na sequência” (“in line”) do código. Isso quer dizer que quando chamamos uma função inline em um programa, o compilador substitui a chamada de função pelo próprio código da função, adaptando automaticamente os parâmetros e retorno da função. Assim, elimina-se a necessidade do

programa pular para outro ponto na memória do computador, executar a função e depois retornar ao ponto onde parou.

Criamos funções inline do mesmo jeito que criamos funções normais: declarando seu protótipo e sua definição, com parâmetros e retorno. Só é preciso adicionar o prefixo inline antes do protótipo ou da definição da função, como é mostrado na função abaixo:

```
inline int quadrado (long x)
 {
  x =  x * x;
return x;
 }
```

Não podemos criar funções inline recursivas, pois logicamente uma função recursiva tem um tamanho variável de acordo com seu número de iterações, que muitas vezes muda durante a execução do programa.

Ganha-se velocidade com as funções inline, mas perde-se em memória utilizada. Se um programa chama uma função inline 10 vezes, criam-se 10 cópias da função dentro do código, aumentando seu tamanho durante a compilação. É preciso escolher com cuidado quando utilizar funções normais e funções inline, tendo em mente o compromisso rapidez de execução – memória utilizada.

4.10 – Parâmetros Padrão

Outra novidade introduzida por C++ no manuseio de funções é a possibilidade de definir-se um valor padrão para parâmetros de uma função. Isto é, podemos definir um parâmetro para uma função e estabelecer um valor padrão para ele, da seguinte forma:

```
void mensagem (int x = 1);
```

Quando chamamos a função e enviamos um valor como parâmetro para ela, a função é executada normalmente, utilizando o valor de x que enviamos. Porém, caso não enviássemos um valor x como parâmetro para esta função, isto é, fizéssemos a seguinte declaração no programa:

```
mensagem( );
```

Automaticamente a função substituiria x por 1, como especificamos no protótipo da função. Caso não tivéssemos definido um valor padrão para o parâmetro x, o compilador nos retornaria um erro de parâmetros insuficientes para que a função seja executada. Abaixo temos um exemplo de programa utilizando um parâmetro padrão.

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

void mensagem (int x = 1) {
```

```
if (x != 1) cout<<"Voce enviou um parametro para a função! O
parametro X é igual a "<<x<<"\n";
if (x == 1) cout<<"Voce NAO enviou um parametro para função!
O parametro X é igual a "<<x<<"\n";
}

int main()
{
int valor;
cout<<"Entre com um valor. Este valor será repassado para a
função mensagem automaticamente!\n";
cin>>valor;
mensagem(valor);
cout<<"Agora chamaremos a função mensagem sem lhe dar um
parametro. Veja o que acontece: \n";
mensagem();
system("PAUSE > null");
return 0;
}
```

Módulo 5 – Estudo dos comandos cout e cin

A linguagem C++ possui uma ótima biblioteca de classes relacionadas ao controle de entrada e saídas de dados. Desde o início da apostila temos feito uso de algumas facilidades fornecidas por esta biblioteca, especificamente, os comandos cout e cin. Como você deve ter percebido, a classe cout serve para exibir valores - seja o valor de uma variável ou uma frase – enquanto que cin serve para armazenar valores recebidos através do teclado em variáveis. Tínhamos na linguagem C as funções printf e scanf para executar estas mesmas funções. Na verdade, printf e scanf também estão presentes em C++ (assim como todas as funções padrões de C), e podemos utilizá-las caso desejemos. Porém, os comandos – ou, utilizando um termo mais tecnicamente apropriado, classes – cin e cout facilitam muito a vida do programador, por serem mais "inteligentes" que printf e scanf.

5.1 – Utilização de *cout*

Como já dissemos, cout exibe valores na tela. A sintaxe utilizada é:

cout << <valor, string, variável, ponteiro, etc>;

Utilizamos cout em conjunto com o operador de inserção **<<**. Note que símbolo << também é utilizado pelo operador de bitwise shift para a esquerda (move bits de uma variável para a direção esquerda), entretanto não precisamos nos preocupar com isto: C++ sabe diferenciar quando estamos utilizando um operador ou o outro, através do contexto.

O operador << indica ao comando **cout** que um dado deve ser exibido na tela, além de identificar automaticamente qual o tipo deste dado e como ele deve ser formatado para exibição na tela. Assim, não precisamos informar à **cout** que estamos enviando um inteiro, um real ou uma string, como fazíamos em C: o operador << se encarrega desta identificação, bastando para o operador indicar o nome da variável. Abaixo temos a lista de todos os tipos básicos de C++ reconhecidos pelo operador de inserção:

- unsigned char
- signed char
- char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- float

- double
- long double

O operador de inserção também fornece facilidades para a exibição de strings. Além dos tipos básicos mostrados acima, o operador de inserção também reconhece os seguintes tipos de ponteiros:

- const signed char *
- const unsigned char *
- const char *
- void *

Ponteiros serão explicados com maior propriedade no módulo 8, mas precisamos saber que C e C++ representam strings utilizando ponteiros para o endereço de memória da string. Este ponteiro pode ser o nome de uma variável matriz de tipo char, um ponteiro de tipo char ou então uma frase entre aspas. O operador de inserção reconhece cada um destes casos e exibe na tela a string de texto. Por exemplo:

```
char nome[20] = "Jose das Couves";
char * nome2 = "Jose das Galinhas";
cout << "Jose das Dores";
cout << nome;
cout << nome2;
```

Estas 3 utilizações de cout exibem as strings de texto na tela. Veremos mais adiante que toda string possui um caractere de término, "\0", que indica para o compilador que a string terminou. Nestes três casos, o operador de inserção orienta-se por este "\0" para terminar a exibição das strings.

É importante ressaltar que o operador de inserção não reconhece automaticamente matrizes numéricas e não as exibe automaticamente na tela. Para fazer isso, precisaremos utilizar os métodos tradicionais envolvendo loops para mostrar cada membro da matriz de uma vez. Este assunto será discutido no módulo 7, dedicado para matrizes.

5.2 – Overload do operador de inserção

"Overload do operador de inserção" nada mais é do que utilizar o operador de inserção várias vezes na mesma chamada de cout. Por exemplo:

```
cout << "O valor da variável X é :" << X;
```

A linha de comando acima exibe dois valores: a string "O valor da variável X é :" e a variável X. Note que utilizamos duas vezes o operador de inserção, sempre antes do valor a ser exibido. Podemos repetir o operador de inserção quantas vezes precisarmos na mesma linha de cout.

5.3 – Formatação de exibição com cout

A função printf fornecia aos usuários de C múltiplas maneiras de formatar a exibição dos dados na tela do computador. A classe cout também fornece as mesmas facilidades para os usuários de C++. Nas subseções abaixo descreveremos como fazer vários tipos de formatação de dados utilizando cout e o operador de inserção.

5.3.1 – Escolhendo a Base Numérica

Podemos escolher a base numérica que utilizaremos para representar números inteiros. Para isto, devemos utilizar os comandos:

```
cout << hex;
cout<< oct;
cout<< dec;
```

Após utilizar qualquer um destes comandos, sempre que pedirmos para cout exibir um número inteiro na tela, o comando automaticamente converterá o número para a base definida previamente. Por exemplo:

```
int numero = 10;
cout << hex;
cout << numero;
```

Dessa forma, cout exibirá na tela não o número 10 como estamos acostumados, mas a letra a, que representa 10 na base hexadecimal. Note que podemos utilizar o operador de inserção duas vezes para deixar o código mais compacto e obter o mesmo resultado:

```
int numero = 10;
cout << hex << numero;
```

Não se esqueça de retornar para a base decimal com "cout << dec;" após exibir valores na base hexadecimal ou octal!

5.3.2 – Formatação de números reais

Podemos escolher também a notação utilizada para exibição de números reais. Com o comando:

```
cout << fixed;
```

Instruímos o programa a exibir valores reais usando a notação de ponto fixo (por exemplo, 3.1214). Da mesma forma, com o comando:

```
cout << scientific;
```

Instruímos o programa a utilizar a notação científica (por exemplo, 3.21E-2).

5.3.3 – Espaçamento de Texto

O comando cout permite também escolher um número mínimo de caracteres para ser exibido na tela. Isto é feito utilizando o método:

cout.width (x);;

Onde substituímos x pelo número mínino de caracteres a ser exibido na tela. Após a utilização deste método, utilizamos o comando cout para exibir o valor desejado, como no exemplo abaixo:

```
int  variavel = 10;
cout.width ( 5 );
cout << variavel;
```

Neste exemplo, foi especificado *cout.width (5);* e o valor a ser exibido é 10. Assim, cout predecerá o valor 10 com três espaços em branco.

Observe que o valor especifica o ***número mínimo*** de caracteres que a saída consumirá. Se o valor a ser exibido requer mais caracteres do que o especificado, será usado o número de caracteres necessários para exibir o valor corretamente.

É importante observar também que o método cout.width só é válido para a próxima utilização de cout: após isto, o número mínimo de caracteres volta a ser zero.

Podemos também determinar o caractere a ser utilizado para preencher os espaços em branco de um campo de exibição. Isto é feito com o seguinte método:

cout.fill ("caractere");

Onde substituímos "caractere" pelo caractere que será exibido. É necessário utilizar aspas entre o caractere, para indicar para o compilador que não se trata de uma variável.

O exemplo abaixo mostra a utilização conjunta destes dois métodos:

```
int  variavel = 10;
cout.width ( 8 );
cout.fill("0");
cout << variavel;
```

Este exemplo fará a seguinte exibição na tela, preenchendo os espaços em branco determinados por cout.width com o caractere 0, determinado por cout.fill:

00000010

5.3.4 – Precisão de Variáveis Reais

O seguinte método é utilizado para fixar a precisão de variáveis reais, ou seja, o número mínimo de casas decimais à serem exibidas após a vírgula em um valor real:

cout.precision (valor);

Por default, C++ utiliza 6 casas decimais após a vírgula. Quando alteramos o valor

da precisão, este novo valor vale para todas as utilizações futuras de cout.

5.3.5 – Alinhamento de Texto

A escolha da direção de alinhamento de texto é feita da seguinte forma utilizando cout:

Alinhamento à direita: *cout << **right** << <valor a ser exibido>;*
Alinhamento à esquerda: *cout << **left** << <valor a ser exibido>;*

Por default, todos os valores exibidos em um programa são automaticamente alinhados à direita. Quando mudamos o modo do alinhamento de texto, ele permanecerá dessa forma até que o alteremos novamente.

5.4 – Utilização de *cin*

Utilizamos o comando cin para obter valores do usuário através do teclado. A sintaxe utilizada é a seguinte:

cin >> variavel_destino;

Assim como cout, cin utiliza um operador (nesse caso, o operador de extração >>) para identificar o tipo de variável onde o valor será armazenado e encontrar o endereço de memória correto. Ao contrário da função scanf, utilizada na linguagem C, não é preciso especificar qual o tipo de valor será enviado pelo teclado pois o operador de extração faz as conversões necessárias. Podemos utilizar cin para ler valores inteiros, reais e strings de caracteres.

Na maioria dos casos, o comando cin cobre nossas necessidades de entrada de dados via teclado. Entretanto, quando precisamos ler strings com mais de uma palavra, como por exemplo frases ou nomes, cin apresenta certos “problemas”. Isto acontece por causa da maneira que C++ trata os espaços em branco em uma entrada via teclado.

Espaços em branco são considerados fim de entrada pelo comando cin; ao invés de descartar os caracteres que vierem após o espaço em branco, C++ os guarda em um buffer (uma espécie de “reserva” ou pilha de dados). Quando cin for chamado novamente, antes de ler a nova entrada do teclado, o programa primeiro utiliza os dados que estão nesse buffer. Assim, temos a impressão que a nova entrada de dados foi descartada pelo programa, mas na verdade ela foi jogada no buffer, esperando uma nova chamada de cin. Para solucionar este problema, utilizamos o métdo de cin cin.getline.

5.5 – Método de cin: cin.getline

O método cin.getline é muito útil para receber strings de caracteres com espaços, como frases. Este método lê uma linha inteira, marcando o fim da entrada de dados pelo uso da tecla <ENTER> indicando a entrada de uma nova linha. Abaixo temos a sintaxe do método:

cin.getline (<matriz_destino>, <limite de caracteres>);

O primeiro argumento é a matriz de caracteres para onde serão enviados os dados recebidos. É necessário declarar uma matriz de caracteres previamente ao uso deste método. O segundo argumento é o número máximo de caracteres que será lido pelo método, menos o caractere \0 indicando o fim da string. Assim, se especificarmos um número máximo igual a 20, este comando lerá 19 caracteres e descartará os próximos caracteres entrados pelo usuário, até que a tecla <ENTER> seja pressionada. Um espaço será sempre utilizado para marcar o fim da string através do caractere \0.

Um exemplo da utilização de cin.getline:

```
char matriz[60];
cin.getline ( matriz, 50 );
cout >> matriz;
```

Nesse caso, o método cin.getline lerá os próximos 49 caracteres (lembre-se do espaço reservado para o caractere fim_de_string \0) que o usuário entrar através do teclado. A leitura será feita até que ele aperte a tecla <ENTER> interrompendo o comando. Caso o usuário entre mais do que 50 caracteres , os próximos serão descartados pelo programa.

Módulo 6 - Estruturas de Controle de Fluxo

6.1 - Estruturas de Controle de Fluxo

Estruturas de controle de fluxo são comandos utilizados em uma linguagem de programação para determinar qual a ordem e quais comandos devem ser executados pelo programa em uma dada condição. C++ oferece várias opções de estrutura de controle de fluxo, todas elas herdadas da linguagem C. Neste módulo iremos ver como funcionam cada uma destas estruturas em detalhe.

Geralmente, as estruturas de controle utilizam expressões condicionais. Caso a expressão retorne 0, dizemos que ela é falsa. Caso ela retorne qualquer outro valor, dizemos que ela é verdadeira. Nesse contexto, qualquer expressão pode ser utilizada desde que retorne um valor zero ou não zero. Podemos utilizar operadores aritméticos, relacionais, lógicos, desde que no final a expressão nos retorne um valor que possa ser testado. Também é possível testar várias condições ao mesmo tempo, unindo as expressões com o auxílio dos operadores AND e OR.

6.2 – A declaração if

Utilizamos a declaração if quando desejamos que o programa teste uma ou mais condições e execute um ou outro comando de acordo com o resultado deste teste. A sintaxe de if é a seguinte:

```
if (condição)
        {
        comandos;
        }
else
        {
                comandos;
        }
```

A declaração if testará a condição expressa entre parênteses. Caso a condição seja verdadeira, os comandos declarados entre as chaves serão executados.

A declaração else é opcional: podemos utilizá-la para determinar um conjunto de comandos que serão executados caso a condição testada seja falsa. Note que somente um dos conjuntos de comandos será executado, nunca os dois: caso a condição seja verdadeira, o bloco pertencente a if será executado; caso a condição falhe, o bloco pertencente a else será executado.

O programa abaixo ilustra de maneira simples o uso da declaração if-else, obtendo um número do usuário e verificando se este valor é maior ou igual a 50.

```
#include <iostream>
```

```
using namespace std;

int main()
          {
          int teste;
          cout<<"Digite um numero qualquer:\n";
          cin>> teste;
          if (teste > 50)
             {
             cout<<"O numero que voce digitou é maior que 50";
             }
          else
              {
              cout<<"O numero que voce digitou é menor que
50";
              }
          system("PAUSE > null");
          return 0;
 }
```

É possível também aninhar ifs, ou seja, fazer uma declaração ***if*** dentro de outra declaração ***if*** anterior. Este é um método muito útil em programação, mas é preciso tomar cuidado ao utilizá-lo para saber qual bloco else pertence à qual ***if***. Em C++, o ***else*** é ligado ao ***if*** mais próximo dentro do mesmo bloco de código que já não tenha uma declaração ***else*** associada a ele. Para se certificar de que estamos aninhando os ifs e elses corretamente, utilizamos chaves para delimitar os diferentes blocos, como pode ser visto no código abaixo:

```
if ( x > 10)
        {
                if ( x == 17)
                        {
                        cout<< "x é maior que 10 e igual a 17";
                        }
                else
                        {
                                cout<< "x é maior que 10 mas não é igual a 17";
                        }
        }
else
        {
        cout << "x é menor do que 10";
        }
```

Note que a segunda declaração if está totalmente contida pelas chaves da primeira declaração if. Utilizou-se chaves em todas os blocos de comando, para melhor separar as condições dos blocos de comandos. Além disso, note o uso da tabulação para separar visualmente os diferentes blocos e declarações: poderíamos escrever todo o código sem usar tabulação, alinhando todas as linhas à esquerda, mas isto dificultaria a identificação das diferentes declarações.

6.3 – O Encadeamento If – Else if

Utilizamos a variação "If – Else If" quando desejamos que o programa teste várias condições em sequência, até encontrar uma que seja verdadeira. Sua sintaxe é muito parecida com a declaração if simples:

```
if (condição)
{
comandos;
}
else if (condição)
{
        comandos;
}
else if (condição)
{
        comandos;
}
else
{
        comandos;
}
```

Cada bloco "else if" deverá testar uma condição diferente. O programa testará todas as condições na sequência, de cima para baixo, até encontrar uma condição verdadeira. Quando isto acontece, os comandos pertencentes ao bloco "verdadeiro" serão executados enquanto todos os outros blocos do encadeamento são ignorados. Caso nenhuma condição "else if" seja verdadeira, executa-se o bloco de comandos pertencente ao else final. Note que o else é opcional: Se o ***else*** final não estiver presente e todas as outras condições forem falsas, então nenhuma ação será realizada.

Utilizando este encadeamento, ampliaremos o programa anterior para que ele teste várias condições sobre um número enviado pelo usuário, até encontrar uma condição verdadeira.

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

int main()
     {
     int teste;
     cout<<"Digite um numero qualquer:\n";
     cin>> teste;
     if (teste <= 50)
          {
```

```
        cout<<"O número que você digitou é menor que 50\n";
        }
else if (teste > 50 && teste <= 100)
        {
        cout<<"O número digitado é maior que 50 e menor ou
igual a 100\n";
        }
else if (teste > 100 && teste <= 200)
        {
        cout<<"O número digitado é maior que 100 e menor ou
igual a 200\n";
        }
else
        {
        cout<<"O numero digitado é maior que 200\n";
        }
system("PAUSE > null");
return 0;
}
```

6.4 – A Declaração Switch

A declaração switch é uma maneira fácil e elegante de se fazer uma tomada de decisão com múltiplas escolhas. Na declaração ***switch***, a variável é sucessivamente testada contra uma lista de inteiros ou constantes caractere. Quando uma associação é encontrada, o conjunto de comandos associado com a constante é executado. Veja a sintaxe de switch abaixo:

```
switch ( variável )
        {
        case valor1:
                comandos;
                break;

        case valor2:
                comandos;
                break;

        ...

        case valorx;
                comandos;
                break;

        default:
                comandos;
        }
```

A declaração switch testa o valor de uma única variável. Note que existem vários

“case”, cada um associado a um determinado valor. A declaração comparará o valor da variável com o valor de cada um dos “case”: quando uma associação é encontrada, os comandos pertencentes ao “case” relacionado são executados. Se nenhuma associação for encontrada, a declaração switch executará os comandos pertencentes ao bloco default (note que o bloco default é opcional: caso ele não exista, caso nenhuma associação seja encontrada a declaração switch termina sem que nenhum comando seja executado). O exemplo abaixo demonstra uma utilização da declaração switch.

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

int main()
{
      int option;
      cout<<"Digite a opção desejada:\n";
      cout<<"1. Opção 1\n";
      cout<<"2. Opção 2\n";
      cout<<"3. Opção 3\n";
      option = 0;
      cin>> option;
      switch(option)
            {
            case 1:
                cout<<"Você escolheu a primeira opção\n";
                break;
            case 2:
                cout<<"Você escolheu a segunda opção\n";
                break;
            case 3:
                cout<<"Você escolheu a terceira opção\n";
                break;
            default:
                cout<<"Você escolheu uma opção inválida!\n";
            }
system("PAUSE > null");
return 0;
}
```

Após o fim de cada bloco de comandos “case”, é comum utilizar o comando “break;”. Este comando interrompe a execução do laço em que o programa se encontra, fazendo com que o programa prossiga para o próximo comando imediatamente após o laço. No caso de switch, o comando break assegura que a execução da declaração switch termine, forçando o programa à voltar para sua execução normal. Caso omitíssemos os comandos break; no fim de cada bloco, a declaração switch executaria os comandos presentes no “case” em que a associação foi encontrada, e continuaria a executar todos os comandos presentes em todos os “case” na sequência (incluindo o bloco default) até o término da declaração switch. Note que este comportamento pode ser útil em alguns programas, como por exemplo uma sequência de operações matemáticas onde utilizamos a declaração switch para escolher o ponto de partida da sequência.

6.5 – A declaração for

Utiliza-se a declaração ***for*** para realizar tarefas repetitivas dentro de um programa, como somar todos os elementos de uma matriz ou exibir na tela uma sequência grande de valores. A declaração for tem o seguinte formato:

for (valor_inicial; condição_testada; valor_incremento)
{
comandos;
}

A declaração for é o que chamamos de **laço ou loop** em programação: um conjunto de comandos que serão executados repetidamente até que uma determinada condição falhe e termine o laço. Em for, determinamos o número de repetições desejadas através de uma variável de controle que será modificada pelos argumentos da declaração for.

- valor_inicial refere-se à atribuição de um valor inicial para a variável de controle, por exemplo: "controle = 0;"
- condição_testada é uma expressão qualquer contendo a variável de controle, que será testada continuamente. Enquanto a condição for verdadeira, os comandos dentro do laço for serão executados. Quando a condição falhar, o laço termina e o programa continua seu fluxo normal. Por exemplo, "controle < 30;" é uma expressão válida, que testa se a variável controle é menor do que 30.
- valor_incremento é uma expressão que incrementará a variável de controle sempre que a condição testada anteriormente for verdadeira. Por exemplo, "controle = controle + 1" ou mais simplesmente "controle++"
- Como já dissemos, os comandos entre as chaves serão executados sempre que a condição testada for verdadeira, e se repetirão até que a condição se torne falsa e o laço termine.

O exemplo de código abaixo mostra uma declaração de laço for utilizando o exemplo dado anteriormente:

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

int main()
{
int controle;
for ( controle = 0; controle <= 30; controle++)
     {
     cout << "Esta frase se repetirá até que a variável
     controle seja maior do que 30\n";
     cout<< "controle = "<<controle<<"\n";
     }
system("PAUSE > null");
return 0;
}
```

A variável controle começa com valor 0. O laço for testa o valor da variável continuamente, sempre determinando se ele é menor do que 30. Sempre que o valor da variável for menor que 30 (condição verdadeira), a variável de controle é incrementada de 1 e os comandos entre as chaves são executados. Quando a variável controle for maior do que 30 (condição falsa), o laço termina e o programa volta a ser executado normalmente.

O comando for é extremamente flexível. Podemos atribuir qualquer valor que quisermos à variável de controle, e também podemos incrementá-la (ou decrementá-la da forma que desejarmos). No exemplo abaixo, utilizamos o laço for de forma inversa ao exemplo anterior: a variável de controle começa em 60 e vai sendo decrementada de 2 a cada repetição.

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

int main()
{
int controle;
for ( controle = 60; controle >= 0; controle = controle - 2)
      {
      cout << "Esta frase se repetirá até que a variável
      controle seja igual a 0\n";
      cout<< "controle = "<<controle<<"\n";
      }
system("PAUSE > null");
return 0;
}
```

Os três argumentos do laço for são opcionais: podemos omitir qualquer um dos argumentos, mas precisamos estar atentos às consequências que isto traz ao programa. Por exemplo, o código abaixo omite o argumento de incremento da variável de controle. No entando, fazemos o incremento da variável dentro do bloco de comandos do laço for:

```
for (x = 0; x <= 50; )
      {
      cout<<x;
      x++;
      }
```

Caso não fizéssemos o incremento dentro do bloco de comandos, a condição testada seria sempre verdadeira, e o laço for nunca terminaria. É o chamado laço infinito:

```
for (x = 0; x <= 50; )
      {
      cout<<x;
      }
```

Laços infinitos geralmente surgem por algum erro de programação. Quando o programa entra em um laço infinito, pode-se pressionar Ctrl+C no modo MS-DOS para

finalizar o programa.

É importante saber que mesmo omitindo um argumento, é necessário incluir o ponto-e-vírgula correspondente. Podemos omitir também a inicialização da variável de controle, caso ela tenha sido declarada e inicializada anteriormente no programa.

Também podemos criar um laço for nulo, sem um bloco de comandos:

```
for (x = 0; x == 50; x++)
        {
        }
```

Nesse caso, o programa ainda é interrompido para a execução do laço for, e não faz nada durante este intervalo. Esta técnica pode ser utilizada para forçar uma pausa na execução de um programa, porém ela é pouco precisa e desperdiça recursos do microprocessador.

Finalmente, podemos utilizar várias variáveis de controle para controlar um mesmo laço for. Para isto, utilizamos vírgulas para separar as expressões relacionadas a cada variável dentro de um mesmo argumento. O código abaixo abaixo ilustra como isto é feito:

```
for (i=0, j=100; i <= 100; i++, j++)
        {
        cout << i << j;
        }
```

A utilização de várias variáveis dentro de um mesmo laço for é particularmente útil em programas que trabalham com matrizes e vetores.

6.6 – A declaração while

Uma outra forma de laço é a declaração while. Seu funcionamento é muito parecido com a declaração for que estudamos anteriormente. Ao encontrar um laço ***while,*** o programa testa a condição especificada. Se a condição for verdadeira, efetuará os comandos contidos no laço. Quando a condição se torna falsa, o laço termina e o programa passa para o próximo comando. A sintaxe da declaração while é a seguinte:

```
while (condição)
        {
        comandos;
        }
```

A declaração while é diferente é diferente da declaração for em alguns aspectos. Em primeiro lugar, while não utiliza variáveis de controle automaticamente. O único argumento entre os parênteses é a condição a ser testada: caso a condição nunca mude e seja sempre verdadeira, estaremos criando um laço infinito. Assim, cabe ao programador inserir uma variável de controle no laço while para evitar que isso ocorra. Isso pode ser feito declarando-se e inicializando-se uma variável antes do laço while, testando esta

variável de controle no laço while e finalmente modificando esta variável (incrementando-a ou decrementando-a) após a execução dos comandos necessários. O programa abaixo demonstra como inserir uma variável de controle no laço while.

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

int main()
{
int controle = 0;
while (controle < 20)
      {
      cout<<"A variavel de controle funcionará se esta frase
      se repetir somente 20 vezes: ";
      controle++;
      cout<<controle<<" \n";
      }
system("PAUSE > null");
return 0;
}
```

6.7 – A Declaração Do While

A declaração do while é muitíssimo parecida com a declaração while, com uma única diferença fundamental: o teste condicional é feito após a execução dos comandos pertencentes ao laço. Veja a sintaxe do laço do while:

```
do
      {
      comandos;
      }
while ( condição);
```

Note a inversão da ordem: primeiro temos a declaração do seguida do corpo do laço, contendo os comandos à serem executados entre as chaves. Após o símbolo fecha-chaves, temos a declaração while e a expressão que será testada. Os comandos entre as chaves serão executados até que a condição torne-se falsa. Porém, lembre-se que a condição só será testada após a execução dos comandos dentro do laço, ao contrário dos laços que vimos anteriormente que antes testavam uma condição para depois executar qualquer comando. O exemplo abaixo ilustra uma utilização simples do laço do while.

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

int main()
{
int controle = 1;
do   {
```

```
     cout<<"Esta frase foi escrita "<<controle<<" vezes!\n";
     controle++;
     }
while (controle <= 20);

system("PAUSE > null");
return 0;
}
```

6.8 – Laços Aninhados

Qualquer um dos laços estudados acima pode ser aninhados, ou seja, colocados um dentro do outro. Esta técnica pode ser muito útil para trabalhar com matrizes multidimensionais, programas que trabalhem com menus e várias outras situações. Entretanto, é preciso atenção para não confundir quais blocos fazem parte de qual laço ou declaração. Como já dissemos anteriormente, é fundamental a utilização de chaves para separar corretamente os blocos e laços, e também a utilização de tabulação na escrita do código para melhor visualização de cada uma das partes do programa. Também é interessante a utilização de comentários no código, para melhor identificação.

O programa exemplo abaixo demonstra a utilização de vários laços e declarações aninhadas, assim como a aplicação de tabulação e comentários para melhor entendimento do programa.

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

int main()
{
int mult1, mult2;
for (mult1 = 1; mult1 <= 10; mult1++)
     {
     cout<<"Tabuada do numero "<<mult1<<":\n";
     for (mult2 = 1; mult2 <= 10; mult2++)
          {
          cout.width(2);
          cout<<mult1<<" X ";
          cout.width(2);
          cout<<mult2<<" = ";
               cout.width(3);
          cout<<mult1*mult2<<"\n";
          }
     }
system("PAUSE > null");
return 0;
}
```

6.9 – Break e Continue

Utilizamos os comandos break e continue para pular partes de um código. Já vimos a utilização do comando break quando falamos sobre a declaração switch: ele é utilizado para interromper a execução de um laço, pulando imediatamente para o próximo comando após o fim do laço. O comando break "quebra" qualquer teste de condição que esteja sendo feito, forçando o laço a terminar abruptamente.

O comando continue é utilizado em um laço de repetição para pular todos os comandos que seriam executados na sequência do comando continue e forçar o laço a pular para a próxima repetição. O comando break força o laço a terminar; já o comando continue pula os próximos comandos que seriam feitos mas continua a executar o laço. O exemplo abaixo mostra a utilização de um comando continue em um programa que simula um menu de opções.

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

int main()
{
int opcao;
    while (opcao!= 5)
          {
           cout<<"Escolha uma opção entre 1 e 4. Escolha 5
para sair do programa\n";
           cin>>opcao;
           if ((opcao> 5) || (opcao< 1))
              {
              continue;     //opção invalida: volta ao inicio
do loop
              }
           switch (opcao)
                 { //início do switch
                      case 1:
                             cout<<"Opção 1 foi escolhida\n";
                             break;
                      case 2:
                             cout<<"Opção 2 foi escolhida\n";
                             break;
                      case 3:
                             cout<<"Opção 3 foi escolhida\n";
                             break;
                      case 4:
                             cout<<"Opção 4 foi escolhida\n";
                             break;
                      case 5:
                             cout<<"Você saiu do programa\n";
                             break;
                 } //fim do switch
          } //fim do laço while
```

```
system("PAUSE > null");
 }
```

Em primeiro lugar, o laço while fará com que o programa rode repetidamente até que a variável opcao, cujo valor é atribuído pelo usuário a cada repetição do programa, tenha valor igual a 5. A declaração if checa se o valor entrado pelo usuário é maior que cinco ou menor que 1. Caso isto seja verdade, o comando continue interrompe a execução de todos os comandos seguintes, fazendo o laço while ser repetido mais uma vez. Caso o usuário entre um valor entre 1 e 5, o comando continue não é executado e o laço while continua para o próximo bloco de comandos, no caso, a declaração switch. A declaração switch é utilizada para mostrar qual opção o usuário escolheu e, no caso do usuário ter escolhido o valor 5, indicar que o programa terminou. Quando o valor 5 é escolhido, a condição do laço while torna-se falsa, terminando o programa.

Módulo 7 – Matrizes

7.1 – Matrizes

Matrizes são variáveis que contém vários valores de um mesmo tipo. Por exemplo, podemos criar a matriz **notas** para armazenar as notas obtidas por 100 alunos em um exame, ou então utilizar uma matriz chamada **gastos_mensais** para anotar nossos gastos mensais ao longo do ano. Uma matriz armazena vários valores de um mesmo tipo: podemos criar matrizes para armazenar qualquer um dos tipos básicos de variáveis, como int, float e char. Cada valor é armazenado separadamente em um **elemento** da matriz, e pode ser acessado e modificado a qualquer momento.

7.2 – Declaração de uma matriz

Para criar uma matriz, precisamos declarar três atributos dela:

- O tipo de valor que vai ser armazenado na matriz
- O nome da matriz, para que possamos acessá-la
- O número de elementos da matriz

A declaração de uma matriz é muito parecida com a declaração de uma variável, bastando adicionar o número de elementos que desejamos que ela tenha. A sintaxe é a seguinte:

<tipo> <nome> [<numero de elementos>];

Por exemplo, caso quiséssemos criar uma matriz chamada catálogo para armazenar 156 inteiros, a declaração seria assim:

int catalogo [156];

Podemos utilizar qualquer tipo de variáveis já estudadas anteriormente para criar uma matriz, como float, int, char. Uma vez criada uma matriz de um determinado tipo, ela só pode receber valores deste tipo. Note que precisamos definir um tipo para uma matriz: não é possível criar uma matriz "genérica" que aceite um tipo qualquer, ou vários tipos. Isso acontece porque ao declarar uma matriz, o compilador aloca memória suficiente para conter o número de valores especificado de acordo com o tipo da matriz. Por exemplo, uma matriz de 100 elementos do tipo int normalmente irá requerer 100*2 ou 200 bytes de memória. Por outro lado, uma matriz de 100 elementos do tipo float irá requerer 100*4 bytes ou 400 bytes.

Assim como uma variável normal, podemos atribuir valores para uma matriz no momento de sua declaração. Isto é feito utilizando o operador de atribuição "=" seguido dos valores contidos entre chaves e separados por vírgulas. Por exemplo, considere a matriz de inteiros "teste" abaixo:

int teste[5] = { 1, 2, 3, 4 , 5};

Também podemos atribuir apenas parte dos valores de uma matriz, por exemplo,

podemos criar uma matriz que comporte 50 valores do tipo float e atribuir apenas 5 valores à ela, deixando para atribuir o restante dos valores no decorrer do programa.

float notas[50] = { 7.65, 8.48, 4.27, 6.78, 9.10 };

A linguagem C++ faz com que toda matriz parcialmente inicializada tenha seus valores restantes automaticamente transformados em zero. Assim, caso precisemos de uma matriz que só contenha zeros, podemos atribuir o primeiro elemento da matriz como zero e deixar que o compilador transforme os elementos restantes em zero, como vemos abaixo:
int zeros[75] = {0};

7.3 – Acessando Valores de uma Matriz

Após criar uma matriz, podemos acessar qualquer valor dentro dela. Cada valor, ou elemento de uma matriz, possui um número próprio. **Toda matriz começa no elemento 0**. Precisamos ter isso em mente quando acessamos valores dentro de uma matriz, pois o primeiro elemento será o elemento "0", o segundo elemento será o elemento "1".

Cada elemento de uma matriz é tratado como uma variável separada. Assim, podemos atribuir valor para um elemento, exibí-lo na tela, utilizá-lo em operações matemáticas e em laços condicionais. O programa abaixo ilustra estas várias ações:

```
#include <iostream>
using namespace std;
int main() {
int matriz[5] = {1,2,3,4,5};
cout<<"o primeiro valor da matriz é: "<<matriz[0]<<endl;
cout<<"o último valor da matriz é: "<<matriz[4]<<endl;
cout<<"Somando o segundo e o quarto elementos da matriz
temos: "<< matriz[1] + matriz[3]<<endl;
matriz[2] = 27;
cout<<"Mudamos o valor do terceiro elemento da matriz para:
"<<matriz[2]<<endl;
system("PAUSE > null");
return 0;
}
```

7.4 – Utilizando Laços para Percorrer Matrizes

Uma das utilizações mais úteis dos laços condicionais é o acesso à vários (ou todos) elementos de uma matriz rapidamente. Podemos utilizar qualquer um dos laços que estudamos, mas sem dúvida o laço for é o mais prático para trabalhar-se com matrizes. Utilizamos a variável de controle do laço para acessar cada um dos elementos desejados (lembre-se que a matriz sempre começa no elemento 0), como vemos no programa abaixo que percorre os elementos de uma matriz, primeiro preenchendo a matriz com os dados entrados pelo usuário, depois exibindo estes dados na tela.

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

int main()
{
int sequencia[4];
for (int i = 0; i < 4; i++) {
     cout << "Entre com o elemento numero "<<(i+1)<<" da
sequencia: ";
     cin >> sequencia[i];
     cout << endl;
     }
cout << "A sequencia entrada pelo usuario foi: ";
for (int i = 0; i < 4; i++) {
     cout << sequencia[i]<<" ";
     }
system("PAUSE > null");
return 0;
}
```

Importante: como vimos no exemplo anterior, podemos utilizar variáveis para acessar os elementos de uma matriz. Da mesma forma, podemos definir constantes para indicar o número de elementos de uma matriz. Essa técnica é muito útil, pois caso precisemos alterar o número de elementos da matriz ao invés de caçarmos no código todas as referências à este número, tudo que precisamos fazer é alterar o valor da constante. Assim, veja o mesmo programa anterior reescrito utilizando uma definição de constante.

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

int main()
{
const int TAMANHO = 4;
int sequencia[TAMANHO];
for (int i = 0; i < 4; i++) {
     cout << "Entre com o elemento numero "<<(i+1)<<" da
sequencia: ";
     cin >> sequencia[i];
     cout << endl;
     }
cout << "A sequencia entrada pelo usuario foi: ";
for (int i = 0; i < 4; i++) {
     cout << sequencia[i]<<" ";
     }
system("PAUSE > null");
return 0;
}
```

7.5 – Matrizes Multidimensionais

Além das matrizes simples de uma única dimensão, C++ permite a criação de matrizes de múltiplas dimensões. As matrizes bidimensionais são sem dúvida as mais utilizadas e as mais úteis, pois comportam-se como tabelas com linhas e colunas. Ao declarar uma matriz multidimensional, adicionamos um conjunto de colchetes para cada dimensão extra. Entre os colchetes de cada dimensão, colocamos o número de elementos que aquela dimensão terá (ou uma variável que represente o número de elementos). Assim:

```
int tabela [10] [5];  //matriz bidimensional
int horas [12] [30] [24]; //matriz de três dimensões
int minutos [12] [30] [24] [60]; //matriz de quatro dimensões
```

Normalmente trabalhamos no máximo com matrizes bidimensionais, mas podem surgir ocasiões onde matrizes de mais de duas dimensões sejam necessárias. Matrizes multidimensionais funcionam como matrizes dentro de matrizes. Por exemplo, uma matriz bidimensional pode ser vista como uma matriz de uma dimensão cujos elementos são outra matrizes. Esta analogia é útil para entender como é feita a inicialização dos valores de matrizes multidimensionais: inicializamos a matriz separando seus elementos por vírgulas, onde cada elemento é uma matriz individual e é inicializada da mesma forma que a matriz "principal". Por exemplo, seja a matriz bidimensional tabela:

```
int tabela [2] [3] = { { 1, 2, 3} ,  { 4, 5, 6}};
```

Veja que cada "elemento" é fechado por chaves e separado por vírgulas. A mesma coisa acontece com matrizes de três dimensões, e assim por diante. O exemplo abaixo mostra a declaração de uma matriz tridimensional. Preste atenção na presença das chaves e das vírgulas separando cada elemento diferente:

```
int tritabela [2] [2] [2] =  {   {{ 9, 8}, {7,6}} ,  {{5, 4},{3, 2}}  };
```

Cada elemento de uma matriz multidimensional pode ser acessado individualmente, indicando a posição exata do valor dentro da matriz. Como vimos anteriormente no caso das matrizes simples, a utilização dos laços condicionais facilita o acesso aos vários elementos de uma matriz. No caso das matrizes multidimensionais, utilizamos laços aninhados para acessar cada dimensão de uma vez. O programa abaixo declara a matriz tridimensional que vimos anteriormente e utiliza uma sucessão de laços for aninhados para exibir a matriz na tela.

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

int main()
{
int tritabela [2] [2] [2] =  {{{ 9, 8}, {7,6}},{{5, 4},{3,
2}}};
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        for (int k = 0; k < 2; k++) {
          cout << tritabela[i][j][k]<<" ";
          }
      cout<<endl;
      }
  cout<<endl;
  }
system("PAUSE > null");
return 0;
}
```

7.6 – Matrizes em Funções

Podemos utilizar matrizes como parâmetros de funções com alguns cuidados simples. Ao declarar os parâmetros que serão utilizados na função, declaramos um parâmetro para a matriz e um parâmetro para seu tamanho separadamente. Fazemos isso para que possamos utilizar matrizes de qualquer tamanho utilizando a mesma função. Ao criar o parâmetro referente à matriz, usamos um conjunto de colchetes vazios para indicar que se trata de uma matriz e não de uma variável simples. No entanto, quando utilizarmos esta função, passamos para ela somente o nome da matriz, sem os parênteses: o programa "já sabe" que uma matriz será enviada, pois já declaramos isso no protótipo da função. O programa abaixo mostra uma função que soma todos os elementos de uma matriz.

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

int soma( int matriz[], int tamanho) {
int resultado = 0;
for (int i = 0; i < tamanho; i++){
     resultado = resultado + matriz[i];
     }
return resultado;
}

int main()
{
const int TAMANHO = 4;
int sequencia[TAMANHO];
int result = 0;
for (int i = 0; i < 4; i++) {
```

```
      cout << "Entre com o elemento numero "<<(i+1)<<" da
sequencia: ";
      cin >> sequencia[i];
      cout << endl;
      }
result = soma(sequencia, TAMANHO);
cout << "A soma de todos os elementos da matriz e igual a
"<<result<<"."<<endl;
system("PAUSE > null");
return 0;
}
```

Também podemos utilizar funções que trabalhem com matrizes multidimensionais. Entretanto, ao criar o parâmetro da matriz é preciso declarar o tamanho de todas as dimensões da matriz com exceção da primeira. Por exemplo:

int soma_tabela (int matriz [ ] [3] [4], int elementos) { comandos; }

A primeira dimensão é deixada em branco, para ter seu tamanho definido pelo outro parâmetro "elementos". Entretanto, as outras duas dimensões tem seu tamanho definido respectivamente em 3 e 4. Caso deixássemos estas outras duas dimensões sem um tamanho definido, o programa não seria compilado.

É muito importante notar que, ao trabalhar com matrizes dentro de funções, estamos trabalhando com a própria matriz, ou melhor, com o endereço dos dados desta matriz. Normalmente, quando trabalhamos com variáveis dentro de uma função, estamos trabalhando com cópias destes valores. Entretanto, isso não é verdade para matrizes, por razões que veremos no módulo sobre ponteiros. Por enquanto, basta saber que quando trabalhamos com matrizes dentro de funções, qualquer modificação feita na matriz é feita para valer. Se somarmos mais 10 a todos os valores de uma matriz em uma função, quando a função terminar a matriz estará modificada. Esta característica é muito útil para contornar o fato que não podemos utilizar uma matriz como retorno de uma função: não precisamos que a função retorne uma matriz, só precisamos fazer com que a função altere a matriz. Por exemplo, o programa abaixo usa uma função para ordenar os números dentro de uma matriz.

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

int ordena( int matriz[], int tamanho) {
int temp = 0;
for (int i = 0; i < tamanho; i++){
      for (int j = i; j < tamanho; j++){
            if (matriz[j] < matriz [i]) {
                  temp  = matriz[i];
                  matriz[i] = matriz[j];
                  matriz[j] = temp;
                  }
            }
      }
}
```

```
int main()
{
const int TAMANHO = 4;
int sequencia[TAMANHO] = {27, 12, 42, -8};
cout <<"Sequencia original: ";
for (int i = 0; i < 4; i++) {
     cout << sequencia[i]<<" ";
     }
cout<<endl;
ordena(sequencia, TAMANHO);
cout <<"Sequencia ordenada: ";
for (int i = 0; i < 4; i++) {
     cout << sequencia[i]<<" ";
     }
cout<<endl;
system("PAUSE > null");
return 0;
}
```

7.7 – Criando Matrizes Dinamicamente

A linguagem C++ prima pela economia de memória durante a execução de seus programas. Quando declaramos uma matriz no início de um programa, o compilador separa um "pedaço" da memória do computador de tamanho equivalente ao tamanho máximo da matriz. Assim, se criarmos uma matriz de inteiros com 200 elementos, o compilador separa na memória do computador um espaço de 400 bytes (2 bytes x 200 elementos). Entretanto, quando trabalhamos com matrizes frequentemente não sabemos qual o tamanho exato de elementos que precisamos. Ao criar uma matriz, estipulamos um valor que seja maior do que o valor máximo de elementos que precisamos, mesmo que não utilizemos todos os espaços disponíveis na matriz. Isso causa desperdício de memória do computador e lentidão na execução de programas.

Para evitar este problema, a linguagem C++ introduz dois novos operadores de controle de memória. Em primeiro lugar, temos o operador **new**, que é utilizado para alocar espaços na memória de computador durante a execução de um programa. Em segundo lugar, o operador **delete,** que libera a memória alocada com o operador new após sua utilização. Este tipo de criação e destruição de variáveis é chamado de alocação dinâmica. Uma matriz criada dessa forma é chamada de **matriz dinâmica**.

A vantagem de se criar matrizes dinâmicas é que a memória utilizada pela matriz só é "tirada" do sistema após a execução do operador **new,** e pode ser liberada novamente após sua utilização com o operador **delete**. Quando criamos uma matriz do jeito "normal", a memória utilizada por ela é guardada pelo programa durante toda sua execução, consumindo recursos desnecessários do computador.

A sintaxe para criar-se uma matriz utilizando o operador new é a seguinte:

*<tipo> * <nome> = new <tipo> [<numero de elementos>];*

Por exemplo, para criar uma matriz do tipo float com 10 elementos:

*float * lista = new float [10];*

O acesso aos elementos da nova matriz é feito de forma análoga ao de uma matriz criada do jeito "normal": utilizamos o nome da matriz seguido do número do elemento acessado. Como já dito anteriormente, o primeiro elemento de uma matriz é sempre o elemento 0.

Note que não podemos inicializar automaticamente os valores de uma matriz criada dessa forma. Cada elemento da matriz deve ter seu valor atribuído separadamente.

A outra face do operador new é o operador delete. Só podemos utilizar este operador com matrizes criadas pelo operador new, ou seja, matrizes criadas por declaração normal não podem ser apagadas durante a execução do programa. Para utilizar o operador delete para apagar uma matriz, a sintaxe é a seguinte:

delete <nome>;

O programa abaixo pede ao usuário o número de elementos da matriz, cria uma matriz e depois volta a pedir ao usuário os valores de cada um dos elementos da matriz. No final do programa, a matriz é deletada.

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

int main() {
int tamanho = 0;
cout<<"Entre com o tamanho da matriz: ";
cin>>tamanho;
cout<<endl;
int *sequencia = new int[tamanho];
for (int i = 0; i < tamanho; i++) {
     cout << "Entre com o elemento "<<(i+1)<<" da matriz: ";
     cin>>sequencia[i];
     cout<<endl;
     }
cout<<"A matriz entrada e: ";
for (int i = 0; i < tamanho; i++) {
     cout << sequencia[i]<<" ";
     }
cout<<endl;
cout<<"Programa encerrado. A matriz criada será deletada, e a
memória será devolvida para o processador.";
delete sequencia;
system("PAUSE > null");
return 0;
}
```

Módulo 8 – Strings

Não falamos até agora de matrizes de caracteres, apesar de já utilizá-las várias vezes. Uma matriz de caracteres é normalmente chamada de "string" em linguagem de programação. Em C++, existem duas maneiras de se criar e trabalhar com strings. Um deles é o método antigo, já utilizado pela linguagem C, de se declarar e trabalhar com uma matriz de caracteres do mesmo modo que trabalhamos com matrizes numéricas. O outro método é utilizar as facilidades da biblioteca padrão de C++, que define um tipo de variável específico para strings, com várias operações e funções próprias. Nesta apostila trabalharemos com as strings do tipo C++, por serem elas mais simples e ao mesmo tempo apresentarem mais recursos do que as strings tipo C.

8.1 – Cabeçalho de um programa com strings

Todo programa escrito em C++ deve conter a seguinte linha de inclusão em seu cabeçalho:

```
#include <string>
```

Isso faz com que o compilador inclua no programa a biblioteca padrão de strings de C++, que contém todas as definições do tipo string, assim como as funções e facilidades relacionados à este tipo. Assim, o cabeçalho de um programa típico envolvendo strings fica assim:

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

8.2 – Declarando e Inicializando uma String

Declaramos strings da mesma maneira que declaramos variáveis: explicitando o tipo da variável (no caso, string) e seu nome. Veja a sintaxe e o exemplo abaixo:

```
string <nome da string>;
string nacionalidade;
string sobrenome;
```

Uma string declarada desta forma estará vazia até que um valor seja atribuído à ela,

das maneiras já estudadas: através de uma atribuição, ou de uma entrada de dados do usuário, por exemplo.

C++ possui uma série de facilidades para a inicialização de strings. Cada um desses diferentes métodos é chamado de "inicializador" de uma string. A tabela abaixo reúne os quatro principais inicializadores de string:

string s1;	Cria uma string vazia, chamada s1. Esta é a inicialização default de uma string: toda string criada dessa forma está vazia.
string s2 (s1);	Cria a string s2 como uma cópia de outra string (nesse caso, s1).
string s2 ("Esta é uma string literal");	Cria a string s2 como uma cópia da string literal entre os parênteses.
string s2 (x, 'c');	Cria a string s2, que contém x cópias do caractere entre aspas (no caso, c).

O programa abaixo exemplifica os métodos descritos acima:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string vazia;
      string ditado("Casa de ferreiro, espeto de pau");
      string copia_ditado(ditado);
      string letra_z( 42, 'z');
      cout <<"Mostrando o conteúdo da string 'vazia':"<< endl;
      cout << vazia;
      cout  <<"Mostrando  o  conteúdo  da  string  'ditado':"<<
      endl;
      cout << ditado;
      cout    <<"Mostrando    o    conteúdo    da    string
      'copia_ditado':"<< endl;
      cout << copia_ditado;
      cout  <<"Mostrando  o  conteúdo  da  string  'letra_z':"<<
      endl;
      cout << letra_z;
      system("PAUSE > null");
      return 0;
}
```

8.3 – Leitura e Escrita de Strings na Tela

Utilizamos cin e cout para ler e escrever strings, assim como fazemos com variáveis

simples. Basicamente, cin e cout tratam strings como se fossem variáveis simples, facilitando sua manipulação e seu uso em programas.

Já vimos no programa exemplo anterior que utilizamos cout para exibir strings inteiras na tela. cout exibe todos os caracteres da string, e detecta automaticamente o fim dela.

Utiliza-se o comando cin para ler strings através do teclado. A sintaxe é a seguinte:

cin >> <nome da strin>;

Como já vimos, cin lerá os caracteres inseridos através do teclado até encontrar um espaço em branco (tecla barra de espaço) ou o fim da entrada (tecla Enter). Quando cin encontra um espaço em branco, todos os caracteres após este espaço são ignorados. O programa abaixo lê entradas do teclado para armazenar o nome e sobrenome do usuário dentro de strings.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string nome;
      string sobrenome;
      cout<<"Digite seu nome: ";
      cin >> nome;
      cout<<"Digite seu sobrenome: ";
      cin >> sobrenome;
      cout << "Seu nome é "<<nome<<" e seu sobrenome é
      "<<sobrenome<<"."<<endl;
      system("PAUSE > null");
      return 0;

}
```

Esse programa apresenta alguns problemas quando o usuário possui mais de um nome ou sobrenome. Por exemplo, caso o nome “José Ricardo” seja inserido, a leitura do sobrenome será “ignorada” e o programa exibirá “José” como o nome e “Ricardo” como sobrenome. Da mesma forma, no caso da entrada de um sobrenome duplo, somente o primeiro sobrenome é guardado na string “sobrenome”, sendo o segundo “ignorado”. Isto acontece por causa da maneira que C++ trata os espaços em branco em uma entrada via teclado.

Espaços em branco são considerados fim de entrada pelo comando cin; ao invés de descartar os caracteres que vierem após o espaço em branco, C++ os guarda em um buffer (uma espécie de “reserva” ou pilha de dados). Quando cin for chamado novamente, antes de ler a nova entrada do teclado, o programa primeiro utiliza os dados que estão nesse

buffer. Assim, temos a impressão que a nova entrada de dados foi descartada pelo programa, mas na verdade ela foi jogada no buffer, esperando uma nova chamada de cin.

Dessa forma, cin não é o método recomendado para ler frases inteiras, com palavras separadas por espaços. Para este objetivo, utilizamos o método cin.getline, que já estudamos no módulo 5, com algumas alterações.

O método cin.getline lê linhas inteiras de entrada através do teclado, sem se importar com os espaços em branco. Mas a sintaxe do método cin.getline é um pouco diferente, quando trabalhamos com strings:

getline (cin, <nome da string>);

Assim, este método recebe todos os caracteres (incluindo os espaços em branco) entrados pelo usuário, até que ele aperte a tecla Enter. O programa abaixo é uma versão melhorada do programa anterior, utilizando o método cin.getline para receber os dados desejados.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string nome;
      string sobrenome;
      cout<<"Digite seu nome: ";
      getline(cin, nome);
      cout<<"Digite seu sobrenome: ";
      getline(cin, sobrenome);
      cout <<"Seu nome é "<<nome<<" e seu sobrenome é
      "<<sobrenome<<"."<<endl;
      system("PAUSE > null");
      return 0;
}
```

8.4 – Operações com Strings

C++ possui uma série de funções e operações próprias para strings. A tabela abaixo resume as operações mais utilizadas (s é uma string qualquer):

s.empty( )	Função que retorna verdadeiro se a string está vazia, e falso caso contrário.
s.size ( )	Função que retorna o tamanho em caracteres da string
s [n]	Acessa um elemento da string. Funciona exatamente com um elemento de uma matriz.

s1 + s2	Concatena duas strings.
s1 = s2	Atribui o conteúdo de s2 na string s1.
s1 == s2	Testa a igualdade entre s1 e s2 (retorna verdadeiro se as duas strings forem iguais). Duas strings são consideradas iguais se elas tiverem o mesmo número de caracteres e seus caracteres forem iguais.

A primeira função, <string>.empty indica se uma string está vazia ou não. Esta função retorna um valor booleano verdadeiro (true ou 1) caso a string esteja vazia, caso contrário ela retorna falso (false ou 0).

A função <string>.size é bastante útil para trabalhar com strings entradas pelo usuário. Como não podemos saber exatamente o número de caracteres entrados pelo usuário, é útil ter uma função que nos retorne o tamanho de uma string.

Como vemos acima, é possível acessar um elemento individual de uma string do mesmo modo que fazemos com matrizes. Esse tipo de acesso é útil quando precisamos manipular os vários caracteres de uma string, como por exemplo, identificar as letras maiúsculas de uma string e transformá-las em minúsculas. Para esse tipo de manipulação caractere à caractere, contamos com as diversas funções da biblioteca cctype, que veremos no item 8.5.

A concatenação de strings é particularmente útil. Quando utilizamos o sinal de soma entre duas strings, estamos concatenando elas, ou seja, juntando o começo da segunda matriz com o final da primeira. Também é possível concatenar strings literais (frases entre aspas) junto com as variáveis string dessa forma. O programa abaixo ilustra a concatenação de strings.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string nome;
      string sobrenome;
      cout<<"Digite seu nome: ";
      getline(cin, nome);
      cout<<"Digite seu sobrenome: ";
      getline(cin, sobrenome);
      string concatena;
      concatena = nome + sobrenome;
      cout << "O seu nome completo é : " + nome + " " +
sobrenome << endl;
      system("PAUSE > null");
```

```
    return 0;
}
```

8.5 – Biblioteca cctype: operações com caracteres

A biblioteca cctype é uma versão da biblioteca ctype da linguagem C, convertida para C++. Ela contém diversas funções que permitem processar os caracteres de uma string, um por um. Por exemplo, podemos precisar saber se um determinado caracter é uma letra ou um número, se está acentuado ou não, se é minúsculo ou maiúsculo, e transformar este caractere. Para utilizar esta biblioteca, precisamos declará-la no cabeçalho do programa, assim como fizemos com a biblioteca de strings. Para declará-la, a sintaxe é a seguinte:

```
#include <cctype>
```

Assim, o cabeçalho de um programa que utiliza strings e a biblioteca cctype ficaria assim:

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

A tabela abaixo resume algumas das funções mais úteis desta biblioteca (x é um elemento de uma string, por exemplo, "sobrenome[4]") :

isalnum (x)	Retorna verdadeiro (1) caso x for uma letra ou um número.
isalpha (x)	Retorna verdadeiro (1) caso x for uma letra.
iscntrl (x)	Retorna verdadeiro (1) caso x for um dígito de controle.
isdigit (x)	Retorna verdadeiro (1) caso x for um número.
isgraph (x)	Retorna verdadeiro (1) caso x não for um espaço.
islower (x)	Retorna verdadeiro (1) caso x for uma letra minúscula.
isprint (x)	Retorna verdadeiro (1) caso x for um caractere imprimível.
ispunct (x)	Retorna verdadeiro (1) caso x for um caractere acentuado.
isspace (x)	Retorna verdadeiro (1) caso x for um esaço em branco.
isupper (x)	Retorna verdadeiro (1) caso x for uma letra maiúscula
isxdigit (x)	Retorna verdadeiro (1) caso x for um número hexadecimal.
tolower (x)	Transforma um caractere maiúsculo em minúsculo.
toupper (x)	Transforma um caractere minúsculo em maiúsculo.

Com exceção das duas últimas funções que transformam caracteres, todas as outras testam os caracteres de uma string, retornando valores booleanos (true ou false, 1 ou 0). O programa abaixo mostra o uso de algumas dessas funções, através da leitura de uma string entrada pelo usuário. Note que também é feito o uso da função <string>.size para determinar o tamanho da string.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
      string frase;
      int letras = 0, maiusculas = 0, minusculas = 0, numeros
      = 0;
      cout<<"Entre com uma frase qualquer, composta de letras
      maiusculas, minusculas e numeros: "<< endl;
      getline(cin, frase);
      letras = frase.size();
      cout<<"Sua frase tem "<< letras<< " letras."<<endl;
      for (int i = 0; i < letras; i++)
         {
          if (isdigit(frase[i])) numeros++;
          if (islower(frase[i])) minusculas++;
          if (isupper(frase[i])) maiusculas++;
          }
      cout<<"Sua frase tem "<< numeros<< " numeros."<<endl;
      cout<<"Sua  frase  tem  "<<  minusculas<<  "  letras
      minusculas."<<endl;
      cout<<"Sua  frase  tem  "<<  maiusculas<<  "  letras
      maiusculas."<<endl;
      system("PAUSE > null");
      return 0;
}
```

Módulo 9 – Ponteiros

9.1 - Endereços de Memória

Já dissemos várias vezes que as variáveis ficam armazenadas em determinados espaços na memória do computador. Ao trabalhar com uma variável, geralmente não precisamos nos preocupar com o endereço desta variável, pois o compilador cuida destes "detalhes" técnicos. Entretanto, pode ser bastante útil saber o endereço de uma variável. Em C++, utilizamos o operador de endereço & para acessar o endereço de uma variável. Sua sintaxe é muito simples, bastando inserir o operador & na frente do nome de uma variável:

&<variável>

O programa abaixo cria uma variável inteira e utiliza cout para exibir o valor da variável e depois seu endereço, utilizando o operador &.

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

int main( )
{
 int valor = 124;
cout << "A variavel armazena o seguinte valor: ";
cout << valor <<endl;
cout << "A variavel esta armazenada no seguinte endereço: ";
cout <<  &valor <<endl;
system("PAUSE > null");
return 0;
}
```

O operador de endereço é utilizável também em matrizes. Quando utilizamos este operador em uma matriz, temos o endereço inicial da matriz, que nada mais é do que o endereço do primeiro elemento da matriz. Também podemos utilizar o operador & para acessar os endereços de cada elemento de uma matriz, como vemos no exemplo abaixo:

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

int main( )
{
 int matriz[3] = { 42, 37, 28 };
cout << &matriz[0] << " : " << matriz[0] << endl;
cout << &matriz[1] << " : " << matriz[1] << endl;
cout << &matriz[2] << " : " << matriz[2] << endl;
system("PAUSE > null");
return 0;
}
```

9.2 – Ponteiros

Em linguagem de programação, chamamos de **ponteiros** as variáveis especiais que armazenam **endereços de memória**. Como já vimos, as variáveis possuem três atributos básicos: seu nome, o valor armazenado por ela e o endereço de memória onde ela está armazenada. Os ponteiros, ao invés de armazenar valores, armazenam endereços de memória. Ao utilizar um ponteiro, podemos acessar este endereço e manipular o valor que está armazenado lá dentro.

Ponteiros são utilizados constantemente em C++. Por exemplo, quando trabalhamos com matrizes sem saber estamos trabalhando com ponteiros: a linguagem C++ (e a linguagem C também) utiliza ponteiros para acessar as diversas posições de uma matriz.

9.3 – Declarando Ponteiros

A declaração de ponteiros é bem parecido com a declaração de variáveis. Assim como uma variável comum, ponteiros precisam ter tipos (como int, float, long) para indicar ao compilador qual o tipo de valor para o qual o ponteiro aponta. Além disso, utilizamos o asterisco * antes do nome do ponteiro, para indicar ao compilador que estamos declarando um ponteiro e não uma variável simples. A sintaxe da declaração de um ponteiro é vista abaixo:

<tipo> * <nome do ponteiro>;

Ponteiros podem ser de qualquer tipo, como int, float, long e até mesmo sem tipo (void). Abaixo temos alguns exemplos de declaração de ponteiro:

*int *referencia;*
*float *media;*

Após declarar um ponteiro, é importante atribuir o endereço que será armazenado por ele. Isto é feito utilizando o operador & para obter o endereço de uma variável. A atribuição é feita da mesma forma que atribuimos valores para uma variável. Suponha que tenhamos em um programa qualquer um ponteiro chamado “referencia” e uma variável chamada “valor”. Caso desejássemos que o ponteiro referencia armazenasse o endereço da variável valor, faríamos a seguinte atribuição:

referencia = &valor;

O programa abaixo é simplesmente o primeiro programa-exemplo deste módulo, reescrito utilizando-se um ponteiro para armazenar o endereço da variável. Note que cout exibe corretamente o endereço armazenado pelo ponteiro, da mesma maneira que exibiria uma variável:

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

```
int main( )
{
 int valor = 124;
 int *referencia;
cout << "A variavel armazena o seguinte valor: ";
cout << valor <<endl;
cout << "A variavel esta armazenada no seguinte endereço: ";
referencia = &valor;
cout <<  referencia <<endl;
system("PAUSE > null");
return 0;
}
```

9.4 – Desreferenciando um Ponteiro

"Desferenciar" um ponteiro nada mais é do que acessar o valor que está armazenado no endereço de memória armazenado pelo ponteiro. Esta operação é feita pelo operador * (que chamamos formalmente de "asterisco de indireção"). Assim como utilizamos o operador de endereço & para acessar o endereço de uma variável, utilizamos o asterisco de indireção para obter o valor que está dentro da posição de memória guardada pela ponteiro. Para utilizá-lo basta adicionar o asterisco antes do nome de um ponteiro em qualquer operação ou comando dentro um programa:

*<*nome do ponteiro*>;

O programa abaixo é bastante parecido com os anteriores, mas utiliza o asterisco de indireção para obter o valor da posição de memória indicada pelo ponteiro. Além disso, este valor é modificado utilizando este mesmo operador:

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

int main( )
{
int valor = 42;
int * ponteiro;
ponteiro = &valor;
cout << "Endereço apontado pelo ponteiro: ";
cout << ponteiro <<endl;
cout << "Valor guardado por este endereço: ";
cout << *ponteiro <<endl;
cout << "Valor atualizado!";
*ponteiro = 12458;
cout << "Novo valor guardado por este endereço: ";
cout << *ponteiro <<endl;
system("PAUSE > null");
return 0;
}
```

Esta operação é útil para acessar e modificar valores dentro de posições de memória. Um ponteiro com o asterisco de indireção se comporta como uma variável comum. Podemos utilizá-lo em comandos, expressões, atribuições e onde mais for necessário.

9.5 –Ponteiros em Funções: Chamada por Referência

Como vimos no módulo 4, podemos criar funções que utilizam dois tipos de chamada de parâmetros: a chamada por valor, onde a função recebe uma cópia do valor de uma variável e a variável em si não é alterada pela função, e a chamada por referência, onde a função recebe o endereço da variável, que tem seu valor alterado durante a execução da função. Nesta seção explicaremos como é feita a chamada de por referência.

A chamada por referência acontece quando passamos um ponteiro como parâmetro de uma função. Já sabemos que o ponteiro armazena um endereço de uma variável: assim, a função recebe o endereço de uma variável, e pode alterar o valor armazenado neste endereço da maneira que precisar.

É preciso uma série de cuidados quando criamos funções que usam chamada por referência. A primeira delas é declarar os ponteiros que serão utilizados como parâmetros no protótipo da função. Por exemplo:

```
void soma ( int * parcela1, int * parcela2) {

//corpo da função

}
```

A declaração de ponteiros como parâmetros é feita da mesma forma que declararíamos ponteiros em um programa, declarando o tipo e utilizando o asterisco para indicar que se trata de um ponteiro.

Ao escrever o corpo da função, também é preciso atenção. Como estamos trabalhando com um ponteiro, é preciso lembrar que um ponteiro indica um endereço. Quando precisamos do valor guardado neste endereço, é preciso usar o asterisco antes do ponteiro. Assim, qualquer operação envolvendo o valor indicado pelo ponteiro deve conter o ponteiro acompanhado do asterisco. Caso utilizemos o próprio ponteiro, não estamos alterando o valor, e sim o endereço. De forma resumida:

ponteiro → endereço
*ponteiro → valor

É muito importante ter isso em mente ao criar funções que utilizem ponteiros. A confusão entre o uso de um ponteiro como endereço ou como valor é a principal fonte de erros nestas situações. O código abaixo é o corpo de uma função que utiliza ponteiros. Note que utilizamos um ponteiro e uma variável normal como parâmetros. Além disso, sempre que utilizamos o valor referenciado pelo ponteiro durante a função, utilizamos o asterisco de dereferenciação antes do nome do ponteiro.

```
void potencia( int *variavel,int elevado) {
int original = 2;
if (elevado == 0) {
        *variavel = 1;
        return;
        }

if (elevado == 1) return;

        for (int i = 2; i <= elevado; i++) {
        *variavel = *variavel*original;
        }
```

Quando utilizamos uma função deste tipo, é preciso que o programa passe um ponteiro como parâmetro desta função. Isto pode ser feito de duas maneiras: indicando um ponteiro que contenha um endereço, ou indicando simplesmente o endereço de uma variável. Como já vimos, utilizamos o operador & antes do nome de uma variável para obter seu endereço. Assim, uma função que utiliza ponteiros aceita em seu parâmetro um ponteiro ou o endereço de uma variável (&variável).

O programa abaixo mostra a utilização de uma função com chamada por referência, tendo em mente os cuidados descritos anteriormente.

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

void potencia( int *variavel,int elevado) {
int original = 2;
if (elevado == 0) {
    *variavel = 1;
    return;
    }
if (elevado == 1) return;
    for (int i = 2; i <= elevado; i++) {
    *variavel = *variavel*original;
    }
}

int main()
{
int dois = 2;
int j = 5;
potencia( &dois, j);
cout << "dois elevado a "<< j <<" : "<< dois << endl;
system("PAUSE > null");
return 0;
}
```

A utilização da chamada por referência parece não ter muita utilidade quando

trabalhamos com variáveis simples. Porém, este cenário muda quando trabalhamos com matrizes: a utilização da chamada por referência é a melhor forma de se criar funções que trabalham com matrizes.

9.6 – Ponteiros para Matrizes

Já vimos que podemos utilizar matrizes como parâmetros de funções e que não podemos utilizar matrizes como retorno de funções. Também já vimos que essa limitação pode ser superada graças ao fato das funções poderem alterar as matrizes que recebem como parâmetros. Assim, caso precisemos de uma função que some 20 à todos os elementos de uma matriz, não podemos criar uma função que retorne uma cópia da matriz alterada, mas podemos criar facilmente uma função que altere a própria matriz, utilizando a chamada por referência.

Isto acontece porque quando trabalhamos com funções, C++ trata o nome de uma matriz como o endereço de seu primeiro elemento. Assim:

matriz = = &matriz [0]

Assim, quando utilizamos uma matriz como parâmetro de um função, estamos na verdade enviando o endereço de seu primeiro elemento para a função, como numa chamada por referência. A partir deste endereço, a função tem acesso a todos os elementos da matriz, e pode fazer todas as alterações que precisar nesta matriz. Podemos utilizar o nome da matriz como se fosse um ponteiro: utilizando o asterisco para obter seu valor, e realizar operações de aritmética de ponteiros para acessar seus elementos. Entretanto, é mais fácil utilizar a matriz do modo que já conhecemos: utilizando o nome da matriz e o número do elemento entre chaves para acessar e alterar qualquer um de seus elementos. Abaixo explicaremos como funciona a **aritmética dos ponteiros**.

Já sabemos que um ponteiro é um valor que aponta para uma posição de memória específica. Se somar-se o valor 1 a um ponteiro, o ponteiro apontará para a próxima posição de memória. Se somar-se 5 ao valor de um ponteiro, o ponteiro apontará para a posição de memória de cinco posições adiante do endereço atual. No entanto, a aritmética de ponteiro não é tão simples quanto parece. Por exemplo, assuma que um ponteiro contenha o endereço 1000. Se fosse somado 1 ao ponteiro, poderia se esperar que o resultado fosse 1001. No entanto, o endereço resultante depende do tipo de ponteiro. Por exemplo, se fosse somado 1 a um ponteiro do tipo char (que contém 1000), o endereço resultante será 1001. Se fosse somado 1 a um ponteiro do tipo int (que requer dois bytes na memória), o endereço resultante será 1002. Quando incrementamos um ponteiro de um determinado tipo, o próprio compilador já faz a conversão de tipo para nós. Assim, podemos utilizar o incremento de um ponteiro para acessar os vários elementos de uma matriz de qualquer tipo. O programa abaixo utiliza a aritmética de ponteiros para acessar uma matriz. Cria-se um ponteiro que aponta para o endereço do primeiro elemento da matriz (lembre-se sempre que o nome de uma matriz indica o mesmo endereço de seu primeiro elemento), e a partir daí vamos incrementando este ponteiro para exibir todos os elementos da matriz. Inicialmente, incrementamos o ponteiro "manualmente" para ilustrar melhor o que estamos fazendo, mas podemos facilmente utilizar um laço for para incrementar o ponteiro e percorrer a matriz toda.

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

int main()
{

int tabela[5] = { 20, 34, 58, 70, 125 };
int *ponteiro = tabela;
cout << *ponteiro << endl;
ponteiro++;

cout << *ponteiro << endl;

for (int i = 2; i < 5; i++){
ponteiro++;
cout << *ponteiro << endl;
}
system("PAUSE > null");
return 0;
}
```

O programa abaixo ilustra a criação de duas funções que utilizam matrizes. As duas funções fazem a mesma coisa (organizar uma matriz em ordem crescente), mas a primeira função trata a matriz como um ponteiro, enquanto que a segunda trabalha com a matriz do modo já conhecido. Os dois métodos são válidos e tem sua utilidade, mas de modo geral é mais simples e recomendável utilizar matrizes do modo já estudado.

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

int ordenamatriz( int matriz[], int tamanho) {
int temp = 0;
for (int i = 0; i < tamanho; i++){
       for (int j = i; j < tamanho; j++){
          if (matriz[j] < matriz [i]) {
          temp  = matriz[i];
          matriz[i] = matriz[j];
          matriz[j] = temp;
          }
       }
    }
}

int ordenaponteiro (int *matriz, int tamanho)
{
int temp = 0;
    for (int i =0; i < tamanho; i++) {
       for (int j = i; j < tamanho; j++){
          if  ( *(matriz + j) < *(matriz + i ) ) {
          temp = *( matriz + i );
```

```
            *(matriz + i) = *(matriz + j);
            *(matriz + j) = temp;
            }
        }
    }
}

int main()
{
const int TAMANHO = 4;
int sequencia[TAMANHO] = {27, 12, 42, -8};
int sequencia2[TAMANHO] = {21, -27, 1024, 42};

cout<<endl;
ordenamatriz(sequencia, TAMANHO);
ordenaponteiro(sequencia2, TAMANHO);

cout <<"Sequencia ordenada 1: ";
for (int i = 0; i < 4; i++) {
cout << sequencia[i]<<" ";
}
cout<<endl;

cout <<"Sequencia ordenada 2: ";
for (int i = 0; i < 4; i++) {
cout << sequencia2[i]<<" ";
}
cout<<endl;

system("PAUSE > null");
return 0;
}
```

9.7 – Funções que Retornam Ponteiros

Podemos criar funções que retornam ponteiros para o programa que as chamou. Estas funções são úteis, por exemplo, para trabalhar com matrizes e strings, retornando o endereço da matriz ou string modificada para o programa. Para declarar uma função que retorne um ponteiro, é preciso declarar o tipo de retorno no protótipo da função, não se esquecendo do asterisco. Por exemplo, veja o protótipo de função abaixo:

int *calculaseno ( );
char *maisculas( );

9.8 – Ponteiros para Funções

Encerrando os tópicos relacionados com ponteiros e funções, temos os ponteiros para funções. Um ponteiro para uma função armazena o endereço de uma função. Este tipo de ponteiro é útil para ser usado como parâmetro de outra função. Sua declaração é parecida com a declaração de funções que retornam ponteiros, mas não deve ser confundida. Sua sintaxe é a seguinte:

<tipo do ponteiro/função> ***(*<nome da função>) ();***

Basicamente, coloca-se o asterisco antes do nome da função, junto dos parênteses, Alguns exemplos:

int (*seno) ( );
int (*integral) ( );

9.9 – Ponteiros para Ponteiros

Ponteiros, assim como variáveis, ocupam um endereço na memória do computador. Podemos então criar ponteiros que apontam para o endereço de outro ponteiro, como vemos no código exemplo abaixo:

```
int inteiro = 1024;
int *pontinteiro = &inteiro;
int *pontponteiro = &pontinteiro;
```

Declaramos um ponteiro para um ponteiro utilizando dois asteriscos antes de declarar seu nome. No programa acima temos a seguinte sequência:

pontponteiro → pontinteiro → inteiro

Utilizando o operador de desreferência * no ponteiro “pontpvalor”, recebemos o valro armazenado pela variável “inteiro”, como já era de se esperar. Quando utilizamos o operador de desreferência * no ponteiro “pontppont”, o que recebemos é o ponteiro “pontpinteiro”, ou melhor, o valor armazenado por ele: o endereço da variável inteiro. Para acessar o valor da variável “inteiro” utilizando o ponteiro “pontppont”, precisamos utilizar o operador de desreferência * **duas vezes.** O programa abaixo mostra três maneiras diferentes de se acessar o valor de “inteiro”: acessando a própria variável, utilizando um ponteiro e o operador de desreferência, ou utilizando um ponteiro para ponteiro e o operador de desreferência duas vezes.

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

int main()
{

int inteiro = 1024;
int *pontinteiro = &inteiro;
int **pontponteiro = &pontinteiro;
cout << "Acessando o valor da variavel inteiro: "<<endl;
```

```
cout << "De forma direta (pela propria variavel): "<< inteiro
<< endl;

cout << "De forma indireta (pelo ponteiro para variavel
pontinteiro): "<< *pontinteiro << endl;

cout << "De forma duplamente indireta (pelo ponteiro para
ponteiro pontponteiro): "<< **pontponteiro << endl;

 system("PAUSE > null");
return 0;
}
```

Na verdade, C++ nos permites criar infinitas indireções (ponteiros para ponteiros para ponteiros...e assim por diante). Mas na prática, a utilização de ponteiros para ponteiros é geralmente evitada por causar grande confusão e dificultar a compreensão dos programas.

9.10 – Operadores new e delete

Já falamos no módulo sobre matrizes da importância dos operadores new e delete. A linguagem C++ prima pela economia de memória, para possibilitar a construção de programas maiores e/ou mais rápidos. Assim, temos dois operadores que gerenciam a alocação e a liberação de memória dinamicamente, possibilitando a criação e destruição de variáveis durante a execução de um programa para evitar o desperdício de memória. Este tipo de criação e destruição de variáveis é chamado de alocação dinâmica.

A vantagem de se criar variáveis dinâmicas é que a memória utilizada por esta variável só é "tirada" do sistema após a execução do operador **new,** e pode ser liberada novamente após sua utilização com o operador **delete**. Quando declaramos uma variável do jeito "normal", a memória utilizada por ela é guardada pelo programa durante toda sua execução, consumindo recursos desnecessários do computador.

A sintaxe para se criar uma variável com o operador new é a seguinte:

*<tipo> * <nome> = new <tipo>;*

Como você deve ter percebido, estamos criando um ponteiro. A novidade está do lado direito do operador de atribuição: o uso do operador new faz com que o sistema separe um espaço na memória para armazenar uma variável do tipo declarado (ou ainda uma matriz, como vimos anteriormente). Para acessar este espaço, utilizamos o ponteiro que acabamos de criar, utilizando o operador de desreferência * para acessar o valor que está armazenado ali.

A outra face do operador new é o operador delete. Só podemos utilizar este operador com variáveis criadas pelo operador new, ou melhor, com os ponteiros que usamos para acessar a memória alocada com o operador new. O uso do operador delete é bastante simples:

delete <nome>;

Após o uso do operador delete, o espaço alocado por new é totalmente liberado

para o sistema.

Apesar da vantagem da economia de memória apresentada pelos operadores new e delete, é preciso saber aonde utilizá-los. Eles são especialmente úteis para matrizes cujo tamanho exato não conhecemos, ocasionando grande economia de memória. Entretanto, para variáveis que serão utilizadas o tempo todo pelo programa, a economia de memória obtida não é tão grande, e a alocação/liberação constante de memória pode deixar o programa até mais lento. Assim, recomenda-se a utilização da declaração "normal" de variáveis na maioria dos programas que criamos.

Módulo 10 - Entrada e Saída de Dados

A maioria dos programas de computador trabalham com arquivos. Processadores de texto criam e editar arquivos de texto; navegadores de internet interpretam e exibem arquivos HTML; compiladores leêm arquivos-fonte e geram arquivos executáveis. Um arquivo nada mais é do que uma sequência de bytes armazenada em um dispositivo, seja esse dispositivo ,por exemplo, um disco rígido, um CD ou um disquete. Tipicamente, o sistema operacional gerencia os arquivos presentes em um computador, mantendo registro de onde estão armazenados, quais os seus tamanhos, quando foram criados, etc.

Quando trabalhamos com arquivos em programação, precisamos de meios para conectar um programa à um arquivo, de modo que o programa possa ler e escrever dados dentro deste arquivo, e também meios para criar novos arquivos e salvá-los em um dispositivo. A linguagem C++ possui um pacote de classes e funções que trabalham com arquivos de forma bastante semelhante às classes cout e cin já estudadas.

10.1 – A biblioteca fstream

O primeiro passo para manipular um arquivo em C++ é adicionar a biblioteca específica para a manipulação de dados em arquivos ao cabeçalho de nosso programa. Esta biblioteca é chamada de "fstream", de "file stream" (fluxo de arquivos). Para adicioná-la ao cabeçalho, fazemos:

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

Note que os nomes das duas bibliotecas são parecidos: ambas contém classes que trabalham com entrada e saída de dados. No entanto, a biblioteca iostream trabalha apenas com o fluxo de dados via monitor/teclado/periféricos, enquanto que a biblioteca fstream trabalha com o fluxo de dados de arquivos. (Na verdade, atualmente a biblioteca fstream engloba todas as classes contidas em iostream, mas nem todos os compiladores já adotaram essa modificação).

10.2 – Os objetos de fstream

Uma vez adicionada a biblioteca, podemos criar objetos em nossos programas que servirão de intermediários entre o programa e os arquivos manipulados. A biblioteca fstream define três tipos de objeto para esta função, cada um com objetivo definido:

ofstream: objetos que escrevem dados em um arquivo.

ifstream: objetos que leêm dados em um arquivo.

fstream: objetos que podem tanto ler como escrever em um arquivo.

Explicaremos o uso de cada um deles, cada vez nos aprofundando mais no assunto. Por enquanto, veremos como criar um arquivo e escrever algumas strings nele utilizando um objeto **ofstream.**

10.3 – Escrevendo em um arquivo

Podemos resumir assim as etapas necessárias para escrever em um arquivo através de um programa em C++:

- Cria-se um objeto do tipo ofstream.

- Associa-se este objeto com um arquivo em particular (seja criando ou abrindo um arquivo já existente).

- Usa-se o objeto para enviar dados para este arquivo, de forma bem parecida como usamos o comando cout. A diferença é que os dados vão para o arquivo, ao invés de serem exibidos na tela.

Para criar um objeto ofstream, declaramos seu nome de maneira parecida com a declaração de uma variável:

ofstream <nome do objeto>

Por exemplo:

ofstream escreve;

A linha acima cria o objeto "escreve", do tipo ofstream, capaz de escrever em arquivos. O próxima passo é associar este objeto a um arquivo. Para isto, utilizamos a função open( ), que abre o arquivo desejado ou cria um arquivo no disco rígido. A sintaxe de open( ) é a seguinte:

<objeto>.open("nome do arquivo");

Por exemplo, vamos utilizar o objeto "escreve" para criar um arquivo chamado "strings.txt". Para fazer isso, utilizamos a função open da seguinte maneira:

escreve.open("strings.txt");

Dessa forma, estamos associando o objeto escreve ao arquivo strings.txt. Agora podemos enviar dados através do objeto "escreve", e estes dados serão escritos no arquivo "strings.txt". Para fazer isso, utilizamos o objeto que criamos da mesma forma que utilizamos o comando cout. Da mesma forma como escreveríamos na tela do computador com o comando cout utilizando variáveis, strings, strings literais, formatação, etc, podemos utilizar estes recursos todos também com os objeto do tipo ofstream, com a diferença que estas informações serão gravadas em um arquivo. O exemplo abaixo reúne todos os passos anteriores e utiliza o objeto "escreve" para escrever diversos dados em um arquivo, para dar uma idéia melhor de suas possibilidades.

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

```
int main(){

string frase;
cout<<"Escreva uma frase para ser escrita no arquivo
string.txt:";
getline(cin, frase);
cout<<"Obrigado. Escrevendo dados no arquivo
strings.txt...\n";
ofstream escreve;
escreve.open("strings.txt");
escreve << "Utilizamos os objetos ofstream para escrever em
arquivos\n";
escreve<< "Note que podemos utilizar os caracteres \\n pra
quebrar a linha, como fazíamos em cout\n";
int numero = 100;
escreve<<"Podemos escrever o valor de variaveis numericas: "
<<numero <<"\n";
int matriz[3] = {42, 19, 99};
escreve<<"Podemos também escrever matrizes!";
for (int i=0; i < 3; i++){
     escreve.width(6);
     escreve<<matriz[i]<<" ";
     }
escreve<<"\n";
escreve<<"Finalmente, podemos receber dados via cin e
escrever estes dados no arquivo!\n";
escreve<<"A frase que você digitou durante a execução do
programa: "<<frase<<"\n";
escreve.close();
cout<<"Dados escritos no arquivo. Fim do Programa!";
system("PAUSE");
return 0;
}
```

Procure o arquivo strings.txt em seu sistema, e abra ele (ele provavelmente está no mesmo diretório onde está o arquivo executável do programa que compilamos. Por definição, estes arquivos ficam no diretório onde está o compilador, a menos que tenhamos configurado o compilador para salvá-los em outro lugar.). Percebeu a semelhança entre a utilização do objeto ofstream com o comando cout? Todos os métodos que vimos com cout para exibir um dado na tela podem ser utilizado para gravar um dado em um arquivo.

10.4 – Checando se o arquivo abriu sem problemas

Sempre existe a possibilidade de erros quando trabalhamos com arquivos. Talvez o arquivo que desejamos ler tenha sido apagado, ou renomeado, ou esteja sendo usado por outro programa. A biblioteca de C++ possui uma função que checa se um objeto ofstream/ifstream conseguiu abrir um determinado arquivo, e se continua conectado corretamente à este arquivo: é a função is_open( ). Esta função checa o estado do objeto, e retorna 0 para o programa caso tudo esteja certo. Caso um valor diferente de 0 seja enviado ao programa, isso indica que o arquivo não pode ser aberto pelo objeto.

O código abaixo pode ser utilizado em qualquer programa que utilize leitura e escrita de arquivos (basta, logicamente, substituir os nomes do objeto e do arquivo) para checar o estado do objeto. Caso um erro seja encontrado, o programa indica que o arquivo não pode ser aberto e a execução é terminada. Caso contrário, o programa continua sua execução normalmente.

```
if(!leitura.is_open( ))
{
cout<<"Não foi possível abrir arquivo! Programa será terminado!\n";
leitura.clear( ); //reseta o objeto leitura, para limpar memória do sistema
return 0;
}
```

10.5 – Fechando um Arquivo

Utilizamos a função close( ) no programa acima após escrever todos os dados desejados no arquivo. Como você deve ter adivinhado, a função close simplesmente fecha o arquivo a que o objeto estava associado, liberando o objeto e a memória do arquivo. Após fecharmos o arquivo antigo, podemos associar o objeto à outro arquivo no mesmo programa. Dessa forma, um objeto ofstream (na verdade, qualquer um dos objetos derivados de fstream) pode trabalhar com múltiplos arquivos dentro de um mesmo programa, porém não simultaneamente (caso precisemos trabalhar com múltiplos arquivos simultaneamente, é preciso criar mais de um objeto fstream). A sintaxe da função close( ) é a seguinte:

```
<objeto>.close( );
```

Note que não é necessário especificar o arquivo à ser fechado, pois cada objeto só consegue estar conectado à um único arquivo por vez. No programa acima, por exemplo, utilizamos o seguinte comando para fechar o arquivo string.txt:

```
escreve.close( );
```

10.6 – Lendo os dados de um arquivo

Para ler os dados de um arquivo em um programa, é preciso antes de mais nada criar um objeto do tipo ifstream, capaz de ser o intermediário entre o arquivo à ser lido e o programa. A criação do objeto ifstream é muito semelhante à criação de um objeto ofstream: no exemplo de código abaixo, cria-se um objeto ifstream chamado "leitura" e conecta-se ele ao arquivo string.txt criado previamente.

```
ifstream leitura;
leitura.open("string.txt");
```

Após criarmos o objeto e conectarmos ele à um arquivo, podemos começar a ler através deste arquivo. A leitura de dados de um arquivo é parecida com a leitura de dados enviados através do teclado: criamos uma variável para armazenar os dados recebidos, e utilizamos o objeto ifstream para ler e enviar os dados do arquivo para o programa (no caso da leitura via teclado, o objeto cin receberia os dados enviados pelo usuário e os repassaria

para o programa). Assim, podemos utilizar qualquer um dos métodos de leitura que estudamos para cin, seja para ler caracteres, palavras ou frases inteiras.

Para lermos somente um caractere de um arquivo, por exemplo, nossa melhor opção é utilizar o método .get, que serve justamente para esta função. Criamos uma variável do tipo char para armazenar este caractere, e utilizamos o objeto ifstream da forma mostrada no exemplo abaixo:

char armazena;
leitura.get(armazena);

O exemplo de código acima utiliza o objeto "leitura" e o método .get para ler um caractere de um arquivo, e armazena este caractere na variável "armazena". Poderíamos criar um laço de repetição para obter todos os caracteres do arquivo, como vemos no exemplo abaixo. O laço while continuará a executar o método .get para ler o arquivo, até que seja encontrado o fim do arquivo. Quando o fim do arquivo é encontrado, o comando "leitura.get(armazena) retorna "falso" para o laço while, terminando a repetição. Lembre-se que o arquivo strings.txt deve estar no mesmo diretório onde estamos executando o programa, caso contrário erros ocorrerão!

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

int main(){

char letra;
ifstream leitura;
leitura.open("strings.txt");

if(!leitura.is_open( ))
     {
     cout<<"Não foi possível abrir arquivo! Programa será
terminado!\n";
     leitura.clear( ); //reseta o objeto leitura, para limpar
memória do sistema}
     }

while (leitura.get(letra)) {cout << letra;}
leitura.close();
system("PAUSE");
return 0;
}
```

Para lermos uma palavra inteira de um arquivo, ao invés de utilizarmos uma simples variável de tipo char, utilizamos uma matriz do tipo char. Da mesma maneira que o comando cin, o objeto lerá todos os caracteres em seu caminho, até que a matriz atinja seu tamanho máximo especificado OU encontre um espaço em branco, uma quebra de linha ou o fim do arquivo.

char matriz_chars[80];
leitura >> matriz_chars;

Finalmente, para lermos uma linha inteira de um arquivo, utilizamos o método .getline, que já havíamos estudado no módulo 5. O método getline lê uma linha inteira de entrada, até que o tamanho máximo especificado seja atinjido ou até encontrar uma quebra de linha ou o fim de arquivo. Relembrando sua sintaxe:

<nome do objeto>.getline (<matriz_destino>, <limite de caracteres>);

Por exemplo

char matriz_chars[80];
leitura.getline(matriz_chars, 80);

O comando acima leria todos os caracteres de uma linha, até encontrar a quebra da linha. Como vimos com o método .get, podemos utilizar um laço de repetição para lermos todas as linhas de um arquivo utilizando o método getline. No exemplo abaixo, à cada repetição uma linha do arquivo será escrita na tela.

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

int main(){

char depot[300];
ifstream leitura;
leitura.open("strings.txt");

if(!leitura.is_open( ))
{
cout<<"Não foi possível abrir arquivo! Programa será
terminado!\n";
leitura.clear( ); //reseta o objeto leitura, para limpar
memória do sistema}
}

while (leitura.getline(depot, 300)) {cout << depot <<"\n";}
leitura.close();
system("PAUSE");
return 0;
}
```

Uma pequena diferença que vemos entre esses dois métodos de ler um arquivo inteiro, é que o método .get lê todos os caracteres até o fim de arquivo, incluindo quebras de linha. O método getline utiliza as quebras de linha para determinar o fim de sua leitura, e descarta elas. Assim, no exemplo logo acima, tivemos que instruir o programa à

adicionar uma quebra de linha após exibir na tela cada uma das linhas lidas. (Entretanto, lembre-se que para arquivos grandes, o método getline é mais eficiente e mais rápido. Tenha em mente que cada acesso de um objeto fstream à um arquivo demanda um certo processamento: em um arquivo grande, se temos centenas de linhas, então temos milhares de caracteres, o que significa que o método get acessará muito mais vezes o mesmo arquivo do que o método getline).

10.7 – Modos de Arquivo

Quando abrimos um arquivo, podemos querer fazer diferentes coisas com o conteúdo deste arquivo. Podemos querer continuar à escrever no fim do arquivo, dando continuidade aos dados já escritos nele. Podemos querer apagar todos os dados já escritos e começar do zero novamente. Os modos de arquivo servem justamente para isto: para definir como deve ser o comportamento de um arquivo quando acessado pelo programa. Podemos definir que o arquivo será utilizado somente para leitura ou somente escrita, se será utilizado para anexar dados ou se será reescrito totalmente. Definimos o modo do arquivo quando abrimos ele com um objeto, adicionando um parâmetro novo ao método open. Veja a sintaxe abaixo:

<objeto ofstream>.open("nome do arquivo", ofstream::<modo de arquivo>);

ou

<objeto ifstream>.open("nome do arquivo", ofstream::<modo de arquivo>);

A tabela abaixo resume os modos de arquivo disponíveis em C++.

`ios_base::in`	Abre arquivo para leitura.
`ios_base::out`	Abre arquivo para escrita.
`ios_base::ate`	Procura o final do arquivo ao abrir ele.
`ios_base::app`	Anexa os dados à serem escritos ao final do arquivo.
`ios_base::trunc`	Trunca os dados existentes no arquivo.
`ios_base::binary`	Abre e trabalha com arquivos em modo binário.

Note que um arquivo aberto por um objeto ofstream não necessita que definamos o modo de arquivo ofstream::out, pois este modo já é definido para este tipo de objeto por definição. O mesmo ocorre com o modo ifstream::in e os objetos ifstream.

Além disso, por definição, um arquivo aberto por um objeto ofstream irá truncar os dados já existentes no arquivo, escrevendo os dados novos "por cima" dos antigos. Porém, o modo ofstream::app nos permite anexar dados ao final de um arquivo. O exemplo abaixo utliza o modo ofstream::app para criar uma espécie de "agenda" simplificada em um arquivo de texto.

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

int main(){
int dia, mes;
char letra[1000];
cout<<"PROGRAMA AGENDA Versão 0.00042\n";

ofstream agenda;
agenda.open("agenda.txt", ofstream::app);
cout<<"Digite o compromisso no espaço abaixo (): \n";
cin.getline(letra, 1000);
cout << "Digite o dia do compromisso:\n";
cin >> dia;
cout << "Digite o mês do compromisso:\n";
cin >> mes;

agenda << "Compromisso marcado para dia "<<dia<<" de
"<<mes<<": ";
agenda << letra;
agenda<<"\n";
cout<<"Obrigado! Sua agenda foi atualizada com sucesso!";
agenda.close();
system("PAUSE");
return 0;
}
```

Módulo 11 – Programação Orientada à Objetos

Nos módulos anteriores, cobrimos todos os aspectos essenciais da programação estruturada em C++. Entretanto, a linguagem C++ possui muito mais aspectos do que os tratados até agora. Pode-se dizer que tocamos apenas na ponta do iceberg: o verdadeiro poder de C++ está na sua versatilidade e capacidade de ser programada em vários paradigmas diferentes, seja programação estruturada, orientada à objetos ou orientada à templates. O módulo à seguir tenta fazer uma explicação básica sobre o que é a programação orientada à objetos, dando aos leitores uma breve introdução aos conceitos essenciais deste paradigma, sem adentrar em exemplos de código.

11.1 – Paradigmas de Programação

Como dissemos no primeiro módulo desta apostila, paradigmas de programação são conjuntos de idéias que fornecem ao programador uma visão sobre a estruturação e execução de um programa. Assim como ao resolver um problema podemos adotar uma entre variadas metodologias para resolvê-lo, ao criar um programa podemos adotar um determinado paradigma de programação para desenvolvê-lo.

O primeiro paradigma que aprendemos quando começamos à estudar programação, qualquer que seja a linguagem, é o paradigma da programação estruturada. A programação estruturada tem como objetivo escrever programas que sigam uma lógica linear, ou seja, tenham começo, meio e fim. À grosso modo, um programa escrito dessa forma começa com a declaração das variáveis que serão utilizadas, seguindo para a execução de comandos, funções e tomadas de decisão numa sequência linear, até que todas as linhas de código tenham sido executadas e o programa atinja seu final. É a maneira mais simples de se escrever um programa, principalmente porque utilizamos essa mesma lógica linear no dia-a-dia: faça tal tarefa, depois caso tal condição seja verdadeira realize outra tarefa, e assim por diante. Porém, quando precisamos escrever programas realmente grandes ou realmente complexos, a programação estruturada torna-se ineficiente. Imagine um editor de textos, onde se pode escrever em um documento, criar um novo documento, alterar fontes, tamanhos, parágrafos, tabulações, executar correções ortográficas...agora imagine como acomodar todas as diferentes ações que um usuário pode executar em um programa de estrutura linear.

11.1 – Programação Orientada à Objetos

O paradigma da programação orientada à objetos nasceu no começo da década de 1970, e tinha por objetivo propor uma nova maneira de se olhar para a elaboração de programas. Para a programação orientada à objetos, um programa deve ser visto como uma coleção de objetos que trabalham cooperando entre si, ao contrário de uma lista de instruções que deve ser seguida pelo computador. Desse modo, cada objeto dentro de um programa deve ser visto como uma "máquina" independente com um determinado papel ou objetivo na execução do programa. Estes objetos devem se comunicar entre si, recebendo, processando e enviando dados para os outros objetos do programa.

11.2 – Conceitos Básicos

A unidade principal da programação orientada à objetos é a **Classe**. Uma classe é uma representação de um objeto ou idéia **real** em forma de código, separando as características operacionais dos detalhes concretos de sua implementação. Simplificando, definimos uma Classe para representar algo: por exemplo, um relógio. Para o usuário, não é importante como o relógio funciona, tanto na vida real como na programação: é importante apenas que o usuário possa interajir com o relógio, seja olhando as horas, seja ajustando o alarme ou configurando o horário correto. As engrenagens ou funções que fazem o relógio funcionar ficam "escondidas" do usuário: é isso que queremos dizer com "separar as características operacionais dos detalhes da implementação".

Em termos de programação, uma classe é uma estrutura que contém variáveis e funções com o objetivo de representar uma idéia. Dentro de uma classe, as variáveis e funções são chamadas de "membros da classe". Ainda pensando no exemplo do relógio, dentro desta classe teríamos variáveis (ou membros) para as horas, minutos, segundos e também para armazenar o horário do alarme, assim como funções-membro para ver o horário atual, ajustá-lo ou ajustar o alarme. Porém, o usuário não precisa ter acesso à todas estas variáveis: eles só precisa interajir com as características operacionais do relógio. Assim, criamos funções-membro para fazer esta interação: uma função para ver as horas, outra para ajustar o horário e outra para ajustar o alarme. Dessa forma, simplificamos a utilização da classe e também evitamos que o usuário acesse e altere dados que possam atrapalhar o funcionamento do programa.

Dizemos que os membros da classe que o usuário pode acessar são "públicos", enquanto que os membros internos são "privados". Um membro privado não pode ser acessado ou alterado pelo usuário: não queremos o usuário "cutucando" as engrenagens do relógio. Os membros privados são acessados somente pelos membros públicos, que fazem a interação entre as "engrenagens" do programa e o usuário. Assim, mesmo que tenhamos funções como membros privados, elas só podem ser executadas por outras funções que sejam membros públicos da função. A função pública "ver horário" de nossa classe Relógio provavelmente precisa acessar uma função privada que atualiza o horário do relógio, assim como a função "ajusta alarme" altera uma variável privada que contém o horário do alarme.

Formalizando, um membro público, seja ele uma variável ou uma função pode ser acessado a qualquer momento durante a execução de um programa (lido ou alterado no caso das variáveis, executado no caso das funções). Quando tentamos acessar diretamente um membro privado, geramos um erro pois seu acesso é proibido para o usuário/programador! Assim, a única forma de acessar um membro privado é através dos membros públicos. Esta separação entre membros públicos e privados é feita na declaração da classe.

O próximo conceito essencial da programação orientada à objetos é o próprio **objeto**. Um objeto é uma instância específica de uma classe. Podemos pensar na classe como um tipo de variável, enquanto que o objeto é a própria variável. Em C++, definimos classes para depois criar objetos que tenham as características definidas nestas classes. No exemplo acima, definimos a classe Relógio, e agora podemos criar quantos relógios forem necessários: cada relógio será um objeto diferente, com funcionamento idêntico e

características diferentes. Por exemplo, criamos os objetos “relógio de pulso” e “relógio-cuco”. Os dois são relógios, mas podemos ajustá-los em horários diferentes, com alarmes diferentes. Cada objeto possui todos os membros públicos e privados que definimos na declaração da classe. Assim, utilizamos os membros públicos para interajir com os objetos, enquanto que os membros privados são as “engrenagens” do objeto. Aqui temos um novo nome para os membros públicos: chamamos eles de “métodos” do objeto. Um método nada mais é do que uma função que interaja com o objeto, ou seja, são próprios os membros públicos que definimos na declaração da classe.

Para termos um exemplo mais concreto, podemos pensar em um programa qualquer do ambiente Windows. Nas ferramentas de programação Windows, temos várias classes que representam os vários itens que vemos na tela: janelas, menus, cursores, ícones. Quando um programador precisa criar uma janela, ele não precisa ter todo o trabalho para programar uma janela do zero: ele cria um novo objeto da classe Janela, e define suas características através das funções públicas (ou “métodos”) da classe. Ele não precisa saber como a janela é desenhada na tela: basta saber quais métodos definem as características que ele necessita para que seu objeto funcione da maneira desejada.

11.2 – Herança e Polimorfismo

O conceito de herança é muito importante para a reutilização de código em C++. Quando escrevemos uma classe, assim como quando escrevíamos funções, temos em mente o objetivo de não reescrever um código que já foi desenvolvido. Além das classes, temos em C++ o conceito de sub-classes: classes que herdam seus membros e métodos de outras classes. Por exemplo, suponha que tenhamos criado a classe Cachorro. Com base nesta classe, podemos criar subclasses representando as diversas raças de cachorro, como Pastor_Alemão, Doberman e Vira_Lata. Não precisamos reescrever todos os membros e métodos que já havíamos escrito ao desenvolver a classe base: basta indicar ao compilador que as novas subclasses serão herdeiras da classe-base. Feito isso, podemos modificar estas subclasses, modificando seus membros públicos e privados. Assim, podemos alterar a interface da subclasse, afetando a maneira do usuário interajir com ela, ou o próprio funcionamento da classe, alterando seus membros privados.

Note que poderíamos simplesmente copiar a classe-base e modificar seu código, alterando seu nome. Mas o conceito de herança permite alterar só os membros que necessitam ser alterados. Não precisamos nem mesmo ter acesso ao código fonte da classe-base, o que é muito útil quando trabalhamos com bibliotecas comerciais que geralmente não disponibilizam seu códigos.

Quando alteramos uma subclasse, podemos alterar qualquer um de seus métodos, ou seja, podemos alterar a maneira como esta subclasse interaje com seus usuários. Por exemplo, quando definimos as subclasses Pastor_Alemão e Vira_Lata, alteramos também seu método “Latido”. Uma subclasse fará uma determinada tarefa quando for chamado o método “Latido”, enquanto que a outra subclasse fará outra tarefa totalmente diferente, mesmo sendo as duas subclasses derivadas da mesma classe-base. Podemos até criar uma subclasse sem o método “Latido”: um cachorro mudo. Essa possibilidade de se fazer alterações nas subclasses de uma classe base é chamada de Polimorfismo. Basicamente, o Polimorfismo afirma que mesmo que a classe B seja herdeira da classe A, ela não precisa

herdar todos os métodos de A: a classe B pode ter características diferentes, ou seja, métodos diferentes da classe-base. Dessa forma, o mesmo "comando" pode gerar resultados diferentes dependendo da subclasse à qual o objeto pertence.

A idéia por trás dos conceitos de classes, objetos, herança e polimorfismo é possibilitar a elaboração de código mais simples para programas mais complexos, quebrando os diversos aspectos de um programa em blocos independentes entre si. Assim, não precisamos atacar todo o problema de uma vez, podemos dividí-lo e trabalhar com cada uma dessas divisões individualmente. Outro ponto importante por trás da programação orientada à objetos é a reutilização de código: porque reinventar a roda sempre que escrevemos um programa novo? Nesse aspecto, podemos pensar nas classes como grupos de funções: não precisamos saber como elas funcionam, apenas precisamos saber como utilizar seus métodos para obter os resultados desejados. Atualmente existe um número muito grande de bibliotecas de classes para C++, tanto comercial como em código aberto, com as aplicações mais diversas, desde bibliotecas matemáticas até bibliotecas gráficas para a elaboração de jogos 3D.

Referências Bibliográficas

[1] TORRES, Jair Gustavo M. – Curso de Linguagem C – Ilha Solteira: Faculdade de Engenharia de Ilha Solteira – FEIS – UNESP – Departamento de Engenharia Elétrica, 2006.

[2] PRATA, Stephen – C++ Primer Plus, 4º edição – Nova York: Sams Publishing. 2001.

[3] STROUSTRUP , Bjarne - The C++ Programming Language, 3º edição – New Jersey: Addison – Wesley. 1997.

[4] LIPPMAN, Stanley B.; LAJOIE, Josée Lajoie; MOO, Barbara E. – C++ Primer, 4º edição – New Jersey: Addison-Wesley. 2005.