

C / C++ e Orientação a Objetos em Ambiente Multiplataforma

Por: Sergio Barbosa Villas-Boas, Ph.D.

Professor Adjunto do DEL - Departamento de Eletrônica e Computação da UFRJ. (www.del.ufrj.br)

Resumo do Currículo:

Técnico Eletrônico pelo CEFET-RJ	1981	www.cefet-rj.br
Engenheiro Eletrônico pelo DEL, UFRJ	1987	www.del.ufrj.br
M.Sc. em Engenharia Elétrica pela COPPE - PEE - Controle	1991	www.coep.ufrj.br
Ph.D. em Engenharia de Controle por Chiba University - Japão	1998	www.chiba-u.ac.jp
MBA em Inteligência Empresarial e Gestão do Conhecimento (MBKM)	2000	www.crie.coppe.ufrj.br/mbkm/

Email do Autor:

villas@kmail.com.br

URL do Autor:

www.del.ufrj.br/~villas

URL do Livro:

www.del.ufrj.br/~villas/livro_c++.html

Versão do Livro: 5.1, de 17 de Agosto de 2001

Índice

Índice	2
Capítulo 1) Introdução.....	17
1.1 Prefácio.....	17
1.2 Histórico do livro.....	19
1.3 Escolha de uma tecnologia: uma decisão estratégica.....	20
1.3.1 Características do produto “software”	21
1.3.2 Incorporação de novidades	22
1.4 Mercadoria e produto diferenciado	24
1.4.1 Mercadoria escassa	24
1.5 Sociedade do conhecimento	25
1.5.1 Mudança do paradigma de negócios	26
1.5.2 “Mercadorização”	26
1.5.3 Remuneração × agregação de valor na cadeia produtiva	27
1.6 Porque C/C++ ?	27
1.6.1 “Tribos” de tecnologia	27
1.6.2 A escolha tecnológica é uma escolha estratégica	28
1.7 Breve história do C/C++.....	30
1.8 Qualidades da Linguagem C/C++	31
1.9 Classificação de interfaces de programas.....	32
1.10 Programando para console	32
1.11 Linguagens de programação de computador.....	33
1.11.1 Alto nível × baixo nível	33
Capítulo 2) Conheça o Seu Compilador.....	35
2.1 Visual C++ 6.0	35
2.1.1 Reconhecendo o Compilador	35
2.1.2 “Hello world” para DOS	41
2.1.2.1 Adicionando argumentos para a linha de comando.....	44
2.1.3 Usando o Help	45
2.1.4 Projetos (programas com múltiplos fontes)	46
2.1.5 Bibliotecas	46

2.1.5.1	Fazer uma biblioteca	46
2.1.5.2	Incluir uma biblioteca num projeto	48
2.1.5.3	Examinar uma biblioteca.....	48
2.1.6	Debug	48
2.1.7	Dicas extras	49
2.1.7.1	Acrescentando Lib no Project Settings	49
2.1.7.2	Class View.....	50
2.1.7.3	Usando bibliotecas de ligação dinâmica (DLL)	51
2.1.7.4	DLL para Windows	55
2.1.7.5	Otimização do linker para alinhamento de código	55
2.1.8	Detectando vazamento de memória	55
2.2	Borland C++ builder 5.0.....	56
2.2.1	Reconhecendo o Compilador	57
2.2.2	“Hello world” para DOS	57
2.2.2.1	Adicionando argumentos para a linha de comando.....	59
2.3	C++ for win32 gratuito.....	60
2.3.1	Ming (da GNU)	60

Livro patrocinado, que história é essa ?

Vivemos atualmente num mundo conectado e globalizado. Há tecnologia de sobra para copiar conhecimentos em praticamente qualquer forma que possa ser registrado. Portanto, tentar bloquear a cópia do texto pode ser um trabalho de “remar contra a maré”.

Optei por disponibilizar gratuitamente a versão em PDF desse texto. Ao fazê-lo, investigo um modelo de negócios que pode ser o mais adequado ao ambiente atual. O autor do texto pode doar seu material se quiser, e é o que estou fazendo. Mas ao fazê-lo, a atividade de dar autoria ao texto torna-se obviamente não remunerada. Então que estímulo teria o autor em fazer o texto ? Resposta: usa-se o texto como veículo para mensagens de patrocinadores. Isso é feito nesses quadros, inseridos nas páginas de índice.

Esse livro não está pronto ainda. Há trechos por melhorar, e outros a acrescentar. Procure por novas versões a cada início de semestre.

2.3.2	djgpp	60
2.4	g++ (do unix).....	60
2.4.1	“Hello world”	61
2.4.1.1	Adicionando argumentos para a linha de comando.....	61
2.4.2	Usando o Help	61
2.4.3	Projetos (programas com múltiplos fontes)	61
2.4.4	Bibliotecas	62
2.4.4.1	Incluir uma biblioteca num projeto	62
2.4.4.2	Fazer uma biblioteca	62
2.4.4.3	Examinar uma biblioteca.....	62
2.4.5	Fazendo uma biblioteca usando libtool	63
2.4.5.1	Instalando uma biblioteca dinâmica	64
2.4.6	Debug	65
2.4.7	Definindo um identificador para compilação condicional	65
2.4.8	O pacote RPM do linux	66
2.4.8.1	rpm binário e rpm com fonte	66
2.4.8.2	Alguns comandos do rpm.....	66

Patrocine essa publicação

Público Alvo: Profissionais e estudantes de tecnologia de informação.

Circulação: Esse texto é literatura recomendada para cerca de 100 alunos por ano (circulação mínima). Considerando as versões anteriores do livro, pelo menos 200 alunos já usaram esse texto. Os alunos são usuários influenciadores, que mostram o material em empresas que trabalham. Na página web do livro há uma lista de interessados em receber notícia sobre novas versões do livro. O número de pessoas que deseja receber notícias sobre o livro é mostrado na página web. Sendo que o formato do livro em PDF pode ser copiado livremente, não há registro de quantas cópias estão efetivamente em circulação, mas é seguramente muito mais que o número de pessoas mostrado na página web do livro.

Periodicidade: A cada semestre, o texto é atualizado e melhorado. A nova versão corrente do texto pode sempre ser copiada a partir da página web. A próxima versão está programada para início de 2002.

2.4.8.3	Construindo um rpm.....	67
2.4.8.3.1	Introdução.....	67
2.4.8.3.2	O header	68
2.4.8.3.3	Preparação (prep).....	69
Capítulo 3)	Princípios de C/C++	70
3.1	O primeiro programa	70
3.2	Formato livre	70
3.3	Chamada de função	70
3.4	Declaração e definição de funções	71
3.5	Comentários.....	71
3.6	Identificador	72
3.7	Constantes literais.....	72
3.8	Escopo	72
3.9	Tipos de dados padrão (<i>Standard Data Types</i>)	73
3.10	Palavras reservadas do C++ (keywords)	74
3.11	Letras usadas em pontuação	74
3.12	Letras usadas em operadores	74

Surgiram oportunidades profissionais ? lembre-se do autor !

<http://www.del.ufrj.br/~villas>

O primeiro patrocinador é o autor, cujos dados de contato estão divulgados no texto. Espero ser lembrado pelos leitores em **consultorias, projetos, cursos** ou **empreendimentos** que surjam, no ambiente profissional sempre em grande mutação.

Como membro da comunidade acadêmica, sou entusiasta defensor de uma parceria tipo “ganha-ganha” entre universidade e empresas privadas ou públicas. Acredito que implantar com sucesso esse tipo de parceria é fator essencial de desenvolvimento econômico e social. Qualquer país precisa muito disso. No caso do Brasil, a necessidade de que a inteligência das universidades seja traduzida em desenvolvimento econômico e bem estar social é evidente e urgente.

Veja slides desse livro na web em <http://del.ufrj.br/~villas/slides.html>

Capítulo 4) Estrutura do Compilador	75
4.1 Entendendo o Compilador	75
4.2 Protótipos (<i>prototypes</i>)	76
4.3 Projetos em C/C++	77
4.4 Header Files (*.h)	79
4.5 Biblioteca (<i>library</i>)	79
4.5.1 Utilizando Bibliotecas prontas	80
4.5.2 Fazendo bibliotecas	80
4.6 Regras do compilador	82
Capítulo 5) Fundamentos de C /C++	84
5.1 Chamada de função por referência e por valor	84
5.2 Tipos de dados definidos pelo programador	85
5.3 Maquiagem de tipos (<i>type casting</i>)	85
5.4 Operações matemáticas	86
5.5 Controle de fluxo do programa	86
5.6 Execução condicional	87
5.7 Laços (<i>loop</i>) de programação	87

Balico – o mercado de trabalho na web

<http://www.balico.com.br>



O Balico é um sistema na web para cadastrar profissionais a procura de emprego, e empresas com ofertas de emprego.

O sistema permite o cadastro imediato e gratuito tanto de empresas quanto de profissionais. Formulários específicos dão perfil para as empresas e para os profissionais, de forma que as buscas por oportunidades são eficientes e focadas.

Se você está procurando talentos, o Balico é o melhor local para veicular o seu anúncio.

5.7.1	Laço tipo “do-while”	87
5.7.2	while	88
5.7.3	for	88
5.7.4	Alterando o controle dos laços com break e continue	89
5.7.5	Exercício	89
5.8	switch-case	90
5.9	arrays	90
5.9.1	Arrays multidimensionais	92
5.10	Parâmetros da função main	92
5.11	Compilação condicional	93
5.12	Pré processador e tokens (símbolos) usados pelo pré-processador	93
5.13	#define	94
5.14	operador #	94
5.15	operador ##	94
5.16	Número variável de parâmetros.....	95
Capítulo 6) Técnicas para melhoria de rendimento em programação		96
6.1	Reutilização de código	96

Vogsys

<http://www.vogsys.com.br/>



A Vogsys Consultoria e Desenvolvimento de Sistemas, é uma empresa especializada no fornecimento de soluções integradas para o segmento de mídia, possui uma expressiva participação no mercado de publicação impressa e on-line.

6.2	Desenvolvimento e utilização de componentes.....	97
6.3	Programação estruturada	98
Capítulo 7) Programação orientada a objeto		99
7.1	Introdução.....	99
7.1.1	Abstração do software	99
7.1.2	Mudança de nomenclatura	100
7.1.3	Objetos	100
7.2	Introdução a análise orientada a objetos.....	100
7.3	Polimorfismo	101
7.3.1	Argumento implícito (<i>default argument</i>)	102
7.4	Uso de classes.....	102
7.5	Herança.....	103
7.6	Sobrecarga de operadores.....	106
7.7	Construtor e destrutor de um objeto	107
7.7.1	Default constructor (construtor implícito)	108
7.7.2	Ordem de chamada de construtor e destrutor para classes derivadas	108
7.7.3	Inicialização de atributos com construtor não implícito	109

VBMcgi – programação para Internet (CGI) com C++

<http://www.vbmcgi.org>



Para quem sabe C++, ou quer saber, VBMcgi é a melhor opção para programação de sistemas de software para Internet. Trata-se de uma biblioteca gratuita, distribuída com código fonte original. A sua melhor característica é o fato de **isolar o trabalho do webmaster do webdesigner**. Isto é, o webdesigner pode usar programas de HTML comerciais, como Dream Weaver ou Front Page. Apenas o webmaster é programador C++ e lida com banco de dados. Além disso, VBMcgi é multiplataforma, testado em Windows e Unix.

Quem desenvolve usando ASP, PHP, JSP (java), Perl ou Cold Fusion, pode usar com C++ e VBMcgi. Não há problema em usar ao mesmo tempo C++/VBMcgi e outra tecnologia.

7.7.4	Copy constructor (construtor de cópia)	110
7.8	lvalue	111
7.9	Encapsulamento de atributos e métodos	112
7.10	União adiantada \times união atrasada (<i>early bind</i> \times <i>late bind</i>)	114
7.10.1	Destrutores virtuais	116
7.11	Classe base virtual	117
7.12	Alocação de Memória	119
7.12.1	Vazamento de memória	120
7.12.2	Objetos com memória alocada	121
7.12.3	Array de objetos com construtor não padrão	123
Capítulo 8) Biblioteca padrão de C++		125
8.1	Introdução	125
8.2	Entrada e saída de dados pelo console	125
8.3	Sobrecarga do operador insersor (<<) e extrator (>>)	126
8.4	Formatando Entrada / Saída com streams	127
8.4.1	Usando flags de formatação	128
8.4.2	Examinando os flags de um objeto ios	129

VBLib – biblioteca gratuita multiplataforma para uso geral

<http://www.vbmcgi.org/vbllib>

Para quem usa C++, a VBLib oferece algumas classes bastante úteis. A VBMcgi usa VBLib.

- Classe genérica (template) para lista encadeada (class VBList)
- Classe de string fácil e poderosa (class VBString)
- Classe para detectar vazamento de memória (VisualC++ only) VBMemCheck
- Classe para ajudar a medição de tempo (class VBClock)
- Classe para mostrar o conteúdo de um contenedor STL (class VBShowSTLContainer)
- Uma classe de calendário, que retorna o dia da semana para uma data dada (class VBCalendar)

8.4.3	Definindo o número de algarismos significativos	130
8.4.4	Preenchendo os espaços vazios com o método fill	131
8.4.5	Manipuladores padrão	131
8.4.6	Manipuladores do usuário	132
8.4.7	Saída com stream em buffer	133
8.5	Acesso a disco (Arquivos de texto para leitura/escrita)	134
8.5.1	Escrevendo um arquivo de texto usando a biblioteca padrão de C++	134
8.5.2	Escrevendo um arquivo de texto usando a biblioteca padrão de C	134
8.5.3	Lendo um arquivo de texto usando a biblioteca padrão de C++	135
8.5.4	Dica para leitura de arquivo de texto.	135
8.5.5	Lendo um arquivo de texto usando a biblioteca padrão de C	136
8.6	Acesso a disco (Arquivos binários para leitura/escrita)	136
8.6.1	Escrevendo um arquivo binário usando a biblioteca padrão de C++	136
8.6.2	Escrevendo um arquivo binário usando a biblioteca padrão de C	137
8.6.3	Lendo um arquivo binário usando a biblioteca padrão de C++	138
8.6.4	Lendo um arquivo binário usando a biblioteca padrão de C	138
Capítulo 9)	Programação genérica (template)	140

VBMath – biblioteca gratuita multiplataforma para matemática

<http://www.vbmcgi.org/vbmath>

Para quem usa C++, a VBMath oferece classes para matemática matricial. Desenvolvida em programação genérica (template), o usuário pode facilmente criar classes matriciais codificadas com float, double ou long double, por exemplo. As funções básicas de operação matemática matricial estão implementadas (+, -, *, /, inverse, etc).

Essa biblioteca é recomendada para implementação de contas matriciais de pequena dimensão, ou quando o requerimento de desempenho de execução não é severo. Nesse caso, aproveita-se a vantagem de fazer o programa de forma rápida e facilmente compreensível.

Também está implementada uma classe (genérica) VBParser para calcular em tempo de execução o resultado de uma fórmula matemática.

9.1	Uma lista encadeada simples com template	141
9.2	VBMath - uma biblioteca de matemática matricial usando classe template	146
Capítulo 10) Tratamento de exceção (<i>exception handling</i>).....		152
Capítulo 11) RTTI – Identificação em tempo de execução.....		154
11.1	Introdução.....	154
11.2	typeid	154
Capítulo 12) namespace		156
12.1	Introdução.....	156
12.2	Namespace aberto.....	157
12.3	Biblioteca padrão de C++ usando namespace	158
12.4	Adaptando uma biblioteca existente para uso de namespace std	159
Capítulo 13) STL - Standard Template Library		161
13.1	Introdução.....	161
13.1.1	Classes contenedoras	161
13.1.2	Classes iteradoras	162
13.1.3	Algoritmos	162
13.2	Preâmbulo.....	162

Confraria de Mercado – restaurante & palestras: encontre-se aqui

<http://www.confrariademercado.com.br>



A Confraria de Mercado é o seu ponto de encontro no Rio de Janeiro, no bairro de Botafogo (em frente a um estacionamento). É um restaurante que com comida muito boa, no estilo “simple chic”. Mas o grande diferencial é o fato de ter infra-estrutura para apresentação de palestras, com projetor (data-show) sofisticado, ligado a um computador moderno com Internet, a um vídeo cassete e ao sinal de TV a cabo.

A agenda de palestras e eventos (sempre gratuitos) pode ser conferida no site, assim como mapa para chegar lá, informações de contato, etc. Para quem se cadastrar, a confraria envia propaganda dos eventos por email.

Quem precisar de um local para almoço de negócios, pode reservar o local.

Marque sua comemoração de formatura na Confraria de Mercado !

13.2.1	Classes e funções auxiliares	164
13.2.1.1	Par (pair)	164
13.2.2	Operadores de comparação	164
13.2.3	Classes de comparação	165
13.2.4	Classes aritméticas e lógicas	167
13.2.5	Complexidade de um algoritmo	167
13.2.5.1	Algumas regras	168
13.3	Iterador (iterator)	168
13.3.1	Iteradores padrão para inserção em contenedores	169
13.4	Contenedor (container)	170
13.4.1	Vector	170
13.4.2	List	171
13.4.3	Pilha (Stack)	172
13.4.4	Fila (Queue)	173
13.4.5	Fila com prioridade (priority queue)	174
13.4.6	Contenedores associativos ordenados	175
13.4.6.1	Set	176

Kmail – email com inteligência. Venha para K você também.
<http://www.kmail.com.br>



Kmail é a marca que representa tecnologia de email inteligente. Filtros de email, logs especializados de email e segurança de email são alguns dos serviços que se sabe prestar.

Na página da Internet, oferece-se serviço inteligente e gratuito de redirecionamento de email. O cadastro é feito imediatamente, e o redirecionamento é feito para até 5 locais, com filtros independentes.



13.4.6.2	Multiset.....	177
13.4.6.3	Map.....	178
13.4.6.4	Multimap	179
13.5	Algoritmos.....	179
13.5.1	remove_if	179
Capítulo 14) Componentes de Programação		181
14.1	Para Windows & DOS	181
14.1.1	Programa para listar o diretório corrente (Visual C++)	181
14.1.2	Porta Serial	182
14.1.3	Porta Paralela	182
14.2	Componentes para unix (inclui Linux).....	183
14.2.1	Entrada cega (útil para entrada de password)	183
14.3	Elementos de programação em tempo real.....	184
14.3.1	Conceitos de programação em tempo real	184
14.3.2	Programa em tempo real com “status loop”	184
14.3.3	Programa em tempo real com interrupção	186
14.3.4	Programas tipo “watch dog” (cão de guarda)	188

DEL – Departamento de Engenharia Eletrônica e de Computação da UFRJ <http://www.del.ufrj.br>

Em 1934 a UFRJ abriu o Departamento de Engenharia Elétrica (juntamente com a inauguração de outros departamentos). Em 1962 o Departamento de Engenharia Elétrica passou para a sua atual localização física, no Centro de Tecnologia, na Cidade Universitária (Ilha do Fundão, RJ).

Em 1968, surge o DEL. O então Departamento de Engenharia Eletrônica da UFRJ passa a atuar de forma independente do Departamento de Engenharia Elétrica. Em 1994 o DEL implantou uma mudança de currículo e incorporou computação ao seu escopo. A sigla DEL foi mantida, mas o nome passou a ser Departamento de Engenharia Eletrônica e de Computação da UFRJ. O DEL atua em nível de graduação, mas tem forte integração com programas de pós graduação da COPPE.

No ano de 2001, o DEL conta com 33 professores, sendo 23 (70%) com doutorado. Destes 29 (88%) tem dedicação exclusiva. O curso dura 5 anos e está com cerca de 500 alunos inscritos.

Capítulo 15) Boa programação × má programação.....	189
15.1 Introdução.....	189
15.2 Itens de boa programação.....	190
15.2.1 Identação correta	190
15.2.2 Não tratar strings diretamente, mas por classe de string	190
15.2.2.1 Motivos pelos quais é má programação tratar strings diretamente	190
15.2.2.2 Solução recomendada para tratamento de strings: uso de classe de string.....	191
15.2.3 Acrescentar comentários elucidativos	193
15.2.4 Evitar uso de constantes relacionadas	193
15.2.5 Modularidade do programa	193
15.2.6 Uso de nomes elucidativos para identificadores	194
15.2.7 Programar em inglês	195
Capítulo 16) Erros de programação, dicas e truques.....	197
16.1 Cuidado com o operador , (vírgula)	197
16.2 Acessando atributos privados de outro objeto.....	197
16.3 Entendendo o NaN	198
16.4 Uso de const_cast	198

GEI – Gestão Estratégica da Informação

<http://www.engenharia.ufrj.br/gei>

O GEI é um curso de pós graduação *latu sensu* ministrado pelo DEL que surgiu para cumprir o papel de formar profissionais aptos a acompanhar as tendências tecnológicas do mercado. O GEI utiliza a bem sucedida experiência da Escola de Engenharia da Universidade Federal do Rio de Janeiro em cursos de especialização em diversas áreas do conhecimento. O curso tem uma visão estratégica, e estará situado em um ambiente de economia globalizada, onde rapidez e precisão das decisões são considerados requisitos básicos.

O curso é dirigido a profissionais com nível superior e alguma experiência na área de tecnologia da informação, como analistas de sistemas, gerentes e usuários em geral.

No segundo semestre de 2001, inicia-se a 4ª turma do GEI.

16.5	Passagem por valor de objetos com alocação de memória.....	198
16.6	Sobrecarga de insersor e extrator quando se usa namespace	199
16.7	Inicializando um array estático, membro de uma classe	200
16.8	SingleTon	200
16.9	Slicing in C++	201
16.10	Dicas de uso de parâmetro implícito	202
Capítulo 17) Incompatibilidades entre compiladores C++.....		204
17.1	Visual C++ 6.0 SP5	204
17.1.1	for não isola escopo	204
17.1.2	Comportamento do ifstream quando o arquivo não existe	204
17.1.3	ios::nocreate não existe quando fstream é usado com namespace std	205
17.1.4	Compilador proíbe inicialização de variáveis membro estáticas diretamente	205
17.1.5	“Namespace lookup” não funciona em funções com argumento	205
17.1.6	Encadeamento de “using” não funciona	205
17.1.7	Erro em instanciamento explícito de funções com template	205
17.2	Borland C++ Builder (versão 5.0, build 12.34).....	206
17.2.1	Inabilidade de distinguir namespace global de local	206

TELECOM - Curso de Especialização em Sistemas de Telecomunicações

<http://www.lps.ufrj.br/telecom.html>



O TELECOM é um curso de pós graduação *latu sensu* ministrado pelo DEL que oferece a profissionais de nível superior interessados nas modernas técnicas de telecomunicações conhecimentos teóricos e práticos em sistemas de telecomunicações, processamento digital de voz e imagem, sistemas de transmissão e recepção, telefonia móvel e redes de computadores.

17.2.2	Erro na dedução de funções com template a partir de argumentos const	206
17.2.3	Erro na conversão de “const char *” para “std::string”	206
17.3	Gnu C++ (versão 2.91.66).....	206
17.3.1	Valor do elemento em vector<double> de STL	206
Capítulo 18)	Glossário.....	208
Capítulo 19)	Bibliografia.....	209
Capítulo 20)	Índice remissivo	210

Marlin

<http://www.marlin.com.br/>



Empresa que atua na área de acesso a Internet, hospedagem, comércio eletrônico, smart cards, etc.

Desenvolve soluções baseadas em XML.

A empresa é parceira certificada Microsoft.

Capítulo 1) Introdução

1.1 Prefácio

A linguagem C/C++ é um assunto relativamente tradicional no dinâmico mercado de tecnologia de informação, sempre cheio de novidades. Existe um acervo bibliográfico substancial sobre esse assunto. Portanto surge a pergunta: ainda há espaço para um novo livro ensinando C/C++ ? Acredito firmemente que sim, pelos motivos que exponho a seguir.

1. **C/C++ deve ser ensinado como multiplataforma.** Ainda há relativamente poucos livros que enfatizam o aspecto multiplataforma dessa linguagem. Muitos livros se propõem a ensinar C/C++ a partir de um compilador específico. Mas eu considero isso um erro. A ênfase no ensino de C/C++ deve ser em cima dos conceitos, e não no compilador ou no sistema operacional. No capítulo “Conheça o Seu Compilador” (página 35), há um conjunto de explicações sobre operacionalidade dos principais compiladores para Windows e Unix. Além disso, há um capítulo sobre “Diferenças entre compiladores” (página 202), onde são listadas pequenas (porém importantes) diferenças de comportamento entre os compiladores.
2. **Talvez seja um erro abandonar C++ pela linguagem Java.** A linguagem C++ implementa os conceitos de orientação a objetos, que são de grande valor em projetos de tecnologia de informação. Outras linguagens também o fazem. A linguagem Java também implementa muito bem os conceitos de orientação a objetos, e claramente concorre com C++ como alternativa tecnológica para implementação de projetos de sistemas. Há um debate quente sobre C++ versus Java. Várias universidades acreditam que não é mais necessário ensinar C/C++, porque supostamente Java a substituiria com vantagem de ser mais segura, elegante, robusta, portátil, etc. Eu sustento que embora a linguagem Java tenha várias qualidades, é um erro parar de ensinar C/C++. Portanto, o mercado precisa seguir melhorando sempre o material tutorial para C/C++, inclusive porque há novidades acrescentadas à linguagem há relativamente pouco tempo, (e.g. namespace, RTTI, programação genérica com template, STL), e que precisam ser ensinadas com material didático sempre melhorado.
3. **Deve-se ensinar regras práticas para reutilização de código, e o trabalho em equipe.** Ainda há relativamente poucos livros de C/C++ que enfatizam a reutilização de código e o trabalho em equipe. A linguagem C/C++ possui grande tradição. Na prática isso significa que existem várias bibliotecas e componentes disponíveis para apoiar o desenvolvimento de projetos, sendo que boa parte desses componentes é gratuita. Mas para ser capaz de usar esse tipo de componente é preciso conhecimento. O foco é apoiar a formação de profissionais que saibam participar na prática de um esforço coletivo de desenvolvimento de um sistema. O programador deve saber usar um componente de software feito por outra pessoa, e explorar essa habilidade no sentido de maximizar seu desempenho profissional. Analogamente, é preciso saber fazer um componente de software, que possa ser usado em diversos projetos diferentes, inclusive por outras pessoas. No capítulo “Boa programação × má programação” (página 189), enfatiza-se os aspectos que um programa deve ter para atingir as necessidades práticas de empresas e organizações. Nesse capítulo, o objetivo de se obter reutilização de código, trabalho em equipe, bem como métodos práticos para minimizar a possibilidade de bugs é traduzido em regras escritas.

4. **C/C++ como escolha estratégica.** O importante segmento de computação comercial tem geralmente resistência a adotar C/C++ por considerar essa linguagem demasiado sofisticada, e portanto propensa a bugs. Além disso, justamente por essa sofisticação, leva mais tempo para se formar um bom profissional de C/C++ em comparação com linguagens que foram projetadas para serem simples. Para as empresas, optar por trabalhar com C/C++ pode significar depender de profissionais mais escassos e mais caros. Se essa escolha faz aumentar o custo e o risco, sem oferecer uma vantagem palpável em relação a uma linguagem mais fácil, é claro que seria uma má escolha. Contudo, desde que consolida-se o ambiente empresarial de “Sociedade do conhecimento” (veja a seção 1.5, na página 25), outras influências podem fazer um estrategista de empresa querer optar por C/C++ por escolha estratégica. Ainda existem relativamente poucas referências de programação em C/C++ para quem precisa atuar no rápido mercado de computação comercial. O texto tutorial deve ter linguagem deve ser o mais objetiva possível, conter todos os exemplos que elucidam os conceitos usados na prática, expor regras práticas para se obter resultados eficientes em empresas, etc. Esse livro tem essa proposta.
5. **A linguagem C/C++ é forte candidata para ser considerada como habilidade essencial e estratégica** na formação de um profissional de tecnologia de informação, pelos motivos abaixo.
- O termo “novidade” não é sinônimo de “qualidade”,** especialmente no mercado de tecnologia de informação. Várias tecnologias e linguagens mais novas que C++ introduzem diversas funcionalidades, mas não substituem-na em toda a sua potencialidade.
 - O profissional que domina C/C++ é altamente adaptável,** e deve continuar a sê-lo. Na história do mercado de tecnologia de informação, sempre houve espaço para uso de C/C++. Ambientes tão distintos quanto unix (todas as variantes), Windows, programação para Internet (CGI), banco de dados, computadores de grande porte, dispositivos digitais de mão, chips processadores sem sistema operacional, etc. podem valer-se de conhecimentos de C/C++ para que se gere programas executáveis. É a única linguagem com essa abrangência. Portanto, o profissional com conhecimentos de C/C++ sempre conseguiu se adaptar às mudanças que o mercado exigiu. Portanto, desenvolver sistemas em C/C++ é considerado como **escolha estratégica pessoal** por muitos profissionais, isto é, apesar de haver alternativa em outra linguagem, pode ser melhor fazer o sistema nessa linguagem para que se **aumente sempre a habilidade essencial** que se pretende definir numa carreira. No caso a habilidade essencial é a capacidade de resolver o problema em questão usando componentes em C/C++.
 - O mercado de software básico possui tradição sólida.** Há inúmeros projetos já feitos em C/C++. Essa tradição por si só (leia-se: a necessidade de se manter o que já existe) já justifica a necessidade de se seguir formando profissionais capacitados nessa linguagem. Para quem trabalha com software básico, a linguagem C/C++ é absoluta. E há bastante mercado a se explorar no segmento de software básico, que certamente é bastante lucrativo.
 - A linguagem C/C++ é absoluta no segmento de software básico.** A experiência mostra que as tecnologias criam “tribos” de usuários em torno de si e essas tribos simplesmente não param de usar o que já aprenderam. Linguagens tradicionais como FORTRAN e COBOL se recusam a morrer, e continuam bastante vivas nos segmentos em que se estabeleceram há tempos (computação científica e comercial

respectivamente). A linguagem C/C++ é absoluta no segmento de software básico, com inúmeros projetos no currículo, entre eles o kernel do sistema operacional unix. Somente por isso já haveria motivo suficiente para que se continuasse a ensinar C/C++. Ressalte-se que java não é competidor efetivo com C/C++ nesse segmento.

- e. **Há inúmeros grupos de ajuda** no uso, farta bibliografia (inclusive gratuita), famosas aplicações com fonte aberto, ampla disponibilidade de compiladores nas mais diversas plataformas e várias outras indicações claras de que trata-se de um padrão bastante maduro.

1.2 Histórico do livro

Esse é um livro que vem sendo escrito aos poucos. A seção abaixo mostra a evolução das diversas versões do texto.

1. Em 1992 foi feita uma versão chamada “Conceitos Básicos de C / C++”, para um curso oferecido à Rede Ferroviária Federal. Nessa época, estava na moda usar o compilador Borland C++ 3.1. (No período de 1993 a 1998, o autor esteve no Japão, Universidade de Chiba, fazendo curso de Doutorado).
2. Em 1999, foi feita uma versão profundamente modificada, chamada “Conceitos de software para sistemas de armas”, para um curso oferecido à Marinha do Brasil. Esse curso foi efetivamente “C++ e orientação a objetos, em console”. Para esse curso, usou-se o compilador Borland C++ 5.0.
3. Em 31 de agosto de 1999 foi feita a uma versão “esqueleto” desse livro. Na nova abordagem está sendo feito um esforço de depurar todo o material gerado, e fazer um livro chamado “C/C++ Multiplataforma”. Esse livro abordará os aspectos mais importantes da programação em C/C++, com ênfase em deixar o leitor livre de uma plataforma em particular. Basicamente, os trechos de programa de exemplo deverão funcionar em unix ou em Windows sem problemas. Esse livro vai ser produzido aos poucos. Chamou-se essa versão de 1.0, ainda com inúmeras inconsistências. Mesmo assim está sendo tornada pública na Internet a pedido de alunos.
4. Em 15 de Fevereiro de 2000, a pedido de um amigo, eu publiquei a versão 1.1 do livro. Há inúmeras melhorias em relação à versão 1.0, mas essa versão ainda continha inconsistências.
5. Em 15 de Março de 2000, após várias modificações e acréscimos em relação a versão anterior, foi disponibilizada a versão 2.0. Essa versão ainda possuía diversas seções incompletas (marcadas com //todo), e algumas seções inconsistentes. Essa versão trouxe uma mudança de título, que passou a se chamar “C/C++ e Orientação a Objetos em Ambiente Multiplataforma”.
6. Em Julho de 2000, fiz diversas melhorias baseado na experiência de um curso oferecido com esse livro. Versão 3.0.
 - a. Refeito e melhorado o capítulo “Conheça Seu Compilador”.
7. Em Fevereiro de 2001, após um curso para o PCT-DEL, surgiram diversas sugestões de melhorias, que foram implementadas nessa versão 4.0.
 - a. Modificou-se bastante o capítulo 1: “Introdução”, inclusive com o acréscimo de seções sobre estratégia na escolha de tecnologia. Para escrever essa parte, aproveitei material do interessante curso de MBA em “Inteligência Competitiva e Gestão do Conhecimento”, recém concluído.

“C / C++ e Orientação a Objetos em Ambiente Multiplataforma”, versão 5.1

Esse texto está disponível para download em www.del.ufrj.br/~villas/livro_c++.html

- b. Acrescentou-se capítulo de boa programação vs. má programação
 - c. Acrescentou-se capítulo sobre STL
 - d. Acrescentou-se tutorial para DLL em Visual C.
 - e. No capítulo “Conheça o Seu Compilador”, acrescentou-se referência para o compilador Borland Builder 5.0.
 - f. Acrescentou-se índice remissivo. Nas próximas versões, esse índice deve melhorar.
 - g. Acrescentou-se uma seção inicial de introdução dando ao leitor uma justificativa para existência desse livro, face ao ambiente do mercado de tecnologia de informação.
 - h. Revisão geral.
8. Em Agosto de 2001, versão 5.0
- a. Incluiu-se um índice.
 - b. Retirados trechos de código em português do livro. Dessa forma, ilustra-se que é boa programação escrever o código em si em inglês.
 - c. Incluiu-se informação sobre uso da ferramenta libtool, para criação de bibliotecas de ligação dinâmica em unix (embora ainda seja preciso melhorar essa informação).
 - d. Incluiu-se informação sobre desenvolvimento de pacotes RPM para linux.
 - e. Melhorou-se o capítulo sobre tratamento de exceção.
 - f. Melhorou-se o capítulo sobre namespace
 - g. Acrescentou-se informação sobre construtor de cópia, e criação de objetos com memória alocada.
 - h. Acrescentou-se um capítulo sobre dicas de programação.
 - i. Acrescentou-se mensagem do patrocinador

1.3 Escolha de uma tecnologia: uma decisão estratégica

O mundo que vivemos é complexo e possui quantidade extremamente grande de informações. O ambiente profissional no ramo de tecnologia de informação muda rapidamente. Muitas tecnologias nasceram e ficaram obsoletas rapidamente. Os profissionais de tecnologia de informação convive com um problema básico: que estratégia devem ter para manter-se atualizado nos conhecimentos valorizados no mercado ? Heis alguns itens a considerar na hora de definir essa estratégia:

- Conhecer o mais possível o teor das opções tecnológicas disponíveis no mercado.
- Estimar em que setores desse mercado se pretende atuar.
- Conhecer o comportamento humano, e juntando-se o conhecimento técnico que se tem, tentar antecipar tendências do mercado.
- Observar historicamente quais tipos de conhecimentos tornaram-se obsoletos, e quais foram relativamente bem aproveitados, apesar da mudança do ambiente.

1.3.1 Características do produto “software”

O software é um produto com características especiais, diferentes de quase todos os outros produtos. Nessa seção, comenta-se sobre essas diferenças.

Um produto tangível (televisão, computador, etc.) precisa que cada unidade seja produzida para que seja vendida. De forma semelhante, serviço tangível (limpeza, preparação de comida, segurança, etc.) precisa ser feito a cada vez que é vendido. Já um software é um produto que na prática custa apenas para ser desenvolvido. O custo de produção ou cópia é tão pequeno que pode ser considerado nulo. Uma vez desenvolvido, a firma poderia em princípio copiar e vender o software indefinidamente, com excelente lucro. Mas nem tudo são flores para uma firma de software. A concorrência é muito grande, e não se sabe a princípio se um software vai ter boa aceitação no mercado. Portanto, o negócio de desenvolver software é um tanto arriscado. Além disso, o ambiente tecnológico muda rapidamente. Para desenvolver um software, é preciso investir em salários de profissionais caros, e gastar tempo. Quando o produto fica pronto, é preciso convencer o mercado a aceitá-lo, sendo que há inúmeras informações sobre software, que podem confundir o cliente final.

Uma outra diferença fundamental entre o software e os demais produtos é o fato de que há vantagens para o consumidor final em haver padronização do software usado. A existência de pluralidade de padrões geralmente causa confusão, pois não é viável na prática conhecer todas as tecnologias em profundidade. Concretamente: o mundo seria muito confuso se cada indivíduo ou empresa usassem um computador diferente, com um sistema operacional diferente, com editor de texto diferente, etc. A padronização do software faz facilitar a árdua tarefa de se aprender uma nova e complexa forma de se trabalhar, e mesmo de se pensar. A existência de pessoas próximas com as quais pode-se tirar dúvidas a respeito do uso do software facilita muito a efetiva assimilação do seu uso.

Se um consumidor escolher uma televisão da marca x, a probabilidade de que a próxima televisão seja da mesma marca é relativamente pequena. Há várias marcas disponíveis, e muitas delas são muito boas. O produto “televisão” é uma mercadoria. Mas se um consumidor escolhe um software para resolver um problema, se o software é satisfatório, a tendência é o consumidor recomendar sempre o mesmo software para resolver o tal problema. É fácil entender o motivo desse comportamento. A complexidade do uso do software pode ser grande, e portanto há um esforço em se aprender a usá-lo bem. Esse esforço pode ser bastante grande. O consumidor tende a recomendar o software que já usou para não jogar fora o investimento que já fez em aprender a usar o software.

Outra forma de entender a diferença fundamental entre software e a maioria dos outros produtos é a seguinte. Seria possível um futuro em que produtos como televisão, gasolina, energia, roupas, etc. fossem oferecidos gratuitamente? É muito difícil imaginar que isso pudesse acontecer. No entanto, é perfeitamente possível que exista software gratuito. Se o criador de um software não ganha dinheiro diretamente distribuindo um software gratuito, que interesse haveria em fazê-lo? Muito simples: sendo gratuito, o consumidor torna-se mais interessado em experimentar o software. Assim, facilita-se a criação de uma base de usuários do software, e portanto o produto fica mais confiável e também mais valioso. Com a base de usuários consolidada, é muito possível pensar em formas indiretas de se ganhar dinheiro. Um exemplo disso é o linux.

A *inércia* é um termo importado do estudo de física. Sabe-se que é difícil mudar a velocidade de um corpo que possui grande massa. Para tal corpo mudar de velocidade, é preciso que se gaste muita energia. Em outras palavras: o tal corpo possui grande inércia. O comportamento humano é inercial em relação ao relacionamento com a tecnologia. Para efetivamente usar um produto tecnológico – tal como um software – é preciso que se invista energia (tempo, paciência, etc.) no estudo desse produto. Uma vez vencida a inércia e adquirido conhecimento sobre o uso de uma tecnologia, torna-se um tanto

difícil convencer um consumidor a adotar uma solução alternativa, especialmente se não houver uma vantagem palpável. Uma das formas de se interpretar o “frenesi de Internet” que ocorreu no final do milênio é justamente o reconhecimento de que seria preciso explorar rapidamente o mercado ainda virgem de pessoas que precisam de serviços de Internet. Após passada a fase na qual muitos se acostuariam com os serviços, seria cada vez mais difícil para novos entrantes no mercado, pois haveria inércia para se mudar de serviço.

Atualmente vivemos uma batalha entre grandes desenvolvedores de software pago e alternativas gratuitas (por exemplo linux). As grandes firmas do setor de software procuram fazer ver ao cliente que embora seus produtos sejam pagos, o TCO (*Total Cost of Ownership* ou custo total de propriedade) é mais baixo que a alternativa. Supostamente, segundo o ponto de vista das grandes empresas, seria mais barato comprar produtos dela e valer-se da base instalada de usuários, suporte, etc. A alternativa de instalar um produto gratuito para a qual há muito menos disponibilidade de assistência seria mais cara, pois seria necessário pagar caro por consultoria. Além disso, os bugs que supostamente seriam mais numerosos causariam despesas para o cliente.

1.3.2 Incorporação de novidades

É muito comum no mercado de software (na realidade para vários produtos de tecnologia de informação) uma situação tipo “competição Darwiniana”. Isto é, para um problema dado surgem diversas alternativas para solucioná-lo. No início, todas as alternativas são incompletas e insatisfatórias. Com o tempo, vão surgindo evoluções na qualidade do software, e cada vez menos alternativas se sustentam. Geralmente apenas uma alternativa é efetivamente usada, ou no máximo muito poucas. A competição Darwiniana descreve a competição para a definição do hardware básico (arquitetura do computador, CPU, etc.), para sistema operacional para esse hardware, para editor de texto e demais aplicativos de escritório, para comunicação pessoal com cliente de email ou outras, para navegador de Internet, etc.

Uma empresa que tem um produto de software deve ter uma estratégia para fazer o seu produto ser aceito no mercado. Isso deve ser feito rapidamente, pois do contrário a maioria do mercado tenderá a convergir para uma alternativa concorrente, e depois haverá inércia para deixar de usar o que já se sabe.

Há estudos que mostram como uma novidade costuma ser incorporada por um grupo social. Pode-se classificar (de forma simplificada) os usuários em 4 grupos, com suas porcentagens:

1. A - Desbravadores (10%)
2. B - Primeira maioria (40%)
3. C - Maioria atrasada (40%)
4. D - Retardatários (10%)

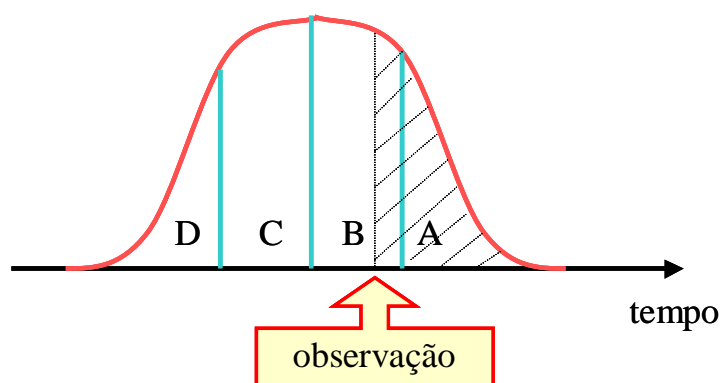


Figura 1: Diagrama mostrando a porcentagem de usuários que incorporam uma novidade em relação ao tempo.

Em software, os desbravadores são os que estão sempre testando software beta, e correndo atrás de todo tipo de novidades. Os da primeira maioria são aqueles que lêem os comentários feitos pelos desbravadores e identificam as tendências consistentes e minimamente maduras, e incorporam-nas. Os da maioria atrasada são os que “correm atrás” dos outros, e apressam-se a incorporar as novidades quando já há muita gente usando-as. Os retardatários são os últimos a adotarem a novidade, e apenas o fazem quando é absolutamente evidente que precisam fazê-lo, sendo que quase todos já o fizeram. As porcentagens de pessoas em cada uma das categorias é mostrado acima.

É interessante cruzar a informação de classificação de usuários com o perfil de sucessos. A experiência mostra que a maior parte dos indivíduos e firmas bem sucedidos são os que se comportam como “primeira maioria”, e não como “desbravadores”. Enquanto os “desbravadores” freqüentemente gastam tempo precioso para depurar problemas de novidades, os da primeira maioria optam pela novidade quando já há cultura suficiente para utilizar a novidade de forma efetiva.

A dinâmica de incorporação de novidades, portanto, começa pelos desbravadores, seguindo-se da primeira maioria, maioria atrasada e retardatários. Uma firma ou indivíduo que queira tornar uma alternativa tecnológica efetivamente usada no mercado deve ser capaz de influenciar os desbravadores, para que então a dinâmica de incorporação de novidades siga seu caminho natural. Em outras palavras: pode-se dizer que há dois tipos de usuários:

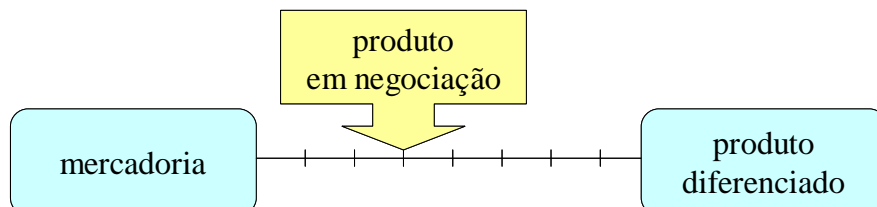
- Usuário influenciador
- Usuário comum.

O usuário influenciador é aquele que tem influência sobre os desbravadores e principalmente sobre a primeira maioria. Trata-se de um usuário que observa atentamente as reportagens dos desbravadores e sintetiza essa informação de forma interpretada para produzir conclusões que são por sua vez observadas pela primeira maioria. As revistas especializadas trazem sempre reportagens sobre novidades tecnológicas. Quem escreve essas reportagens está entre os “usuários influenciadores”.

Há uma categoria de profissionais que precisa tomar decisões sobre uma tecnologia a adotar para solucionar determinado problema. São por exemplo os gerentes de informática de empresas. Ter capacidade de convencer esse tipo de pessoa é absolutamente importante para o sucesso comercial de uma firma de software, pois são eles quem decidem como alocar recursos financeiros da firma para adquirir tecnologia no mercado.

1.4 Mercadoria e produto diferenciado

Quando ocorre uma transação econômica, há dois atores – o comprador e o vendedor. O produto que está sendo negociado está em algum ponto de uma escala, cujos extremos estão marcados com: “mercadoria”¹ e “produto diferenciado”.



Quando o produto é uma mercadoria, quem está em boa situação para o negócio é o comprador.

Quando trata-se de um produto diferenciado, a situação se inverte.

A capacidade profissional de um indivíduo é um produto no mercado. O preço é a remuneração. Para o profissional pretender obter boa remuneração, é necessário que esse profissional seja um “produto diferenciado no mercado”.

1.4.1 Mercadoria escassa

Quando uma mercadoria é necessária e escassa, o seu valor tende a subir. Quem pode suprir a mercadoria escassa pode praticar um preço que lhe é vantajoso (se for bom negociador). Mas vivemos numa sociedade com abundância de informação. Se uma mercadoria de boa demanda no mercado torna-se escassa e seu preço sobe, muitos saberão disso. Portanto, será provavelmente bastante lucrativo investir no sentido de atender a demanda. Costuma haver bastante investidores dispostos a quase todo tipo de investimento. Portanto, com o tempo a tendência é que uma mercadoria torne-se plenamente atendida, e portanto o preço de comercialização tende a cair até o mínimo possível para o mercado em questão.

Considere os conceitos acima aplicados a profissionais de tecnologia de Internet. Desde que a Internet tornou-se muito importante para empresas, em torno do ano de 1995, surgiu demanda para desenvolvimento de páginas e sistemas de Internet. Como essa profissão não existia até então, os poucos que sabiam fazer o serviço tornaram-se mercadoria escassa, e puderam cobrar alto. Há não muito tempo atrás, o simples desenvolvimento de uma página html simples era muito bem remunerado. Com essa situação bastante reportada pela imprensa, surgiu interesse em criação de cursos para webdesign, e também desenvolvimento de software para facilitar o desenvolvimento de html. Tudo isso foi feito e hoje há muito mais profissionais habilitados a desenvolver páginas html, usando software muito mais fácil de usar. Com isso, o preço do profissional de webdesign caiu um pouco, mesmo com uma demanda sólida por profissionais desse tipo.

Uma das características na nossa época é que quase tudo tornou-se uma mercadoria, inclusive nossa capacidade profissional. Essa é uma das conseqüências do fenômeno econômico conhecido como “globalização”. Como estamos comentando, a longo prazo, a única forma de se manter um preço *premium* para um produto é torna-lo um produto diferenciado.

Portanto, qual deve ser a estratégia de um profissional de tecnologia de informação para evitar ser transformado numa mercadoria e portanto jogado numa situação onde ganhará pouco ? É

¹ “mercadoria” em inglês é *commodity*

evidentemente uma pergunta complexa, mas a resposta certamente estará na capacidade de o profissional reciclar-se mais rápido que a média, mantendo-se sempre como um produto diferenciado. Na hora de escolher uma linguagem de programação para estudar, procure pensar a longo prazo e avaliar em quais mercados pretende atuar.

1.5 Sociedade do conhecimento

A humanidade está vivendo a “sociedade do conhecimento”. O conhecimento tornou-se o mais valorizado ativo das empresas. Obter, assimilar, classificar, recuperar, acessar, interpretar, gerenciar e manipular conhecimentos são algumas das habilidades valorizadas nesse ambiente. Dentre os conhecimentos que se deve ter não incluem-se apenas os conhecimentos técnicos. Igualmente importante é ter conhecimentos sociais (relacionamentos), conhecimentos de mercado, conhecimento de como e onde se produz e se acessa (ou se compra) conhecimentos de boa qualidade, saber distinguir um conhecimento de boa qualidade de um conhecimento de qualidade desconfiável, saber para que servem os conhecimentos técnicos que se dispões, etc.

Alguns conceitos fundamentais:

- Dados: é um valor derivado da observação de objeto ou referência. Exemplo: 15 graus celcius. Para serem registrados, guardados e disponibilizados os dados devem estar ligados a um *quadro referencial*. No caso o quadro referencial é a escala de temperatura.
- Informação: é uma coleção de dados **relacionados** que, dentro de um determinado **contexto**, podem transmitir algum significado a que os consulta. Exemplo: guia telefônico, tabela de temperaturas. Para quem não sabe o que é um telefone, o guia telefônico não contém informação alguma.
- Comunicação: é a transferência de informações de um agente de processamento dessa informação para outro.
- Conhecimento: É a familiaridade, a consciência ou entendimento conseguido seja através da experiência ou do estudo. O conhecimento pode ser classificado nas formas abaixo:
 - Tácito - é algo pessoal, formado dentro de um contexto social e individual. Não é propriedade de uma organização ou de uma coletividade. Em geral é difícil de ser transmitido. Por exemplo: um craque de futebol sabe dar uma bicicleta e marcar um gol. Por mais que se veja e reveja o craque fazendo isso, é difícil fazer o mesmo.
 - Explícito - envolve conhecimento dos fatos. É adquirido principalmente pela informação, quase sempre pela educação formal. Está documentado em livros, manuais, bases de dados, etc. Por exemplo: o conhecimento da sintaxe e da programação de C/C++ é um conhecimento explícito. Já a boa programação é em parte explícito, mas envolve uma certa dose de conhecimento tácito. A capacidade de trocar informações com um grupo (tribo) do mesmo segmento de tecnologia é um complemento fundamental para o profissional, e envolve muito conhecimento tácito.
- Sabedoria: É a qualidade de possuir grande conhecimento, geralmente de mais de uma área do saber. A associação de compreensão de diversos conhecimentos faz surgir uma “compreensão geral”, ou “compreensão da grande figura”, que é exatamente o aspecto que configura a sabedoria. Uma das vertentes da sabedoria é conseguir ver o mundo pelo ponto de vista de outras pessoas.

1.5.1 Mudança do paradigma de negócios

Pode-se chamar de “sociedade do conhecimento” a sociedade que em grande parte adotou para si os “paradigmas economia do conhecimento”, mais atuais, que se opõem aos “paradigmas da economia do conhecimento”. Um quadro comparativo desses paradigmas é mostrado abaixo.

Paradigmas da Economia Industrial	Paradigmas da Economia do Conhecimento
Fordismo, economia de escala, padronização	Escala flexível, customização, pequenos estoques, <i>just in time</i> .
Grandes tempos de resposta	Tempo real
Valorização de ativos tangíveis	Valorização de ativos intangíveis
Atuação local, regional	Atuação global
Intensivo em energia (petróleo) e materiais	Intensivo em informação e conhecimento
Ramos matrizes: petrolíferas, petroquímicas, automóveis, aeroespacial	Ramos matrizes: eletrônico, informática, telecom., robótica, genética
Qualidade do produto	Qualidade do processo (além do produto)
Grandes organizações verticais	Sub-contratação (tercerização), <i>downsizing</i> , redes de relacionamento entre empresas
Rigidez da hierarquia administrativa, separada da produção	Flexibilidade e integração de empresa com clientes, colaboradores e fornecedores
Taylorismo. Decomposição do processo de trabalho	Trabalhador polivalente
Empregado como gerador de custo	Empregado como geradores de receita (capital intelectual)
Pleno emprego	Empregabilidade
Pouco ou nenhum aprendizado no trabalho	Aprendizado contínuo
Qualificação formal	Competência (capacidade efetiva de resolver problemas)

1.5.2 “Mercadorização”

Tradicionalmente, se um segmento do mercado estiver na situação de escassez e com ganhos relativamente altos, esse segmento torna-se atrativo para negociantes. Desde que exista informação disponível sobre o ambiente de negócios, a tendência é de o segmento atrair novos entrantes, que farão aumentar a oferta de produtos, até que o preço se estabilize num patamar razoável (com taxas de lucro relativamente pequenas), quando então diz-se que o segmento de mercado está maduro.

No ambiente atual, a Internet ajudou a tornar as informações mais abundantes e baratas. Portanto como nunca há reportagens informando qualquer segmento de mercado que esteja “dando dinheiro”. Além disso, a facilidade de comunicação tornou o conhecimento rapidamente transferível de um local

a outro. Com isso, serviços de conhecimento passaram a ser globalizados. Por exemplo: pode-se desenvolver software num país para um cliente em outro país.

O efeito final desses fatores é a chamada “mercadorização” do ambiente de negócios. Isto é, tudo transforma-se numa mercadoria, e rapidamente. A concorrência em todos os setores parece ser incrivelmente grande, e o consumidor exige preço e qualidade como nunca, sabendo que há inúmeras opções caso não se sinta satisfeito. Isso é bom para quem compra, mas torna-se difícil a vida de quem quer vender. E todos nós temos que vender alguma coisa, nem que seja a nossa capacidade profissional no mercado.

1.5.3 Remuneração × agregação de valor na cadeia produtiva

O processo de produção de um produto (ou serviço) é geralmente composto de várias etapas, ou elos. Pode-se dizer que há uma “cadeia produtiva” que faz surgir o produto. Com a comercialização do produto final, faz-se caixa (ganha-se dinheiro), com o qual se remunera direta ou indiretamente toda a cadeia produtiva.

Uma das características da sociedade do conhecimento é que as melhores remunerações da cadeia de valor acabam sendo dos elos que agregam alto valor de conhecimento. Por exemplo: na produção de um carro, o ato de montar as peças não agrega tanto valor de conhecimento quando o projeto de um carro novo. E a remuneração do projetista é maior do que a do montador. Na escala mercadoria-produto_diferenciado, o projetista de carro está mais para produto diferenciado, enquanto o montador está mais para mercadoria. Mas gradativamente mesmo o projetista torna-se também uma mercadoria, e passa a ter menores remunerações. Apenas um projetista que saiba fazer algo difícil (entenda-se: que poucos sabem fazer) e útil (entenda-se: valorizado no mercado) consegue obter boa remuneração.

1.6 Porque C/C++ ?

De tudo o que foi dito (até agora) nesse capítulo de introdução, o que se pode concluir sobre a escolha de uma linguagem de programação para um estudo profundo ?

1.6.1 “Tribos” de tecnologia

O uso de uma tecnologia - tal como uma linguagem de computador - geralmente cria uma “tribo” de usuários que gostam dessa tecnologia. Os membros da “tribo” costumam achar que a “sua” linguagem é a melhor, por isso é que torna-se difícil julgar de forma isenta qual seria a linguagem melhor. A mentalidade de tribo é de certa forma semelhante a mentalidade que une os torcedores de um time de futebol, ou os crentes de uma religião.

Por exemplo: quem está habituado a usar unix (e.g. linux) como servidor, geralmente não tolera usar Windows. Vice versa também acontece. Uma empresa com cultura Windows bem consolidada muitas vezes reluta em usar servidores unix, ainda que gratuitos.

Para quem conhece bem uma certa linguagem de computador, e essa linguagem atende às suas necessidades, a tal linguagem é em princípio a melhor opção. Mas a experiência mostra que o ambiente de trabalho com tecnologia de informação muda constantemente, e muito rapidamente. Uma linguagem de computador não se “deteriora”, no mesmo sentido que uma máquina velha o faz. Mas considerando-se a constante mudança do ambiente, se uma linguagem de computador deixa de ter suporte às novidades que o mercado exige, então essa linguagem de certa forma está se deteriorando. Há inúmeros exemplos de linguagens de programação muito boas que entraram em desuso, pois não houve uma “manutenção”. O dBase ou o Clipper para sistema operacional DOS, são um bom exemplo. São duas linguagens que foram muito importantes no seu tempo. Mas o ambiente passou a

valorizar o Windows, e as versões para Windows desses programas custaram muito a sair. Além disso, com o crescimento meteórico do tamanho das bases de dados que para as quais se desenvolvem sistemas, era imperioso que houvesse suporte para uso de hardware mais moderno (que é lançado a toda hora). Assim, os usuários ficaram de certa forma pressionados a migrar para uma alternativa.

Uma boa tribo de tecnologia costuma trocar muitas informações entre seus membros. Na era da Internet, essa troca de informações costuma dar-se por web sites e listas de email.

Quem trabalha com recursos humanos para tecnologia deve estar sempre ciente do comportamento de “tribo” que esses profissionais tem. Não basta oferecer salário (embora isso seja bastante importante). É preciso oferecer para o profissional de tecnologia um ambiente em que ele se desenvolva profissionalmente a partir das tribos a que pertence ou que gostaria de pertencer.

1.6.2 A escolha tecnológica é uma escolha estratégica

A escolha de uma linguagem de computador é a escolha de uma tecnologia. Essa escolha é estratégica, pois uma escolha errada pode fazer perder tempo e/ou dinheiro.

Não se escolhe um software apenas pelos seus méritos e pelo seu preço. Se seus amigos usam o software x, você tende a escolher essa opção também. **Se um software é muito usado, apenas por isso ele tem mais valor.** Um software que não tem intrinsecamente muitos méritos (mas que funciona), mas que é usado por uma comunidade grande é geralmente a melhor opção de escolha. Exemplo: Windows.

É sabido que o ser humano tende a acrescentar conhecimentos a sua mente, em forte associação com os conhecimentos que já possui. No caso de linguagem de programação isso significa que em geral aprende-se uma linguagem de programação em tende-se a não querer mudar de linguagem, para aproveitar o que já se sabe.

Para um profissional de tecnologia em formação (um aluno), ou um profissional trainee (com relativamente pouca experiência), a questão que se coloca é a seguinte: é preciso estudar, e há coisas demais para aprender. Portanto o ideal é optar por um caminho de alta flexibilidade, que dê ao profissional condições de atuar de forma bastante ampla.

Para ter flexibilidade na vida profissional, é conveniente adotar uma “tribo” que tenha a maior abrangência possível.

O nosso curso visa ensinar conceitos de desenvolvimento de software. Para que isso seja feito, é importante que o aluno faça programas práticos, de forma a ilustrar os conceitos. A linguagem escolhida é C++. Essa é a opção de escolha principalmente pelas razões abaixo.

1. O compilador é fácil de se fazer. Isso não é uma vantagem direta, pois não se está falando em fazer um compilador. Contudo trata-se de uma grande vantagem indireta, pois há inúmeros compiladores C / C++ no mercado, e isso é bom para o programador. Há compilador C / C++ para praticamente qualquer CPU e sistema operacional que exista.
2. Não havendo um “dono” da linguagem, acaba que o padrão C / C++ tem inúmeros patrocinadores, famosos e anônimos, o que aumenta o desenvolvimento e o suporte.
3. É estrategicamente mais seguro para uma empresa confiar seu sistema de informação em um padrão de múltiplos “donos” que comprar um pacote que pertença a um único fornecedor, por

mais renomado que seja. Há muito mais fornecedores de bibliotecas para ligar com C / C++ que em qualquer outra linguagem.

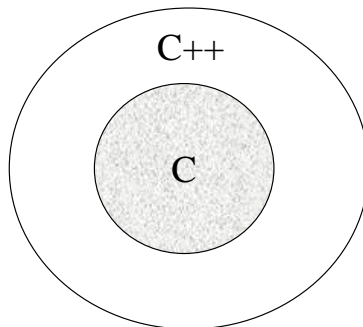


Figura 1: C++ é um super-set de C

As linguagens C / C++ possuem uma filosofia de retirar da própria linguagem tudo o que pode ser escrito em cima dela. Isto quer dizer que muitos comandos que o leitor deverá usar (tais como aqueles necessários para colocar mensagens na tela ou manipular arquivos) e que em outras linguagens correspondem a comandos intrínsecos da linguagem, passarão a ser em C / C++ simples chamadas de funções escritas na própria linguagem. Estas funções, que serão mostradas mais tarde, estão disponíveis na biblioteca padrão da linguagem e podem ser usadas em qualquer compilador C / C++. Esta sutil diferença de filosofia levará a facilitar o trabalho daquele que cria o compilador de uma linguagem de muito poucos comandos onde se compila a biblioteca padrão. O mercado comprova este fato oferecendo ao comprador muito mais opções de compilador C / C++ que qualquer outra linguagem.

Acredito que para se obter qualquer aprendizado, é fundamental que se esteja motivado. Quando o objeto do aprendizado é uma tecnologia, isso é mais verdade do que nunca. O mundo tecnológico é tumultuado de opções, que competem entre si pelo direito a sua atenção. Essa seção de introdução para o aprendizado de C/C++ pretende motivar o leitor para o aprendizado, e situar a alternativa dessa linguagem no contexto das demais tecnologias que lhe são alternativas.

A linguagem de programação C/C++ *não* é especialmente fácil de ser aprendida. Existem alternativas que tem curva de aprendizado mais rápida. Então porque utilizar esta linguagem ? Acontece que há uma vantagem indireta no uso de C/C++, que é a seguinte: **trata-se de uma linguagem para a qual é fácil de se fazer um compilador**. Esse fato só pode ser considerado como vantagem (direta) para quem se interessa pela tarefa de fazer um compilador. Mas pouca gente se interessa por isso. E esse texto é voltado para quem quer aprender a programar a partir de um compilador já pronto, e não para quem pretende desenvolver um novo compilador C/C++. Então porque seria considerado vantagem usar uma linguagem para a qual é fácil de se fazer um compilador ? Porque graças a esse fato, **sempre há um compilador C/C++ para qualquer computador ou dispositivo programável que se pense !** Sendo que em muitos casos, há pelo menos uma opção de compilador que é gratuita. E essa característica não existe para nenhuma outra linguagem ! Para alguns dispositivos (DSPs, microcontroladores, etc.) há apenas compilador C, não havendo compilador C++. Essa é uma das poucas razões para ainda se estudar C puro (sem ++).

Tendo dito isso, o propósito do título do livro torna-se mais claro. Um bom programador em C/C++ deve, sempre que possível, abstrair-se da CPU, do compilador e do sistema operacional que está usando. Em alguns casos, não é sequer necessário que exista um sistema operacional. Fazendo isso, o programador estará **pensando de forma multiplataforma**. Dessa forma, **o conhecimento adquirido**

na linguagem C/C++ levará o programador a uma versatilidade enorme. Esse conhecimento pode ser usado para programar inúmeros ambientes, incluindo Windows, Unix (Linux, FreeBSD, etc.), computador de grande porte, implementação de interatividade em páginas Internet, diários portáteis tipo Palm Pilop ou Windows CE, dispositivos programáveis como DSPs, microcontroladores, etc. Se surgir no futuro algum tipo novo de processador ou computador que apresente alguma vantagem sobre o que existe hoje, com certeza uma das primeiras coisas que se fará para esse novo processador é o compilador mais fácil de se fazer - da linguagem C/C++.

1.7 Breve história do C/C++

A linguagem de programação C / C++ está em constante evolução, e isso é bom. Tudo começou com a linguagem BCPL, por Martin Richards, que rodava no computador DEC PDP-7, com sistema operacional unix (versão da época). Em 1970, Ken Thompson fez algumas melhorias na linguagem BCPL e chamou a nova linguagem de “B”. Em 1972, Dennis Ritchie e Ken Thompson fizeram várias melhorias na linguagem B e para dar um novo nome a linguagem, chamaram-na de “C” (como sucessor de “B”).

A linguagem C não foi um sucesso imediato após a sua criação, ficando restrita a alguns laboratórios. Em 1978 (um ano histórico para os programadores C), é lançado um livro famoso por Brian Kernigham e Dennis Ritchie. Esse livro serviu de tutorial para muita gente e mudou a história da programação em C. De fato, essa primeira versão da linguagem C é até hoje conhecida como “C Kernigham e Ritchie” ou simplesmente “C K&R”.

Em 1981, a IBM lança o IBM PC, iniciando uma família de computadores de muito sucesso. A linguagem C estava no lugar certo e na hora certa para casar-se com o fenômeno dos computadores pessoais. Logo foram surgindo inúmeros compiladores para PC e a linguagem C estava livre do seu confinamento inicial no sistema operacional unix.

Conforme a tecnologia de software foi se desenvolvendo, o mercado exigiu que se introduzisse mudanças na linguagem C. Talvez nós devêssemos nos referir as várias versões de C como “C 1.0”, “C 2.0”, etc., mas o mercado geralmente não usa essa nomenclatura. O fato é que modificações importantes foram pouco a pouco sendo introduzidas na linguagem. Uma modificação importante foi a padronização feita pelo *American National Standards Institute* (ANSI), em 1983, que criou uma linguagem conhecida como “C ANSI”.

Entre outras modificações, o C ANSI mudou a forma de se definir um protótipo de função, para que se possa conferir esse protótipo na hora da compilação e acusar erros no caso de se chamar uma função com parâmetros errados. No quadro abaixo é mostrado um exemplo de como se escrevia uma função (`fun1`) em C K&R e como passou a se escrever em C ANSI. Se segunda função (`fun2`) chama a primeira com parâmetros errados, o C K&R não tem como saber do erro na hora da compilação, mas o C ANSI acusa o erro e com isso ajuda o programador a depurar o seu código.

C K&R	C ANSI
<pre>void fun1 (i, k) int i, k; { ... }</pre>	<pre>void fun1 (int i, int k) { ... }</pre>

```
void fun2 () {  
    int a1, a2; float b1, b2; /* variáveis locais */  
    fun1(a1,a2); /* OK em ambas as versões de C */  
    fun1(b1,b2); /* é um erro, pois os parâmetros  
    estão errados. Mas somente surgirá um erro de compilação  
    no C ANSI, no C K&R não há esse tipo de checagem */  
}
```

No C K&R, como não havia como checar se uma função foi chamada com os argumentos certos, não se proibia o uso de identificadores que não previamente declarados. No exemplo acima significa que podia-se chamar a função `fun1` a qualquer momento, sem necessidade de declarar o identificador da função. Com o C ANSI, passou a se recomendar (como técnica de boa programação) que toda função fosse declarada antes de ser usada, para que o compilador pudesse checar a chamada da função. A partir do C++ essa recomendação tornou-se uma obrigação, isto é, o compilador simplesmente não aceita compilar um identificador que não tenha sido previamente declarado².

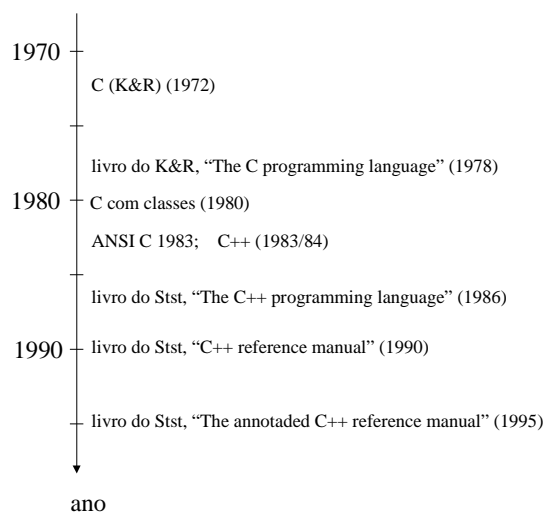


Figura 2: Alguns fatos relevantes na evolução da linguagem C / C++

1.8 Qualidades da Linguagem C/C++

Há ainda outras qualidades interessantes na linguagem C/C++, listadas abaixo.

1. Não há um "dono" desta linguagem. Há vários sistemas operacionais e compiladores 100% gratuitos que usam o padrão C/C++ (o exemplo mais visível é o Linux). Isso significa na prática que essa linguagem tem inúmeros patrocinadores, famosos e anônimos. Isso mantém o suporte sempre muito atualizado e amplamente disponível na Internet.
2. Escolher C/C++ é uma opção estratégica muito interessante para uma empresa ou consultor, pois sendo um padrão amplamente difundido e com opções gratuitas. Portanto pode-se reproduzir infinitamente (sem custo adicional e sem fazer pirataria) qualquer solução que se aprenda ou se desenvolva a partir dessa linguagem. Há inúmeras opções tecnológicas de

² Para quem está no começo da aprendizagem, pode ser um tanto difícil compreender agora a diferença entre o C K&R e o C ANSI. O importante é lembrar que o C K&R é mais antigo e mais sem proteção. O C ANSI é mais moderno e dá algumas dicas a mais para facilitar encontrar erros de programação. O C++ é mais moderno ainda, e melhor que C.

programas (com fonte incluído) disponíveis na Internet. Os links de algumas delas estão listadas na página web desse livro.

3. É verdade que a curva de aprendizado de C/C++ tende a ser mais lenta (especialmente no início) que a de outras linguagens. Mas isso pode ser compensado com um bom material didático. Esse livro, que é gratuito e distribuído na Internet, é uma contribuição no sentido de se aumentar a oferta de material didático para o estudo de C/C++.
4. Conforme será explicado em detalhes em outro trecho do livro, é possível “encapsular” complexidades de programação de forma a tornar o uso de C/C++ perfeitamente digeríveis mesmo para um programador inexperiente. Isso pode ser feito a partir do uso de bibliotecas (que existem em enorme quantidade a disposição na Internet, e que podem ser desenvolvidas/adaptadas para necessidades específicas).
5. Pode-se usar C/C++ com técnicas apropriadas para trabalho em equipe. Dessa forma, um gerente de projeto de software pode alocar tarefas para diversos programadores, que trabalham em paralelo. Com essas técnicas pode-se obter resultados excelentes no tempo de desenvolvimento de um software.

1.9 Classificação de interfaces de programas

Na informática há hardware e software. O hardware é fisicamente tangível, ao contrário do software. A palavra software significa um programa de computador ou um *sistema*, que é um conjunto de programas de computador.

Há basicamente 3 tipos de interface para programas, como mostrado abaixo em ordem de complexidade (1 = menos complexo, 3 = mais complexo).

1. Console (ou linha de comando), tela de texto tipo *scroll* (rolar). Exemplo: xcopy, grep.
2. Console (ou linha de comando), tela de texto cheia. Exemplo: Lotus 1-2-3, WordStar (para DOS).
3. GUI (*graphics user interface*), também chamado de “interface Windows”. Exemplo: sistema operacional Windows e qualquer aplicação para Windows ou o ambiente xWindows do unix.

Uma interface mais complexa agrada mais ao usuário, porém complica a vida do programador. Quando o objetivo é estudar conceitos de programação, a interface é algo que a princípio não está no foco. Portanto, a interface mais adequada para estudo é a interface tipo 1 – console do tipo *scroll*.

Em Windows, é necessário que se abra uma janela de DOS para se ter acesso ao console e poder usar programas com linha de comando (como se fazia na época que não havia Windows). Em unix, há o xWindows, que é um dos tipos de interface gráfica, e também o console de unix.

1.10 Programando para console

Ultimamente, há particular interesse por aplicativos gráficos, com uso intensivo de GUI e multimedia. Há também muito interesse em programação para Internet. Para quem está em fase de aprendizado, é importante que se tenha uma base sólida de programação em console antes de se pretender programar em ambientes mais complexos.

Programar para console é a abordagem que historicamente surgiu primeiro que a programação GUI. Programas para console NÃO estão caindo em desuso, apesar da popularidade dos ambientes gráficos. Um exemplo importante de como a programação para console é muito presente na modernidade do

mercado de informática, é o caso de programação para Internet com interface CGI. Outro exemplo é a programação para dispositivos programáveis como DSPs, microcontroladores, etc.

Mesmo para quem pretende programar para GUI (como o Windows ou xWindows), é extremamente importante saber programar para console. Fazendo programas para console que se consegue-se depurar trechos de programa facilmente, e com isso pode-se produzir componentes de software que serão aproveitados em programas quaisquer, incluindo programas gráficos.

1.11 Linguagens de programação de computador

Um computador é acionado por um processador central, conhecido como CPU (*central processing unit*). A CPU é um circuito eletrônico programável, que executa instruções em um código próprio, chamado de código de máquina. Esse código é fisicamente uma sequência de bytes³. Esse conjunto de bytes é conhecido como programa executável de computador. “Programa” porque é uma sequência de instruções; “executável”, porque essa sequência está na forma binária já traduzida para o código de máquina da CPU, portanto está pronto para ser executado (um programa escrito em arquivo de texto não está pronto para ser executado).

Quase sempre, usa-se um computador como um sistema operacional (Windows, Linux, etc.). Esse sistema operacional possibilita que se trabalhe com arquivos. Quando um usuário comanda a execução de um programa, o que acontece fisicamente (de forma resumida) é que o sistema operacional copia para a memória o arquivo com o programa executável e pula para o seu início.

A CPU somente executa código de máquina binário. É muito difícil para um programador trabalhar diretamente com a informação em binário. Portanto, a técnica básica de programação consiste em utilizar uma “linguagem de programação”. Ou seja, o programador escreve um arquivo de texto (ou um conjunto de arquivos) chamado de programa fonte. Em seguida converte esses arquivos texto no código de máquina num processo chamado de “compilação”

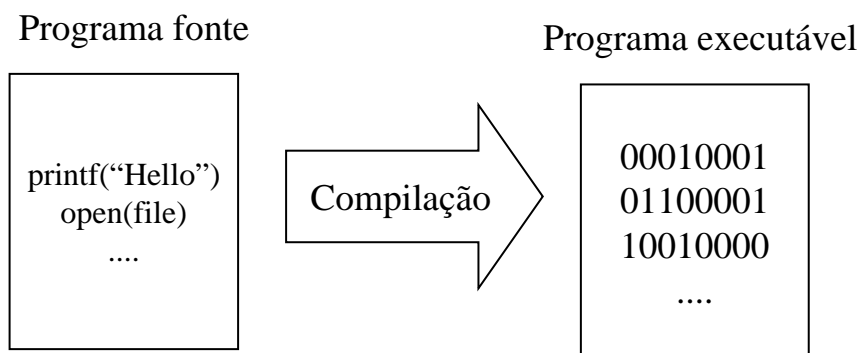


Figura 3: Visão gráfica da compilação

1.11.1 Alto nível x baixo nível

Existem inúmeras linguagens de computador. As linguagens são (simplificadamente) divididas em 2 grupos:

1. Linguagens de baixo nível.
2. Linguagens de alto nível.

³ Um byte é um conjunto de 8 bits.

As linguagens de baixo nível são aquelas onde se programa diretamente com as instruções da linguagem de máquina da CPU. Essas linguagens são também chamadas de *assembly* ou *assembler*⁴. O programador escreve um texto fonte e compila-o para gerar o código de máquina, mas o nível com que pode escrever a linguagem é bastante baixo. Ao se programar em assembler, é muito fácil que se perca a idéia geral de conjunto do programa. Além disso, o texto fonte gerado é específico para uma CPU, portanto o programa não se aproveita caso se queira usar outra CPU diferente. A menos de certos casos específicos, é em geral uma má idéia tentar programar em assembler.

As linguagens de alto nível são independentes da CPU e só por esse motivo já são bem superiores às linguagens de baixo nível. Há muitas linguagens de alto nível no mercado, e é difícil fazer um julgamento isento para saber delas qual é a melhor. Algumas linguagens de alto nível existentes são:

1. C
2. C++
3. FORTRAN
4. PASCAL
5. PL/1
6. ALGOL
7. PROLOG
8. JAVA
9. ADA

⁴ *assembler* significa “montador”

Capítulo 2) Conheça o Seu Compilador

Esse capítulo explica como conhecer o compilador que se vai usar. Cada seção fala de um compilador diferente. Portanto use esse capítulo assim: verifique dentre os compiladores abordados qual é o que você vai usar e leia essa seção. As demais seções podem ser ignoradas (a princípio), pois falam de outros compiladores.

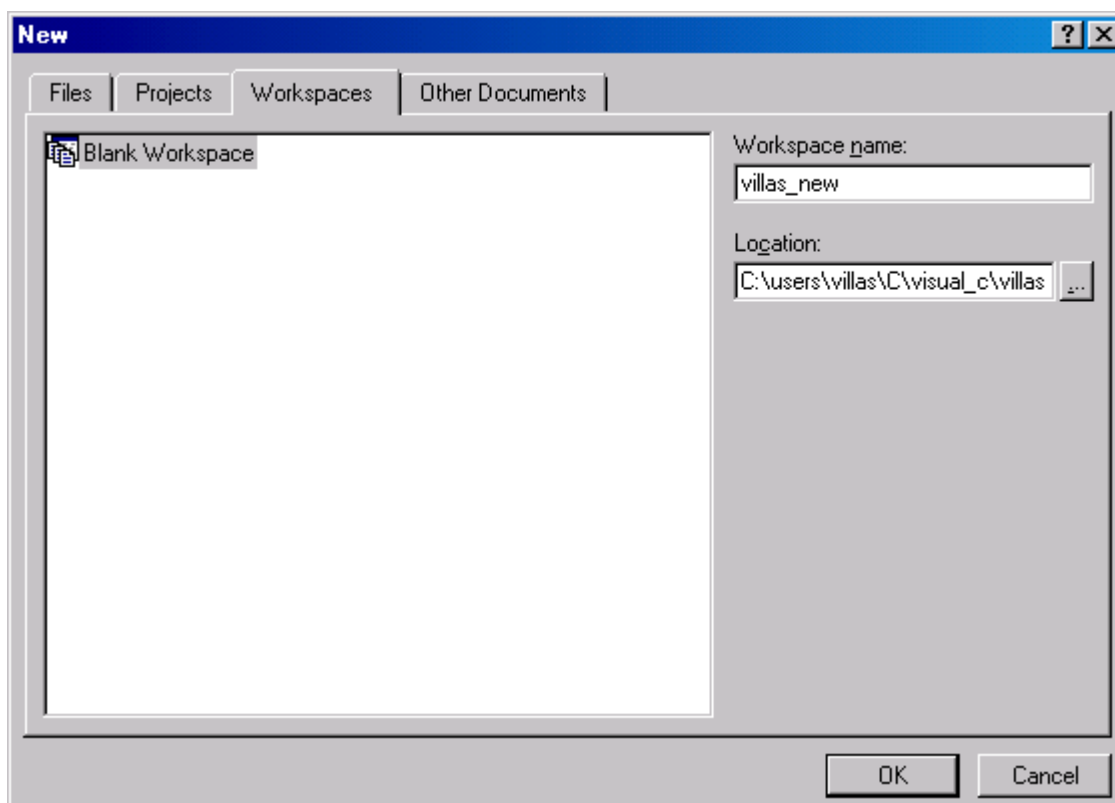
Para um compilador qualquer, é importante que se adquira as seguintes habilidades:

1. Fazer um programa tipo “hello world” para console.
2. Adicionar argumentos na linha de comando.
3. Usar o help do programa.
4. Usar “projetos” (fazer programas com múltiplos fontes).
5. Usar bibliotecas (*libraries*), isto é,
 - o Incluir no projeto uma biblioteca num projeto.
 - o Fazer uma biblioteca para uma terceira pessoa.
 - o Examinar (listar) o conteúdo de uma biblioteca qualquer.
6. Usar ferramenta de debug.
7. Definir um identificador para permitir compilação condicional.
8. Escrever um código para detectar vazamento de memória

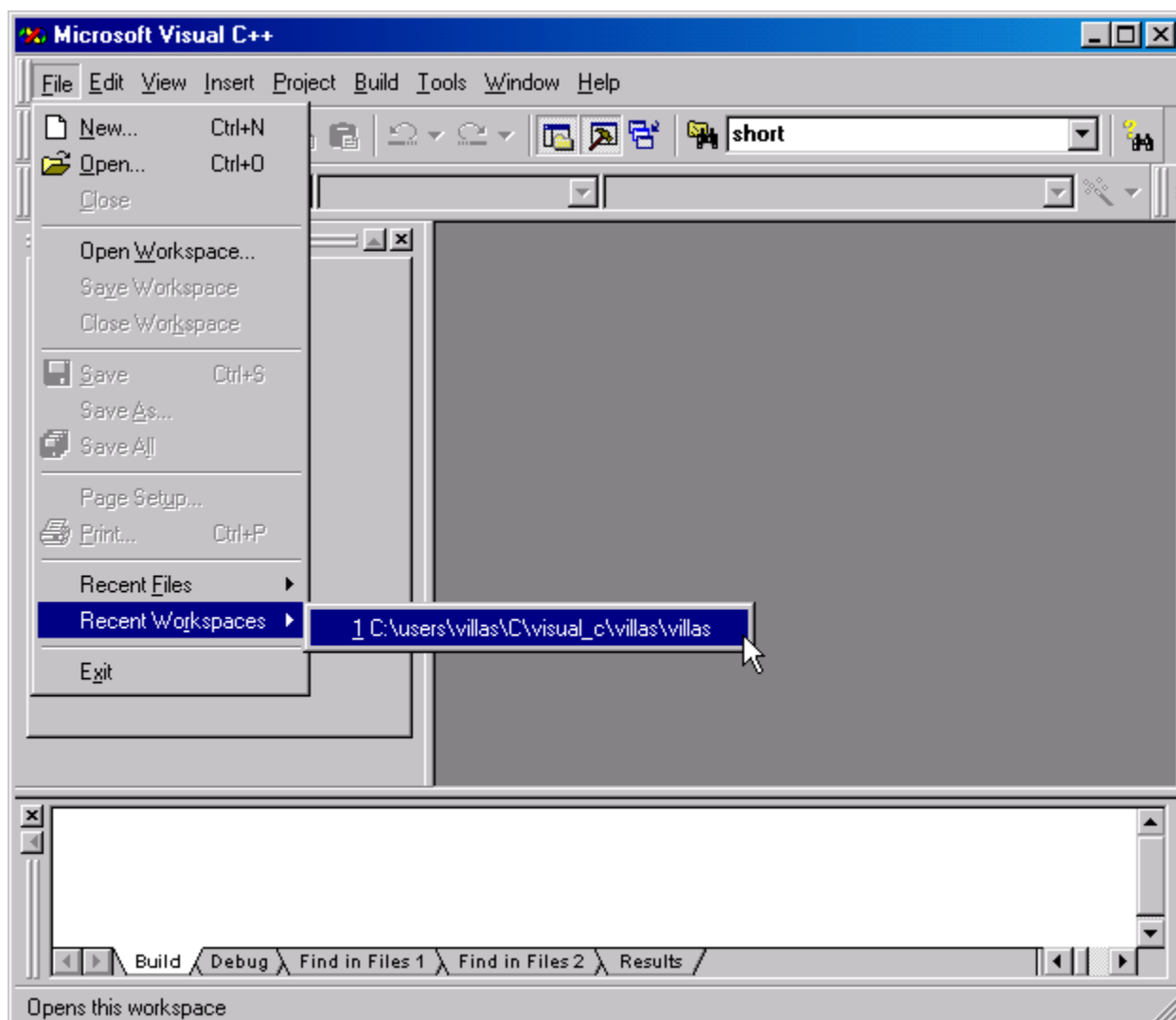
2.1 Visual C++ 6.0

2.1.1 Reconhecendo o Compilador

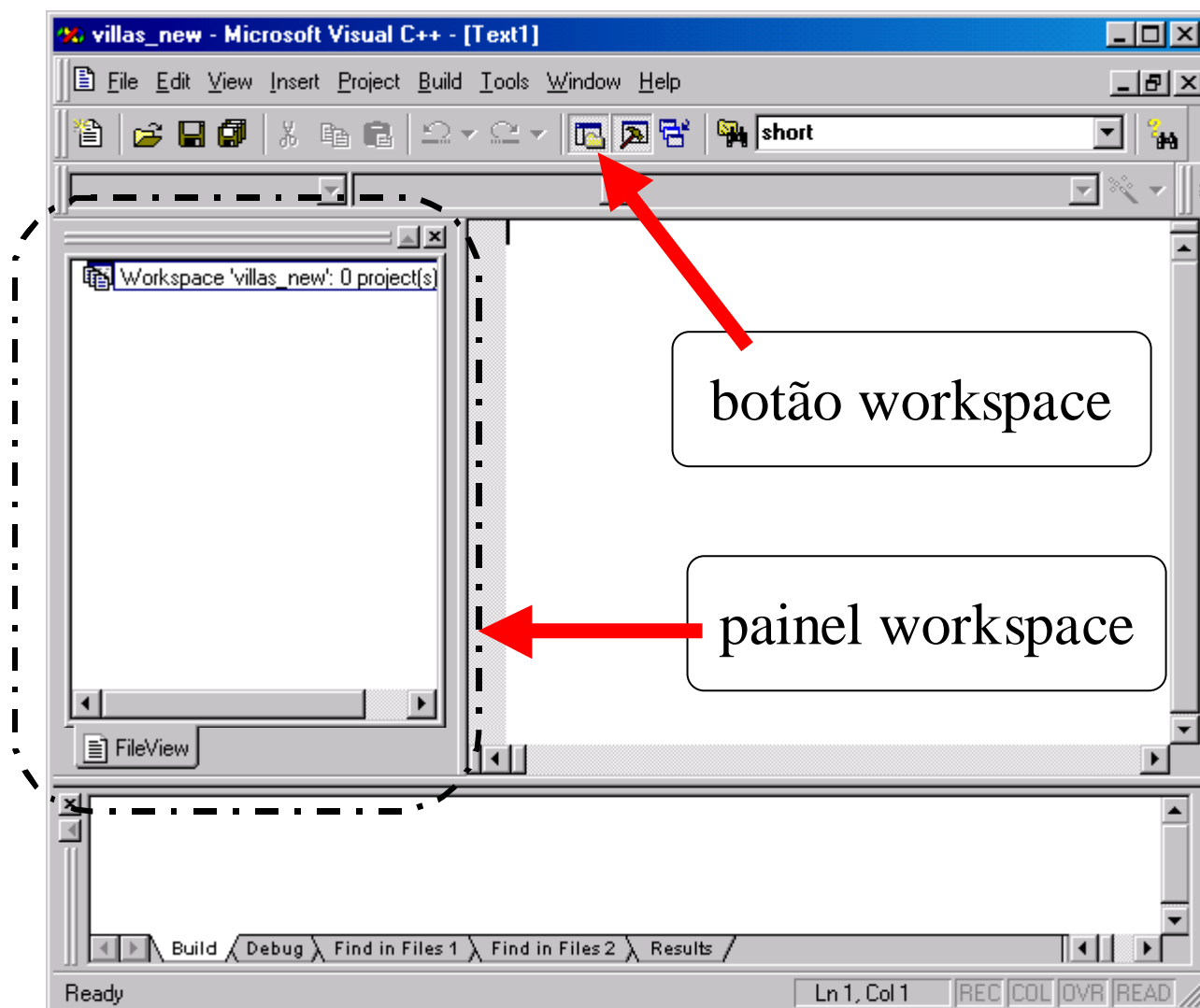
O VisualC trabalha com o conceito de “workspace”. A idéia é que várias pessoas podem trabalhar com o mesmo compilador e o que uma pessoa faz não afeta o ambiente de trabalho de outra pessoa. A primeira coisa a fazer com esse compilador é criar um workspace para iniciar o trabalho. Para isso, vá no menu File - New. Na caixa de diálogo “New” que aparecer, selecione o tab “workspaces”. Digite um nome para o workspace no campo “Workspace name” (selecione também o local para o workspace). Um bom nome para o workspace é o seu nome pessoal (o nome pelo qual é conhecido, possivelmente o sobrenome; no meu caso uso “villas”; nesse exemplo, usei “villas_new”). Cria-se assim o arquivo “villas_new.dsw”



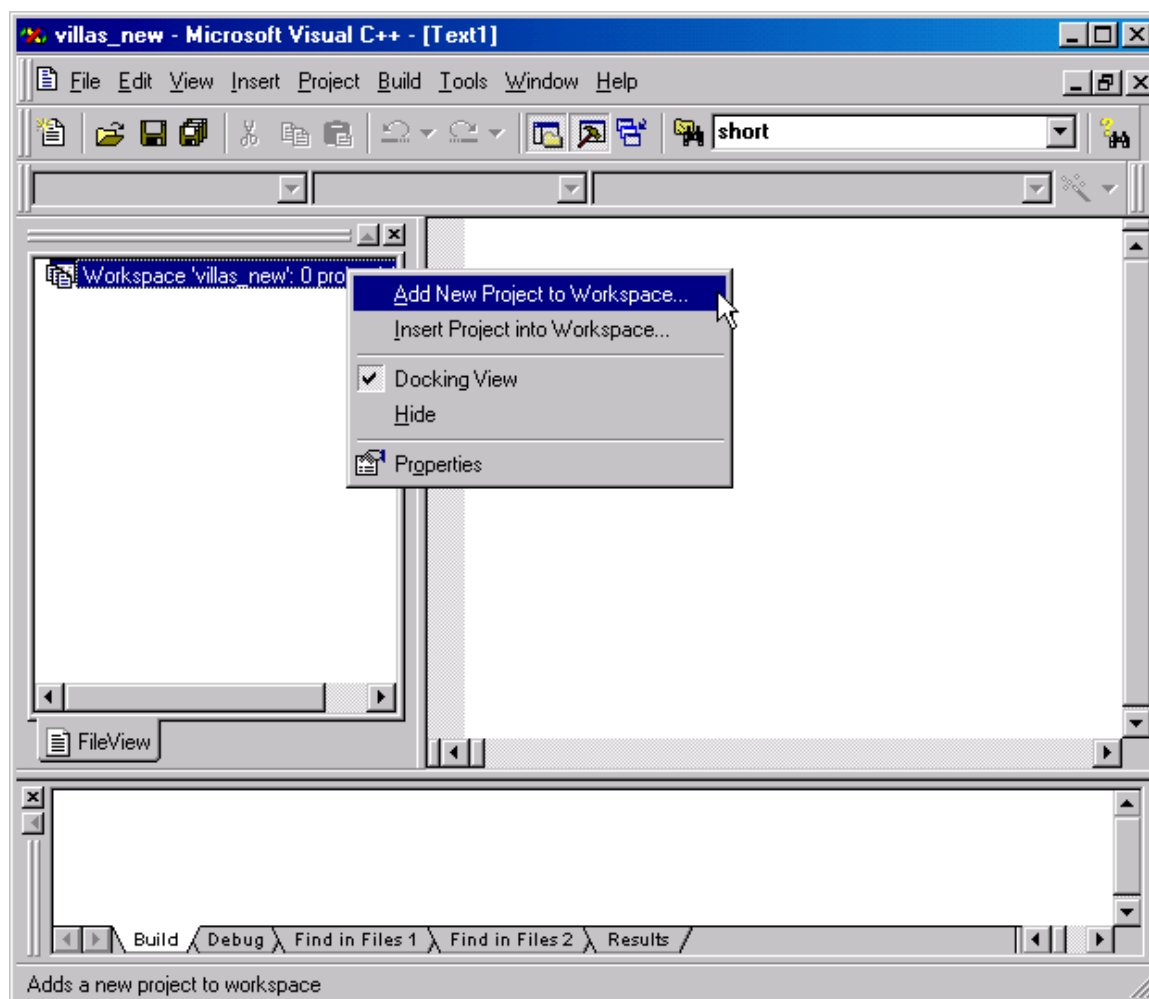
Ao entrar no VisualC uma próxima vez, abra o seu workspace e trabalhe dentro dele. Uma forma fácil de fazer isso é usar o comando pelo menu File - Recent Workspaces.



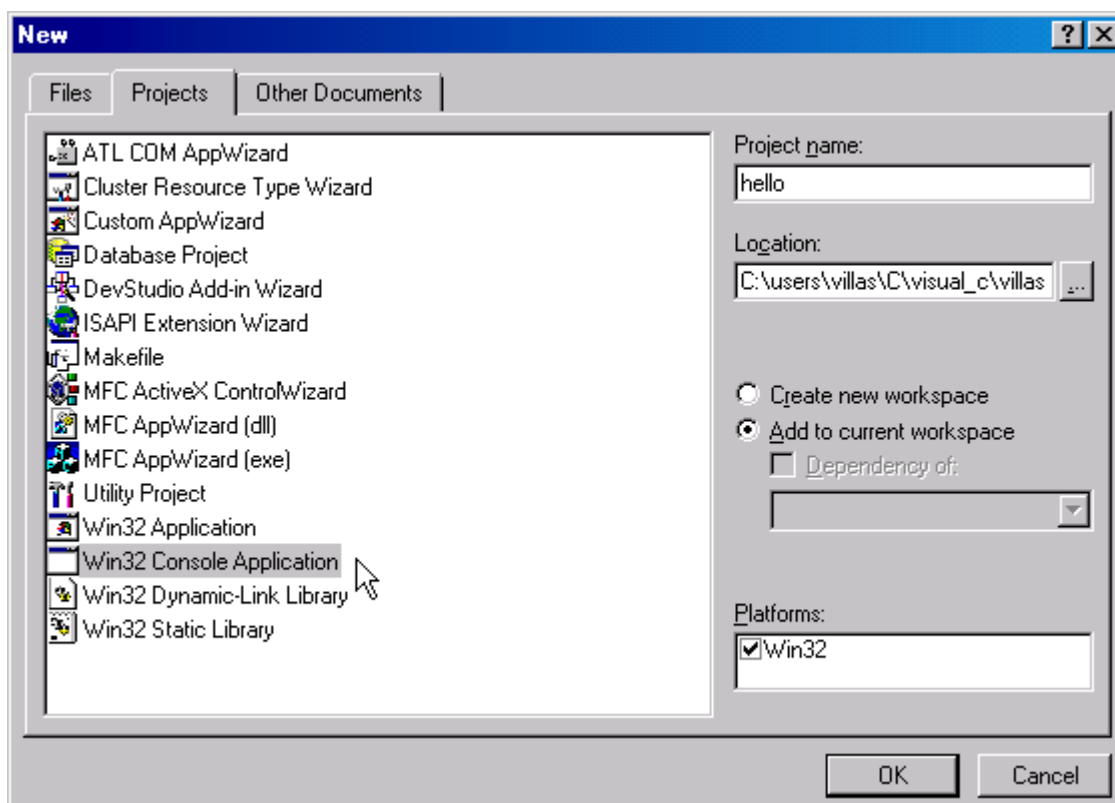
Um workspace pode conter diversos projetos. Cada projeto é uma aplicação, biblioteca, componente, etc., que se pretende desenvolver. Mantenha o “painel workspace” a mostra (use o botão “workspace” na barra de ferramentas para comutar (*toggle*) entre ver e não ver o painel workspace, que fica do lado esquerdo da tela).



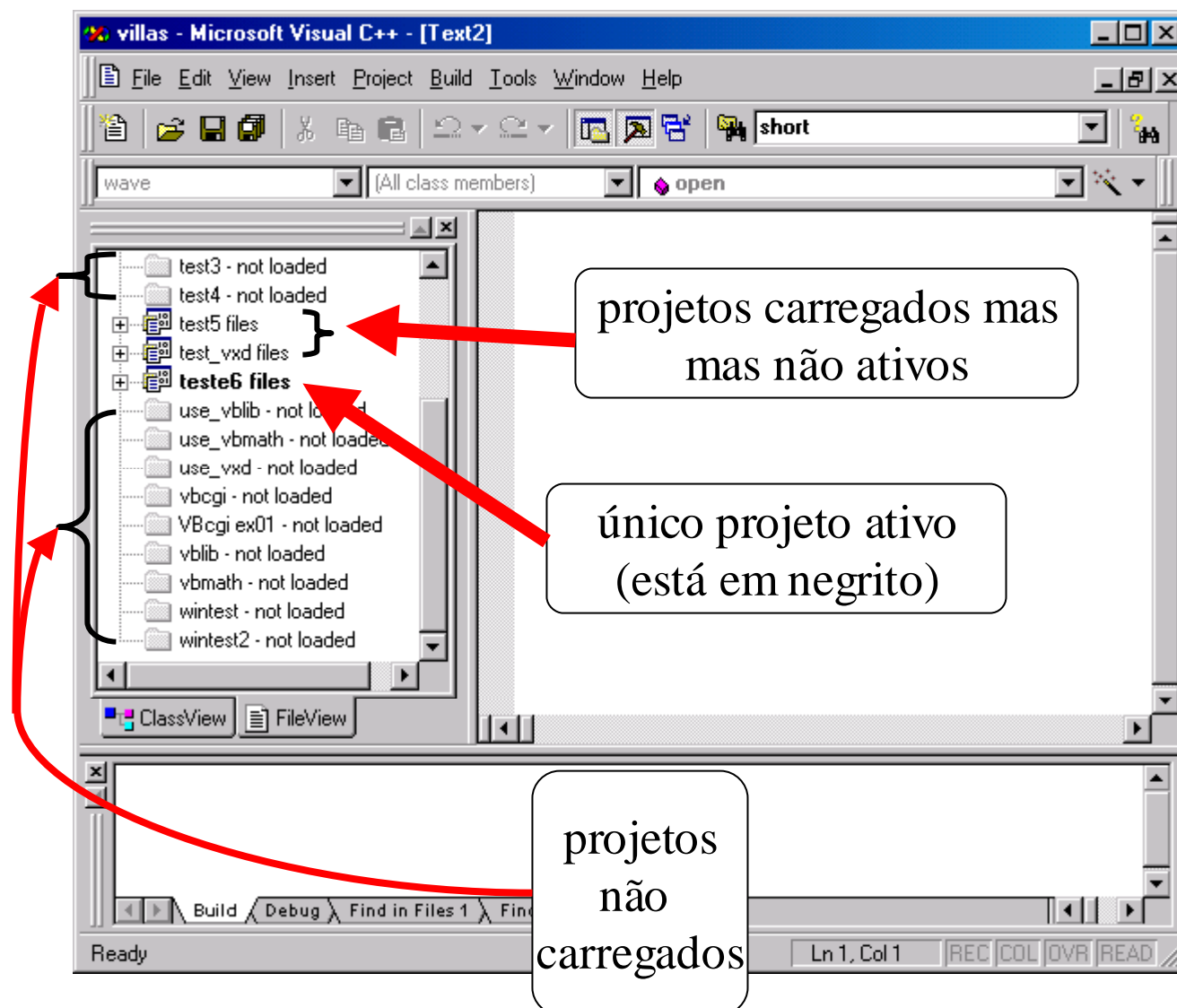
No painel workspace, clique com o botão direito sobre o ícone com nome “workspace ‘nome’”, e selecione “Add New Project to Workspace”.



No tab “projects” escolha o tipo de projeto que deseja fazer. Para iniciantes, escolha “win32 console application”. Dê um nome ao projeto (por exemplo “hello”) e clique OK.



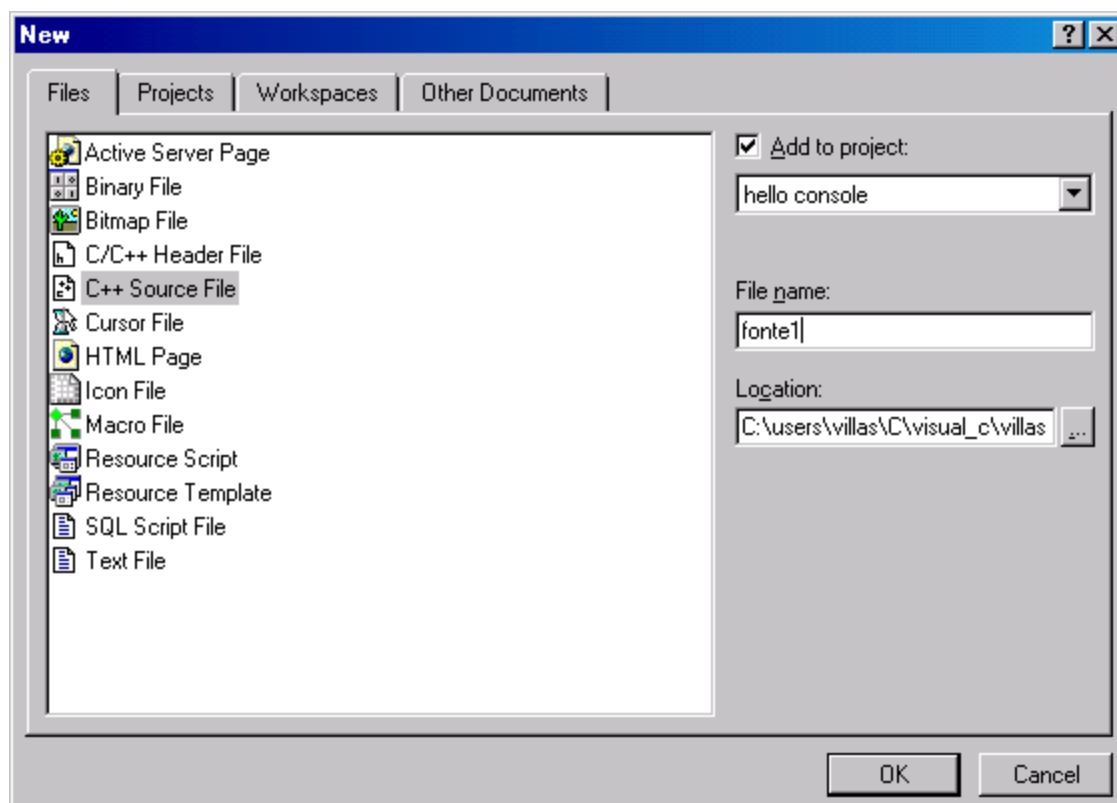
Dentro de um mesmo workspace, podem existir vários projetos. Dentre os projetos existentes, pode haver mais de um carregado (*loaded*). Dentre os projetos carregados, somente um está “active” (no foco). O único projeto ativo aparece com letras em negrito (*bold*), e é esse projeto que está sendo compilado e “debugado”.



2.1.2 “Hello world” para DOS

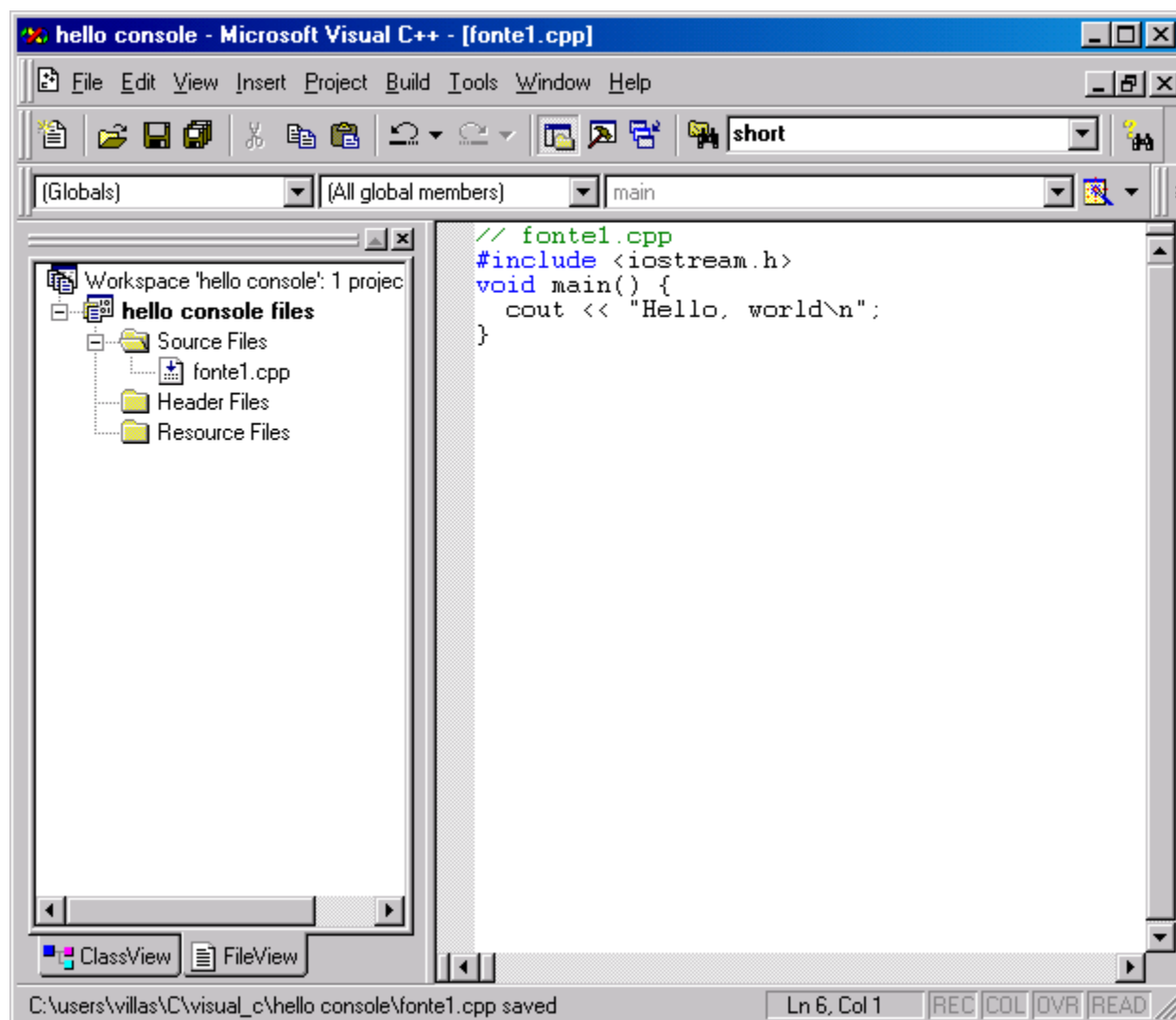
Crie um projeto com nome “hello console” do tipo “win32 console application”. No step 1 do application wizard, escolha “An empty project” (um projeto vazio, apesar da opção “Typical hello world application”). Clique “Finish”.

O seu projeto precisa de pelo menos um arquivo de código fonte. Crie-o com **File - New -** (no tab Files) **<C++ source file>** - no campo **File name**, escreva o nome do arquivo, digamos, “fonte1” (na verdade fonte1.cpp, mas a extensão .cpp é automática) - deixe ativo o flag “**A**dd to project”, para que o seu arquivo fonte1.cpp seja automaticamente incluído no projeto em questão - **<OK>**.

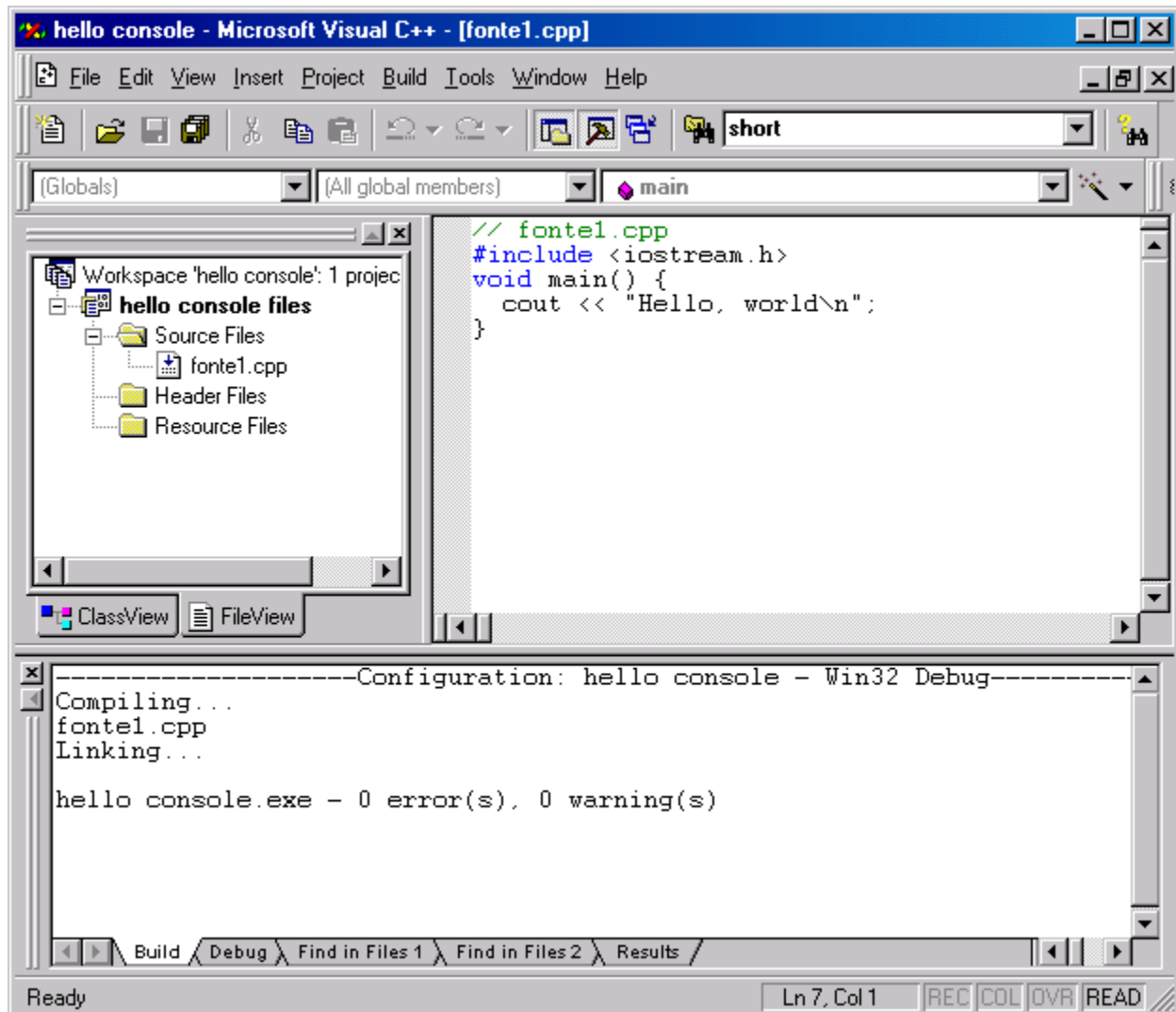


Confirme que a janela da direita está editando o arquivo fonte1.cpp olhando a barra de título do VisualC++, com “[fonte1.cpp]” do lado direito. Escreva o texto abaixo no arquivo que criou.

```
// fonte1.cpp
#include <iostream.h>
void main() {
    cout << "Hello, world\n";
}
```



Compile e ligue (*link*) o programa com Build - Build (F7). Durante a compilação, na parte de baixo se abrirá uma janela mostrando as informações de compilação.



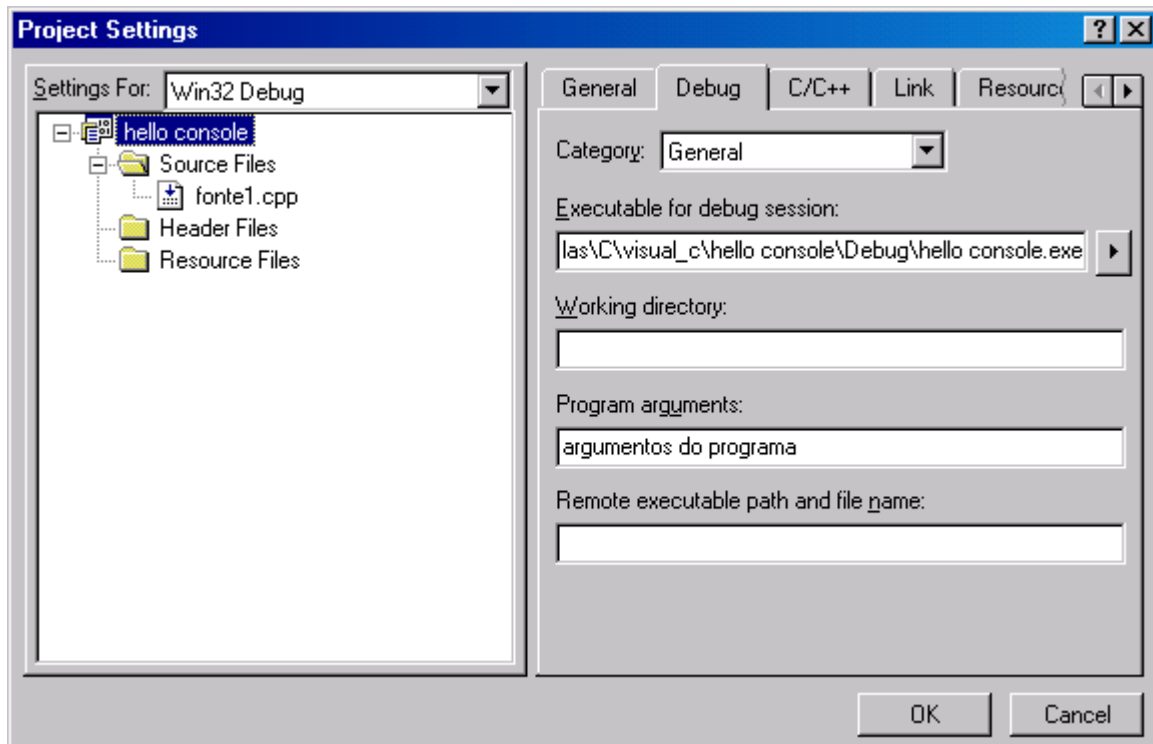
Pode-se rodar o programa sem debug, de dentro do VisualC++, com **Build - Execute** (<ctrl>F5). O mesmo programa também pode ser rodado “manualmente” a partir de uma janela DOS. Para isso, abra uma janela de DOS, vá para o diretório mostrado abaixo (ou o equivalente no seu sistema)

“C:\users\visual_c\ villas_new\hello console\Debug”

Lá há o programa “hello console.exe” que você acabou de compilar (no DOS aparece como “helloc~1.exe” devido ao problema de compatibilidade do DOS com arquivos com nomes maiores que 8 letras). Digite “helloc~1.exe” no prompt do DOS e veja o seu “hello world”.

2.1.2.1 Adicionando argumentos para a linha de comando

Comande **Projects - Settings** (<alt>F7) - <Debug tab> - no campo de <Program arguments>, digite os argumentos para o seu programa.



O programa abaixo lista os argumentos.

```
#include <iostream.h>
void main(int argc, char**argv) {
    cout << "Hello, world\n";
    int i;
    for (i=0; i<argc; i++)
        cout << "Arg[" << i << "]: " << argv[i] << endl;
}
```

Resultado:

```
Hello, world
Arg[0]:C:\USERS\villas_C\visual_c\villas_new\hello console\Debug\hello console.exe
Arg[1]:argumentos
Arg[2]:do
Arg[3]:programa
```

2.1.3 Usando o Help

Aprender a usar bem o help é uma habilidade estratégica, pois é um conhecimento que permite ao programador usar o seu tempo para pesquisa pessoal no sentido de se aprimorar no uso da linguagem. Aprender a usar o help é “aprender a aprender”.

Como é fundamental usar o help do programa, isso torna-se mais um motivo para que nessa apostila se use o idioma inglês misturado com português. Para se encontrar algo no help, é preciso se ter em mente as palavras chave em inglês.

O help do Visual C++ 6.0 é muito grande, ocupando 2 CDs. Para não ocupar lugar demais no disco rígido do computador, a ferramenta de help vai copiando para o disco os pedaços do help a medida que vão sendo solicitados. Por isso, é sempre bom trabalhar com o Visual C++ tendo-se a mão os seus 2 CDs de help.

Uma forma direta de se usar o help é comandar **Help - Search** - e digitar o que se quer. Nessa apostila, em vários lugares recomenda-se que mais informação seja pesquisada no help, tópico “tal”, sendo que

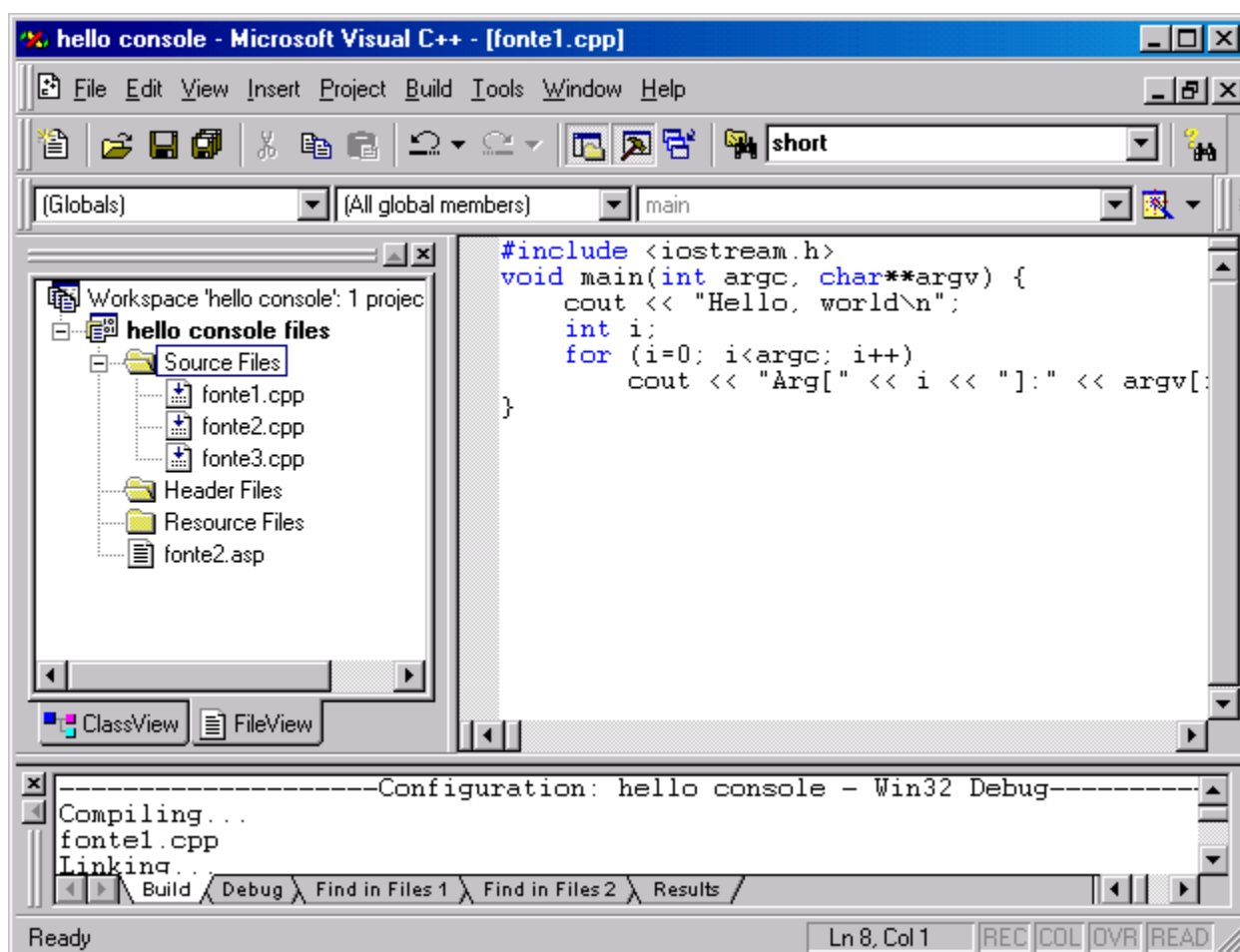
“tal” é uma palavra chave ou expressão chave faz o Visual C++ apontar para a parte correspondente do help.

2.1.4 Projetos (programas com múltiplos fontes)

Num “Workspace” do Visual C++, há diversos projetos. Cada projeto desses pode ter um ou mais arquivos fonte. Para compilar um programa que possui mais de um arquivo fonte, basta colocar todos os arquivos fonte que compõem o programa debaixo do mesmo folder “Source Files”.

Não é necessário acrescentar manualmente os arquivos header no folder “Header Files”. O Visual C++ faz isso automaticamente ao compilar os arquivos *.cpp.

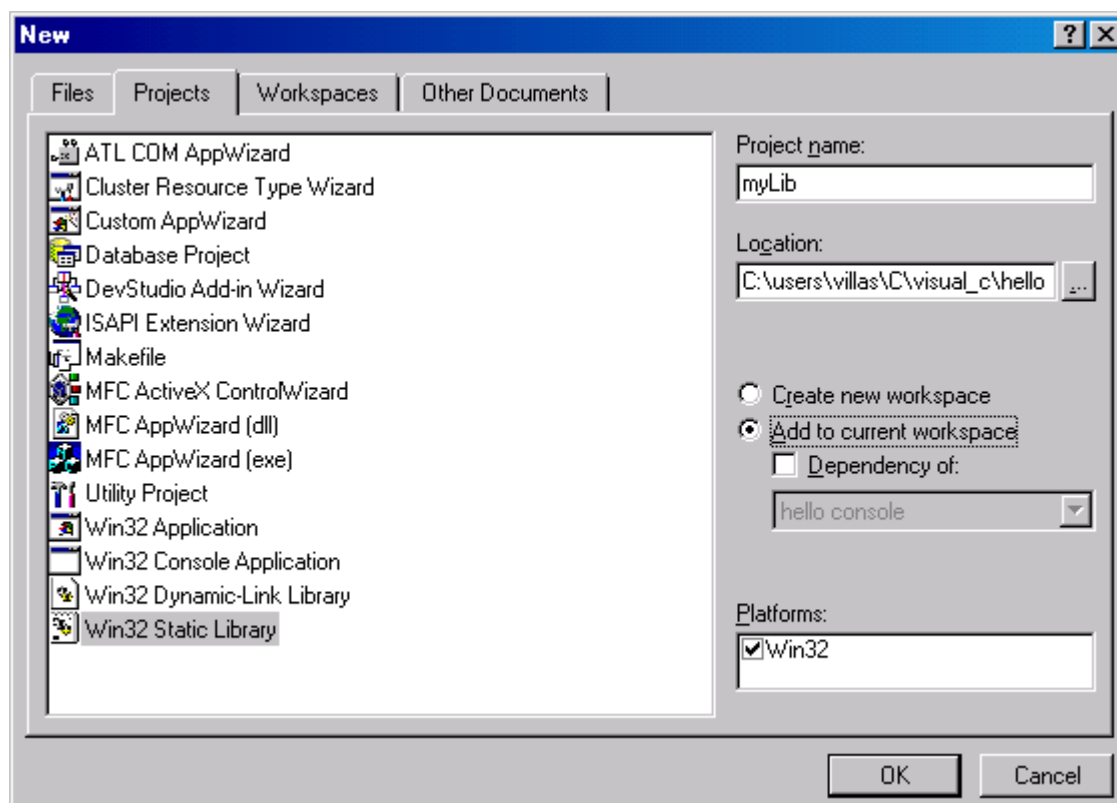
Além de arquivos *.cpp, pode-se também acrescentar no folder “Source Files” arquivos *.obj e *.lib, desde que compilados pelo Visual C++.



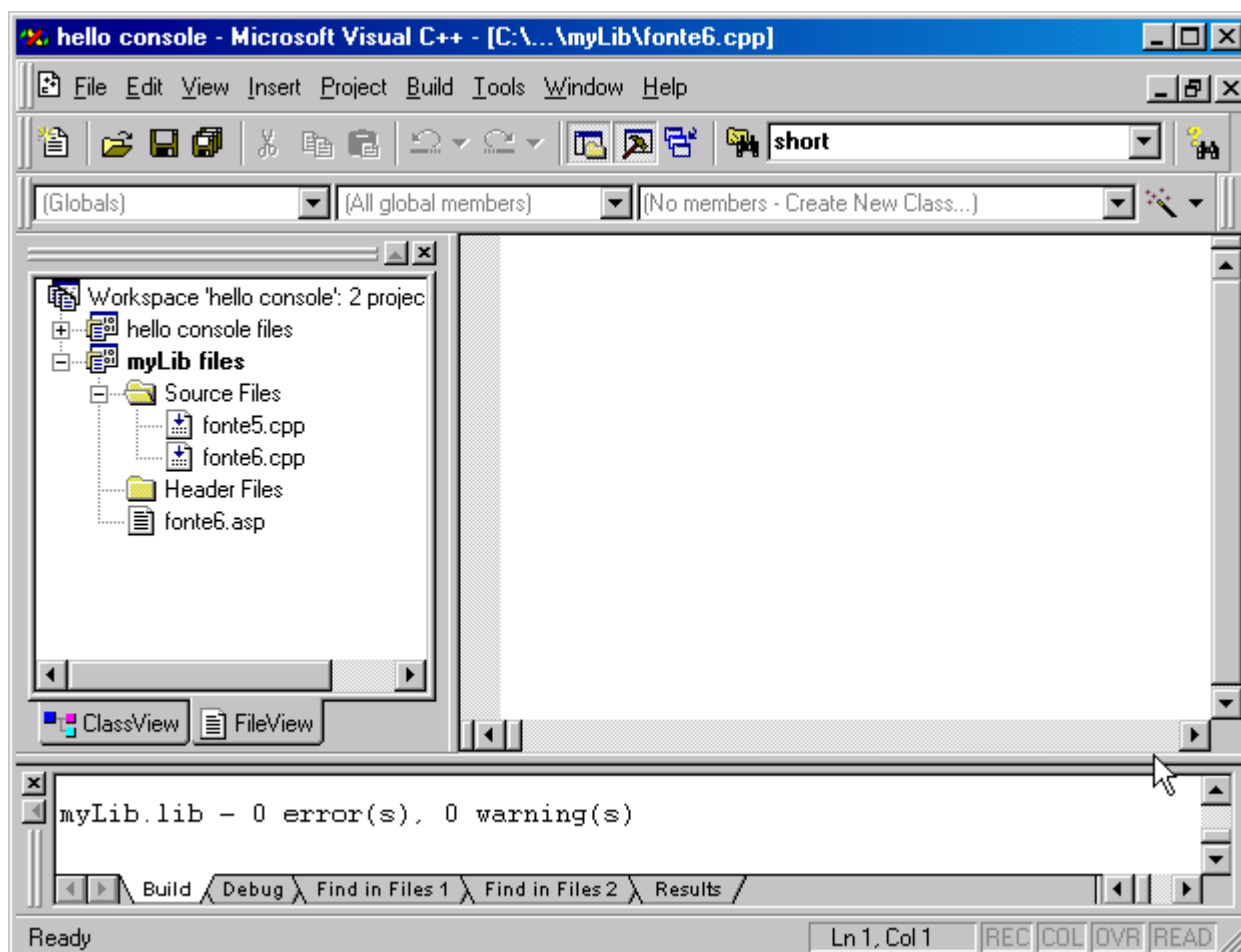
2.1.5 Bibliotecas

2.1.5.1 Fazer uma biblioteca

Para criar uma biblioteca estática (*.lib) com o Visual C++, comande **File - New**, e no tab “Projects” escolha “win 32 Static Library”. No exemplo da figura abaixo, foi escolhido o nome “myLib” para a biblioteca.



Acrescente arquivos fonte na biblioteca, como se o projeto fosse gerar um executável. Compile a biblioteca da mesma forma como se compila um executável (F7). Também da mesma forma como num programa executável, escolha a versão (debug, release) da biblioteca (veja página 42).



2.1.5.2 Incluir uma biblioteca num projeto

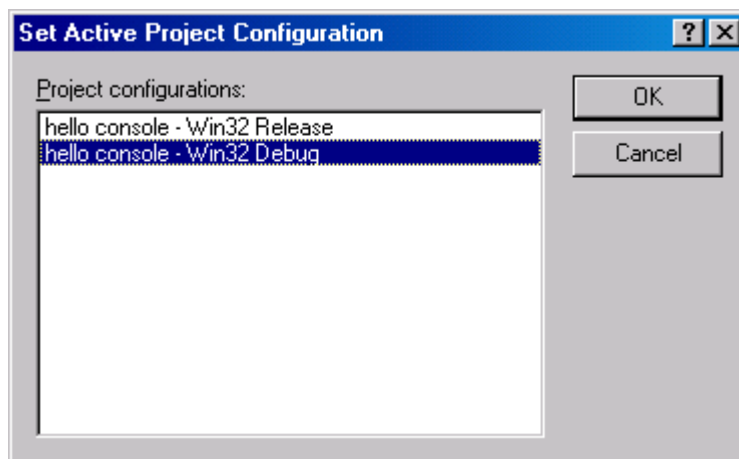
Criar uma biblioteca é um projeto, mas a lib gerada não é executável. Para incluir uma biblioteca num outro projeto que gera um programa executável, basta acrescentar a biblioteca no folder “Source Files” do projeto que gera executável e mandar compilar.

2.1.5.3 Examinar uma biblioteca

```
// todo
```

2.1.6 Debug

Debugar um programa significa executa-lo passo a passo, observando as variáveis e conferindo se o comportamento encontra-se conforme esperado. Para que um programa possa ser debugado, é preciso que seja compilado para essa finalidade, isto é, que o compilador salve no arquivo executável final informações para possibilitar o debug. Obviamente essa informação de debug ocupa espaço, e esse espaço é desnecessário na versão final (*release version*) do programa. Portanto, quando se compila um programa é preciso saber se o compilador está configurado para compilar a debug version ou a release version. No Visual C++, isso é feito com **Build - Set Active Configuration**, e selecionando a versão desejada. Para um mesmo programa, compile-o na versão release e debug e compare os tamanhos do arquivo executável gerado. Para um programa “hello world”, o tamanho da versão debug é 173K, enquanto o tamanho da versão release é 36K.



Quando se inicia o debug de um programa, surge o menu de Debug. As opções de debug desse menu estão mostradas abaixo.

Resumo dos Comandos do Debug	
Go	F5
Restart	Ctrl+Shift+F5
Stop Debugging	Shift+F5
Step Into	F11
Step Over	F10
Step Out	Shift+F11
Run to Cursor	Ctrl+F10
Quick Watch	Shift+F9
Toggle Break Point	F9

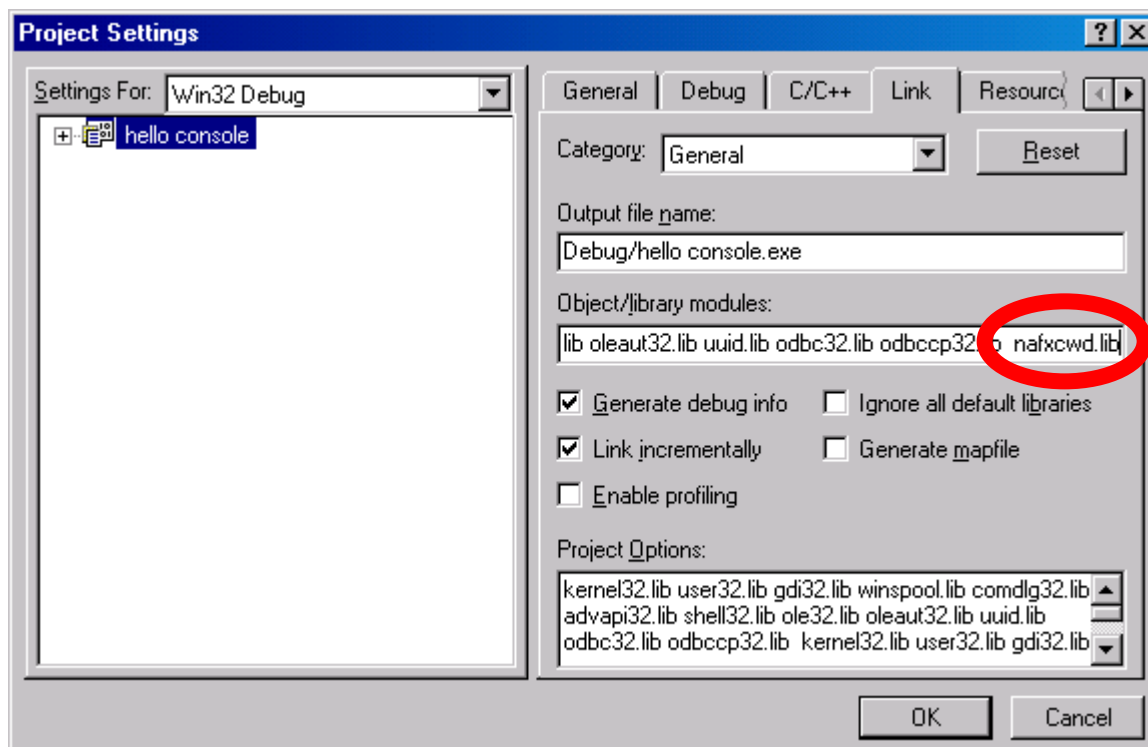


A única forma de compreender o funcionamento do debug é praticando-o. Portanto, pratique os comandos de debug acima até que se sinta a vontade com a sua utilização.

2.1.7 Dicas extras

2.1.7.1 Acrescentando Lib no Project Settings

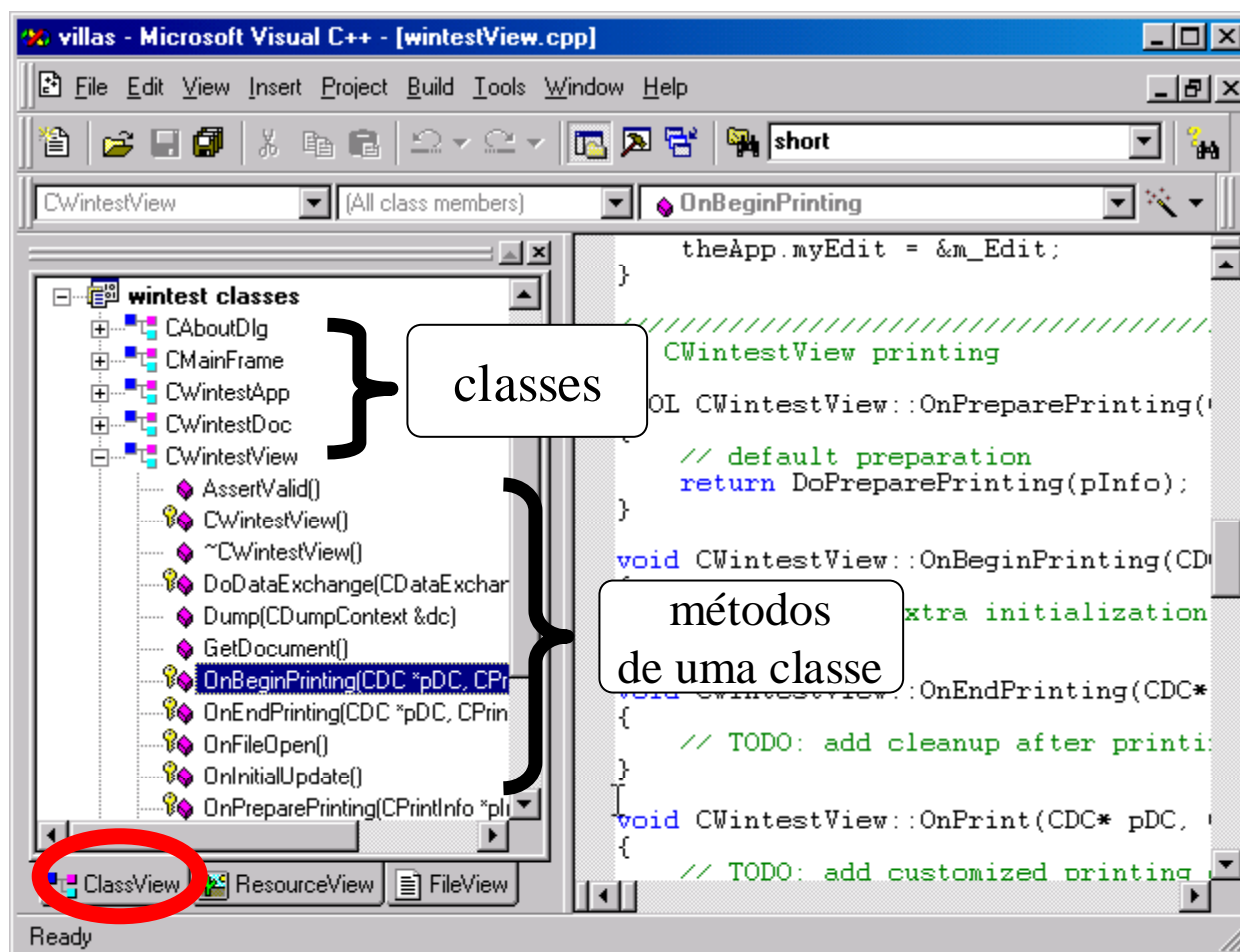
Ao usar certas funções, é preciso incluir a biblioteca correspondente. Por exemplo, para se usar a função "GetCurrentDirectory", é preciso se incluir a biblioteca "nafxcwd.lib". Para acrescentar uma biblioteca a ser linkada no projeto, comande Projects - Settings (<alt>F7)- <Link tab> - no campo de <Object/library modules>, clique no campo e acrescente no final da linha o nome da biblioteca a acrescentar "nafxcwd.lib".



2.1.7.2 Class View

A funcionalidade “Class View” do Visual C++ é um interessante diferencial desse produto. A idéia é “enxergar” um projeto de um programa C++ a partir da descrição de suas classes, mais que pela descrição de seus arquivos (o que corresponde a “File View”).

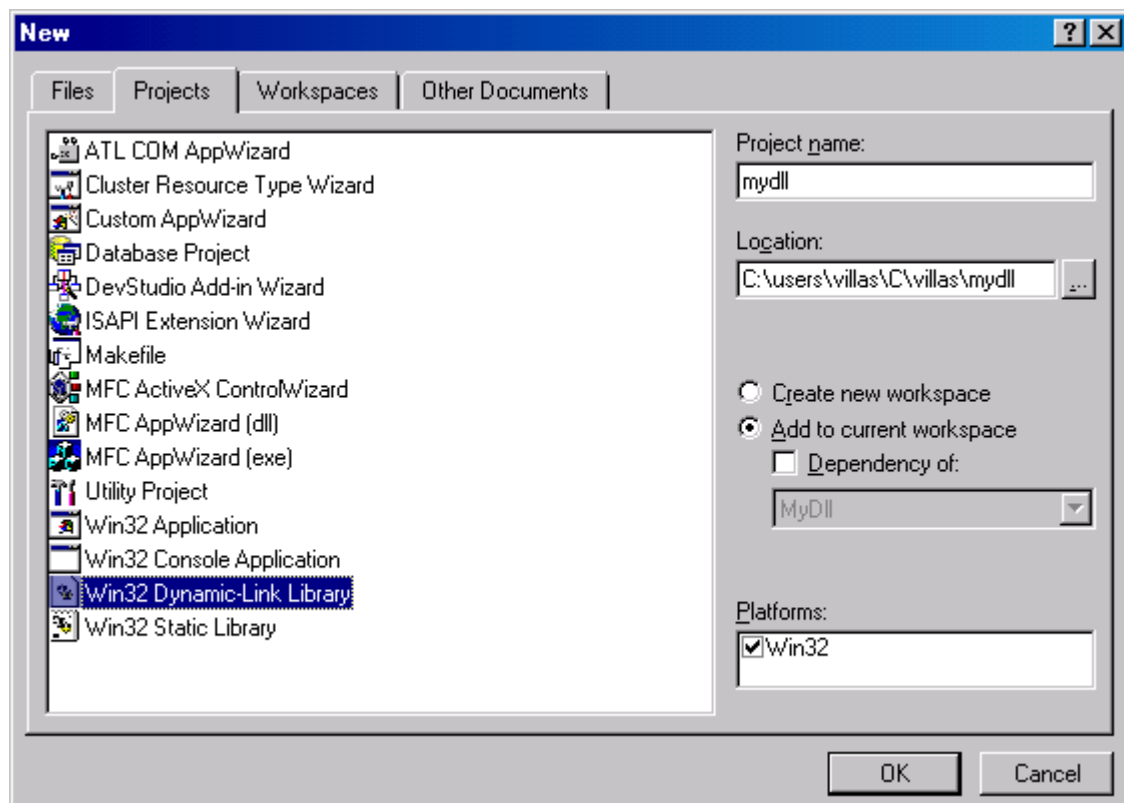
Há inúmeras funcionalidades de navegação no código C++ a partir da funcionalidade do Class View. Por exemplo: clicando-se num método de uma classe, automaticamente o editor é aberto e o cursor posicionado na classe em questão.



2.1.7.3 Usando bibliotecas de ligação dinâmica (DLL)

Uma biblioteca de ligação dinâmica ("Dynamic Link Library - DLL") é uma biblioteca que será ligada ao programa executável em tempo de execução. Para fazer uma dll, a primeira coisa a fazer é um projeto novo, em que se escolhe "Win32 Dynamic-Link Library", como mostrado na figura. A dll deve ter um nome (no caso "mydll", e pertencer a algum workspace). No wizard, escolha "An empty DLL project".

Abaixo pode-se ver um exemplo de dll. Nesse exemplo, a biblioteca contém 3 funções globais. Uma é `dllTest`, outra é `plus_1`, e a terceira é `plus_2`. A `dllTest` apenas coloca uma constante string no console. As outras duas, recebem um inteiro como parâmetro e retornam um inteiro (somando 1 e 2 respectivamente). São funções simples de teste, úteis para um tutorial.



No arquivo mydll.cpp, escreve-se o corpo da dll, acrescentando-se a macro WINAPI ao protótipo da função. As regras de compilação da dll são as mesma de qualquer arquivo cpp.

```

////////////////////////////////////
// mydll.cpp
#include <iostream.h>
#include "mydll.h"
void WINAPI dllTest() {
    cout << "You're inside a Dll" << endl;
}
int WINAPI plus_1(int i) {
    return i+1;
}
int WINAPI plus_2(int i) {
    return i+2;
}

```

No arquivo mydll.h, coloca-se apenas os protótipos das funções globais da dll, acrescentando-se a mesma macro WINAPI. Uma outra macro _MYDLL_H_ controla a possibilidade de inclusão múltipla do header mydll.h.

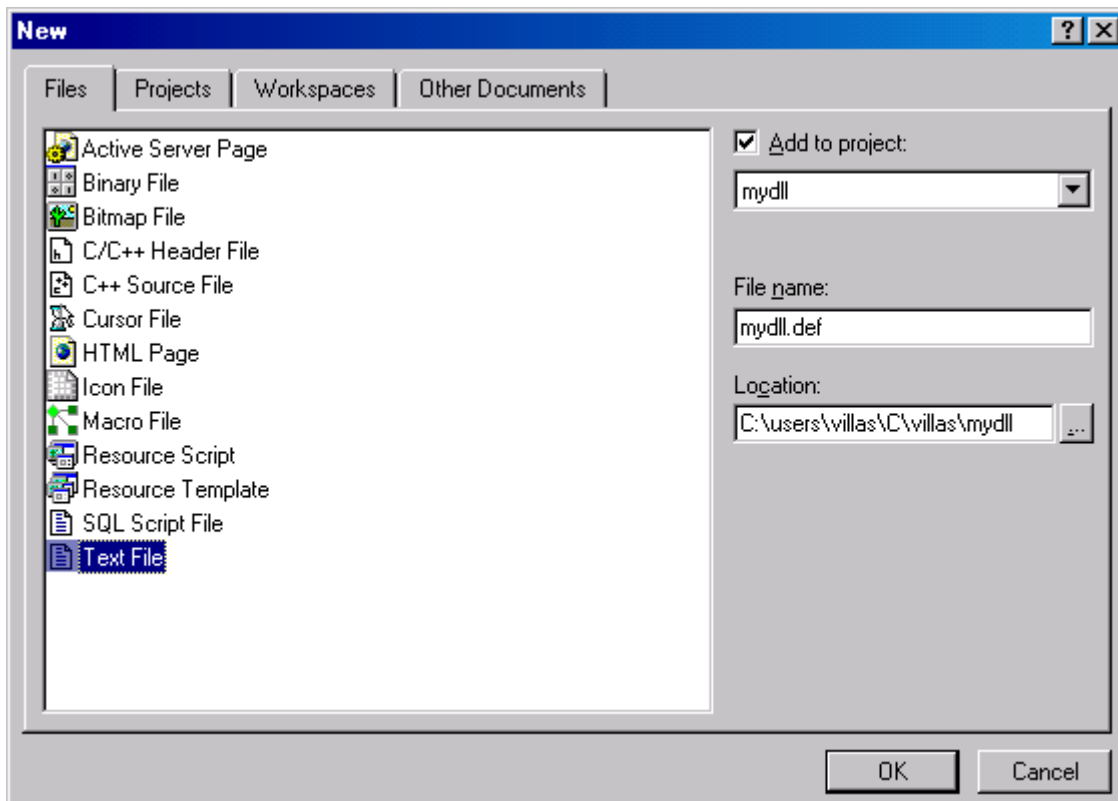
```

////////////////////////////////////
// mydll.h
#if !defined(_MYDLL_H_)
#define _MYDLL_H_
#include <afxwin.h>
// place the prototypes of dll functions
void WINAPI dllTest();
int WINAPI plus_1(int i);
int WINAPI plus_2(int i);
#endif // _MYDLL_H_

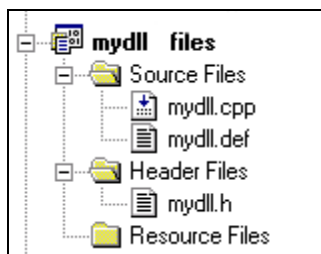
```

Além do cpp e do header, é preciso que exista um arquivo texto de definição. No caso esse arquivo é mydll.def. O modelo abaixo mostra como fazer esse arquivo. Além do nome e descrição, é preciso que se faça uma lista dos identificadores a serem exportados (no caso o nome das 3 funções globais). Para

criar o arquivo def, basta escolher no menu “File-New” e no tab “files” escolher o nome com terminação *.def.



O arquivo mydll.def aparece no projeto, como se fosse um arquivo *.cpp.



```

////////////////////////////////////
; mydll.def : Declares the module parameters for the DLL.
LIBRARY      "mydll"
DESCRIPTION  'MyDll Windows Dynamic Link Library'
EXPORTS
; Explicit exports can go here
dllTest @1
    plus_1 @2
    plus_2 @3

```

Para usar a dll, é preciso fazer a seguinte sequência de operações. Em primeiro lugar é preciso carregar a dll. Isso é feito com a função `LoadLibrary`. Essa operação pode falhar (como a abertura de um arquivo). Portanto, um programa bem feito deverá tratar a possibilidade de o carregamento da dll falhar.

O programa deve ter variáveis tipo “ponteiro para função”, para receber as funções da dll. Para usar as funções, a segunda coisa a fazer é ligar os ponteiros para função locais com as funções na dll. Isso é feito com a função `GetProcAddress`. Novamente, essa operação pode falhar e um programa bem feito

deve tratar a possibilidade de falha. Repare que o retorno da função `GetProcAddress` deve ser maquiado (*type cast*) para ser compatível com a variável local tipo “ponteiro para função”.

Com os ponteiros para função carregados, basta usa-los como se fossem funções do próprio programa. Note que como o carregamento da dll é feito em tempo de execução, seria bastante simples que alguma lógica escolhesse carregar uma ou outra dll equivalente. Por exemplo: a dll poderia conter mensagens ou recursos, em que as frases seriam escritas em um determinado idioma. Assim um software poderia ser internacionalizado apenas por escrever uma nova dll (e fazer o programa principal encontrar essa dll, o que seria feito na instalação do programa).

Para conseguir ligar (linkar) o `usedll.cpp`, é preciso acrescentar a biblioteca “`nafxcwd.lib`”. Veja a seção 2.1.7.1, na página 49. No caso desse exemplo, foi feito um novo projeto, tipo “win32 console application”, chamado `usedll`. Portanto, considerando que as saídas dos projetos ficam no diretório “Debug”, ou “Release”, a forma de apontar a dll a partir do projeto `usedll` é a string “`..\..\mydll\Debug\mydll.dll`”. Uma dll pode ser executada sob debug tanto quanto um programa normal, desde que tenha sido compilada para debug. Também como um programa ou biblioteca normal, caso não haja interesse de debug (e.g. entregando o arquivo para um cliente), é recomendável compilar como Release.

```

////////////////////////////////////
// usedll.cpp (mydll.dll)
// use_dll.cpp
#include <iostream.h>
#include <afxwin.h>
void main() {
    HINSTANCE hDll; // dll handler
    hDll = LoadLibrary("../..\mydll\Debug\mydll.dll"); // load dll

    // handle error while loading dll handler
    if (!hDll) {
        cout << "Error while loading DLL handler" << endl;
        exit(1);
    }

    // function pointers equivalent to dll functions
    void (WINAPI * c_dllTest)();
    int (WINAPI *c_plus_1)(int);
    int (WINAPI *c_plus_2)(int);

    // link function pointers to dll functions
    c_dllTest = (void (WINAPI *)())GetProcAddress(hDll, "dllTest");
    c_plus_1 = (int (WINAPI *)())GetProcAddress(hDll, "plus_1");
    c_plus_2 = (int (WINAPI *)())GetProcAddress(hDll, "plus_2");

    // handle error while loading functions
    if(!c_dllTest || !c_plus_1 || !c_plus_2) {
        cout << "Error while loading functions from DLL" << endl;
        exit(1);
    }

    // now, use the functions freely
    c_dllTest();
    cout << "3+1=" << c_plus_1(3) << endl;
    cout << "5+2=" << c_plus_2(5) << endl;
}

```

Saída do programa:

```

You're inside a Dll
3+1=4
5+2=7

```

2.1.7.4 DLL para Windows

Tipos de dll com Visual C para Windows - Projeto “MFC AppWizard (dll)”.

1. Regular DLL with MFC static linked
2. Regular DLL using shared MFC DLL
3. MFC Extension DLL (using shared MFC DLL)

A dll tipo “regular” (caso 1. e 2.) pode em princípio ser usada em outra linguagem que não Visual C++ (como Delphi). A dll tipo “extension” somente pode ser usada para programas com Visual C++.

2.1.7.5 Otimização do linker para alinhamento de código

Na versão 6.0 do Visual C++, há uma opção de otimização do linker que é “/OPT:WIN98”, que é definida como verdadeira implicitamente. Com essa definição, o alinhamento do código é feito com 4K de tamanho (sem essa opção ativa, o alinhamento é de 512 bytes). O alinhamento em 4K torna o código um pouco maior. Pode-se comandar esse flag diretamente pelo código usando #pragma. No exemplo abaixo, o flag é resetado (desativado). Esse código deve ser acrescentado no início do código fonte em questão.

```
#pragma comment(linker, “/OPT:NOWIN98”)
```

2.1.8 Detectando vazamento de memória

Para detectar vazamento de memória, é preciso duas funcionalidades:

1. Salvar a condição da memória alocada
2. Checar a condição corrente da memória alocada. Se for igual a que está salva, conclui-se que entre o momento que se salvou e o momento que se checkou não houve vazamento de memória. Ao contrário, se não for igual, conclui-se que houve vazamento de memória.

Uma forma prática de lidar com o problema de vazamento de memória é escrever uma classe para essa finalidade. Abaixo está escrita essa classe específica para o compilador Visual C++ 6.0. A classe `vb_memCheck` possui apenas 2 métodos. O construtor salva a condição de memória alocada. O outro método é `check`, que checa o vazamento e manda para o console informação de vazamento de memória caso haja algum. Se não houver vazamento, o método `check` não faz nada.

```
#include <crtdbg.h> // include necessary to handle memory leak debug (win32 only)

// use this class for Visual C++ 6.0 only !!
class VBMemCheck {
    _CrtMemState s1;
public:
    VBMemCheck() { // store memory on constructor
        // Send all reports to STDOUT
        _CrtSetReportMode( _CRT_WARN, _CRTDBG_MODE_FILE );
        _CrtSetReportFile( _CRT_WARN, _CRTDBG_FILE_STDOUT );
        _CrtSetReportMode( _CRT_ERROR, _CRTDBG_MODE_FILE );
        _CrtSetReportFile( _CRT_ERROR, _CRTDBG_FILE_STDOUT );
        _CrtSetReportMode( _CRT_ASSERT, _CRTDBG_MODE_FILE );
        _CrtSetReportFile( _CRT_ASSERT, _CRTDBG_FILE_STDOUT );

        // Store memory checkpoint in s1 (global variable) memory-state structure
        _CrtMemCheckpoint( &s1 );
    };

    void check() {
        _CrtDumpMemoryLeaks();
    }
};
```

```
};  
};
```

No exemplo de utilização abaixo, não há vazamento de memória.

```
void main () {  
    cout << "hello" << endl;  
  
    VBMemCheck c;    // construtor salva condição da memória  
  
    int *p_i=new int; // código que pode ter vazamento de memória  
    delete p_i;  
  
    c.check(); // manda aviso para console caso haja vazamento  
}
```

Nesse caso, o resultado é:

```
hello
```

Caso haja vazamento de memória, como no caso abaixo,

```
void main () {  
    cout << "hello" << endl;  
  
    VBMemCheck c;    // construtor salva condição da memória  
  
    int *p_i=new int; // código que pode ter vazamento de memória  
  
    c.check(); // manda aviso para console caso haja vazamento  
}
```

O resultado é:

```
hello  
Detected memory leaks!  
Dumping objects ->  
{20} normal block at 0x00780DA0, 4 bytes long.  
Data: <    > CD CD CD CD  
Object dump complete.
```

2.2 Borland C++ builder 5.0

Essa versão do compilador C++ da Borland foi lançada no ano de 2000. Trata-se de um produto com muitas funcionalidades. A mesma Borland possui também um produto chamado Delphi, que é uma variante de Pascal, e é considerado um RAD (*Rapid Application Environment*, ou ambiente de [desenvolvimento] rápido de aplicação) muito aceito no mercado. O Borland builder é uma tentativa de refazer o ambiente RAD do Delphi em C++. As semelhanças entre o ambiente gráfico do Delphi e do builder são óbvias.

Com respeito a programação para Windows, a Borland já recomendou o uso da biblioteca OWL. Posteriormente, já há algum tempo, vem recomendando o uso de outra biblioteca chamada VCL. Tanto OWL quanto VCL são de autoria da Borland. É interessante lembrar que a Microsoft recomenda o uso da biblioteca MFC, de sua própria autoria. Não é viável na prática desenvolver programas para Windows em C++ sem uma biblioteca de apoio. A decisão de qual biblioteca usar é estratégica, pois trata-se de algo que requer tempo para se aprender. Por várias razões, principalmente por razões estratégicas e de mercado, eu recomendo o uso de MFC (que pode ser usado tanto pelo compilador da Borland quanto pelo compilador da Microsoft).

Nessa seção, fala-se de como usar o Borland C++ builder para programas DOS.

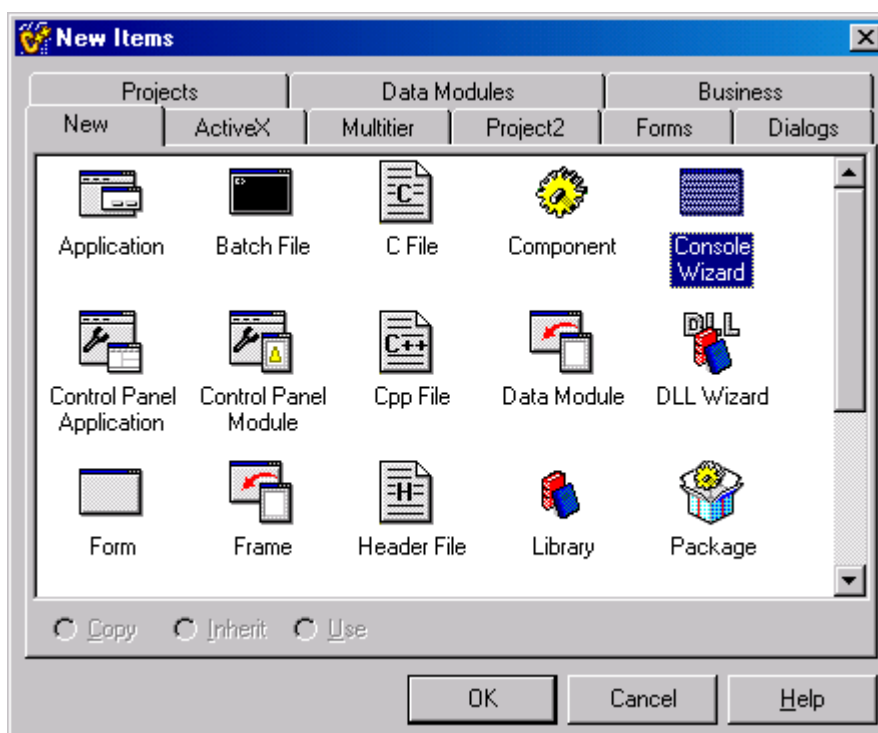
2.2.1 Reconhecendo o Compilador

O builder 5 é um programa que não usa a estrutura (*framework*) padrão de SDI nem de MDI. O programa em si é uma pequena janela apenas (na forma padrão) com barra de título (*title bar*), menu, e duas barras de ferramentas (*toolbars*) (veja figura abaixo). Pode-se editar mais de um arquivo ao mesmo tempo, por isso o programa é de certa forma parecido com a estrutura MDI. Geralmente, ao abrir-se o builder 5, automaticamente outras janelas de edição são automaticamente inicializadas. Mas nesse tutorial se está partindo do programa em si (considerando que os arquivos que foram abertos automaticamente foram fechados).

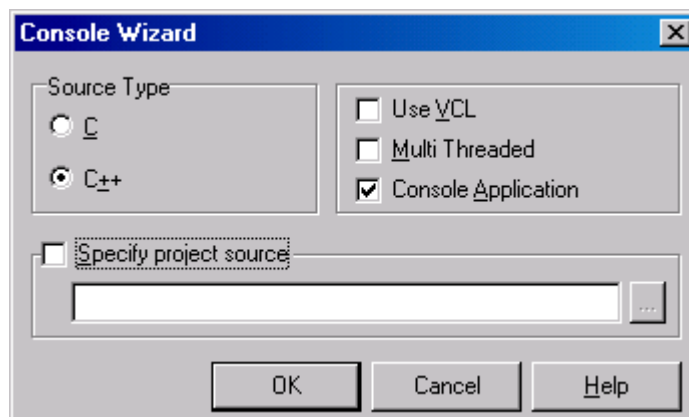


2.2.2 “Hello world” para DOS

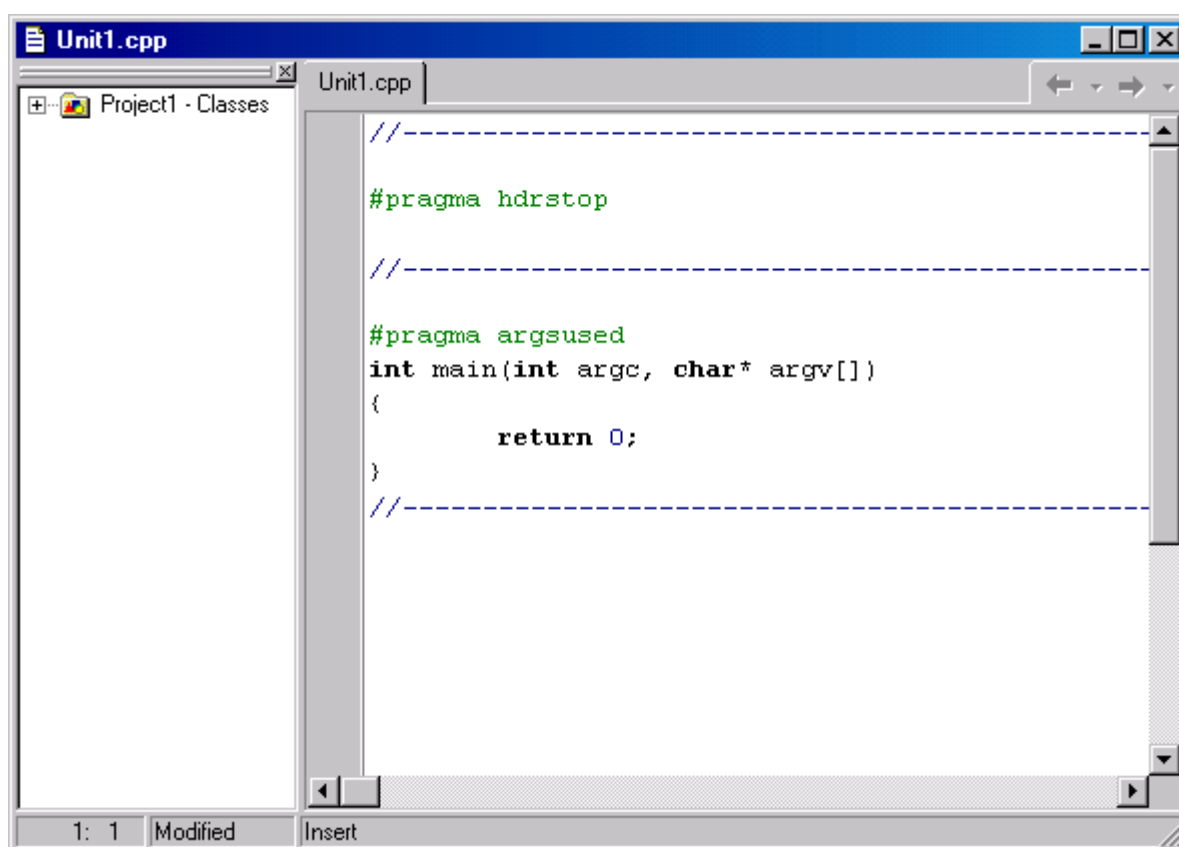
Para fazer um programa para console com builder 5, escolha no menu File-New. Na caixa de diálogo “New Items”, escolha “console Wizard”.



Na caixa de diálogo “Console Wizard”, escolha as opções como mostrado na figura abaixo.



O builder então cria um programa de esqueleto, num arquivo chamado Unit1.cpp, como mostrado abaixo. Já existe um programa mínimo nesse arquivo fonte.



Apesar da recomendação da Borland, eu recomendo que se apague o conteúdo do arquivo e substitua-o pelo texto abaixo.

```
#include <iostream.h>
void main() {
    cout << "Hello, world" << endl;
}
```

A resposta do programa é como mostrado abaixo.

Hello, world

Mas como o programa acaba, haverá se poderá perceber uma janela piscando rapidamente na tela. Para que a janela fique esperando uma tecla ser apertada, mude o programa para o texto abaixo. Nessa versão, foi acrescentada a linha “cout << “Press any key”; getch;” no final, e também

“`#include <conio.h>`” no início. Assim, o programa ficará sempre esperando uma tecla para terminar (como o Visual C++).

```
#include <conio.h>
#include <iostream.h>
void main() {
    cout << "Hello, world" << endl;
    cout << "Press any key"; getch();
}
```

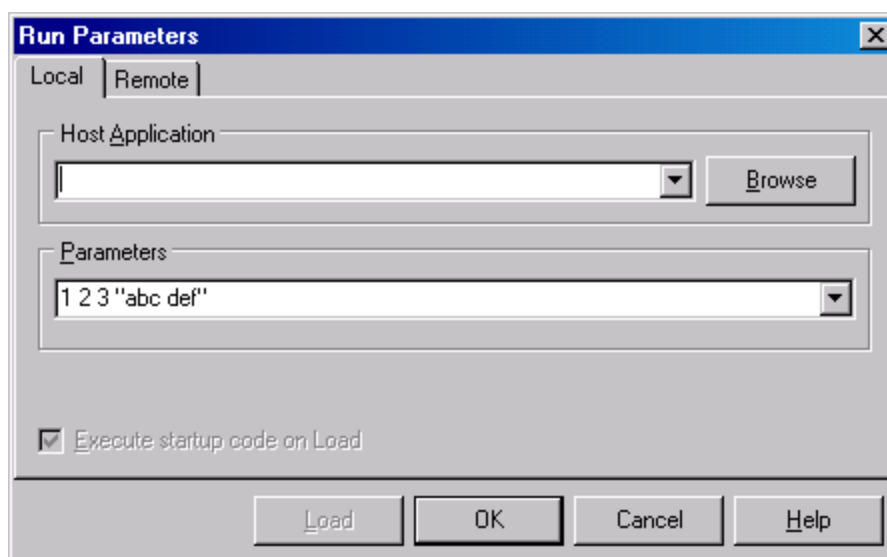
A resposta do programa é como mostrado abaixo.

```
Hello, world
Press any key
```

Nos exemplos que se seguem para builder 5, a linha “`cout << "Press any key"; getch();`” não será mais mencionada, ficando o seu uso a critério do leitor.

2.2.2.1 Adicionando argumentos para a linha de comando

No menu, selecione Run-Parameters. Surge a caixa de diálogo “Run Parameters” como mostrado abaixo.



No caso dos parâmetros mostrados acima, rodando-se o programa abaixo,

```
#include <iostream.h>
void main(int argc, char**argv) {
    cout << "Hello, world\n";
    int i;
    for (i=0; i<argc; i++)
        cout << "Arg[" << i << "]: " << argv[i] << endl;
}
```

obtem-se o resultado como mostrado abaixo.

```
Hello, world
Arg[0]:C:\Program Files\Borland\CBuilder5\Projects\Project1.exe
Arg[1]:1
Arg[2]:2
Arg[3]:3
Arg[4]:abc def
```

2.3 C++ for win32 gratuito

2.3.1 Ming (da GNU)

O compilador Ming é gratuito, baseado no compilador da GNU e funciona em DOS (também em Windows). A página web desse compilador é mostrada abaixo.

<http://www.xraylith.wisc.edu/~khan/software/gnu-win32/>

2.3.2 djgpp

A página do djgpp é www.delorie.com/djgpp/. Nessa página, há todas as informações necessárias para copiar, instalar e usar esse compilador. Abaixo há um resumo das informações de instalação.

O uso do djgpp é muito parecido com o do g++ do unix.

Pegue num dos servidores ftp listados, no diretório v2gnu, os arquivos abaixo (pequenas variações nas versões dos arquivos em princípio são aceitas). As descrições do significado desses arquivos estão disponíveis na página do djgpp.

- bnu2951b.zip
- djdev203.zip
- gcc2953b.zip
- gpp2953b.zip
- txi40b.zip

Descompacte todos esses arquivos para um diretório de instalação. Digamos que esse diretório de trabalho. Digamos que o nome desse diretório seja c:\winap\djgpp. Para simplificar, se quiser, crie um arquivo chamado djgpp.bat com o conteúdo abaixo.

```
@echo off
```

```
set PATH=C:\winap\DJGPP\BIN;%PATH%
```

```
set DJGPP=c:/winap/djgpp/djgpp.env
```

```
echo DJGPP enviromnent installed (SBVB)
```

Um problema dessa versão de compilador é que muitas bibliotecas são acrescentadas no programa executável final. Com isso mesmo um programa pequeno (tipo “hello world” fera um executável grande (maior que 300K).

2.4 g++ (do unix)

O compilador geralmente é chamado de cc, ou de gcc ou de g++. Verifique como é no seu sistema. A menos que se comande de forma diferente, o arquivo executável resultante é a.out. Lembre-se que em unix qualquer arquivo pode ser executável e não apenas os que tem extensão *.exe ou *.com.

Para ver a versão do seu compilador, digite como mostrado abaixo. Uma das últimas versões disponíveis é 2.95.

```
g++ -v
```

Para compilar e linkar

```
g++ main.cpp // programa com 1 só arquivo fonte
```

```
g++ main.cpp file1.cpp file2.cpp // programa com 3 arquivos fonte
```

Caso se modifique apenas o main.cpp, pode-se compilar apenas esse arquivo e linkar com os demais com

```
g++ main.cpp file1.o file2.o
```

Ensina-se o compilador a procurar arquivos include em mais um diretório com a opção “-I<diretório>”. No Exemplo abaixo acrescenta-se o diretório ./include na busca para includes.

```
G++ -I./include main.cpp file1.cpp file2.cpp
```

2.4.1 “Hello world”

Crie o arquivo de texto f1.cpp (veja na página 42). Para criar esse arquivo de texto, use um editor de texto, como o vi ou emacs. Compile-o com cc f1.cpp. Execute-o com o comando a.out, que é o arquivo executável final (o nome a.out é padrão, mas nada impede que seja mudado posteriormente).

2.4.1.1 Adicionando argumentos para a linha de comando

Chamar um programa passando argumentos pela linha de comando faz particularmente muito sentido no caso de compiladores que tem interface tipo linha de comando, como é o caso do g++ do unix (que também tem versão para Windows). Ao construir-se um programa, surge um arquivo executável com um nome, digamos “myprog”. Suponha que na linha de comando escrever-mos como abaixo.

```
myprog 1 2 3 "abc def"
```

Suponha também que o programa myprog seja como abaixo

```
#include <iostream.h>
void main(int argc, char**argv) {
    cout << "Hello, world\n";
    int i;
    for (i=0; i<argc; i++)
        cout << "Arg[" << i << "]: " << argv[i] << endl;
}
```

Nesse caso, a saída do programa será como abaixo.

```
Hello, world
Arg[0]:myprog
Arg[1]:1
Arg[2]:2
Arg[3]:3
Arg[4]:abc def
```

2.4.2 Usando o Help

O help do g++ em unix é em grande parte baseado no comando man (manual), que adicionalmente dá ajuda em todo o sistema operacional unix. Por exemplo: para saber ajuda sobre a função c chamada “printf”, na linha de comando digite como abaixo.

```
man printf
```

2.4.3 Projetos (programas com múltiplos fontes)

Para construir programas com g++ a partir de múltiplos fontes, basta adicionar todos os fontes que pertencem ao projeto na linha de comando. Por exemplo: caso um projeto seja composto pelos arquivos f1.cpp, f2.cpp e f3.cpp (sendo que um desses arquivos contém a função main), construa o programa digitando como abaixo na linha de comando.

```
g++ f1.cpp f2.cpp f3.cpp
```

Caso se esteja depurando o f1.cpp, enquanto se mantém constante o conteúdo de f2.cpp e f3.cpp, pode-se compilar apenas o f1.cpp e ligar com f2.o (versão compilada de f2.cpp) e f3.o (idem f3.cpp). Para isso, compile apenas o f2.cpp e f3.cpp com o comando como na linha abaixo.

```
g++ -c f2.cpp f3.cpp
```

Com esse comando, a opção “-c” (compile only) irá fazer gerar o f2.o e f3.o. Para construir o programa executável final a partir da compilação de f1.cpp com f2.o e f3.o, basta comandar como mostrado na linha abaixo.

```
g++ f1.cpp f2.o f3.o
```

Caso exista uma biblioteca estática que se deseje usar num projeto, simplesmente acrescentando-a à linha de comando se está incluindo a biblioteca no projeto. No exemplo abaixo, uma biblioteca chamada “libsomething.a” é acrescentada ao projeto.

```
g++ f1.cpp libsomething.a
```

2.4.4 Bibliotecas

Em unix, as bibliotecas estáticas têm extensão .a e as dinâmicas tem extensão .so (como referência, em DOS/Windows, geralmente a extensão das bibliotecas estáticas é .lib, e das dinâmicas é .dll). No caso de unix, é comum que as bibliotecas dinâmicas coloquem a versão após o .so.

```
unix_static_lib.a
unix_dynamic_lib.so
```

```
dos_static_lib.lib
dos_dynamic_lib.dll
```

O utilitário para criar e manter bibliotecas é o ar. Para referência completa do ar, comande man ar.

Para facilitar a compreensão, heis uma sugestão de convenção de nomes para bibliotecas em unix. Pense num nome que identifique a biblioteca. A sua biblioteca deve se chamar lib_nome.a.

2.4.4.1 Incluir uma biblioteca num projeto

Na linha abaixo, compila-se o fonte m1.cpp, que tem chamadas a biblioteca lib_teste.a. O arquivo de include usado dentro de m1.cpp está no diretório corrente (.), portanto esse diretório foi incluído na busca de includes.

```
g++ -I. m1.cpp libmylib.a
```

2.4.4.2 Fazer uma biblioteca

Para se criar a biblioteca libmylib.a contento os arquivos myfile1.o, myfile2.o, etc usa-se o comando abaixo:

```
ar -q libmylib.a myfile1.o myfile2.o ...
```

Exemplo: Para se criar a biblioteca lib_teste.a com 2 arquivos f1.cpp e f2.cpp

```
g++ -c f1.cpp f2.cpp // compila os fontes, criando f1.o e f2.o
ar -q libmylib.a f1.o f2.o // cria library "libmylib.a" contendo f1.o e f2.o
```

2.4.4.3 Examinar uma biblioteca

Para verificar o conteúdo de uma biblioteca estática, use o comando abaixo. Tente esse comando com uma biblioteca que já esteja pronta do seu sistema. Provavelmente há várias no diretório /usr/lib.

```
ar -t unix_static_lib.a

// exemplo
$ ar -t /usr/lib/libz.a
__SYMDEF
uncompr.o
gzio.o
crc32.o
compress.o
inflate.o
```

```
deflate.o
adler32.o
inffblock.o
zutil.o
trees.o
infcodes.o
inftrees.o
inffast.o
infutil.o
```

2.4.5 Fazendo uma biblioteca usando libtool

Libtool é uma ferramenta (gratuita) da gnu para bibliotecas portáteis. Essa ferramenta substitui com vantagens o tradicional ar para gerenciamento de bibliotecas. Veja a referência original sobre libtool em [2]. O Libtool permite linkar e carregar bibliotecas estáticas e também compartilhadas (*shared*), isto é, bibliotecas dinâmicas (o que em Windows corresponde a *dll*). Por usar o libtool para gerenciar as bibliotecas do seu projeto, há que se preocupar apenas com a interface do libtool. Se o projeto faz uso de uma biblioteca com estrutura diferente, o libtool cuida de chamar o compilador e linker para acertar a estrutura da biblioteca. Com isso, a biblioteca binária será instalada de acordo com as convenções da plataforma em questão.

Para usar o libtool, é preciso que essa ferramenta esteja instalada no seu sistema. Caso não esteja, peque-a em [2] e instale-a.

Enquanto uma biblioteca estática é criada a partir de código objeto (*.o), uma biblioteca dinâmica deve ser criada a partir de código independente de posição (*position-independent code* [PIC]). A nível de arquivo, a informação PIC consiste do código objeto normal (*.o), e de um arquivo adicional tipo *library object* (*.lo). Por exemplo: seja os arquivos fonte `plus_1_file.cpp` e `plus_2_file.cpp` abaixo.

```
// plus_1_file.cpp
int plus_1(int i) {
    return i+1;
}
```

```
// plus_2_file.cpp
int plus_2(int i) {
    return i+2;
}
```

Para esses arquivos, pode-se gerar o código independente de posição (`plus_1_file.o`, `plus_1_file.lo`, `plus_2_file.o` e `plus_2_file.lo`) com o comando abaixo. No exemplo abaixo, usa-se libtool e g++ para gerar os arquivos objeto e independente de posição para `plus_1_file.cpp` e `plus_2_file.cpp`.

```
libtool g++ -c plus_1_file.cpp
libtool g++ -c plus_2_file.cpp
```

Uma biblioteca dinâmica precisa de um diretório para se instalada. Um diretório comum para se instalar uma biblioteca dinâmica é `/usr/local/lib`. No exemplo abaixo, 2 arquivos (`plus_1_file` e `plus_2_file`) compõe a biblioteca “`libplus_n.la`”. A biblioteca é construída no diretório corrente.

```
libtool g++ -o libplus_n.la plus_1_file.lo plus_2_file.lo -rpath /usr/local/lib
```

Seja um programa de teste da biblioteca, como mostrado abaixo.

```
// use_plus_n.cpp
#include <iostream.h>
int plus_1(int);
int plus_2(int);
void main () {
    int k = plus_1(4);
    cout << "k=" << k << endl;
    k = plus_2(3);
    cout << "k=" << k << endl;
}
```

```
}

```

A saída do programa deve ser como mostrado abaixo. Esse programa usa 2 funções globais que estão definidas na biblioteca. Portanto o problema é conseguir linkar o programa.

```
k=5
k=5

```

Para usar a biblioteca diretamente (no caso de a biblioteca não estar instalada no sistema), basta compilar o programa que usa a biblioteca e posteriormente linkar com a biblioteca.

```
g++ -c use_plus_n.cpp
libtool g++ -o test use_plus_n.cpp libplus_n.la

```

Se a biblioteca estiver instalada, digamos no diretório /usr/local/lib, basta indicar o local onde a biblioteca está instalada com a opção -L, como mostrado no exemplo abaixo.

```
g++ -c use_plus_n.cpp
libtool g++ -L/usr/local/lib -o test use_plus_n.o libplus_n.la

```

Nos dois exemplos acima, é gerado o arquivo executável de saída “test”.

2.4.5.1 Instalando uma biblioteca dinâmica

Para se instalar uma biblioteca dinâmica, é preciso usar o aplicativo install, que geralmente encontra-se em /usr/bin/install. Seguindo o exemplo anterior, para que se instale uma biblioteca no sistema, no diretório /usr/local/lib, a partir de um arquivo libplus_n.la já criado, executa-se o comando abaixo.

```
libtool --mode=install /usr/bin/install -c libplus_n.la /usr/local/lib/libplus_n.la

```

Com esse comando, o arquivo libplus_n.la é copiado para /usr/local/lib, e são criados os arquivos libplus_n.so, libplus_n.so.0 e libplus_n.so.0.0.0 também no diretório /usr/local/lib. Isso significa que a versão 0.0.0 da biblioteca está instalada.

Para se gerenciar a versão de uma biblioteca instalada, é preciso em primeiro lugar ser capaz de definir a versão da biblioteca que se pretende instalar.

Exemplo para instalar a vllib. Primeiramente vá para um diretório de trabalho, onde existam os arquivos vllib.cpp e vllib.h (esses arquivos podem ser baixados de www.vbmcgi.org/vllib). Em primeiro lugar, compile todos os fontes (no caso há somente um), com a linha abaixo.

```
libtool g++ -c vllib.cpp

```

O resultado deve ser parecido com o mostrado abaixo.

```
rm -f .libs/vllib.lo
g++ -c vllib.cpp -fPIC -DPIC -o .libs/vllib.lo
g++ -c vllib.cpp -o vllib.o >/dev/null 2>&1
mv -f .libs/vllib.lo vllib.lo

```

Em seguida, crie o arquivo libvllib.la com o comando abaixo

```
libtool g++ -rpath /usr/local/lib -o libvllib.la vllib.lo

```

O resultado deve ser parecido com o mostrado abaixo.

```
rm -fr .libs/libvllib.la .libs/libvllib.* .libs/libvllib.*
gcc -shared vllib.lo -lc -Wl,-soname -Wl,libvllib.so.0 -o .libs/libvllib.so.0.0.0
(cd .libs && rm -f libvllib.so.0 && ln -s libvllib.so.0.0.0 libvllib.so.0)
(cd .libs && rm -f libvllib.so && ln -s libvllib.so.0.0.0 libvllib.so)
ar cru .libs/libvllib.a vllib.o
ranlib .libs/libvllib.a
creating libvllib.la
(cd .libs && rm -f libvllib.la && ln -s ../libvllib.la libvllib.la)

```

Por último, instale a biblioteca dinâmica no diretório adequado (do sistema), no caso /usr/local/lib. Para isso, use o comando abaixo.

```
libtool --mode=install /usr/bin/install -c libvllib.la /usr/local/lib/libvllib.la

```

O resultado deve ser parecido com o mostrado abaixo.

```
/usr/bin/install -c .libs/libvblib.so.0.0.0 /usr/local/lib/libvblib.so.0.0.0
(cd /usr/local/lib && rm -f libvblib.so.0 && ln -s libvblib.so.0.0.0 libvblib.so.0)
(cd /usr/local/lib && rm -f libvblib.so && ln -s libvblib.so.0.0.0 libvblib.so)
/usr/bin/install -c .libs/libvblib.lai /usr/local/lib/libvblib.la
/usr/bin/install -c .libs/libvblib.a /usr/local/lib/libvblib.a
ranlib /usr/local/lib/libvblib.a
chmod 644 /usr/local/lib/libvblib.a
PATH="$PATH:/sbin" ldconfig -n /usr/local/lib
ldconfig: warning: /usr/local/lib/libdb_cxx-3.1.so is not a symlink
ldconfig: warning: /usr/local/lib/libdb-3.1.so is not a symlink
-----
Libraries have been installed in:
  /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use '-LLIBDIR'
flag during linking and do at least one of the following:
- add LIBDIR to the 'LD_LIBRARY_PATH' environment variable
  during execution
- add LIBDIR to the 'LD_RUN_PATH' environment variable
  during linking
- use the '-Wl,--rpath -Wl,LIBDIR' linker flag
- have your system administrator add LIBDIR to '/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
-----
```

// ainda há o que melhorar !

// Exemplo para usar a plus_n após ter sido instalada:

// Usando a biblioteca sem fazer a sua instalação no sistema.

2.4.6 Debug

Existe uma ferramenta de debug gráfica para unix e sistema GUI tipo X. Mas na prática, eu acabo usando o debug de outro sistema (um bom compilador em Windows) e passo para unix os componentes de programação já testados. Essa é uma estratégia.

Quando é fundamental depurar programas no próprio unix, pode-se usar uma série de artifícios para depurar um programa sem o uso de ferramentas de debug. Por exemplo, pode-se mandar exteriorizar variáveis no meio do programa para verificar o seu conteúdo. Diretivas de compilação podem criar versões de “debug” e “release” do sistema em desenvolvimento, sendo que a versão debug é igual a versão release, a menos do acréscimo de algumas linhas de exteriorização de valores de variáveis.

2.4.7 Definindo um identificador para compilação condicional

A compilação condicional é a compilação de trechos de código fonte a partir de uma condição.

```
// vb100.cpp
// in unix compile it with the comand line below
// g++ -DGCC vb100.cpp
#include <iostream.h>
void main () {
    cout << "hello" << endl;
#ifdef WIN32
    cout << "I am compiling for WIN32" << endl;
#endif
}
```

```
#endif
#ifdef GCC
    cout << "I am using g++ compiler (UNIX)" << endl
        << "and I defined the \"GCC\" constant in the compiler line command " << endl;
#endif
}
```

2.4.8 O pacote RPM do linux

Originalmente, o unix exige uma quantidade de conhecimentos considerável para a instalação de um programa. Geralmente um arquivo compactado é fornecido. Ao ser aberto em um diretório, há que se ler um as instruções num arquivo geralmente nomeado como INSTALL. Essas instruções são algo como executar “./config”, e depois “make install”. Mas na prática, não é incomum que se exija vários parâmetros de configuração a mais na hora da instalação.

O sistema operacional linux possui uma forma de instalação de programas que é bastante mais fácil que a forma tradicional. Essa forma é conhecida como rpm (uma extensão de arquivo). O significado de rpm é Redhat package manager (Redhat é uma distribuição famosa de linux). Procure “rpm” no site do Redhat, ou vá direto para a página deles ensinando a usar rpm, em [1]. O resumo de rpm é mostrado nessa seção.

O pacote rpm é compatível com diversos variantes de unix, e não apenas para linux.

Dentre os serviços que o rpm oferece incluem-se

- Instalar ou dar upgrade do software, verificando dependências.
- Enquanto instala o software, verificar se o executável fica pronto para usar.
- Recuperar arquivos que por ventura tenham sido acidentalmente apagados de uma instalação prévia.
- Informar se o pacote já está instalado.
- Descobrir a qual pacote corresponde um determinado arquivo.

2.4.8.1 rpm binário e rpm com fonte

Há dois tipos de arquivos rpm. Um deles é binário (pré-compilado), e o outro vem com os fontes, e instala-se a partir da compilação (forma mais lenta de instalar, contudo mais segura e mais portátil).

A designação do rpm binário é *.rpm. A designação do rpm com fonte é *.src.rpm.

2.4.8.2 Alguns comandos do rpm

Veja todos os comandos do rpm rodando simplesmente

rpm

- Instalando um pacote rpm:

```
rpm -i <package>.rpm
```

- Desinstalando um pacote rpm:

```
rpm -e <package_name>
```

- Instalando um rpm diretamente da rede (sem haver cópia local).

```
rpm -i ftp://<site>/<directory>/<package>.rpm
```

Exemplo:

```
rpm -i ftp://ftp.redhat.com/pub/redhat/rh-2.0-beta/RPMS/foobar-1.0-1.i386.rpm
```

- Verificando o sistema (conferindo o que está instalado)

```
rpm -Va
```

- Digamos que você encontra um arquivo e quer saber de que pacote rpm ele pertence. No exemplo abaixo, o arquivo “/home/httpd/icons/uuencoded.gif” está sendo.

```
rpm -qf /home/httpd/icons/uuencoded.gif
```

A saída deve ser como abaixo

```
apache-1.3.12-2
```

- Pegando informações de um rpm sem instala-lo

```
rpm -qpi <package>.rpm
```

- Verificando quais

```
rpm -qpl <package>.rpm
```

2.4.8.3 Construindo um rpm

2.4.8.3.1 Introdução

Construir um pacote rpm é relativamente fácil de se fazer, especialmente se você tem o fonte do software para o qual quer fazer o pacote. O procedimento passo a passo é mostrado abaixo.

1. Prepare o código fonte do pacote, deixando-o pronto para compilar.
2. Faça um arquivo de especificação (*spec file*) para o pacote.
3. Execute o comando de construir (*build*) o pacote rpm.

O arquivo de especificação é um arquivo texto requerido para se construir um pacote rpm. Nesse arquivo texto há uma descrição do pacote, instruções para permitir contruí-lo, e uma lista de arquivos binários que são instalados com o pacote. O melhor é seguir a convenção do arquivo de especificação, como mostrado no exemplo abaixo. Com essa convenção, o sistema poderá lidar com múltiplas versões de um pacote com o mesmo nome. Recomendo que a descrição do software, nome, etc. seja feita no idioma inglês.

```
Summary: A program that ejects removable media using software control.
Name: eject
Version: 2.0.2
Release: 3
Copyright: GPL
Group: System Environment/Base
Source: http://metalab.unc.edu/pub/Linux/utils/disk-management/eject-2.0.2.tar.gz
Patch: eject-2.0.2-buildroot.patch
BuildRoot: /var/tmp/%{name}-buildroot

%description
The eject program allows the user to eject removable media
(typically CD-ROMs, floppy disks or Iomega Jaz or Zip disks)
using software control. Eject can also control some multi-
disk CD changers and even some devices' auto-eject features.

Install eject if you'd like to eject removable media using
software control.

%prep
%setup -q
%patch -p1 -b .buildroot

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

%install
```

```

rm -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT/usr/bin
mkdir -p $RPM_BUILD_ROOT/usr/man/man1

install -s -m 755 eject $RPM_BUILD_ROOT/usr/bin/eject
install -m 644 eject.1 $RPM_BUILD_ROOT/usr/man/man1/eject.1
Prep
%clean
rm -rf $RPM_BUILD_PreROOT

%files
%defattr(-,root,root)
%doc README TODO COPYING ChangeLog

/usr/bin/eject
/usr/man/man1/eject.1

%changelog
* Sun Mar 21 1999 Cristian Gafton <gafton@redhat.com>
- auto rebuild in the new build environment (release 3)

* Wed Feb 24 1999 Preston Brown <pbrown@redhat.com>
- Injected new description and group.

```

2.4.8.3.2 O header

Explicação item a item:

- Summary: explicação em uma linha sobre o propósito do pacote
- Name: um identificador para o pacote
- Version: a versão do pacote
- Release: é como uma sub-versão
- Copyright: diga como quer a licença do pacote. Você deve usar algo como GPL, BSD, MIT, public domain, distributable ou commercial.
- Group: o grupo ao qual o pacote pertence num nível do instalador RedHat.
- Source: essa linha aponta para o diretório raiz do programa fonte. É usado para o caso de você precisar pegar o fonte novamente e checar por novas versões. É importante lembrar que o nome do arquivo dessa linha precisa ser exatamente igual ao que você tem de fato no sistema (ou seja, não mude o nome dos arquivos fonte dos quais fez download). Os arquivos fonte estão geralmente compactados com tar e depois com gzip, ou seja, tem a extensão *.tar.gz. Você pode especificar mais de um arquivo fonte, como mostrado no exemplo abaixo.

```

source0: mySource.tar.gz
source1: myOtherSource.tar.gz
source2: myYetOtherSource.tar.gz

```

Esses arquivos vão para o diretório SOURCES. (A estrutura de diretórios do pacote rpm é mostrada mais abaixo.)

- Patch: Esse é o local onde você encontra os arquivos que “concertam” uma versão já disponibilizada (esse procedimento é chamado de *patch*). Da mesma forma que para os arquivos fonte, o nome dos arquivos precisa ser exatamente igual ao original, e pode haver mais de um. O exemplo abaixo ilustra-o. Os arquivos de patch serão copiados para o diretório SOURCES.

```
Patch0: myPatch.tar.gz
Patch1: myOtherPatch.tar.gz
Patch2: myYetOtherPatch.tar.gz
```

- Group: Essa linha é usada para dizer para programas instaladores de alto nível (tal como gnorpm) onde salvar o programa do pacote e sua estrutura hierárquica. Você pode encontrar ...
- BuildRoot: É uma linha que permite especificar um diretório como “root” para construir e instalar o pacote. Você pode usar essa opção para ajudar a testar seu pacote antes de realmente instalar na máquina.
- %description: É onde se escreve a descrição do pacote. Recomenda-se enfaticamente o uso do idioma inglês para a descrição. Nesse campo pode-se usar várias linhas, como mostrado no exemplo.

2.4.8.3.3 Preparação (prep)

É a segunda seção do arquivo de especificação (*spec file*). É usado para obter parâmetros para construir o pacote. O que a seção prep faz realmente é executar scripts no shell. Você pode simplesmente fazer um script seu para construir o pacote, e colocar após o tag %prep. Contudo, há alguns macros que podem ser úteis, e são mostrados abaixo.

- -n name. Esse macro define o nome do diretório para construir o pacote com o nome indicado. O nome implícito (*default*) é \$NAME\$VERSION. Outras possibilidades são \$NAME, ou \${NAME}\${VERSION}, ou o que for usado nos arquivos tar. Note que o uso da letra “\$” *não* são variáveis especificadas no spec file. Estão apenas usadas aqui no lugar de um nome de exemplo. Você precisa usar um nome real no seu pacote, e não uma variável.
- -c. Essa opção irá fazer mudar para o diretório de trabalho *antes* de executar a descompactação com untar.
- -b. Essa opção irá descompactar os fontes antes de mudar de diretório. É útil apenas quando se usa múltiplos arquivos fonte.
- -a. Essa opção irá descompactar os fontes após mudar para o diretório.
- -T. Essa opção sobre-escreve a ação implícita de descompactar com untar os fontes, e requer o uso da opção -a ou -b para obter os principais fontes descompactados. Você precisa disso quando são usados fontes secundários.
- -D. Use essa opção para *não* apagar o diretório após desempacota-lo.

// ainda há o que melhorar

Capítulo 3) Princípios de C/C++

3.1 O primeiro programa

```
#include <stdio.h>
void main() {
    printf("Hello world\n");
}
```

A saída do programa é:

```
Hello world
```

Basicamente, um compilador é uma ferramenta que transforma um arquivo de texto (conhecido como texto fonte) num arquivo executável. Chama-se esse procedimento “construir (*build*) um executável”. As seções abaixo abordam os conceitos básicos sobre como se pode escrever o arquivo de texto fonte.

3.2 Formato livre

Formato livre (*free format*) que significa que o compilador observa o texto fonte sem atenção a posicionamento relativo dos caracteres. Em FORTRAN, que por exemplo não possui formato livre, o código fonte só pode ser escrito a partir da coluna 7 do texto e o return no final da linha significa que o comando acabou. A maioria das linguagens modernas como C e C++ podem ser escritas em formato livre o que significa também que no lugar de 1 espaço podem haver qualquer número de espaços, returns e tabs sem qualquer alteração na compilação.

Com o formato livre, as funções `myinc` e `myinc2` abaixo são exatamente iguais.

```
int myinc(int in) { return in+1; }

int
myinc2
(
int
in
)
{
return
in
+
1
;
}
```

3.3 Chamada de função

A chamada de uma função corresponde à chamada de uma sub-rotina, como ilustrado na figura abaixo.

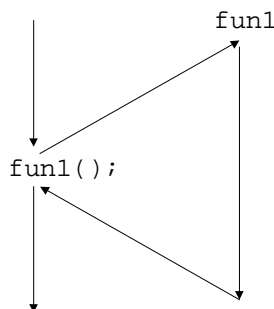


Figura 4: representação gráfica do fluxo de um programa na chamada de uma função

Para se chamar uma função em C, basta escrever o nome da função seguido de `()`. Caso haja parâmetros, eles devem ser colocados dentro dos parênteses, separados por vírgulas. Mesmo que não haja parâmetros é necessário que se escreva o abre-e-fecha parêntesis vazio `()`. Depois da chamada da função, como depois de qualquer comando, é necessário que se pontue com ponto-e-vírgula `;`.

Caso haja retorno da função, uma variável de retorno colocada antes da função e o uso do operador `=` fazem com que a variável receba o retorno da função. Não é necessário que o retorno de uma função seja usado. Caso não seja necessário usar o retorno de uma função, basta ignorar esse valor de retorno.

Exemplo:

```
double triple(double d) {
    return 3*double;
}
void main () {
    double d=2.2;
    double dd = triple(d);
    triple(d); // o retorno não é usado. Nesse caso não faz nada, mas é legal.
}
```

3.4 Declaração e definição de funções

É importante entender bem a diferença desses dois conceitos.

1. “Declarar uma função” significa dizer qual é o protótipo dessa função, isto é, quantos são os parâmetros e de que tipo, bem como o tipo de retorno.
2. “Definir uma função” significa escrever o conteúdo da função. Ao se definir uma função, se está declarando também. Mas não o contrário.

O compilador somente pode compilar uma função se ela tiver sido declarada antes de ser usada.

Declaração	Definição
<code>int plus_3(int);</code>	<code>int plus_3(int in) { return in+3; }</code>

3.5 Comentários

É muito útil que se possa inserir comentários num texto fonte. Em C, os comentários são tudo que se situa entre a abertura de comentário `/*` e o fechamento de comentário `*/`. Veja o exemplo abaixo.

```
int num_alunos; /* variável que armazena o numero de alunos */
num_alunos = 4; /* carrega o número de alunos que há na primeira fila */
/* etc */
```

Uma das utilidades de um comentário é que se retire temporariamente do programa um trecho de código. Chama-se isso de “comentar para fora”⁵. Mas pela definição da linguagem, os comentários não aninham⁶ (embora em algumas implementações do compilador os comentários aninhem). Portanto uma tentativa de se comentar para fora um trecho de programa que contenha comentários pode levar a problemas. Veja o exemplo abaixo. Nesse caso, o início do comentário para fora é fechado no fim da segunda linha. Portanto o código `num_alunos=4;` *será* compilado.

```
/* ===== comentado para fora - início
int num_alunos; /* variável que armazena o numero de alunos */
num_alunos = 4; /* carrega o número de alunos que há na primeira fila */
===== comentado para fora - fim */
/* etc */
```

Em C++, além de se usar o comentário como em C, usa-se também outro tipo de comentário. C++ considera comentário tudo que estiver a direita de `//`, sendo desnecessário fechar o comentário, pois o fim da linha já significa fim de comentário. Veja o exemplo abaixo.

```
int num_alunos; // variável que armazena o numero de alunos
num_alunos = 4; // carrega o número de alunos que há na primeira fila
// etc
```

3.6 Identificador

Identificador (*identifier*) se refere as palavras que o programador coloca no programa para dar nome a suas variáveis, funções, tipos, etc. Em C e C++, que são linguagens fortemente tipadas, todos identificadores devem ser declarados antes de serem usados, conforme será visto.

```
void g() {
int i;
// int é palavra reservada; "i" é um identificador que representa uma
// ocorrência (instância) de uma variável do tipo int (inteiro).
i=3;
// "i" é um identificador que representa uma variável tipo int, declarada acima.
// essa variável recebe uma constante inteira 3.
float area_do_circulo(float raio);
// float é palavra reservada. Essa linha de programa é a declaração de uma
// função, cujo identificador é "area_do_circulo". Essa função
// recebe um parâmetro que é uma variável tipo float identificador é "raio".
}
```

3.7 Constantes literais

Pode-se carregar valores constantes literais em variáveis, conforme mostrado no exemplo abaixo.

```
int i = 3;
float f = 4.5;
double pi = 3.14;
char ch = 'c';
char *string = "Bom dia"; // uma constante string retorna um ponteiro para char
```

3.8 Escopo

O escopo significa “mira, alvo, intenção”⁷. Para a programação em C++, significa um trecho de código onde é permitido que novas definições tomem precedência sobre definições feitas anteriormente. Esse trecho de código é delimitado por `{` (abre escopo) e `}` (fecha escopo).

⁵ Pode-se também tirar temporariamente trechos de código com diretivas de compilação.

⁶ Aninhar em inglês é *nest*

⁷ Segundo o dicionário Aurélio

Uma variável declarada fora de qualquer função é uma variável global. Qualquer função é definida dentro de um escopo. As variáveis que são parâmetros da função, bem como as variáveis declaradas dentro da função existem apenas enquanto existir o escopo da função. As definições dentro da função tem precedência sobre declarações globais.

```
int i; // variável global, definida fora de uma função
void f1() {
    int i; // variável no escopo da função f1, diferente da variável global
    i=3;    // a constante 3 é carregada na variável local da função f1
}
void f2() {
    i=5;    // não há variável local chamada "i", portanto a constante 5
           // será armazenada na variável global "i".
}
```

Outro exemplo:

```
void f1() {
    int i; // declaro a variável "i"
    i=3;    // atribuo uma constante a essa variável

    { // abro um escopo interno
        int i; // as variáveis definidas no escopo interno não são as mesmas
              // que aquelas fora do escopo; a declaração interna toma precedência

        i=5;    // atribuo uma constante a variável interna
    } // fecho o escopo que foi aberto

    // o que aconteceu dentro do escopo interno não interessa fora dele
    // aqui fora, "i" continua a valer 3;
}
```

3.9 Tipos de dados padrão (*Standard Data Types*)

Data Type	n de bytes	range
Char	1	-128..127
unsigned char	1	0..255
Int	2	-32768..32767
unsigned int	2	0..65535
Long	4	$-2^{31}..2^{31}-1$
unsigned long	4	$0..2^{32}$
Float	4	
Double	8	
long double	10	

Declaração:

```
<data_type> <instance_user_identifier_list>;           // em inglês
ou
<tipo_de_dados> <lista_de_ocorrências_de_identificadores_do_usuario>; // em português
```

Exemplo:

```
int i;           // i é ocorrência de variável tipo int
double d1,d2,d3; // 3 ocorrências de variáveis tipo double
```

3.10 Palavras reservadas do C++ (keywords)

As palavras reservadas do C++ são um conjunto de identificadores que são previamente definidas antes de começar a compilação.

Palavras reservadas de C++					
asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

É proibido usar uma palavra reservada como um identificador definido pelo usuário. Por exemplo, é proibido que se crie uma função como abaixo.

```
void continue () { // ERRO ! o nome da função é uma palavra reservada
    printf("continue"); // isso não é um erro, pois a palavra reservada
                        // está dentro de uma constante string
}
```

Além das palavras reservadas da linguagem, há uma lista de identificadores que estão definidos na biblioteca padrão de C / C++, por exemplo `printf`. Não é recomendável que se use um identificador que pertença a essa lista. É proibido que se declare 2 identificadores com significados diferentes. Caso se use um identificador que também existe na biblioteca padrão, poderá haver conflito entre as duas declarações e isso gera um erro de compilação.

3.11 Letras usadas em pontuação

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

3.12 Letras usadas em operadores

-> ++ -- .* ->* << >> <= >= == !=
 && || *= /= %= += -= <<= >>= &= ^=
 |= ::

Capítulo 4) Estrutura do Compilador

4.1 Entendendo o Compilador

Caso se utilize o sistema operacional Windows [unix], o que o compilador fez no exemplo “hello world” foi compilar o seu texto fonte (*.c ou *.cpp) criando um código compilado (*.obj [*.o]). Em seguida foi chamado o linker que ligou este último arquivo com a biblioteca padrão (*standard library*) do C / C++ (c.lib) e criou um arquivo executável (*.exe [qualquer extensão]). Um compilador comercial (como Visual C++, ou Borland C++ Builder) possui geralmente ambiente integrado que chama a execução de (*.exe) sob debug, mas como não foi ativada nenhuma função de debug, nada além da execução normal ocorreu e o programa rapidamente chega ao fim, retornando então ao ambiente integrado.

Um programa fonte passa por um processo de construção (*build*), para que seja produzido o programa executável. O processo de construção do programa é separado em duas etapas – compilação e ligação, como será explicado com mais detalhes. A figura abaixo ilustra o processo de construção de um programa.

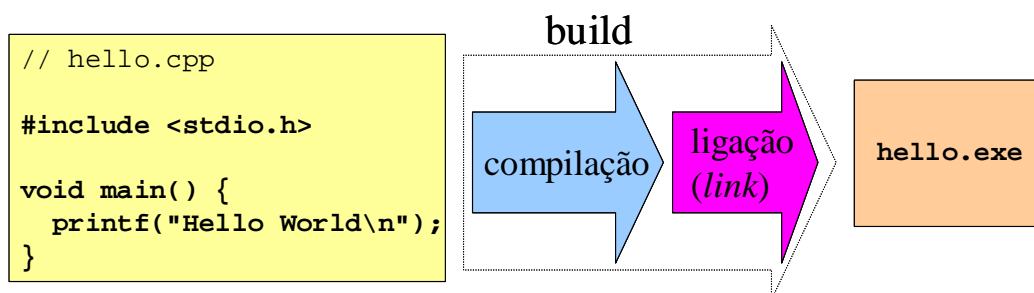


Figura 2: Processo de construção de um programa executável

Os programas em C++ são escritos como um conjunto de funções. Uma função em C++ é uma subrotina que retorna um valor do tipo `return_type`, possui um nome identificador `function_name` e possui um número maior ou igual a zero de argumentos (mesmo que não haja argumento nenhum, é necessário o abre e fecha parênteses `()`). O corpo da função é escrito no *block statement*, que é o espaço entre as chaves `{}`. Os comentários (não geram código compilado) são o que estiver entre `/*` e `*/` ou a direita de `//`.

```
<return_type> function_identifier (<arguments>) {
// function body
}
```

Ou em português.

```
<tipo_de_retorno> identificador_da_função (<argumentos>) {
// corpo da função
}
```

Em C++, o programa é a execução da função `main`. A função `main`, bem como todas as funções, pode chamar quaisquer funções definidas no universo do programa. O arquivo `hello.cpp` possui apenas a função `main`. Como não se deseja retorno desta função, escreve-se a palavra reservada `void` (cancelado) como o tipo de retorno da função `main`. O corpo da função contém apenas a chamada da função `printf`, da biblioteca padrão, cuja finalidade é colocar no console de saída (tela) o argumento

que recebeu, que é a string "Hello, world\n". O \n no final significa <return>, isto é, mudar de linha. A linha `#include <stdio.h>` no início do programa serve para declarar o protótipo (*prototype*) da função `printf`, conforme será visto mais tarde.

4.2 Protótipos (*prototypes*)

Conforme foi dito, os identificadores precisam ser declarados antes de serem usados⁸, ou seja, antes que se possa gerar código a partir deles. Caso o identificador seja uma função global, essa função pode estar definida antes de ser usada, ou pode ser apenas declarada (protótipo) antes de ser usada. No segundo caso, é preciso que a definição exista em algum lugar do projeto, pois do contrário ocorre erro de ligação (*link*). A definição de uma função inclui a declaração dessa função, mas não vice-versa.

Para declarar um identificador como função deve-se declarar o protótipo (*prototype*) da função. O protótipo da função contém apenas o tipo do retorno, o identificador da função, os argumentos entre parêntesis e um ponto-e-vírgula (;) no final. O protótipo não contém o corpo propriamente dito da função nem gera código executável, **apenas faz a sua declaração** para permitir que o programa abaixo possa usar a função.

Seja o programa abaixo. Nesse programa, define-se (e portanto declara-se) a função global "hello", e em seguida a função global "bye". Por último, define-se a função main, que chama a função hello e a função bye. O programa está correto (pode ser construído sem erros), pois quando se está gerando código a partir dos identificadores das funções hello e bye, esses identificadores já estão definidos.

```
#include <iostream.h>
// definição da função hello (inclui declaração)
void hello() {
    cout << "Hello, world" << endl;
}

// definição da função bye (inclui declaração)
void bye() {
    cout << "Bye bye, world" << endl;
}

// definição da função main
void main() {
    hello(); // chama-se a função hello
    bye();   // chama-se a função bye
}
```

Saída do programa

```
Hello, world
Bye bye, world
```

Considere agora uma variante desse programa, como mostrado abaixo. Nessa variante, primeiro declara-se os protótipos das funções globais hello e bye. A declaração desses protótipos não gera código, mas faz o compilador aprender o significado dos identificadores. Com isso, quando o compilador pode gerar código normalmente na função main (que chama hello e bye). Posteriormente à função main está a definição das funções hello e bye, em compatibilidade com os protótipos previamente declarados. Esse programa é construído sem problemas, e dá o mesmo resultado do programa anterior.

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

⁸ Em C não é obrigatório declarar os protótipos das funções antes de usá-las, é apenas fortemente recomendável. Em C++ é obrigatório.

```
// declaração de funções globais (não gera-se código)
void hello(); // declaração (protótipo) da função hello
void bye();   // declaração (protótipo) da função hello

// definição da função main
void main() {
    hello(); // chama-se a função hello
    bye();   // chama-se a função bye
}

// definição da função hello
void hello() {
    cout << "Hello, world" << endl;
}

// definição da função bye
void bye() {
    cout << "Bye bye, world" << endl;
}
```

Como exercício, imagine que o programa acima fosse modificado e as linhas após a função main fossem apagadas, de forma que a definição das funções hello e bye não existisse mais. Nesse caso, o programa compilaria, mas não passaria pela ligação (*link*), que acusaria a falta das funções que foram apagadas.

4.3 Projetos em C/C++

Um projeto em C/C++ é a forma de construir um programa executável a partir de mais de um arquivo fonte (*.c ou *.cpp). A forma como se representa a informação de quais arquivos fonte fazem parte de um dado projeto não é padrão. Ou seja, compiladores diferentes tem formas diferentes de informar quais arquivos fonte fazem parte de um projeto (ainda que usem os mesmos arquivos fonte).

Para construir o programa executável a partir de um projeto, o compilador compila cada um dos arquivos fonte (*.c ou *.cpp), gerando arquivos objeto de mesmo nome e extensão *.obj (no caso de Windows) ou *.o (no caso de unix). Cada arquivo objeto contém a informação do código compilado de funções globais (e outras) definidas no arquivo fonte correspondente.

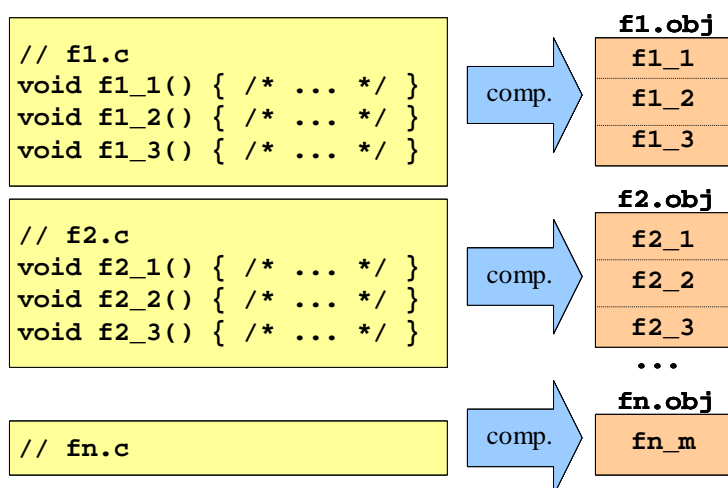


Figura 3: Etapa de compilação de um projeto em C/C++

Após concluída a etapa de compilação de todos os arquivos fonte, ocorre a etapa de ligação. Nessa etapa, todos os arquivos objeto do projeto são incluídos, e também a biblioteca padrão (*standard library*). A biblioteca padrão é automaticamente incluída em todos os projetos. Um e somente um

arquivo fonte deve conter a função main (do contrário, ocorre um erro de *link*). O programa executável gerado geralmente tem o nome do projeto.

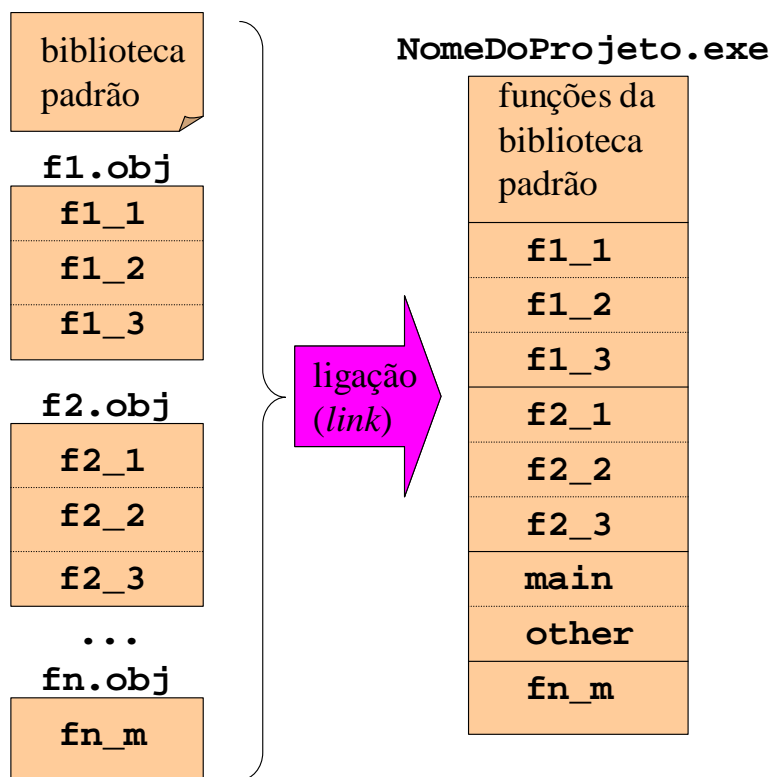


Figura 4: Etapa de ligação de um projeto em C/C++

Funções do programa compilador

- Cada arquivo fonte *.c (*.cpp) é compilado gerando *.obj (ou *.o).
- Cada arquivo fonte *.c (*.cpp) é compilado como se outros arquivos do projeto não existissem.
- No início da compilação de cada arquivo *.c (*.cpp), o compilador apenas conhece o vocabulário das palavras reservadas, e a gramática da linguagem C/C++.
- O compilador deve conseguir compilar o código fonte numa única passagem, ou haverá erro de compilação.
- Ao analisar um trecho de código fonte, o compilador faz uma das 2 coisas:
 - Gera código.
 - Aprende o significado de novos identificadores, e com isso passa a poder compilar com o que aprendeu.
- O compilador não pode gerar código a partir de identificadores que não tenham sido previamente declarados. A tentativa de fazê-lo gera erro de compilação.
- Um mesmo arquivo fonte *.c (*.cpp) pode pertencer a mais de um projeto.

4.4 Header Files (*.h)

Agora pode-se explicar melhor a necessidade da linha `#include <iostream.h>` no início de vários arquivos fonte. Esta linha faz incluir um arquivo chamado `iostream.h`, que é um dos *header files* padrão (sempre fornecidos com o compilador C++). O arquivo `iostream.h` é um arquivo de texto (pode ser examinado com o editor) em que está escrito (entre outras coisas) o protótipo da função `printf` que foi usada no programa de exemplo. Desta forma, quando o compilador encontra a chamada da função `printf`, já sabe o seu protótipo e pode conferir se a chamada da função foi feita corretamente.

Os arquivos com a extensão `*.h` são chamados de *header* e sua finalidade é de serem incluídos nos programas fonte com a diretiva `#include <*.h>`, para includes padrão do compilador e `#include "*.h"` para includes que o programador desenvolver. Os arquivos de *header* geralmente contém, além de protótipos de funções, definições de tipos, macros (`#define`), etc. Em C os *headers* não devem conter trechos que gerem código executável como o corpo de funções, já em C++ podem conter códigos curtos `inline`, conforme será visto mais tarde.

Os arquivos header outro tipo de arquivo fonte. Esses arquivos são feitos para serem incluídos em arquivos `*.c`, `*.cpp` ou mesmo outro arquivo `*.h`. A inclusão é feita com a diretiva `#include <nome.h>`. O compilador substitui essa diretiva pelo conteúdo do arquivo `nome.h` e segue a compilação. Os headers do sistema são incluídos com o nome entre `< >`, enquanto os headers do programador são incluídos com `#include "nome.h"`

Nesses arquivos geralmente são escritas declarações úteis para permitir a compilação do código que vem a seguir. No exemplo abaixo, há um programa que foi gerado a partir de 2 arquivos fonte chamados `plus_1_file.cpp` e `main.cpp`. Para que se possa compilar o arquivo `main.cpp` é necessário que se declare o identificador `plus_1`, e isso é feito no arquivo `myheader.h`.

```
// myheader.h
int plus_1(int in); // protótipo de uma função
```

```
// plus_1_file.cpp
int plus_1(int in) {
    return in+1;
}
```

```
// main.cpp
#include <iostream.h>
#include "myheader.h"
void main() {
    int i=2;
    cout << "i+1=" << plus_1(i) << endl;
}
```

4.5 Biblioteca (*library*)

Um arquivo executável (também chamado de “aplicativo”) é um produto para o usuário. Analogamente, uma biblioteca é um produto para o programador. Uma biblioteca não é executável. É um conjunto de funções compiladas (`*.obj`) mas ainda não ligadas a nenhum arquivo executável. Para se criar uma biblioteca, todo compilador possui uma ferramenta de criação de bibliotecas.

Todo compilador possui uma biblioteca padrão (*standard library*), que é uma biblioteca como outra qualquer. Em alguns compiladores os arquivos fonte para a criação da biblioteca padrão estão disponíveis. A única diferença da biblioteca padrão e outras bibliotecas é o fato de que a biblioteca

padrão por definição está sempre disponível em qualquer compilador. Qualquer outra biblioteca que se use deve ser adicionada manualmente. Por isso, geralmente quem usa uma biblioteca não padrão “x”, costuma ter (ou querer ter) os arquivos fonte dessa biblioteca. Assim, pode-se criar a biblioteca sempre que necessário, em qualquer compilador.

4.5.1 Utilizando Bibliotecas prontas

Uma biblioteca é um conjunto de funções prontas para serem usadas. Geralmente uma biblioteca se apresenta como 2 arquivos:

1. nome_da_biblioteca.lib
2. nome_da_biblioteca.h

Para se usar uma biblioteca feita por outra pessoa, basta acrescentar a biblioteca no projeto. Um projeto de teste típico com uma biblioteca conterá 2 arquivos

1. nome_da_biblioteca.lib
2. main.cpp (um arquivo fonte com a função main)

Geralmente a biblioteca vem com pelo menos 1 arquivo header, com as declarações das funções que contém a biblioteca. Portanto um programa que use as funções da biblioteca deverá incluir o tal header.

Exemplo:

```
#include "nome_da_biblioteca.h"
void main () {
    fun_da_biblioteca();
}
```

4.5.2 Fazendo bibliotecas

Além de se criar arquivos executáveis, o compilador pode também criar uma biblioteca. Se um executável (um aplicativo) é um produto para o usuário, uma biblioteca deve ser encarada como um produto para o programador. Uma biblioteca não é executável, é um conjunto de funções compiladas (*.obj) mas ainda não ligadas a nenhum arquivo executável. Para se criar uma biblioteca, todo compilador possui uma ferramenta de criação de bibliotecas.

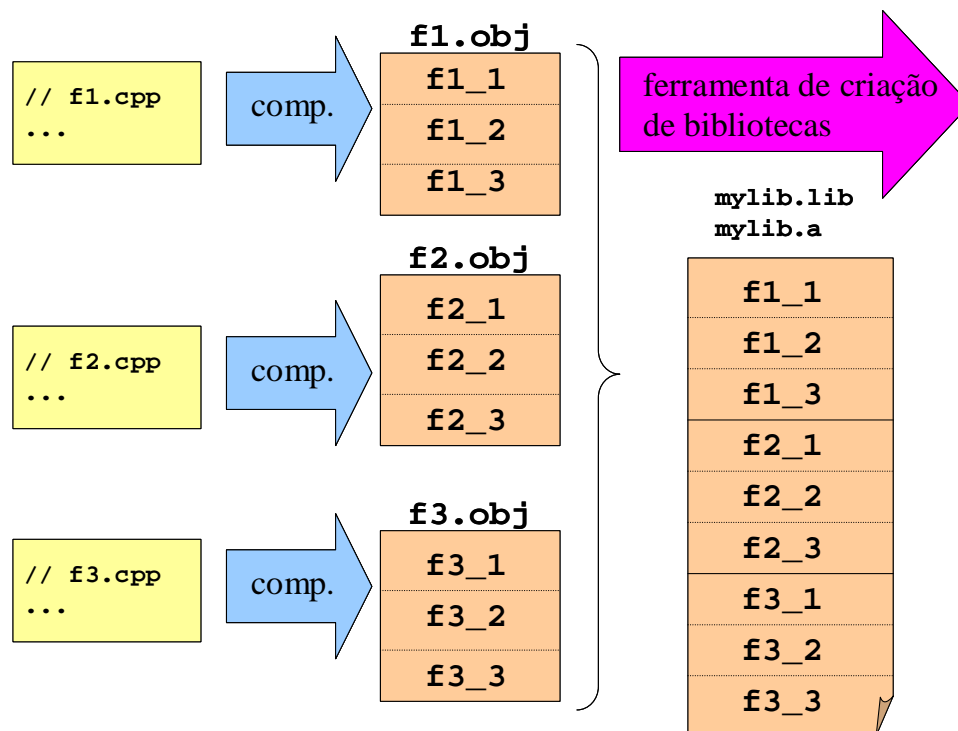


Figura 5: diagrama da criação de bibliotecas

O objetivo dessa prática é entender a criação e uso de bibliotecas. Nesse sentido, será feita uma biblioteca que não tem utilidade prática. Isto é, as funções da biblioteca não são úteis realmente; são apenas funções bastante simples para teste.

Prática: Fazer uma biblioteca com as especificações abaixo:

1. Nome da biblioteca: `plus_n`, contendo 5 funções: `plus_1`, `plus_2`, `plus_3`, `plus_4`, `plus_5`. Cada uma dessas funções deve somar 1, 2, 3, 4 ou 5 a entrada (conforme o nome) e

Resposta: Para a implementação das 5 funções serão criados 5 arquivos diferentes (não é necessário que os arquivos sejam diferentes, mas é recomendável para que a biblioteca seja mais “fatiada”, isto é, que cada fatia da biblioteca entre ou deixe de entrar no programa executável final de acordo com a necessidade do linker). Os 5 arquivos `plus_?.cpp` estão mostrados abaixo.

```
// plus_1.cpp
int plus_1 (int in) { return in+1; }
```

```
// plus_2.cpp
int plus_2 (int in) { return in+2; }
```

```
// plus_3.cpp
int plus_3 (int in) { return in+3; }
```

```
// plus_4.cpp
int plus_4 (int in) { return in+4; }
```

```
// plus_5.cpp
int plus_5 (int in) { return in+5; }
```

Para permitir o uso da biblioteca, é preciso que haja um arquivo de header com os protótipos das funções que existem. O arquivo merece o mesmo nome da biblioteca, com a extensão .h. O arquivo `plus_n.h` é mostrado abaixo.

```
// plus_n.h
int plus_1 (int in);
int plus_2 (int in);
int plus_3 (int in);
int plus_4 (int in);
int plus_5 (int in);
```

Utilizando-se a ferramenta de criação de bibliotecas do seu compilador, a partir dos arquivos fonte mostrados acima, cria-se a biblioteca `plus_n.lib` (ou `libplus_n.a`). Para se testar a biblioteca, é preciso que exista um programa principal com a função `main` (a biblioteca não contém a função `main`). O programa abaixo `test_plus_n.cpp` usa funções da biblioteca.

```
// test_plus_n.cpp
#include <iostream.h>
#include "plus_n.h"
void main() {
    int k = 10;
    cout << "k = " << k << "; k+3 = " << mais_3(k) << endl;
}
```

A saída do programa é como mostrado abaixo.

```
k = 10; k+3 = 13
```

Exercício:

Acrescente à biblioteca `plus_n`, feita na prática anterior, as funções `plus_6` e `plus_7`. Altere o programa principal para que use uma dessas duas funções para testar a nova biblioteca.

4.6 Regras do compilador

O compilador segue um conjunto pequeno de regras simples para fazer o seu trabalho de construir o programa executável. Nessa seção, fala-se desse pequeno conjunto de regras. O nome “compilador” é ligeiramente inadequado.

O nome mais correto do programa compilador seria programa “construtor” (*builder*), pois é isso (construir) que o programa compilador realmente faz. A construção é o procedimento que inclui “compilar” e “ligar (*link*)”.

- Quando analisa um código, o compilador faz uma das duas coisas:
 - Gera código.
 - Aprende uma informação, que poderá ser útil para gerar código mais tarde.
- O compilador somente pode gerar código a partir de identificadores que tenham sido previamente declarados.
- Ao definir-se uma função, também se está declarando-a. Mas ao declarar-se uma função, não se está definindo. Declarar uma função é o mesmo que escrever o seu protótipo.
- O compilador precisa conseguir compilar cada arquivo fonte (*.c ou *.cpp) em uma única passagem, de cima para baixo. Do contrário, ocorre erro de compilação.
- A construção de um executável é feita em duas etapas – compilação e ligação.

- Um projeto é a construção de um executável a partir de um conjunto de programas fonte (*.cpp). Para construir um projeto, cada fonte é compilado como se os demais não existissem. Em seguida, a etapa de ligação faz construir o executável final.
- Ao iniciar a compilação de cada arquivo fonte (*.c ou *.cpp), o compilador somente reconhece as palavras reservadas e a gramática da linguagem.
- Os arquivos de cabeçalho (*header*) *.h *não devem* fazer parte diretamente do projeto⁹, e portanto *não* geram arquivo objeto. Mas isso não quer dizer que esses arquivos não sejam importantes. A finalidade desses arquivos é serem incluídos em arquivos fonte (*.c ou *.cpp).

⁹ O leitor deve ter percebido que usou-se o termo “não devem” e não “não podem”. Isso porque o compilador segue regras simples e geralmente não realmente verifica a extensão dos arquivos fonte. Contudo, é considerado má programação fazer constar um arquivo *.h diretamente no projeto. É também má programação incluir (usando #include) um arquivo *.c ou *.cpp dentro de um outro arquivo fonte de mesma extensão, como se fosse um arquivo *.h.

Capítulo 5) Fundamentos de C /C++

5.1 Chamada de função por referência e por valor

Numa função, as variáveis que são parâmetros e as variáveis internas da função são chamadas de variáveis automáticas, pois são automaticamente criadas no momento de chamada da função e automaticamente destruídas quando a função termina.

Há basicamente 2 formas de se qualificar as variáveis automáticas que são parâmetros de uma função.

- 1) Parâmetros passados por valor.
- 2) Parâmetros passados por referência.

Quando um parâmetro é passado por valor, a variável parâmetro dentro da função é uma **cópia**¹⁰ da variável que chamou a função. Portanto qualquer alteração que ocorra na variável valerá apenas dentro da função e com o fim desta, o programa que chamou a função não perceberá nada.

Quando um parâmetro é passado por referência, a variável que se passa como parâmetro é na realidade o ponteiro para a variável que chamou a função. O que interessa na prática é que as alterações que se faz na variável dentro da função **são** percebidas fora da função.

No exemplo abaixo, 3 funções são definidas. A função f1 tem passagem de parâmetros por valor. A função f2 tem passagem de parâmetros por valor, mas como está recebendo o ponteiro para uma variável (um objeto), significa que colocar dados na referência do ponteiro corresponde a colocar dados na variável original. Ou seja, é como se a passagem de parâmetros fosse por referência. Mas como o referenciamento dos ponteiros deve ser feito explicitamente (tanto dentro da função f2, como na linha “*x = 2.2;”, quando na chamada da função f2, como na linha “f2(&a);”), essa forma de se passar parâmetros pode ser considerada como “pseudo por referência”. A função f3 recebe o parâmetro como um lvalue do tipo em questão (no caso float). Portanto, trata-se de uma passagem por referência verdadeira, pois tanto dentro da função f3 quanto na chamada da função f3, o uso de lvalue é implícito. Para modificar f3 para que a passagem de parâmetros seja por valor, basta apagar uma letra, que é o “&” no protótipo de f3.

```
#include <iostream.h>
void f1(float z) {
    z = 1.1;
}
void f2(float *x) {
    *x = 2.2;
}
void f3(float & v) {
    v = 3.3;
}
void main() {
    float a = 10.1;
    cout << "a=" << a << endl;
    f1(a);
    cout << "a=" << a << endl;
    f2(&a);
}
```

¹⁰ Cópia no sentido que, no caso de programação orientada a objeto, o construtor de cópia é chamado.

```
cout << "a=" << a << endl;  
f3(a);   cout << "a=" << a << endl;  
}
```

Resultado:

```
a=10.1  
a=10.1  
a=2.2  
a=3.3
```

Em C, a única forma de se passar parâmetros por referência é como mostrado em f2. Em C++, pode-se usar como mostrado em f3, que é mais elegante.

Os motivos para se optar por passar um parâmetro por valor ou por referência são vários (não apenas o fato de a função alterar ou não alterar o valor do parâmetro). Um deles trata do tempo de execução. Como as variáveis passadas por valor recebem uma **cópia** do seu valor quando é chamada, se essa variável é grande (pode ocorrer no caso de variáveis estruturadas, veja a seção 7.4) o tempo de cópia pode ser significativo, principalmente se essa chamada é feita num trecho repetitivo de programa, como um loop. Uma variável passada por referência passa somente o ponteiro da variável para a função, e o ponteiro tem valor fixo independente do tamanho da variável.

5.2 Tipos de dados definidos pelo programador

Além dos tipos padrão, o programador pode definir tipos próprios com diversas finalidades, que após definidos passarão a ser tão válidos como os demais. Pode-se, por exemplo, criar o tipo `REAL` que será usado no lugar dos tipos de ponto flutuante com finalidade de facilitar a substituição do tipo básico de ponto flutuante em todo o programa. No exemplo abaixo, foram definidas duas funções com parâmetros em ponto flutuante onde foi usado o tipo do usuário definido como `REAL`, que corresponde ao tipo padrão `float`. Caso se deseje substituir em todo o programa, para efeito das funções definidas pelo usuário, o tipo `float` pelo tipo `double`, basta trocar a definição do tipo do usuário `REAL` de `float` para `double`.

Exemplo:

```
typedef float REAL;    // defino o tipo real  
REAL myinverse(REAL x) { // declaro e defino uma função usando o meu tipo REAL  
    return (1./x);  
}  
REAL mydoubleinverse(REAL x) { // outra função com o tipo REAL  
    return (1./x/x);  
}
```

Também é considerado tipo do usuário (*data type*) uma lista de identificadores com a palavra reservada `enum`. Por exemplo:

```
typedef enum days {sun,mon,tue,wed,thu,fri,sat}; // cria o tipo days  
days today; // cria ocorrência do tipo days com today  
ou em português  
typedef enum dias {dom,seg,ter,qua,qui,sex,sab}; // cria o tipo dias  
dias hoje; // cria ocorrência do tipo dias com hoje
```

5.3 Maquiagem de tipos (*type casting*)

Muitas vezes é preciso converter um tipo para outro. Por exemplo: `int` para `float`, etc. Em alguns casos o compilador sabe que tem que fazer a conversão e faz automaticamente para você. Algumas vezes a conversão não é feita automaticamente mas pode-se forçar a conversão maquiando o tipo. Para fazer a maquiagem, basta colocar antes da variável ou função em questão o novo tipo entre parênteses. No exemplo abaixo, a maquiagem de tipo garante a divisão em ponto flutuante.

```
#include <iostream.h>
void main () {
    float f;
    f = 2/5; // divisão de ints
    cout << "f = " << f << endl; // resultado errado
    f = (float)2/(float)5; // type cast garante divisão de floats
    cout << "f = " << f << endl; // resultado certo
}
```

Muitos autores estão recomendando evitar o uso direto de maquiagem de tipos, como mostrado no programa acima. É mais seguro a utilização de `static_cast` (palavra reservada), como mostrado no programa abaixo.

```
#include <iostream.h>
void main () {
    float f;
    f = 2/5; // divisão de ints
    cout << "f = " << f << endl; // resultado errado
    f = static_cast<float>(2)/static_cast<float>(5); // type cast garante divisão de floats
    cout << "f = " << f << endl; // resultado certo
}
```

Em ambos os programas, a saída é como mostrado abaixo.

```
f = 0
f = 0.4
```

5.4 Operações matemáticas

Para se escrever operações matemáticas básicas, basta escreve-las no programa da mesma forma como no papel, respeitando-se a precedência dos operadores da mesma forma também.

```
double z,y,x;
y=3.33; z=4.5677;
x=y*y+z*3-5;
```

A biblioteca padrão oferece também inúmeras funções matemáticas de interesse geral, listadas abaixo. Para maiores informações sobre essas funções, consulte qualquer referência ou o help do compilador.

Funções matemáticas da biblioteca padrão de C

acos	asin	atan	atan2	sin	tan
cosh	sinh	tanh	exp	frexp	ldexp
log	log10	modf	pow	sqrt	ceil
fabs	floor	fmod			

5.5 Controle de fluxo do programa

Em todos os casos o teste do controle de fluxo, que é interpretado como booleano, é qualquer variável ou função de qualquer tipo (inclusive ponteiro). 0 significa falso e 1 significa verdadeiro.

Em todos os lugares onde há `<statement>` pode-se substituir por `<block statement>`, que é um conjunto de *statements* dentro de um escopo de chaves.

```
{ // block statement
    <statement1>;
    <statement2>;
}
```

Exemplo:

```
void fun2() { /* ... */ }
```

```
void fun3() { /* ... */ }
void fun4() { /* ... */ }

// uma maneira de se chamar 3 procedimentos em seqüência
void fun1 () {
    if (i==1) {
        fun2();
        fun3();
        fun4();
    }
}

// função auxiliar
void fun_auxiliary() {
    fun2();
    fun3();
    fun4();
}

// outra maneira de se chamar 3 procedimentos em seqüência
void fun1_new () {
    if (i==1)
        fun_auxiliary();
}
```

5.6 Execução condicional

```
if (<variavble>) <statement>
else <statement>;    // optional
```

Exemplo:

```
#include <iostream.h>
float a,b,c; // variáveis globais
void main () {
    a=2; b=1.5;
    if (b==0)
        cout <<"c is undefined" << endl;
    else {
        c = a / b;
        cout << "c = " << c << endl;
    }
} // fim da função main
```

5.7 Laços (loop) de programação

Os laços de programação são estruturas de uma linguagem de programação para comandar instruções que se repetem num programa. Há 3 estruturas de laço em C++, mostradas abaixo.

5.7.1 Laço tipo “do-while”

```
do
    <statement>;
while (<variable>);
```

Exemplo:

```
#include <iostream.h>
void main() {
    int i=5;
    do {
        cout << "i = " << i << endl;
        i--; // i = i - 1
    }
}
```

```
    while (i>0);  
}
```

Resultado:

```
i=5  
i=4  
i=3  
i=2  
i=1
```

5.7.2 while

```
while (<variavble>)  
    <statement>;
```

Exemplo:

```
#include <iostream.h>  
void main() {  
    int i=5;  
    while (i>0) {  
        cout << "i = " << i << endl;  
        i--; // i = i - 1  
    }  
}
```

Resultado:

```
i=5  
i=4  
i=3  
i=2  
i=1
```

5.7.3 for

```
for (<inic> ; <test> ; <set>)  
    <statement>;  
  
/*===== o mesmo que  
{  
    <inic>;  
    while (<test>) {  
        <statement>;  
        <set>;  
    }  
}  
=====*/
```

Exemplo:

```
#include <iostream.h>  
void main() {  
    for (int i=5 ; i>0 ; i--)  
        cout << "i = " << i << endl;  
}
```

Resultado:

```
i=5  
i=4  
i=3  
i=2  
i=1
```

5.7.4 Alterando o controle dos laços com break e continue

Caso se inclua a palavra reservada “break” (parar) num laço controlado, o fluxo de execução é quebrado e o programa sai do laço que está executando. No exemplo abaixo, há dois laços do tipo for, e o break é ativado numa condição específica ($j==3$). Quando isso ocorre, o laço interno é quebrado, mas a execução continua no laço externo.

```
#include <iostream.h>
void main() {
    for (int i=0 ; i < 2 ; i++) {
        for (int j=5 ; j>0 ; j--) {
            if (j==3)
                break;
            cout << "i = " << i << ", j = " << j << endl;
        }
    }
}
```

Resultado:

```
i = 0, j = 5
i = 0, j = 4
i = 1, j = 5
i = 1, j = 4
```

A palavra reservada “continue” tem o significado de mandar o laço de programação corrente se encerrar (mas não se quebrar). Isto é, mesmo que haja mais comandos a serem executados antes do fim do laço, o laço se encerra caso encontre a palavra reservada “continue”. Se o exemplo acima substituir a palavra reservada “break” por “continue”, o que ocorre é que caso $j==3$, o laço interno se encerra, isto é, passa para $j=4$. Com isso, a linha que exterioriza o valor de $j=3$ não é executada. Portanto a saída é como mostrado abaixo.

```
i = 0, j = 5
i = 0, j = 4
i = 0, j = 2
i = 0, j = 1
i = 1, j = 5
i = 1, j = 4
i = 1, j = 2
i = 1, j = 1
```

5.7.5 Exercício

Prática: Fazer um programa para escrever a tabuada de 5.

Solução:

```
#include <iostream.h>
void main() {
    int tab=5;
    for (int i=1 ; i<=10 ; i++) {
        cout << i << "*" << tab << "=" << i*tab << endl;
    }
}
```

Prática:

Fazer um programa para escrever as tabuada de 3 a 6.

Solução:

```
#include <iostream.h>
void imprime_tabuada(int tab) {
    cout << "-----" << endl;
    for (int i=1 ; i<=10 ; i++) {
        cout << i << "*" << tab << "=" << i*tab << endl;
    }
}
```

```

}
}
void main() {
    int tab_min=3;
    int tab_max=6;
    for (int i=tab_min ; i <= tab_max ; i++)
        imprime_tabuada(i);
}

```

O Visual C++ possui uma sutil diferença na implementação do for, como mostrado na seção 17.1.1 (página 204).

5.8 switch-case

O switch, case e default (implícito) são palavras reservadas que servem para definir múltiplo redirecionamento do fluxo do programa. Frequentemente também se usa a palavra reservada break. Heis a sintaxe:

```

switch (<expr>) {
    case <item>: <statements>;
    case <item>: <statements>;
    default: <statements>;    // quando nenhum dos itens coincide
}

```

Exemplo:

```

#include <iostream.h>
enum days {sun,mon,tue,wed,thu,fri,sat}; // cria o tipo days
days today; // cria ocorrência do tipo days com today
void m() {
    today = sat;
    switch (today) {
        case mon:
        case tue:
        case wed:
        case thu:
        case fri:
            cout << "Vá para o trabalho" << endl;
            break;
        case sat:
            cout << "Limpe o jardim e ";
        case sun:
            cout << "relaxe." << endl;
            break;
        default: cout << "Erro, dia não é definido";
            // quando nenhum dos itens coincide
    }
}
void main () {
    m();
}

```

5.9 arrays

Os arrays estáticos são declarados como:

```
<tipo_do_array> <nome_do_array>[<dimensão>;
```

Um array reserva espaço na memória para <dimensão> elementos do tipo <tipo_do_array> desde a posição [0] até a posição [<dimensão> - 1].

Exemplo:

```
double vector_a[10]; // vector_a[0] até vector[9]
```

```
vector_a[2] = 1.23; // uso do array
```

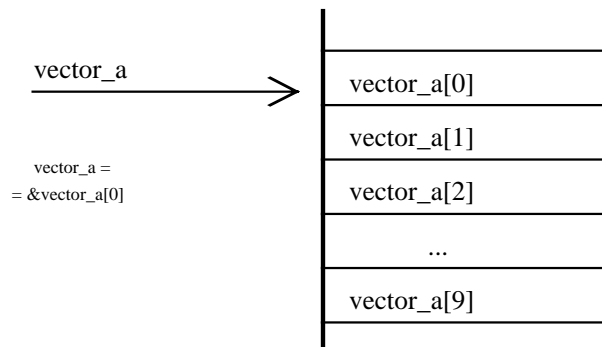


Figura 6: Visualização gráfica do array na memória

No exemplo acima, o identificador `vector_a` é uma constante do tipo `(double *)`, ou seja, ponteiro para `double`. Ao declarar `double vector_a[10]`, o compilador reserva espaço para 10 variáveis do tipo escolhido `double` e a própria variável `vector_a` é o ponteiro `double*` que aponta para o início das 10 variáveis.

Ao usar uma das variáveis do array com o uso do operador `[]`, o que realmente se está fazendo é uma indireção do ponteiro somado ao índice escolhido, conforme será explicado na seção abaixo.

Ao se utilizar um array, não há *range-check*, isto é, checagem de tentativa de acesso fora da dimensão máxima do array.

Exemplo:

```
#include <iostream.h>
void main() {
    double valores[4]; // define-se um array
    // carrega-se valores no array
    valores[0] = 1.0;
    valores[1] = 1.1;
    valores[2] = 1.2;
    valores[3] = 1.3;

    for (int i=0; i<4 ; i++)
        cout << "valores[" << i << "] = " << valores[i] << endl;
}
```

Resultado:

```
valores[0] = 1.0;
valores[1] = 1.1;
valores[2] = 1.2;
valores[3] = 1.3;
```

No exemplo abaixo, define-se um array de strings com 4 posições, devidamente carregado com constantes. Nesse caso, existem as posições `[0]`, `[1]`, `[2]` e `[3]`. Em seguida, executa-se um laço que manda imprimir o conteúdo do array, mas o programa comanda o laço indo até a posição `[4]`. Isso não gera um erro de compilação ou de execução, mas é um erro de lógica, pois o valor do array na posição `[4]` não é definido.

Exemplo:

```
#include <iostream.h>
void main() {
    char* nomes[4] = {"Manuel", "Joaquim", "Maria", "Bruno"};
    for (int i=0; i<5 ; i++)
```

```
    cout << "Nome[" << i << "] = " << nomes[i] << endl;
}
```

Resultado:

```
Manuel
Joaquim
Maria
Bruno
sldflaskf j;alsdkj (indefinido)
```

5.9.1 Arrays multidimensionais

São muito semelhantes aos arrays de 1 dimensão. Para a definição, apenas acrescenta-se a segunda dimensão após a primeira. A utilização é feita da mesma forma que a definição, com múltiplas indireções (`[i][j]...`).

Na declaração de um array estático, caso se queira carrega-lo com um valor de inicialização na compilação, pode-se declarar o array omitindo-se a primeira dimensão, pois os dados declarados a seguir contém esta informação. Todas as dimensões exceto a primeira precisam ser declaradas. Veja o exemplo:

Exemplo:

```
void fun() {
    // aqui também se exemplifica como inicializá-lo na compilação
    int a[][4] = { {1,2,3,4} , {5,6,7,8} , {9,10,11,12} };
    int b[][4][5] = { {1,2,3,4} , {5,6,7,8} , {9,10,11,12} };

    int i = a[1][1]; // uso de array multivariável
}
```

5.10 Parâmetros da função main

Na verdade, o programa gerado por um compilador C / C++ deve ser entendido como o código fonte abaixo:

```
// inicializações do compilador
exit_code = main (argc, argv);
// finalizações do compilador
```

A função `main` é aquela que chama o próprio programa, conforme já foi explicado. Esta função admite ter parâmetros (argumentos) que são as palavras escritas na linha de comando que chamou o programa. O protótipo da função `main` é:

```
int main (int argc, char *argv[]);
```

O retorno da função `main` é o código de retorno ao sistema operacional, que só interessa para que um arquivo `*.bat` do DOS (em programas para Windows), ou um script em um programa em unix, mas isso não será abordado aqui.

O primeiro argumento `int argc` é o número de parâmetros passados ao programa (incluindo o próprio nome dele) e o segundo `char **argv` é um array de strings (`char*`) cujos conteúdos são exatamente os parâmetros passados. Veja o exemplo.

Exemplo:

```
#include <iostream.h>
void main (int argc, char *argv[]) {
    for (int i=0 ; i < argc ; i++)
        cout << "o argumento de chamada numero " << i << " é " << argv[i] << endl;
}
```

Compile o código acima, chame-o de `prog`, por exemplo. Verifique o seu funcionamento utilizando a linha de comando abaixo:

```
prog a1 a2 a3 a4 a5 "one sentence"
```

O resultado será:

```
o argumento de chamada numero 0 é C:\USERS_1\VILLAS_C\BORLAND_C\T3.EXE
o argumento de chamada numero 1 é a1
o argumento de chamada numero 2 é a2
o argumento de chamada numero 3 é a3
o argumento de chamada numero 4 é a4
o argumento de chamada numero 5 é a5
o argumento de chamada numero 5 é one sentence
```

É muito comum que um programa seja usado a partir dos parâmetros passados pela linha de comando e nestes casos é de bom alvitre dar informações rápidas sobre a utilização do programa quando não se passa parâmetro algum. Veja o exemplo de uma estrutura genérica de um programa deste tipo.

Exemplo:

```
#include <iostream.h>
void main(int argc,char *argv[]) {
    // prints the always seen message
    cout << "Prog" << endl;
    cout << "Version 1.3 - by Author, in Jan 20th, 1998" << endl;

    if (argc<2) {
        // called with no arguments. Display immediate help
        cout << "Usage: Prog [-options] def[.def]"
            << " source[.ex1] [destin[.ex2]]";
        cout << "Something Else ..." << endl;
    }
    else {
        // do what the program should do
        // do_the_thing(argv[1],argv[2]);
    }
}
```

5.11 Compilação condicional

A diretiva `#if` e `#endif` compila condicionalmente um trecho de programa. A sintaxe é mostrada abaixo.

```
#if <condição>
// trecho de programa
#endif
```

```
#if 0
    int pi=3.1416;
#endif
```

5.12 Pré processador e tokens (símbolos) usados pelo pré-processador

A linguagem C / C++ compila os arquivos fonte após esses arquivos serem pré-processados a partir de algumas diretivas, iniciadas pelos símbolos abaixo.

```
#    ##
```

```
#define PI 3.1416
#ifndef PI
int pi=3.1416;
```

```
#endif
```

As diretivas de pré-comilação mais importantes são:

5.13 #define

O `#define` é uma diretiva que troca seqüências de texto no código fonte, para em seguida esse texto ser compilado. O uso dessa diretiva também é chamada de macro. É um estilo recomendável de boa programação que as macros sejam escritas com maiúsculas.

```
#define PI 3.1416
float area_circulo(float raio) {
    return raio*raio*PI; // expandido para return raio*raio*3.1416;
}
```

O `#define` pode ser parametrizado. Veja o exemplo abaixo.

```
#define FORMULA(x,y) (x)*3 + (y)*4 + 5
void main () {
    float r = FORMULA(1.2,3.4); // expandido para r = (1.2)*3 + (3.4)*4 + 5;
}
```

É recomendável que se escreva a parametrização das macros usando parêntesis envolvendo os parâmetros, para se evitar erros como no exemplo abaixo.

```
#define FORMULA(x) x*3
void main () {
    float r = FORMULA(1+2); // expandido para r = 1+2*3,
                          // que é diferente de (1+2)*3
    // melhor seria escrever a macro como #define FORMULA(x) (x)*3
}
```

5.14 operador

Para a construção de strings a partir de parâmetros dentro de um `#define`, usa-se o operador `#`

```
#define PATH(logid,cmd) "/usr/" #logid "/bin/" #cmd

char *mytool = PATH(francisco,readmail);
// expandido para: char *mytool = "/usr/" "francisco" "/bin/" "readmail";
// é equivalente a: char *mytool = "/usr/francisco/bin/readmail";
```

5.15 operador

O operador `##` é uma diretiva para o pré processador da linguagem C/C++. Esse operador aparece como um token de troca entre 2 tokens, concatenando-os. Veja o exemplo abaixo.

```
#include <iostream.h>
#define TIPO(a) FUT ## a
#define FUTSAL "futebol de salão"
#define FUTBOL "tradicional futebol de campo"

void main () {
    cout << TIPO(SAL) << endl;
    cout << TIPO(BOL) << endl;
}
```

resulta em:

```
futebol de salão
tradicional futebol de campo
```

5.16 Número variável de parâmetros

Uma função pode ter número variável de parâmetros. Mas para usar esse recurso, a função deve de alguma forma saber quantos parâmetros serão chamados. Veja o exemplo abaixo.

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

double sum_parameters(int num,...) {
    double sum = 0.0;
    double t;
    va_list argptr;
    va_start(argptr,num); // initialize argptr
    // sum the parameters
    for ( ; num ; num--) {
        t = va_arg(argptr,double);
        sum += t;
    }
    va_end(argptr);
    return sum;
}

void main () {
    cout << "the sum is " << sum_parameters(2, 1.2, 3.4) << endl;
    cout << "the sum is " << sum_parameters(3, 1.2, 3.4, 4.4) << endl;
}
```

A saída do programa é como mostrado abaixo.

```
the sum is 4.6
the sum is 9.3
```

Capítulo 6) Técnicas para melhoria de rendimento em programação

6.1 Reutilização de código

O trabalho de programação é um trabalho caro. Projetos grandes na área de software requerem uma quantidade muito significativa de homem-hora e com isso a produção de um programa pode representar um investimento vultuoso. Além disso, a crescente complexidade do software torna praticamente impossível que apenas um indivíduo possa desenvolvê-lo completamente. Até mesmo software de “média complexidade” tende a ser feito por um time.

Uma das chaves para se melhorar o rendimento da programação é o reaproveitamento de “blocos de código”, também conhecido como “componentes”. Concretamente isso significa ser capaz de usar uma biblioteca feita por outra pessoa ou de entregar para outra pessoa uma biblioteca desenvolvida por você. A biblioteca, no caso, é um conjunto de componentes prontos para serem reaproveitados.

Para que se possa aproveitar trabalho de outros e permitir que outros usem nosso trabalho, é preciso entre outras coisas definir uma nomenclatura para o que se está trabalhando. Para isso, define-se os termos abaixo.

1. **Componente** - Um componente de software é um bloco de código que pode ser usado de diversas formas dentro de um programa. Por exemplo: listas, arrays e strings são componentes que podem ser usados em muitos programas. Também são considerados componentes de software elementos de interface, como check box, ou radio button.

Os componentes devem ser escritos como uma função de propósito genérico, de forma que possa ser usado das mais variadas formas. Quem desenvolve uma aplicação e quer usar um componente de software, não precisa saber como é o funcionamento interno desse componente.

Tipicamente “descobre-se” um componente quando se está programando e percebe-se que há trechos de programação que estão se repetindo naturalmente. É como se algo estivesse pedindo que todos esses trechos se transformassem num componente genérico que possa ser usado sempre que aquela necessidade surja.

2. **Biblioteca (*library*)** - É um conjunto de componentes de software prontos para serem usados pelo programador.
3. **Estrutura (*framework*)** - Um estrutura é um esqueleto de programa contendo algum código e indicação para a introdução de código extra. Uma estrutura é usada para se gerar uma aplicação com determinadas características.

Uma estrutura deve ser de fácil entendimento para que um programador possa usá-la para fazer um aplicativo. Um exemplo de estrutura é um conceito usado pelo VisualC, chamado “application wizard”, que cria uma estrutura pronta para o programador.

4. **Aplicação** - uma aplicação é um programa completo. Os exemplos são inúmeros - editores de texto, jogos, planilhas, etc. Uma aplicação frequentemente baseia-se numa estrutura e usa diversos componentes.

Seja um projeto de software cuja especificação requer que se faça gráficos a partir de dados que vem de um sensor. Nesse caso, seria muito útil ter em mãos uma biblioteca pronta com componentes que fazem gráficos de forma genérica a partir de dados também genéricos. Assim, o programador precisaria apenas escrever a parte do código para ler os dados do sensor em seguida chamar os componentes da biblioteca gráfica para que os gráficos fossem feitos.

Conclusão: deve-se pensar em programação sempre considerando se um trecho de programa é de aplicação genérica ou de aplicação específica. Caso se perceba que uma funcionalidade do programa é de aplicação genérica, essa funcionalidade é candidata a um componente.

No exemplo acima, as funções de uso do sensor são (a princípio) específicas. São também específicas os detalhes de como os dados devem ser mostrados no gráfico. Mas as funções que desenham gráficos são genéricas, pois desenham qualquer coisa. Nesse sentido, faz sentido colocar as funções gráficas como componentes numa biblioteca, que poderia se chamar `grafico`. Qualquer novo problema que precise de gráficos poderia usar esses mesmos componentes.

Caso o sensor que se está usando seja ligado a uma placa de aplicação genérica acoplada ao computador, poderá ser uma boa idéia que se faça uma biblioteca também para as funções de uso desta placa. Assim, qualquer outro projeto que precise usar a tal placa com sensor poderá partir de funções de uso básico do sensor.

Não se esqueça: *programar é usar e desenvolver bibliotecas*. Portanto, antes de se iniciar um esforço de programação, pense se o problema em questão é genérico ou específico. Caso seja genérico, procure por uma biblioteca pronta, que possivelmente existe. Caso não exista uma, ou a que exista não seja satisfatória, considere desenvolver uma biblioteca para o problema genérico em questão. Assim, a próxima pessoa que pensar nesse assunto possa aproveitar parte do seu esforço de programação.

É importante também que se saiba encomendar um serviço de programação. Para um certo problema pode-se solicitar a terceiros (um profissional autônomo, uma empresa, etc) que se desenvolva uma biblioteca de forma a atender as necessidades do problema, contendo funções básicas. A partir de uma biblioteca, torna-se relativamente fácil fazer “em casa” (isto é, dentro da instituição em que se trabalha) a *customização* do programa que se está desenvolvendo. O termo *customização* é um inglesismo que vem de *customization*, isto é, fazer alguma coisa de acordo com o gosto do *custom* (cliente, freguês). Portanto, *customizar* um programa significa implementar ao detalhe as solicitações de quem solicita o trabalho. Por exemplo: escrever as telas de entrada e saída de dados, efeitos de multimedia, etc.

Pense neste exemplo: há uma placa com um sensor que se deseja usar. Pode-se solicitar a terceiros que desenvolva uma biblioteca, com fonte, contendo funções básicas de uso da placa com sensor — seleção do sensor, inicialização do sensor, leitura do sensor, etc. Tudo feito com tutorial ensinando a usar as funções desenvolvidas. A partir das funções da biblioteca, seria em princípio fácil se desenvolver um programa para usar a placa com sensor.

6.2 Desenvolvimento e utilização de componentes

Para que se possa desenvolver e utilizar componentes, é preciso se pensar se um problema a ser resolvido é específico ou genérico. É muito comum que um problema possa ser dividido em uma componente específica e uma componente genérica.

Por exemplo: Um programa precisa fazer uns gráficos a partir de dados que vem de um sensor. Os detalhes de como o gráfico deve ser (valor mínimo e máximo do eixo das abcissa e da ordenada, tipo

do gráfico, etc) são um problema específico. Mas o traçado do gráfico em si é um problema genérico. Esse programa provavelmente pode ser resolvido com o seguinte ordenação:

1. Adquire-se os dados.
2. Chama-se a rotina gráfica (genérica) com parâmetros que tornem o gráfico exatamente como a especificação manda.

Como o mercado de software está muito desenvolvido, é bem possível que qualquer coisa que seja de interesse geral (como o traçado de um gráfico) já tenha sido desenvolvida e que haja várias bibliotecas disponíveis. Geralmente há bibliotecas algumas gratuitas e outras pagas. Inicialmente as gratuitas eram simples demais ou careciam de um manual ou help mais elaborado, e as pagas eram as que (supostamente) seriam completas e com suporte melhor. Ultimamente, contudo, está se observando um fenômeno que alguns chamam de “software livre”¹¹.

Esse fenômeno se caracteriza pelo aparecimento de vários grupos que estão literalmente dando o seu trabalho de programação. O nível de qualidade do software desses grupos está crescendo muito, assim como o número de usuários. Em vários casos, o grupo dá o software e o texto fonte. Com isso, não há realmente nenhuma razão para não se experimentar usar o tal software e isso permite que o número de usuários cresça rápido. Os usuários logo se organizam em “tribos” para trocar informações, e isso corresponde a existência de um bom suporte para o produto.

Para o programador (ou para quem coordena programadores), é importante saber sobre o “software livre” para que se possa alocar otimamente o esforço de programação. O uso de uma biblioteca pronta, feita por um grupo externo ao seu, *pode* ser uma boa alternativa. Mas também pode não ser. A biblioteca é boa se ela resolve o problema em questão e se é sabido como usa-la (aliás, como qualquer software). É preciso que haja sentimento para decidir se é melhor tentar procurar por uma biblioteca pronta, aprender a usa-la, testa-la, etc, ou se é melhor desenvolver a biblioteca internamente, para que as especificações sejam atendidas perfeitamente. Essa é uma decisão difícil.

6.3 Programação estruturada

Para facilitar o desenvolvimento e uso de componentes reaproveitáveis de código, surgiu o conceito de programação estruturada. A programação estruturada consiste basicamente de se estimular o programador a usar “estruturas”, isto é, classes (`class` ou `struct`, veja sobre uso de classes na seção 7.4) que contém um conjunto de variáveis. Por exemplo: uma classe “pessoa” pode conter vários campos como nome, telefone, endereço, etc.

Com essa filosofia de trabalho, o programador iniciaria o seu trabalho definindo como devem ser as estruturas (tipos, classes) necessárias para se resolver um problema. Isto é, quais exatamente devem ser os campos que a classe “pessoa” deve ter, por exemplo. Em seguida, cuida de desenvolver um conjunto de funções para processar variáveis do tipo “pessoa”.

A programação estruturada já é um grande avanço para o aumento da produtividade da programação. Mas a experiência mostrou que alguns conceitos ainda faltavam. O desenvolvimento da tecnologia de software acabou produzindo um conjunto de conceitos ainda mais evoluídos que a programação estruturada. O uso desses novos conceitos ficou conhecido como programação “orientada a objeto”. Desde que esse texto vai abordar diretamente os conceitos de orientação a objeto, creio que não convém “perder tempo” comentando sobre programação estruturada. É melhor ir direto para a programação orientada a objeto, que é mais fácil e mais poderosa.

¹¹ Em inglês é “free software”, que significa ao mesmo tempo “software livre” e “software gratuito”.

Capítulo 7) Programação orientada a objeto

7.1 Introdução

7.1.1 Abstração do software

A programação orientada a objeto é um passo a mais na direção do aumento da abstração na tecnologia de desenvolvimento de programas. E a programação orientada a objeto não é a última novidade nessa direção. Veja a figura abaixo.

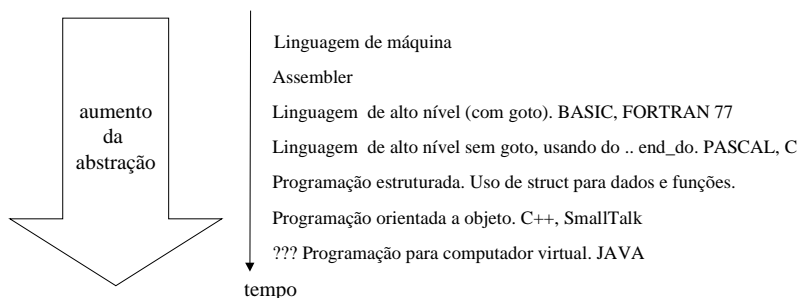


Figura 7: Aumento do nível de abstração da tecnologia de desenvolvimento de software.

Com o aumento da abstração ...

1. consegue-se desenvolver programas maiores e mais complexos. 😊
2. o esforço de programação torna-se mais eficiente. 😊
3. reaproveita-se mais facilmente componentes já desenvolvidos. 😊
4. o tamanho do código executável fica maior. ☹
5. a velocidade de execução dos programas geralmente fica um pouco menor. ☹

Como se vê, o aumento da abstração não é uma estratégia que traz apenas vantagens. Mas o que se percebe é que o mercado deseja muito obter as vantagens do aumento da abstração e não se incomoda muito com as desvantagens. Portanto o aumento da abstração tem se mostrado uma direção adequada para a tecnologia de software.

De fato, a primeira das vantagens listadas acima é um dos grandes determinadores da estratégia a adotar no desenvolvimento de software. Atualmente, com o fabuloso desenvolvimento do hardware, os computadores são tão velozes, têm tanta memória e tanto disco que proporcionalmente quem é pequeno é a nossa capacidade de usar tanto poder. Um hardware atual é potente suficiente para resolver problemas complexos, que requerem programas grandes.

Por exemplo: se um certo programa está sendo executado com muita lentidão, pode-se pensar em comprar um computador com CPU mais rápida ou com mais memória e com isso resolve-se o problema rapidamente. Mas se esse programa precisa ser alterado (isso sempre acontece . . .) e a tecnologia do programa usa baixo nível de abstração e o programa é grande, pode ser uma grande dor de cabeça conseguir implementar na prática as alterações solicitadas. Em muitos casos é requerido uma quantidade enorme de homem-hora para solucionar um problema desses e isso equivale a perder dinheiro.

“C / C++ e Orientação a Objetos em Ambiente Multiplataforma”, versão 5.1

Esse texto está disponível para download em www.del.ufrj.br/~villas/livro_c++.html

7.1.2 Mudança de nomenclatura

Uma das confusões na compreensão de programação orientada a objetos é o fato de a nomenclatura mudar em relação ao que geralmente se usa na programação procedural. Na tabela abaixo encontram-se as principais mudanças de nomenclatura

Programação procedural	Programação orientada a objetos
função	método
ocorrência (instância)	objeto
tipo	classe

7.1.3 Objetos

A programação estruturada enfoca a criação de estruturas (tipos) do usuário e de funções para manipular essas estruturas. Na programação orientada a objeto, as funções que manipulam os dados ficam dentro da mesma estrutura que contém os dados. Essa nova estrutura que contém dados e funções passou a ser chamada de objeto.

O fundamento da programação orientada a objetos é o uso de estruturas com dados e funções (os objetos), o conceito de herança e de polimorfismo, como mostrado posteriormente

7.2 Introdução a análise orientada a objetos

A idéia da análise orientada a objeto é de facilitar a classificação de tipos e com isso dar maior flexibilidade à manutenção desta classificação. Esta seção discutirá o seu aspecto conceitual (um pouco diferente do implementacional) com um exemplo.

Seja uma análise de um quadro funcional de uma empresa, onde se pode separar os integrantes em 4 classes - presidente, gerente financeiro, gerente comercial e secretário. As características deles estão no quadro central da figura abaixo e os métodos que eles exercem estão no quadro inferior. Por exemplo: o presidente tem por característica “salário e “tem_carro”, e seu método é “decide”.

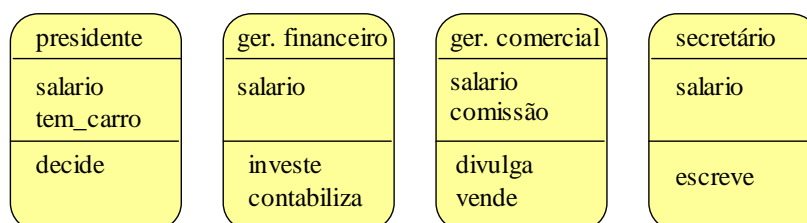


Figura 8: Visão gráfica das entidades (classes) de uma empresa

Como todos têm salário, pode-se conceber uma classe que possua esta característica, chamada funcionário por exemplo, que levará esta propriedade às demais por herança. Neste caso a figura ficará como no quadro abaixo, já assinalado com as ocorrências das classes.

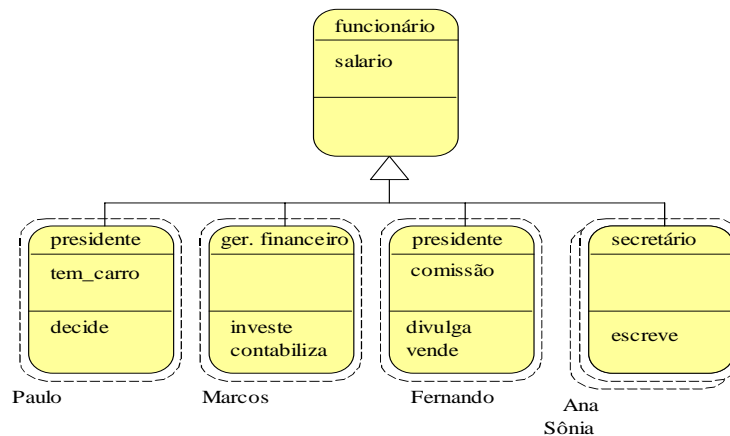


Figura 9: Visão gráfica das entidades (classes) de uma empresa (2)

Em mais um passo na manutenção da classificação do quadro funcional, pode-se imaginar o crescimento da empresa e a necessidade de se criar uma nova classe que seria um gerente financeiro trainee, que faz o que o gerente financeiro faz e ainda estuda. O nosso quadro de classificação ficará assim:

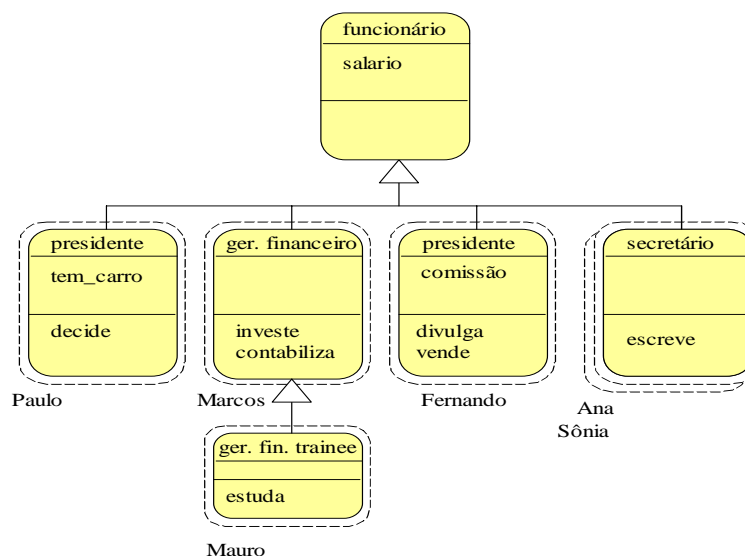


Figura 10: Visão gráfica das entidades (classes) de uma empresa (3)

Objetivamente, pode-se dizer que a linguagem orientada a objeto oferece os seguintes conceitos de programação a mais (serão explicados):

1. Polimorfismo.
2. Encapsulamento de dados e métodos, isto é, uso de classes.
3. Herança (*inheritance*).

7.3 Polimorfismo

Polimorfismo significa, pelo radical da palavra, “muitas formas”. Em programação quer dizer que o mesmo nome pode ser dado a funções diferentes, desde que o contexto possa discernir de qual se trata pelo *prototype*. Por exemplo: uma função de inicialização de variáveis globais pode preencher com zeros (se não é passado parâmetro) ou com o valor do parâmetro passado. Veja !

Exemplo:

```
int x1,x2,x3; // variáveis globais a inicializar
void inicializa() { // chamada sem parâmetros
    x1=x2=x3=0; // preenche variáveis com zeros
}
void inicializa(int value) { // chamada com parâmetro
    x1=x2=x3=value; // preenche variáveis com parâmetro passado
}
```

Neste caso, a função `inicializa` pode ter dois códigos diferentes. O compilador determina em tempo de compilação qual é a que será usada pelo *prototype*. Por exemplo:

```
inicializa(); // chama a função de cima
inicializa(3); // chama a função de baixo
```

7.3.1 Argumento implícito (*default argument*)

Neste caso específico, as duas funções poderiam ser uma só, com a inicialização na linha de *prototype* do código da função. A inicialização de `value` só ocorre se não se passar parâmetro. Isto se chama argumento implícito (*default argument*). Veja:

Exemplo:

```
void inicializa(int value=0) { // chamada sem parâmetro implica value=0
    x1=x2=x3=value; // preenche variáveis com value
}
```

7.4 Uso de classes

É o mais importante conceito introduzido pela POO. O *objeto* é definido como: “uma ocorrência de uma entidade lógica, contendo dados e funções que manipulam estes dados.” Veja o exemplo:

Exemplo:

```
class myclass { // define a classe (entidade)
public:
    double d; // a classe pode ter campos de dados
    void fun(){ /* ... */ }; // a classe pode ter funções (métodos)
};

void main() {
    myclass myobj; // myobj é uma ocorrência da classe myclass,
                  // ou seja, é um objeto
    myobj.fun(); // o objeto myobj pode chamar um método (função) membro
}
```

Dentro de um objeto, cada dado e método pode ser privado (`private`), o que protege os objetos contra acessos acidentais ou incorretos não previstos originalmente. Este tipo de proteção de dados e código é freqüentemente chamada de encapsulamento.

O objeto é, para todos os propósitos e finalidades, uma ocorrência (*instance*) de um tipo definido pelo usuário, em outras palavras: é uma variável. Pode parecer estranho para programadores tradicionais aceitar que um variável possa conter métodos (código executável) além de dados, mas é exatamente isso que ocorre na POO.

Um método de um objeto é uma função membro (*member*) o função amigas (*friend*), funções estas que podem ser chamadas de métodos do objeto. A forma de se chamar uma função membro de um objeto é sintaticamente semelhante ao acesso a um dado, usando o "." entre o nome da ocorrência e o nome do dado ou função membro.

Exemplo:

```
mytype myinstance; // define 1 ocorrência do tipo mytype
a = myinstance.i;  // acesso a dado do objeto
myinstance.fun();  // chama método do objeto
```

Os objetos podem ser manipulados por funções em geral sendo passados como parâmetros ou sendo retornados. Para estas funções, a noção de `public`, `private` e `protected` está presente, ou seja, não se pode acessar os dados e funções protegidos. Já para funções amigas (com `prototype` devidamente declarado na definição da classe) é permitido o acesso a código e dados privados da classe.

7.5 Herança

A herança é o processo pelo qual um objeto adquire (herda) propriedades de outro, conforme foi analisado conceitualmente no caso da empresa. Isto é muito importante pois abarca a noção de classificação. Por exemplo: um fusca é um automóvel (*automobile*) e um automóvel é um veículo (*vehicle*); um caminhão (*truck*) também é um veículo e um FNM é um caminhão. Sem a classificação e a herança, cada objeto (as palavras sublinhadas) teria de ser definido por todas as suas propriedades. Fusca e FNM são ocorrências (instâncias) de automóvel e caminhão, respectivamente. Vejamos um diagrama de classificação para este caso.

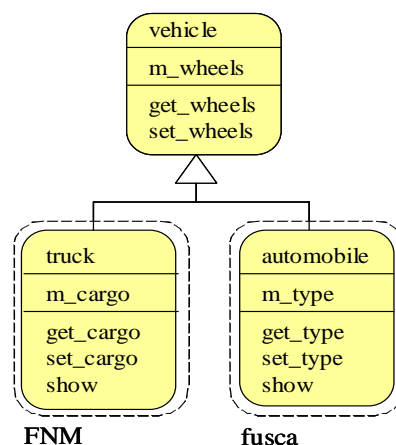


Figura 11: Visão gráfica das entidades (classes) para carros

Vejamos agora uma implementação disto em C++:

Exemplo:

```
#include <iostream.h>
// the base class and its base properties
class vehicle {
    int m_wheels;
public:
    void set_wheels(int wheels) { m_wheels = wheels; };
    int get_wheels() { return m_wheels; };
};

// the first derived class and its specific additional properties
class truck : public vehicle {
    int m_cargo;
public:
    void set_cargo(int cargo) { m_cargo = cargo; };
    int get_cargo() { return m_cargo; };
    void show() { cout << "wheels = " << get_wheels() << endl
                    << "cargo = " << get_cargo() << endl; };
};
```

```
};

// the second derived class and its specific additional properties
enum a_type {trail, city, city_luxury};
class automobile : public vehicle {
    a_type m_auto_type;
public:

    void set_type(a_type auto_type) { m_auto_type = auto_type; };
    a_type get_type() { return m_auto_type; };
    void show() { cout << "wheels = " << get_wheels() << endl
                      << "type = " << get_type() << endl; };
};

void main() {
    truck FNM;
    automobile fusca;
    FNM.set_wheels(18);
    FNM.set_cargo(3200);
    FNM.show();
    fusca.set_wheels(4);
    fusca.set_type(city);
    fusca.show();
}
```

Resultado:

```
wheels = 18
cargo = 3200
wheels = 4
type = 1
```

Caso seja interessante definir o tipo de freio (*breaks*) e o tipo de motor do veículo, se poderia acrescentar tais características na classe veículo e elas passariam a pertencer também às classes caminhão e automóvel. Mas poderia ser também que seu quadro de classificação já possuísse as classes freio e motor e que você quisesse aproveitá-las. Neste caso, a classe veículo herdaria de mais de uma classe, o que se chama de herança múltipla.

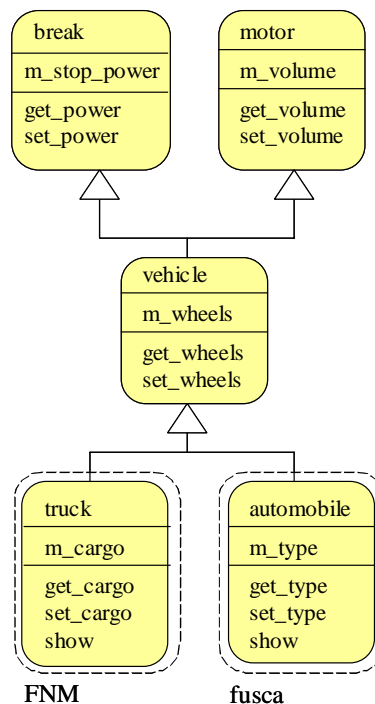


Figura 12: Visão gráfica das entidades (classes) para carros (2)

Para implementar isto em C++ a partir do código anterior, basta definir as novas classes e fazer a classe veículo herdar delas. Para ter graça, os comandos show deverão mostrar também as novas características. Heis o programa completo.

Exemplo:

```
#include <iostream.h>
class breaks {
    float m_stop_power;
public:
    void set_spower(float stop_power) { m_stop_power = stop_power; };
    float get_spower() { return m_stop_power; };
};

class motor {
    float m_volume;
public:
    void set_volume(float volume) { m_volume = volume; };
    float get_volume() { return m_volume; };
};

// the base class and its base properties
class vehicle : public breaks, public motor {
    int m_wheels;
public:
    void set_wheels(int wheels) { m_wheels = wheels; };
    int get_wheels() { return m_wheels; };
};

// the first derived class and its specific additional properties
class truck : public vehicle {
    int m_cargo;
public:
    void set_cargo(int cargo) { m_cargo = cargo; };
    int get_cargo() { return m_cargo; };
    void show() { cout << "wheels = " << get_wheels() << endl
                    << "stop power = " << get_spower() << endl
                    << "motor volume = " << get_volume() << endl
                    << "cargo = " << get_cargo() << endl; };
};

// the second derived class and its specific additional properties
enum a_type {trail, city, city_luxury};
class automobile : public vehicle {
    a_type m_auto_type;
public:
    void set_type(a_type auto_type) { m_auto_type = auto_type; };
    a_type get_type() { return m_auto_type; };
    void show() { cout << "wheels = " << get_wheels() << endl
                    << "stop power = " << get_spower() << endl
                    << "motor volume = " << get_volume() << endl
                    << "type = " << get_type() << endl; };
};

void main() {
    truck FNM; automobile fusca;
    FNM.set_wheels(18);
    FNM.set_spower(333.3);
    FNM.set_volume(5.5);
    FNM.set_cargo(3200);
    FNM.show();
    fusca.set_wheels(4);
}
```

```

fusca.set_spower(100.2);
fusca.set_volume(1.5);
fusca.set_type(city);
fusca.show();
}

```

Resultado:

```

wheels = 18
stop power = 333.3
motor volume = 5.5
cargo = 3200
wheels = 4
stop power = 100.2
motor volume = 1.5
type = 1

```

7.6 Sobrecarga de operadores

Quando se escreve um código com uma operação, o compilador internamente converte esse código pela chamada de uma função correspondente. Por exemplo:

```

a=b; // a.operator=(b);
a=b+c; // a.operator=(b.operator+(c));

```

Em C++, pelo princípio de polimorfismo, pode-se atribuir um comportamento qualquer a uma operador da lista abaixo. A palavra “sobrecarregar” abaixo deve ser entendida como “escrever uma função do usuário para operator...”.

Operadores de C++ que podem ser sobrecarregados

+	-	*	/	%	^
&		~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=
>>=	==	!=	<=	>=	&&
	++	--	,	->*	->
()	[]	new	delete		

Exemplo:

```

#include <iostream.h>
class vetor2 {
public:
    float f[2]; // array com 2 float
    vetor2(){}; // construtor sem parâmetros
    // sobrecarrega operator+ - ret = a+b - ret = operator+(a,b)
    friend vetor2 operator+(vetor2 & a, vetor2 & b) {
        vetor2 ret;
        ret.f[0] = a.f[0] + b.f[0];
        ret.f[1] = a.f[1] + b.f[1];
        return ret;
    };
    // sobrecarrega insersor (operator<<)
    friend ostream & operator<<(ostream & stream, const vetor2 &obj) {
        stream << "f[0]=" << obj.f[0] << endl << "f[1]=" << obj.f[1] << endl;
        return stream;
    };
}; // end of class vetor2

void main() {
    vetor2 a, b, ret;
    a.f[0]=1.1; a.f[1]=1.2; b.f[0]=2.5; b.f[1]=2.6; // carrega dados

    ret = a+b; // ret = a.operator+(b), ou ret.operator=(a.operator+(b))

    cout << "a" << endl << a; // imprime a
}

```

```
cout << "b" << endl << b;    // imprime b
cout << "ret" << endl << ret; // imprime ret
}
```

Resultado:

```
a
f[0]=1.1
f[1]=1.2
b
f[0]=2.5
f[1]=2.6
ret
f[0]=3.6
f[1]=3.8
```

7.7 Construtor e destrutor de um objeto

Os objetos, quando são definidos, podem ser automaticamente inicializados por funções conhecidas como construtores. Geralmente cabe a estas funções a responsabilidade de alocar memória, inicializar variáveis, abrir arquivos, etc. Pode ocorrer de os construtores serem mais de um (pelo princípio de polimorfismo), de maneira a possibilitar que o objeto seja construído com os dados disponíveis no momento.

O destrutor é uma função chamada quando um objeto sai do escopo, ou é deletado com `delete` (no caso de alocação dinâmica). Sua tarefa é limpar o ambiente de trabalho do objeto, o que geralmente significa desalocar memória, fechar arquivos ou sinalizar a outros objetos que está sendo destruído.

O construtor tem o mesmo nome de sua classe e não retorna. O destrutor tem o mesmo nome da classe precedido por `~`, não recebe parâmetros e não também não retorna.

Exemplo:

```
#include <iostream.h>
class foo {
    double m_d;
    int m_i;
public:
    foo();           // Construtor sem parametros
    foo(int)         // Construtor com um parâmetro
    foo(int,double); // Construtor com dois parâmetro
    ~foo();          // Destrutor
};
foo::foo() {
    cout << "Construtor sem parâmetros" << endl;
}
foo::foo(int i) {
    cout << "Construtor com 1 parâmetro" << endl;
    m_i=i;
}
foo::foo(int i, double d) {
    cout << "Construtor com 2 parâmetros" << endl;
    m_i=i;
    m_d=d;
}
foo::~~foo(){
    cout << "destrutor" << endl;
}
void m() {
    foo f1;
    foo f2(1);
    foo f3(1,3.14);
}
void main() {
```

```
m();
}
```

Resultado:

```
Construtor sem parâmetros
Construtor com 1 parâmetro inteiro
Construtor com 1 parâmetro inteiro e um double
destrutor
destrutor
destrutor
```

7.7.1 Default constructor (construtor implícito)

Um construtor especial que existe num objeto é o default construtor, que é o construtor que não tem parâmetros. Se uma classe não define explicitamente construtor algum o compilador cria um construtor implícito (isto é, sem parâmetros), que não faz nada. Algo como mostrado abaixo.

```
myclass() {}; // empty default constructor
```

O default constructor é o construtor chamado quando se declara um objeto sem nenhum parâmetro no construtor, como é muito usual (mostrado abaixo).

```
void f() {
    myclass myObj; // default constructor called
}
```

Caso uma classe declare um construtor implícito, o construtor declarado é usado no lugar do construtor vazio que o compilador usaria. Mas atenção: caso a classe tenha um construtor com parâmetros e não declare explicitamente um construtor implícito, então torna-se erro de compilação declarar um objeto sem parâmetros no construtor.

```
class myclass {
    int m_i;
public:
    myclass(int i) { m_i = i; } // non-default constructor defined
};

void f() {
    myclass myObj; // error: no appropriate default constructor available
}
```

Esse erro de compilação poderia ser evitado (caso seja conveniente para a lógica de programação em questão) com a introdução de um default constructor explicitamente, como mostrado abaixo.

```
class myclass {
    int m_i;
public:
    myclass () {}: // explicit definition of default constructor
    myclass(int i) { m_i = i; } // non-default constructor defined
};

void f() {
    myclass myObj; // OK
}
```

7.7.2 Ordem de chamada de construtor e destrutor para classes derivadas

Caso haja uma classe derivada de uma classe base, sendo ambas com construtor e destrutor, como é de se esperar,

```
#include <iostream.h>
class base {
public:
    base () { cout << "constructor base" << endl; }
    ~base () { cout << "destructor base" << endl; }
};
```

```

class derived : public base {
public:
    derived () { cout << "constructor derived" << endl; }
    ~derived () { cout << "destructor derived" << endl; }
};

class other {
public:
    other () { cout << "constructor other" << endl; }
    ~other () { cout << "destructor other" << endl; }
    base b;
};

void main () {
    derived a;
    cout << "----" << endl;
    other o;
    cout << "----" << endl;
}

```

A saída do programa é como mostrado abaixo.

```

constructor base
constructor derived
---
constructor base
constructor other
---
destructor other
destructor base
destructor derived
destructor base

```

7.7.3 Inicialização de atributos com construtor não implícito

Desde que uma classe pode ter mais de um construtor (usando polimorfismo) pode-se querer controlar qual construtor uma classe derivada chamará. Implicitamente o compilador tentará sempre usar o construtor sem parâmetro; para forçar a chamada de construtor com parâmetro, basta declara-lo na sequência do construtor da classe derivada separado por `:`. Veja o exemplo.

Exemplo:

```

#include <iostream.h>
class base {
public:
    int m_i;
    base() {m_i=0;};           // default constructor
    base(int i) {m_i=i;};      // constructor with one parameter
}; // end of base class

class derived : public base { // inherits from base class
    int m_h;
public:
    derived(){m_h=0;};         // default constructor
    // constructor with one parameter calls non-default constructor of base class
    derived(int h) : base (h) {m_h=h;};
    void show() {
        cout << "m_i = " << m_i << ", m_h = " << m_h << endl;
    };
};

class A {
    base base_obj,base_obj2; // one attribute of the base class
public:
    A(int z=0) : base_obj(z), base_obj2(z+1) {};
}

```

```

// default constructor and a constructor with one default parameter
// this class has an attribute of a class that has non-default constructor
void show() {
    cout << "m_i = " << base_obj.m_i << ", m_i2 = " << base_obj2.m_i << endl;
};

};

void main () {
    derived d1;
    derived d2(2);
    A d3(3);

    d1.show();
    d2.show();
    d3.show();
}

```

Resultado:

```

m_i = 0, m_h = 0
m_i = 2, m_h = 2
m_i = 3, m_i2 = 4

```

7.7.4 Copy constructor (construtor de cópia)

O copy constructor é um construtor que tem o protótipo abaixo.

```
myclass(const myclass & obj); // prototype of copy constructor
```

O copy constructor é chamado automaticamente nas seguintes situações

- Quando um objeto é instanciado e imediatamente atribui-se a ele um valor

```

// exemplo
myClass myObj2 = myObj; // call copy constructor

```

OBS: Quando o objeto é atribuído após a sua instanciação, chama-se o copy constructor e em seguida o método operator=.

- Quando chamado explicitamente

```

// exemplo
myClass myObj3(myObj); // explicit call to copy constructor

```

- Na passagem de um objeto como parâmetro para uma função
- No retorno de um objeto de uma função

Caso uma classe não tenha um construtor de cópia definido explicitamente, o compilador cria automaticamente um construtor de cópia para ser usado quando necessário. O construtor de cópia do compilador copia byte a byte os atributos de um objeto na sua cópia. Esse procedimento resolve muitos problemas mas não todos. Por exemplo: se um objeto aloca memória, o atributo do objeto é o ponteiro para a memória alocada. A cópia byte a byte dos atributos não resulta na cópia da região de memória alocada. Portanto, considerando que cada objeto deve ter sua própria região de memória alocada, é preciso escrever um construtor de cópia adequado para um objeto que aloca memória.

O código abaixo mostra claramente os momentos em que chama-se o copy constructor, o default constructor, e o operator=. Repare que a cópia da referência (lvalue) ou do ponteiro, como mostrado nas duas últimas linhas do código, não implica na chamada do construtor de cópia nem do operator=.

```

#include <iostream.h>
class myClass {
public:
    myClass () { // default constructor
        static int count = 0;
        cout << "default constructor (" << ++count << ")" << endl;
    };
};

```

```

    myClass(const myClass & obj) {
        static int count = 0;
        cout << "copy constructor (" << ++count << ")" << endl;
    }
    void operator=(myClass obj) {
        static int count = 0;
        cout << "operator= (" << ++count << ")" << endl;
    };
};

void userFunction(myClass a)
// myClass a = myObj; call copy constructor
{ /* do nothing */ }

myClass userFunction2() {
    myClass a; // default constructor (2)
    return a; // myClass returned_object = a; call copy constructor
}

void main() {
    myClass myObj; // default constructor (1)
    myClass myObj2 = myObj; // call copy constructor (1)
    myObj2 = myObj; // call copy constructor (2) and operator= (1)
    myClass myObj3(myObj); // explicit call to copy constructor (3)
    userFunction(myObj); // myClass as parameter, call copy constructor (4)
    myObj2 = // call operator= (2)
        userFunction2(); // myClass as return object, call copy constructor (5)
    myClass & l_obj = myObj; // don't call any of these. Reference copy only.
    myClass * p_obj = &myObj; // don't call any of these. Pointer copy only.
}

```

A saída do programa é como mostrado abaixo.

```

default constructor (1)
copy constructor (1)
copy constructor (2)
operator= (1)
copy constructor (3)
copy constructor (4)
default constructor (2)
copy constructor (5)
operator= (2)

```

7.8 lvalue

Um *objeto* é uma região de memória onde pode-se armazenar dados e/ou funções. Um *lvalue* (o termo é a abreviação *left value*, isto é “valor a esquerda”) é uma expressão referindo-se a um objeto ou função. Um exemplo óbvio de lvalue é o nome de um objeto.

Numa operação tipo $a = b$, é necessário que a seja um lvalue. A generalização desse conceito é a seguinte: É necessário que o identificador a esquerda de um operador de designação seja um lvalue. Os operadores de designação (*assignment operator*) são os seguintes:

```

operator=      operator*=      operator/=      operator%=      operator+=
operator-=      operator>>=      operator<<=      operator&=      operator^=      operator|=

```

Alguns operadores retornam lvalues. Por exemplo o `operator()`, quando é definido, geralmente retorna um lvalue. No exemplo abaixo, uma classe simples de vetor de inteiros é definida. Essa classe `int_vector` possui o método `operator()` que retorna `int &`, isto é, um lvalue de `int`. No caso, o retorno de `operator()` é um ponteiro para inteiro que aponta para o array interno da classe, na posição indicada pelo parâmetro de entrada. Um objeto dessa classe pode chamar `operator()` estando tanto a direita quanto a esquerda de um operador de designação (*assignment operator*).

```
#include <iostream.h>

class int_vector {
    int m_a[10];
public:
    int & operator()(int n) {
        return m_a[n];
    }
};

void main () {
    int_vector obj;
    obj(3) = 4;
    int k = obj(3);
    cout << k << endl;
}
```

Resultado

4

Seja agora uma variação do programa acima, apenas retirando o “&” do retorno do método `operator()`. Nesse caso, o retorno é um `int`, ou seja, não é um `lvalue`. Portanto, esse operador pode estar do lado direito de um operador de designação, mas não do lado esquerdo.

```
#include <iostream.h>

class int_vector {
    int m_a[10];
public:
    int operator()(int n) { // essa linha foi
                           // modificada em relação ao programa anterior

        return m_a[n];
    }
};

void main () {
    int_vector obj;
    obj(3) = 4; // erro de compilação !
    int k = obj(3);
    cout << k << endl;
}
```

7.9 Encapsulamento de atributos e métodos

A idéia básica de encapsular dados e métodos é definir cuidadosamente a *interface* de um objeto. Define-se como interface de um objeto o conjunto de atributos e métodos aos quais se tem acesso. Quando se descreve uma classe, pode-se definir 3 níveis de acesso aos atributos e aos métodos:

- `public` – o acesso é permitido para métodos da classe em questão, para classes que herdem da classe em questão e para usuários do objeto.
- `private` – o acesso é permitido apenas para métodos da classe em questão. O usuário de objetos dessa classe não podem acessar um atributo privado diretamente. Classes que herdam propriedades da classe em questão também não podem acessar atributos privados.
- `protected` – o acesso é permitido para métodos da classe em questão, e para classes que herdem dela. Mas o usuário não pode acessar atributos protegidos diretamente.

Para um programador iniciante, pode haver a idéia que tornar todos os atributos como públicos sempre é uma boa técnica de programação. Contudo, a experiência mostra que há necessidade de se proteger o acesso a certos métodos e atributos.

Quando se projeta uma classe ou um conjunto de classes, deve-se pensar em como se pretende que um programador irá usar essas classes. Em princípio, deve-se permitir o acesso somente a atributos e métodos que sejam de interesse, para permitir que a classe cumpra a sua finalidade. Tudo o que não for necessário para o programador deve ficar fora da interface do objeto. Como regra geral, costuma ser boa programação colocar atributos como privados, e oferecer métodos públicos de acesso aos atributos privados, como no exemplo abaixo.

```
#include "vblib.h" // VBString

class person {
    // private attributes
    VBString m_name;
    VBString m_address;
    VBString m_telephone;
    float m_weight;
public:
    // public access methods
    void setName(VBString name) { m_name = name; };
    void setAddress(VBString address) { m_address = address; };
    void setTelephone(VBString telephone) { m_telephone = telephone; };
    void setWeight(float weight) { m_weight = weight; };

    VBString getName() { return m_name; };
    VBString getAddress() { return m_address; };
    VBString getTelephone() { return m_telephone; };
    float getWeight() { return m_weight; };
};

void main () {
    person p1;
    p1.setName("Sergio");
    p1.setAddress("Rua x, número y");
    p1.setTelephone("222-3344");
    p1.setWeight(89.5);

    // ...

    VBString userName = p1.getName();
    VBString userAddress = p1.getAddress();
    VBString userTelephone = p1.getTelephone();
    float userWeight = p1.getWeight();
}
```

O programa acima não faz nada útil, sendo apenas um programa de exemplo. Existe uma classe chamada `person`, que possui 4 atributos privados. Na interface dessa classe, há 8 métodos públicos, para ler e escrever em cada um dos atributos privados da classe.

Qual seria a vantagem de se encapsular (tornar privado) os atributos da classe, já que os métodos tornam-nos públicos de qualquer forma? A vantagem é que tornando os atributos acessíveis apenas através de métodos, com certeza sabe-se que o usuário da classe não irá acessar diretamente os atributos, mas o fará usando os métodos de acesso. Assim, caso surja uma necessidade de upgrade no software (isso sempre ocorre, a experiência o mostra), em que seja necessário, digamos, fazer uma estatística de quantas vezes é acessado o atributo `m_name`. Isso pode ser feito apenas mudando a descrição interna da classe, sem mudar a interface.

No exemplo abaixo, que é um upgrade do exemplo anterior, a classe `person` ganhou um novo atributo chamado `m_getNameTimes`. A classe ganhou também um construtor padrão, chamado automaticamente no momento da criação do objeto, que atribui zero a `m_getNameTimes`. O método `getName` foi alterado internamente, sem que sua funcionalidade fosse alterada, de forma a incrementar

`m_getNameTimes` antes de retornar o valor solicitado. Um novo método foi criado para ler o valor de `m_getNameTimes`. Repare que um trecho do programa principal não foi alterado. Esse trecho poderia ser muito grande num exemplo real de sistema. Caso não existisse a opção de encapsulamento, nada impediria que o usuário da classe `person` acessasse diretamente o atributo `m_name`, e com isso seria impossível esse tipo de upgrade.

```
#include "vblib.h"

class person {
    // private attributes
    int m_getNameTimes;
    VBString m_name;
    VBString m_address;
    VBString m_telephone;
    float m_weight;
public:
    // public access methods
    void setName(VBString name) { m_name = name; };
    void setAddress(VBString address) { m_address = address; };
    void setTelephone(VBString telephone) { m_telephone = telephone; };
    void setWeight(float weight) { m_weight = weight; };

    VBString getName() { m_getNameTimes++; return m_name; };
    VBString getAddress() { return m_address; };
    VBString getTelephone() { return m_telephone; };
    float getWeight() { return m_weight; };

    int getNameTimes() {return m_getNameTimes; };

    // default constructor
    person() { m_getNameTimes = 0; }
};

void main () {
    person p1;
    p1.setName("Sergio");
    p1.setAddress("Rua x, número y");
    p1.setTelephone("222-3344");
    p1.setWeight(89.5);

    // ...

    VBString userName = p1.getName();
    VBString userAddress = p1.getAddress();
    VBString userTelephone = p1.getTelephone();
    float userWeight = p1.getWeight();

    cout << "No objeto p1, o campo name foi lido " << p1.getNameTimes()
         << " vezes" << endl;
}
```

7.10 União adiantada × união atrasada (*early bind* × *late bind*)

Um ponteiro para uma classe base pode apontar para todas as respectivas classes derivadas (i.e. que herdem da classe base). Veja o exemplo abaixo.

```
#include <iostream.h>

class baseClass {
public:
    void printout() {
        cout << "baseClass" << endl;
    }
};
```

```

    }
};

class derivedClass : public baseClass {
public:
    void printout() {
        cout << "derivedClass" << endl;
    }
};

void main () {
    baseClass *myPointer;    // pointer to base class
    derivedClass myObj;      // object of derived class
    myPointer = &myObj;      // pointer to base class points to derived object
    myPointer->printout();    // (using pointer) base class method
    myObj.printout();         // (using object) derived class method
}

```

Resultado

```

baseClass
derivedClass

```

No caso do exemplo acima, a união do ponteiro base com a classe derivada é chamado de “união adiantada” (*early bind*). O tempo adiantado (*early*) se refere ao fato de que a união ocorre em tempo de compilação. Mas é possível um outro tipo de “união atrasada” (*late bind*), que ocorre em tempo de execução. Em C++ programa-se a união atrasada com a palavra reservada “virtual”. Considere agora o mesmo código acima, apenas acrescentando a palavra “virtual” no local indicado na classe base.

```

class baseClass {
public:
    virtual void printout() {
        cout << "baseClass" << endl;
    }
};

```

Agora o resultado é:

```

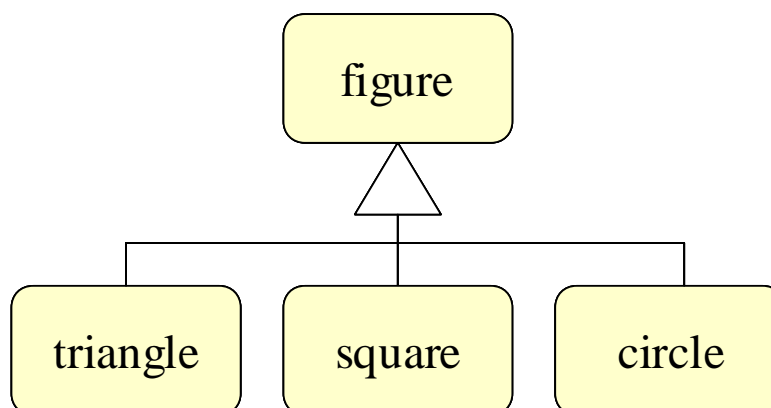
derivedClass
derivedClass

```

Isso significa que embora o ponteiro `myPointer` apontasse para a classe base, como o método em questão “printout” foi programado como virtual, o ponteiro para esse método foi, em tempo de execução (i.e. união atrasada), apontado para a classe derivada.

A inclusão da palavra reservada `virtual` antes de um método torna-o elegível para ser sujeito de uma união atrasada. Isto é, se um método é “marcado” com `virtual`, isso significa que caso ocorra a união de um ponteiro para classe base recebendo o endereço de um objeto derivado, o compilador procurará por um método como mesmo identificador, e se encontrar, em tempo de execução, o ponteiro será carregado com o valor que o faz apontar para o método da classe derivada.

O uso de união atrasada dá grande flexibilidade no projeto de bibliotecas para uso geral. Pode-se por exemplo criar classes que contém diversos métodos pré-programados. O programador usa a biblioteca definindo uma classe do usuário que herda propriedade de uma das classes da biblioteca. Os métodos podem ser sobre escritos facilmente com o uso de união atrasada. Desta forma, pode-se usar a biblioteca de forma “customizada”, o que a torna muito flexível.



// comentar mais sobre o uso de união tardia

7.10.1 Destrutores virtuais

Suponha o caso clássico de união tardia (*late bind*). Um ponteiro para classe base aponta para um objeto da classe derivada. De acordo com as regras de união tardia, caso a classe derivada tenha destrutor, esse destrutor somente será chamado caso o destrutor da classe base seja virtual.

Portanto, por segurança, é aconselhável que os destrutores sejam sempre virtuais (a menos que se tenha certeza que a classe em questão nunca servirá como classe base para uma outra, que possa ser conectada via união tardia).

O exemplo abaixo ilustra o problema. A classe “base” possui destrutor não virtual, e a classe “derived” deriva de “base”. A classe “derived” aloca memória no construtor e apaga a memória no destrutor. Portanto, qualquer objeto da classe derived instanciado diretamente não provoca vazamento de memória. Mas se a classe “derived” for instanciada com união tardia a partir de um ponteiro para classe base (no caso “pb”), então, o destrutor da classe “derived” nunca é chamado. Portanto o programa vaza memória.

A classe “base2” é quase idêntica a “base”, exceto que tem destrutor virtual. A classe “derived2” é quase idêntica a “derived”, exceto que herda de “base2” e não de “base”. O objeto apontado por “pb2”, apesar de ter união tardia, chama corretamente o destrutor da classe “derived2” (além do destrutor da classe “base2”). Essa é a situação segura.

```

class base {
public:
    base () { cout << "constructor base" << endl; }
    ~base () { cout << "destructor base" << endl; }
};

class derived : public base {
    float *m_p;
public:
    derived () {
        cout << "constructor derived, allocating of 100 floats" << endl;
        m_p = new float [100];
    }
    ~derived () {
        cout << "destructor derived (delete floats)" << endl;
        if (m_p) {
            delete [] m_p;
            m_p = 0;
        }
    }
};
  
```

```

    }
};

class base2 {
public:
    base2 () { cout << "constructor base2" << endl; }
    virtual ~base2 () { cout << "destructor base2" << endl; }
};

class derived2 : public base2 {
    float *m_p;
public:
    derived2 () {
        cout << "constructor derived2, allocating of 100 floats" << endl;
        m_p = new float [100];
    }
    ~derived2 () {
        cout << "destructor derived2 (delete floats)" << endl;
        if (m_p) {
            delete [] m_p;
            m_p = 0;
        }
    }
};

void main () {
    cout << "--- non virtual destructor" << endl;
    base *pb = new derived;
    delete pb;

    cout << "--- virtual destructor" << endl;
    base2 *pb2 = new derived2;
    delete pb2;
}

```

Esse programa gera a saída como mostrado abaixo.

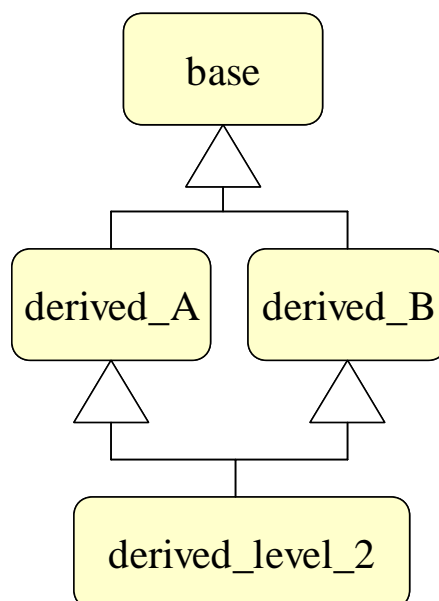
```

--- non virtual destructor
constructor base
constructor derived, allocating of 100 floats
destructor base
--- virtual destructor
constructor base2
constructor derived2, allocating of 100 floats
destructor derived2 (delete floats)
destructor base2

```

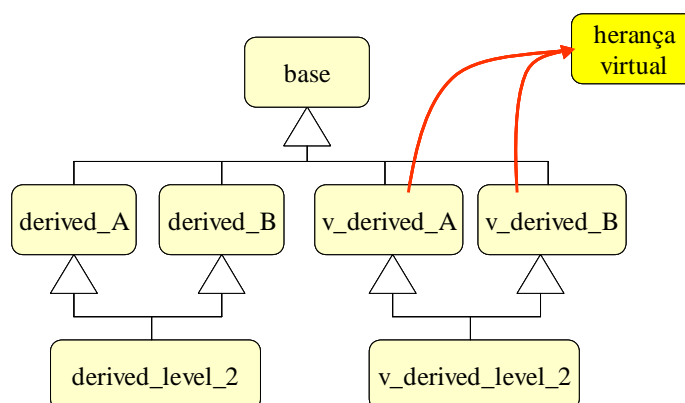
7.11 Classe base virtual

Outra aplicação da palavra reservada *virtual* é a definição de *classe base virtual*. Isto é necessário para uma classe derivada em nível 2 possa lidar com heranças múltiplas de mesma classe base. Em termos de diagrama de herança de classes seria como na figura abaixo.



Em princípio um objeto da classe `derived_level_2` possuirá 2 vezes o conteúdo da base, uma vez em função da herança de `derived_A`, e outra vez em função da herança de `derived_B`. A tentativa de acessar diretamente um atributo ou método da classe base pelo objeto da classe `derived_level_2` gera um erro de ambigüidade, a menos que seja feita referência explicitamente sobre de qual das duas classes derivadas nível 1 se herdou o membro da base. Mas quase nunca tal recurso é efetivamente útil. É bem mais provável que se queira apenas 1 cópia da base na classe `derived_level_2`, e para tanto será necessário definir nas classes derivadas de nível 1 (`derived_A` e `derived_B`) uma herança virtual da classe base. Desta forma o compilador sabe que antes de criar qualquer herdeiro desta classe, os membros da classe base poderão ser fundidos de modo a criar apenas 1 cópia na classe herdeira.

Veja o exemplo (o programa compila mas não gera resultados na tela). Nesse exemplo, são definidas 7 classes, com diagrama de herança como mostrado na figura abaixo.



Devido ao problema exposto, a classe `derived_level_2` possui duas cópias do atributo `m_base_i` (da classe base). Um dos atributos foi herdado de `derived_A` e o outro de `derived_B`. O objeto “a”, da classe `derived_level_2`, precisa explicitar de qual das cópias de `m_base_i` se refere (pois do contrário causa um erro de ambigüidade).

Já a classe `v_derived_level_2` possui apenas uma cópia do atributo `m_base_i`, pois embora herde de `v_derived_A` e de `v_derived_B`, essas duas classes herdam de forma virtual da classe base. Portanto, o

objeto “v” da classe `v_derived_level_2` pode acessar o atributo `m_base_i` diretamente. Há apenas uma cópia de `m_base_i` no objeto “v”.

```
class base {
public:
    int m_base_i;
};

class derived_A : public base {
public:
    int m_derived_A_i;
};

class derived_B : public base {
public:
    int m_derived_B_i;
};

class derived_level_2 : public derived_A, public derived_B {
public:
    int m_derived_level_2_i;
};

// inherits from base as virtual
class v_derived_A : virtual public base {
public:
    int m_derived_A_i;
};

// inherits from base as virtual
class v_derived_B : virtual public base {
public:
    int m_derived_B_i;
};

class v_derived_level_2 : public v_derived_A, public v_derived_B {
public:
    int m_derived_level_2_i;
};

void main () {
    derived_level_2 a;
    v_derived_level_2 v;

    // fill all attributes of object "a"
    a.derived_A::m_base_i = 1;
    a.derived_B::m_base_i = 5;
    a.m_derived_A_i = 2;
    a.m_derived_B_i = 3;
    a.m_derived_level_2_i = 4;

    // fill all attributes of object "v"
    v.m_base_i = 1;
    v.m_derived_A_i = 2;
    v.m_derived_B_i = 3;
    v.m_derived_level_2_i = 4;
}
```

7.12 Alocação de Memória

Existem 3 tipos de variáveis

1. variáveis globais
2. variáveis locais (também chamadas de automáticas)

3. variáveis alocadas dinamicamente

As variáveis **globais** ocupam uma posição arbitrária na memória e podem ter qualquer tamanho, sendo limitadas apenas pela memória do computador. Geralmente não é considerada boa prática de programação a definição de muito espaço em variáveis globais. Isso porque o espaço ocupado pelas variáveis globais é alocado já na carga do programa pelo sistema operacional. Se o espaço de memória global é grande, o programa poderá nem carregar numa máquina com pouca memória.

As variáveis **locais** ocupam uma posição específica da memória - a pilha (*stack*). Quando um programa qualquer é carregado, há um parâmetro que define o espaço de pilha que é reservado para esse programa. A pilha serve também para outras finalidades que guardar variáveis locais. Na pilha guarda-se o endereço das funções de retorno. O tamanho da pilha é fator limitante para saber quantas funções dentro de funções podem ser chamadas, por exemplo. As variáveis locais são também chamadas de automáticas porque essas variáveis são automaticamente criadas quando se entra numa função e automaticamente destruídas quando se retorna da função. Há um parâmetro do compilador que é o *check stack overflow*. Se um programa é compilado com esse tipo de checagem, toda função primeiro checa o tamanho da pilha e depois usa-a. Se não houver espaço, sinaliza-se erro. Essa checagem toma tempo de processamento. Portanto, para programas com tempo de processamento crítico, uma possibilidade é resetar o flag de *check stack overflow*. Mas sem essa checagem, caso haja *overflow* na pilha, o sistema operacional pode congelar.

As variáveis **alocadas** comportam-se como variáveis globais, pois ocupam posição arbitrária na memória. Contudo, há uma diferença importante entre variáveis globais e alocadas. As variáveis globais são definidas em tempo de compilação, enquanto as variáveis alocadas são definidas em tempo de execução. Portanto, o tamanho de uma variável alocada pode ser definido em função de parâmetros que somente são conhecidos em tempo de execução. Isso dá enorme flexibilidade para o programador.

O sistema operacional pode estar executando mais de um programa ao mesmo tempo. A memória, que pode ser alocada dinamicamente, é um dos recursos que o sistema operacional deve gerenciar de forma centralizada para todos os programas que o solicitem. O *heap manager* (*heap* significa monte, pilha, enchimento) é um serviço do sistema operacional que gerencia o espaço livre de memória. Quando o *heap manager* recebe uma solicitação de alocação de algum programa, escreve numa tabela própria a informação de que aquele pedaço de memória está alocado (e portanto não está mais disponível).

A alocação de memória é uma operação que pode falhar, pois pode ser que não haja disponível no momento da alocação um segmento contínuo de memória com tamanho requerido. Portanto um bom programa deve sempre testar se uma tentativa de alocação foi bem sucedida ou não. Um procedimento deve ser previamente definido para ser executado no caso de a alocação de memória fracassar.

7.12.1 Vazamento de memória

Os programas que alocam memória devem ter o cuidado de liberar a memória que não precisam mais. A memória alocada por um programa permanece indisponível para outros usos, mesmo quando o programa que alocou a memória já tenha terminado. Diz-se que um programa que aloca memória e termina sem libera-la possui um “vazamento de memória”. Um conceito análogo existe para um objeto. Se um objeto aloca memória (digamos no construtor) e não a libera (digamos no destrutor), o objeto tem um “vazamento de memória”.

Em C++, a alocação é feita com a palavra reservada `new`, e a liberação é feita com a palavra reservada `delete`. O programa abaixo define uma classe chamada `mydata`, com alguns dados. A função `main` aloca dinamicamente um objeto da classe `mydata`, usa-o e em seguida libera-o.

```
class mydata {
public:
    double d[333444];
    float f[333];
};

void main () {
    mydata *p_mydata=new mydata; // alloc
    p_mydata->d[222111] = 1.1; // escreve um dado dentro de mydata
    double z = p_mydata->d[222111]; // lê um dado dentro de mydata
    cout << 3 + z << endl; // a resposta será 4.1
    delete p_mydata; // delete
}
```

Além de objetos simples, como acima, pode-se alocar e liberar arrays de objetos. Para isso, basta passar como parâmetro para o operador `new` uma constante ou variável com o número (inteiro positivo) de objetos contidos no array. No exemplo abaixo, uma variável `n` tipo `long` é carregada com a constante 333444 (esse valor poderia ter sido entrado pelo usuário, ou lido de disco). Em seguida, aloca-se um array de `double` de tamanho `n`, usa-se um valor no meio do array para leitura e escrita, e em seguida libera-se a memória.

```
void main () {
    long n=333444;
    double *p_double=new double[n]; // alloc
    p_double[222111] = 1.1; // um valor qualquer no meio
    cout << 3 + p_double[222111] << endl; // 4.1
    delete [] p_double; // delete
}
```

Pode-se verificar se um programa está ou não vazando memória. Para maiores detalhes, veja a seção “verificando vazamento de memória” (página 50). Usando as técnicas descritas nessa seção, remova (ou comente) a linha que deleta a memória nos 2 programas anteriores e verifique que sem essa linha o programa fica vazando memória.

7.12.2 Objetos com memória alocada

Caso se esteja desenvolvendo uma classe que faça alocação de memória, geralmente haverá alocação de memória em algum método (pode ser o construtor), e o destrutor cuida de ver se há memória alocada e se houver, deletar essa memória.

Caso essa classe seja usada em alguma situação de cópia, é preciso que se defina o construtor de cópia e o operador `=` para essa classe. Caso não se defina esses métodos, o compilador automaticamente os substituirá por métodos que copiam os atributos do objeto byte a byte. Mas esse tipo de cópia automática não faz nova alocação de memória para os dados, portanto o objeto copiado não é uma cópia do original no sentido que geralmente se supõe que deveria ser, para a maioria das aplicações práticas de programação.

Conclusão: ao se desenvolver uma classe que faça alocação de memória, além de definir um método para alocar memória e de deletar essa memória no destrutor, é preciso definir o construtor de cópia e o operador `=` para que a cópia do objeto seja feita de acordo com o que se espera dele.

Veja o exemplo abaixo. Nesse exemplo, é feita uma classe chamada `myVectorAllocMem`, que é um vetor de `float`, sendo que os dados são alocados (não estão definidos estaticamente na classe). A classe portanto possui um atributo que é ponteiro para os dados, no caso `m_data`, que é um `float*`. Para simplificar, foi definido um método privado chamado `myDelete`, que deleta os dados caso o atributo ponteiro para os dados esteja não nulo (isto é, apontando para dados).

Nessa classe há um default constructor que também aceita parâmetro implícito. Esse parâmetro é o tamanho do vetor. Se o tamanho é maior que zero, aloca-se dados. Caso contrário, carrega-se zero no

ponteiro membro, para garantir que não há nada alocado. O construtor de cópia aloca memória para o novo objeto e copia os atributos (m_size e o conteúdo de memória apontado pelo ponteiro, elemento a elemento).

```
#include <iostream.h>12
#include "vplib.h" // VbMemCheck (Visual C++ only)
class myVectorAllocMem {
    float *m_data;
    int m_size;
    void myDelete() {
        if (m_data) {
            delete [] m_data; // delete mem
            m_data = 0;
        }
    }
public:
    myVectorAllocMem(int size=0) { // default constructor with default parameter
        m_size = size;
        m_data = 0; // if size <= 0
        if (size>0)
            m_data = new float [m_size]; // alloc mem
    }

    myVectorAllocMem(const myVectorAllocMem & obj) { // copy constructor
        if (obj.m_size > 0) {
            m_size = obj.m_size;
            m_data = new float [m_size]; // alloc mem
            for (int i=0 ; i < m_size ; i++)
                m_data[i] = obj.m_data[i]; // copy data contents
        }
    }

    ~myVectorAllocMem() { // destructor
        myDelete();
    }

    void operator=(const myVectorAllocMem & obj) { // operator=
        if (obj.m_size > 0) {
            m_size = obj.m_size;
            if (m_size>0) {
                myDelete();
                m_data = new float [m_size]; // alloc mem
            }
            for (int i=0 ; i < m_size ; i++)
                m_data[i] = obj.m_data[i]; // copy data contents
        }
        else {
            myDelete();
            m_size = 0;
        }
    }

    // the fill method fills the vector with a given value
    void fill(float f) {
```

¹² copy data contents = copia conteúdo dos dados, default constructor with default parameter = construtor implícito com parâmetro implícito, the fill method fills the vector with a given value = o método fill (preencher) preenche o vetor com o valor dado, the show method copies the contents of the vector to the console = o método show (mostrar) copia o conteúdo do vetor para o console, a user function that receives a myVectorAllocMem as a parameter, and returns myVectorAllocMem = uma função do usuário que recebe um myVectorAllocMem como parâmetro, e retorna um myVectorAllocMem

```

        if (m_size > 0) {
            for (int i=0 ; i < m_size ; i++)
                m_data[i] = f;
        }

// the show method copies the contents of the vector to the console
void show() {
    cout << "--- size=" << m_size << endl;
    for (int i=0 ; i < m_size ; i++)
        cout << m_data[i] << endl;
}
}; // end of myVectorAllocMem

// a user function that receives a myVectorAllocMem as a parameter,
// and returns myVectorAllocMem
myVectorAllocMem userFun(myVectorAllocMem a) {
    a.show();
    myVectorAllocMem b(5);
    b.fill(4.4);
    return b;
}

// the main code isolated, to allow memory leak check
void do_all() {
    myVectorAllocMem z(2);
    z.fill(2.2);
    z.show();
    myVectorAllocMem x;
    x = userFun(z);
    x.show();
}

void main() {
    VBMemCheck w; // mark the memory condition
    do_all(); // do actual processing
    w.check(); // check to see if there is memory leak
}

```

A saída do programa é como mostrado abaixo. O primeiro vetor é o “z”. O segundo é o “a” (cópia de “z”). O terceiro é “x” (cópia de “b”).

```

--- size=2
2.2
2.2
--- size=2
2.2
2.2
--- size=5
4.4
4.4
4.4
4.4
4.4

```

7.12.3 Array de objetos com construtor não padrão

Um array de objetos pode ser inicializado com um construtor não padrão, desde que seja estático. Não há sintaxe para que se faça alocação de um array de objetos, sendo que cada objeto recebe parâmetros de um construtor não padrão.

No exemplo abaixo, uma classe é definida com um construtor com parâmetro, mas sem construtor padrão. Portanto, não se pode criar um objeto diretamente, como feito com o “obj2” no exemplo abaixo. Mas pode-se criar um obj chamando diretamente o construtor e passando um parâmetro. Pode-

se também criar um array estático de objetos, desde que cada objeto seja devidamente inicializado com a chamada do construtor e tendo um parâmetro para cada construtor de cada objeto.

Mas como não há sintaxe para que se defina a chamada de construtor não padrão para um array de objetos, não se pode alocar myClass.

```
// this class has no default constructor
class myClass {
    int m_i;
public:
    myClass(int i) {
        m_i = i;
    }
};

void main () {
    myClass obj2; // error: no appropriate default constructor available
    myClass obj = myClass(20);
    myClass myArray[3] = { myClass(1), myClass (4), myClass (10) };
    myClass *p;
    p = new myClass [3]; // error: no appropriate default constructor available
}
```

Capítulo 8) Biblioteca padrão de C++

8.1 Introdução

A biblioteca padrão de C já existia quando foram desenvolvidos os conceitos de orientação a objetos. Para a linguagem C++ uma nova biblioteca padrão foi criada, onde os conceitos de orientação a objetos são usados. A biblioteca padrão de C++ é mais fácil de ser aprendida e mais segura que a de C. Em princípio, não há motivo para que se use a biblioteca padrão de C nas funcionalidades em que a biblioteca de C++ tem aplicação (a menos que seja para manter código de legado). Nesse capítulo será dada ênfase no uso da biblioteca padrão de C++, e não de C.

Em alguns exemplos, será usada biblioteca não padrão `vblib`. A única diferença técnica entre uma biblioteca não padrão e a biblioteca padrão é o fato de que o programa compilador incorpora automaticamente a biblioteca padrão em qualquer projeto. Uma biblioteca não padrão precisa ser incorporada manualmente. Quase sempre, a finalidade de incorporar essa biblioteca não padrão é usar a classe `VBString`, que é uma classe de string de uso geral. Essa classe é bastante fácil de usar e não requer uso de namespace (como a classe `string` da biblioteca padrão). Para usar a `vblib`, é preciso incluir o header `vblib.h` no fonte onde se deseja usar funcionalidades da `vblib`, e acrescentar o arquivo `vblib.cpp` no projeto em questão.

Tradicionalmente o C manipula strings como array of char. Muitos livros ensinam isso. Mas a experiência mostra que essa forma de programar faz aumentar a probabilidade de bugs grandemente, portanto é geralmente considerada como má programação. A solução recomendada é o uso de uma classe de string na manipulação de strings é boa programação.

Ao usar `VBString` nos exemplos abaixo, além de se exemplificar o que se quer, a mensagem indireta que se está passando é a seguinte:

- Evite o uso de array of char, e manipule strings com uma classe de string como `VBString`.
- Não tenha medo de usar uma biblioteca não padrão, especialmente se o fonte dessa biblioteca estiver disponível.

8.2 Entrada e saída de dados pelo console

Para saída de dados pelo console, usa-se uma corrente (*stream*), que termina em `cout` (console output). A corrente é uma seqüência de elementos separados pelo operador insersor `<<`. Qualquer elemento pode ser inserido na corrente, desde que para esse elemento esteja definido o operador insersor. Todos os tipos padrão já possuem funções para o operador insersor. Os tipos do usuário também podem entrar na corrente, desde que seja feita a definição de uma função mostrando como o tipo entra na corrente. Esse procedimento é conhecido como sobrecarga do operador insersor e está explicado na seção 8.3.

Veja o exemplo:

Exemplo 1:

```
#include "vblib.h"
void main () {
    int i=2;
```

```
float f=2.2;
double d=3.333;
VbString str = "anything at all";
cout << "text" << i << " , " << f << " , " << d << " , " << str << endl;
}
```

A saída do programa é mostrada abaixo.

```
text2 , 2.2 , 3.333 , anything at all
```

Analogamente, o extrator >> é o operador inverso ao insersor, e a corrente se inicia em cin (console input). O extrator também pode ser sobrecarregado, da mesma forma que o insersor. Veja o exemplo.

Exemplo:

```
#include <iostream.h>
void main () {
    int i; float f; double d;
    cout << "Entre com i:";
    cin >> i;
    cout << endl << "Entre com f:";
    cin >> f;
    cout << endl << "Entre com d:";
    cin >> d;
    cout << endl << "i=" << i << "f=" << f << "d=" << d << endl;
}
```

8.3 Sobrecarga do operador insersor (<<) e extrator (>>)

Quando se cria uma classe, pode-se querer utiliza-la em entradas e saídas do console ou arquivo de forma análoga a tipos padrão. Como o compilador não sabe como usar o insersor e extrator para a classe do usuário, tudo o que se tem a fazer é sobrecarga-los para que entendam a sua classe. Veja o exemplo.

Exemplo:

```
#include <iostream.h>
#include <string.h>
class myclass {
    int m_i;
    double m_d;
    char m_name[100];
public:
    myclass () {m_i=0; m_d=0; strcpy(m_name,"Unnamed");};
    friend ostream & operator<<(ostream & stream, const myclass &obj);
    friend istream & operator>>(istream & stream, myclass &obj);
};
// overload insersor
ostream & operator << (ostream & stream, const myclass & obj) {
    stream << obj.m_i << "\n" << obj.m_d << "\n" << obj.m_name << "\n";
    return stream;
}
// overload extrator
istream & operator >> (istream & stream, myclass &obj) {
    stream >> obj.m_i;
    stream >> obj.m_d;
    stream >> obj.m_name;
    return stream;
}
void main() {
    myclass a;
    cout << a;
    cout << "Entre i d nome : ";
    cin >> a;
    cout << a;
}
```

Resultado:

```
0
0
Unnamed
Entre i d nome : 1 1.11111 um_nome
1
1.11111
um_nome
```

As funções de sobrecarga do insersor e do extrator *não* podem ser membros da classe respectiva. Isso porque o insersor e o extrator são operadores (`operator`). Quando um operador é membro de uma classe, o operando esquerdo (implicitamente passado usando o ponteiro `this`) é assumido como o objeto da classe que chama a função `operator`. Lembre-se:

```
a=b; // a.operator=(b)
```

Contudo, no insersor e no extrator o argumento a esquerda é uma stream, enquanto o argumento a direita é um objeto da classe. O insersor (e extrator) é uma função não membro que precisa ter acesso a elementos internos de uma classe. No caso geral pode haver elementos de acesso privado na classe. A solução para esse dilema é declarar o insersor (e extrator) como uma função `friend` da classe.

8.4 Formatando Entrada / Saída com streams

A forma como C++ gerencia a entrada e saída de dados para console, arquivos, etc. foi toda remodelada em relação a forma como C o fazia. Na biblioteca padrão de C++, existem classes para saída e para entrada, como mostrado abaixo. Além de classes, na biblioteca padrão de C++ há também objetos (ocorrência de classes), como `cout` e `cin`.

Classe de saída da bib. padrão	Objeto	Função
<code>ostream</code>	<code>cout</code> , <code>cerr</code> , <code>clog</code>	Saída em console
<code>ofstream</code>		Saída em arquivo em disco
<code>ostrstream</code>		Saída bufferizada em string

Classe de saída da bib. padrão	Objeto	Função
<code>istream</code>	<code>cin</code>	Entrada em console
<code>ifstream</code>		Entrada em arquivo em disco
<code>istrstream</code>		Entrada bufferizada em string

O diagrama de heranças entre as classes da biblioteca padrão de C++, no que se refere a entrada e saída formatada, é mostrado na figura abaixo.

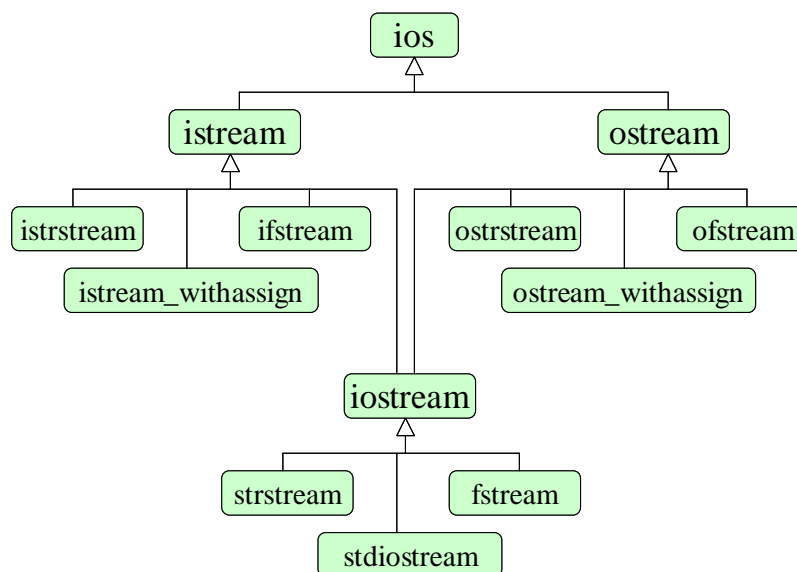


Figura 5: diagrama de heranças entre as classes da biblioteca padrão de C++.

8.4.1 Usando flags de formatação

Formatar a saída significa por exemplo escolher com quantas casas decimais um número vai ser exteriorizado. Em C++ isso é feito usando formatadores e manipuladores de stream, que são usados igualmente para console, arquivo, etc. Todos os streams herdam propriedades da classe ios. Abaixo está mostrado os flags dessa classe que afetam a entrada / saída com streams.

```
// flags de formatação
enum = {
skipws    = 0x0001, // pula espaço em branco na entrada
left      = 0x0002, // saída ajustada a esquerda
right     = 0x0004, // saída ajustada a direita
internal  = 0x0008, // preenche após o sinal ou indicador de base
dec       = 0x0010, // conversão decimal
oct       = 0x0020, // conversão octal
hex       = 0x0040, // conversão hexadecimal
showbase  = 0x0080, // usa o indicador de base na saída
showpoint = 0x0100, // força o ponto decimal (saída flutuante)
uppercase = 0x0200, // saída hexadecimal maiúscula
showpos   = 0x0400, // mostra "+" aos inteiros positivos
scientific = 0x0800, // notação científica
fixed     = 0x1000, // notação ponto flutuante fixa
unibuf    = 0x2000, // descarrega o buffer após a inserção
stdio     = 0x4000, // descarrega stdout e stderr após inserção
boolalpha = 0x8000, // insere/extrai booleanos como texto ou numérico
};
```

No exemplo abaixo, o flag showpos foi ativado para forçar a exibição do sinal no caso de número positivo.

```
#include <iostream.h>
void main () {
    int i = 100;
    cout.setf(ios::showpos);
    cout << i << endl;
}
```

A saída do programa é mostrada abaixo.

```
+100
```

No exemplo abaixo, o flag hex foi ativado para forçar a exibição do número em hexadecimal.

```
#include <iostream.h>
void main () {
    int i = 100;
    cout.setf(ios::hex);
    cout << i << endl;
}
```

A saída do programa é mostrada abaixo.

64

Desde que os flags são definidos em cada bit de uma palavra de 16 bits, mais de um flag pode ser usado ao mesmo tempo, a partir do or bitwise (usando |). Por exemplo: para usar o uppercase o scientific ao mesmo tempo, pode-se fazer como mostrado abaixo. No mesmo exemplo, mostra-se como desligar um flag usando-se o método unsetf.

```
#include <iostream.h>
void main () {
    float f = 100.12;
    cout.setf(ios::uppercase | ios:: scientific);
    cout << f << endl;
    cout.unsetf(ios::uppercase);
    cout << f << endl;
}
```

A saída do programa é mostrada abaixo. Note o ‘E’ (maiúsculo) e o ‘e’ (minúsculo).

```
1.001200E+002
1.001200e+002
```

8.4.2 Examinando os flags de um objeto ios

Para um dado objeto que herda da classe ios, pode-se examinar o conteúdo dos seus flags, a partir do retorno do método “flags”. A partir disso, pode-se criar uma função para exibir a condição de um dado objeto que herda de ios, mesmo que seja o cout da biblioteca padrão. É exatamente isso que faz a função global showIosFlags do exemplo abaixo. Experimente alterar o programa e chamar showIosFlags passando como parâmetro cin.

```
#include <iostream.h>

void showIosFlags(ios & iosObj) {
    cout << "==== ios flag status" << endl;
    long flags = iosObj.flags();
    const char *flagNames[] = {
        "skipws",
        "left",
        "right",
        "internal",
        "dec",
        "oct",
        "hex",
        "showbase",
        "showpoint",
        "uppercase",
        "showpos",
        "scientific",
        "fixed",
        "unibuf",
        "stdio",
        "boolalpha"
    };

    long k = 1;
    for (int i=0 ; i < 16 ; i++) {
        cout << flagNames[i] << " is ";
        if (k & flags)
```

```

        cout << "ON" << endl;
    else
        cout << "off" << endl;
    k <= 1; // rotate k
}
cout << endl;
}

void main () {
    showIosFlags(cout);
    cout.setf(ios::uppercase | ios::scientific);
    showIosFlags(cout);
}

```

A saída do programa é mostrada abaixo.

```

==== ios flag status
skipws is off
left is off
right is off
internal is off
dec is off
oct is off
hex is off
showbase is off
showpoint is off
uppercase is off
showpos is off
scientific is off
fixed is off
unibuf is off
stdio is off
boolalpha is off

==== ios flag status
skipws is off
left is off
right is off
internal is off
dec is off
oct is off
hex is off
showbase is off
showpoint is off
uppercase is ON
showpos is off
scientific is ON
fixed is off
unibuf is off
stdio is off
boolalpha is off

```

8.4.3 Definindo o número de algarismos significativos

O método “precision” serve para definir o número de algarismos significativos que se vai usar ao converter um número em ponto flutuante para a stream (corrente). Antes que se defina o número, o objeto cout assume-o como 6.

```

#include <iostream.h>
void main () {
    double d = 1234.123412341234;
    cout << "d=" << d << endl;
    cout.precision(3);
    cout << "d=" << d << endl;
    cout.precision(8);
    cout << "d=" << d << endl;
}

```

```
}
```

A saída do programa é mostrada abaixo.

```
d=1234.12
d=1.23e+003
d=1234.1234
```

8.4.4 Preenchendo os espaços vazios com o método fill

O método “width” (largura) força a largura com que um campo deve ser copiado para a stream. O método “fill” (preencher) define o char que será usado para o preenchimento de espaços vazios (o char padrão é ‘ ’ [espaço]). O exemplo abaixo ilustra o uso desses métodos.

```
#include <iostream.h>
void main () {
    double d = 1234.123412341234;
    cout.precision(3);
    cout.width(10);
    cout.fill('*');
    cout << "d=" << d << endl;
    cout.precision(8);
    cout.width(10);
    cout.fill('*');
    cout.setf(ios::left);
    cout << "d=" << d << endl;
}
```

A saída do programa é mostrada abaixo.

```
*****d=1.23e+003
d=*****1234.1234
```

8.4.5 Manipuladores padrão

Além de se chamar funções membro dos objetos que herdam de ios, pode-se também acrescentar à *stream* (corrente) os manipuladores. Na biblioteca padrão de C++ há um conjunto de manipuladores que podem ser usados. Esses manipuladores estão, em geral, declarados no arquivo header padrão iomanip.h.

Manipulador	Entrada/Saída (i/o)	Propósito
dec	i/o	Faz a <i>stream</i> a usar o formato decimal
hex	i/o	Faz a <i>stream</i> a usar o formato hexadecimal
oct	i/o	Faz a <i>stream</i> a usar o formato octal
resetiosflags(long s)	i/o	Desliga os <i>flags</i> especificados por s
setiosflags(long s)	i/o	Liga os <i>flags</i> especificados por s
endl	o	Faz a <i>stream</i> inserir um char de fim-de-linha e também descarregar o seu buffer
ends	o	Faz a <i>stream</i> inserir um char nulo
flush	o	Faz a <i>stream</i> descarregar o seu buffer
setbase(int b)	o	Define b como base
setfill(char ch)	o	Define o char de preenchimento como ch
setprecision(int p)	o	Define o número de algarismos significativos como p
setw(int w)	o	Define a largura (<i>width</i>) como w
ws	i	Pula espaço em branco a esquerda

O programa da seção 8.4.4 pode ser re-escrito com manipuladores como mostrado abaixo.

```
void main () {
```

“C / C++ e Orientação a Objetos em Ambiente Multiplataforma”, versão 5.1

Esse texto está disponível para download em www.del.ufrj.br/~villas/livro_c++.html

```
double d = 1234.123412341234;
cout << setprecision(3) << setw(10) << setfill('*') << "d=" << d << endl;
cout << setprecision(8) << setw(10) << setfill('*')
    << setiosflags(ios::left) << "d=" << d << endl;
}
```

A saída é a mesma que a da versão original do programa.

```
*****d=1.23e+003
d=*****1234.1234
```

8.4.6 Manipuladores do usuário

Uma das diferenças no uso de manipuladores e de funções membro de ios é o fato de que é possível criar-se um manipulador do usuário para *streams*. A estrutura para se fazer um manipulador do usuário é muito parecido com a da sobrecarga do operador inserção (operator<<).

```
// estrutura para se fazer um manipulador do usuário
ostream & myManip(ostream & str [, parameters] ) {
    // user code
    return str;
}
```

No exemplo abaixo, um conjunto de formatações é colocada num manipulador do usuário chamado *myStyle*.

```
#include <iostream.h>
#include <iomanip.h>
ostream & myStyle(ostream & str) {
    str.precision(3);
    str.width(10);
    str.fill('*');
    cout.setf(ios::left);
    return str;
}
void main () {
    double d = 1234.123412341234;
    cout << myStyle << "d=" << d << endl;
}
```

A saída do programa é mostrada abaixo.

```
d=*****1.23e+003
```

Vamos agora repedir a funcionalidade do programa da seção 8.4.4, com um novo manipulador do usuário *myStylec* com a passagem de um parâmetro, que é a precisão (número de algarismos significativos). Para isso, é preciso usar-se a macro *OMANIP*, conforme mostrado no exemplo.

```
#include <iostream.h>
#include <iomanip.h>
ostream & myStyle(ostream & str, int precision) {
    str.precision(precision);
    str.width(10);
    str.fill('*');
    cout.setf(ios::left);
    return str;
}

OMANIP(int) myStyle(int precision) {
    return OMANIP(int) (myStyle, precision);
}

void main () {
    double d = 1234.123412341234;
    cout << myStyle(3) << "d=" << d << endl;
    cout << myStyle(8) << "d=" << d << endl;
}
```

A saída do programa é mostrada abaixo.

```
d=*****1.23e+003
d=*****1234.1234
```

8.4.7 Saída com stream em buffer

Por várias razões, pode ser necessário que um buffer seja criado com o resultado de um *stream* de saída. Isso pode ser feito com a classe *ostrstream*, que está declarada no header padrão *strstream.h* (em Windows), e *strstream.h* (em unix). Há também a classe *istrstream*, para entrada. Admitindo que os programas são compilados em Windows com a diretiva WIN32 ativada, os programas que usam *strstream* abaixo são multiplataforma.

É preciso tomar cuidado no uso de *ostrstream*, para que se inclua um char nulo de término de string no fim de cada stream, usando-se o manipulador padrão “ends”. Do contrário, a string não estará terminada e o comportamento do programa poderá ser aleatório. Caso não se inclua o ends no final da stream, não haverá nenhum erro ou aviso por parte do compilador. No exemplo abaixo, a mesma funcionalidade da seção 8.4.6 foi repetida, usando-se um objeto da classe *ostrstream*, que por sua vez relaciona-se com um buffer.

```
#include <iostream.h>
#include <iomanip.h>
#ifdef WIN32
    #include <strstream.h>
#else
    #include <strstream.h>
#endif

ostream & myStyle(ostream & str, int precision) {
    str.precision(precision);
    str.width(10);
    str.fill('*');
    cout.setf(ios::left);
    return str;
}

OMANIP(int) myStyle(int precision) {
    return OMANIP(int) (myStyle, precision);
}

void main () {
    char buffer[500];
    ostrstream myCout(buffer, sizeof(buffer));
    double d = 1234.123412341234;
    myCout << myStyle(3) << "d=" << d << ends;
    cout << buffer << endl;
    myCout << myStyle(8) << "d=" << d << ends;
    cout << buffer << endl;
}
```

A saída do programa é mostrada abaixo.

```
d=*****1.23e+003
d=*****1234.1234
```

A vantagem de se usar o recurso da classe *ostrstream* é o fato de que pode-se manipular strings com todas as facilidades da biblioteca padrão de streams.

A desvantagem de se usar o recurso da classe *ostrstream* é o fato de que é necessário o uso de buffer externo de tamanho fixo. Não é possível que o tamanho do buffer seja alocado automaticamente como no caso de uma classe de string. Caso a stream ultrapasse o tamanho do buffer, o comportamento do programa torna-se aleatório. Isso é portanto uma fonte potencial de bugs.

8.5 Acesso a disco (Arquivos de texto para leitura/escrita)

Há dois tipos básicos de arquivos

1. Arquivos de texto
2. Arquivos binários

A biblioteca padrão de C++, usa as classes `ifstream` e `ofstream` para leitura e escrita em arquivo respectivamente, que são definidas em `fstream.h`. Estas classes possuem funções membro para abrir (`open`) e fechar (`close`) o arquivo em questão e sobrecarrega os operadores `>>` e `<<` respectivamente para a manipulação do arquivo.

Não é preciso, como em C, ajustar uma letra numa string de formato, pois o operador utilizado foi sobrecarregado para todos os tipos padrão do compilador. Para manter a lógica do uso de arquivos com a classe do usuário, basta sobrecarregar o operador `>>` ou `<<` na nova classe.

8.5.1 Escrevendo um arquivo de texto usando a biblioteca padrão de C++

Exemplo:

```
#include <fstream.h>
void main() {
    float pi=3.14;
    double d = 2.3456;
    ofstream myfile;
    const char* fname = "fileout.dat";
    myfile.open (fname);
    if (!myfile)
        cout << "File " << fname << " not open";
    myfile << pi << endl << d;
    myfile.close();
}
```

Resultado: cria-se o arquivo `fileout.dat` com o seguinte conteúdo

```
3.14
2.3456
```

8.5.2 Escrevendo um arquivo de texto usando a biblioteca padrão de C

Exemplo:

```
#include <stdio.h>
void main() {
    float pi=3.14;
    double d = 2.3456;
    FILE *myfile;
    char* fname = "fileout.dat";
    myfile = fopen (fname, "w");
    if (!myfile)
        printf("File %s not open\n",fname);
    fprintf(myfile,"%f\n%lf",pi,d);
    fclose(myfile);
}
```

Resultado: cria-se o arquivo `fileout.dat` com o seguinte conteúdo

```
3.140000
2.345600
```

Observe que a versão do programa que usa a biblioteca padrão de C++ é substancialmente mais simples que a versão em C. Essa é uma das vantagens do uso do conceito de “orientação a objetos”.

Na versão em C++, usa-se uma corrente (*stream*) direcionada para `cout`. O operador inseridor `<<` é definido para todos os tipos padrão (`int`, `float`, `double`, etc) e portanto todos esses tipos podem ser

usados na corrente sem necessidade de uma “string de formato” adequada para cada tipo. E mais: qualquer tipo ou classe do usuário pode sobrecarregar o operador inseridor de forma que as variáveis do tipo do usuário podem entrar na corrente da mesma forma que os tipos padrão.

Na versão em C, qualquer variável requer o uso de uma string de formato adequada para si. Um erro na definição da string de formato não gera qualquer erro de compilação, mas é gera erros de execução imprevisíveis. No caso do exemplo, o a string de formato contém %f associado ao tipo float e %lf associado ao tipo double. Os tipos do usuário devem ser quebrados em elementos que sejam de tipos padrão e então usa-se a função fprintf com cada um dos elementos. Isso é muito mais trabalhoso e aumenta muito a possibilidade de erros de programação.

No caso de arquivos de leitura, de forma análoga ao caso de escrita, o uso de C++ e orientação a objetos mostra também a sua vantagem, pois o código é mais fácil de se entender e mais difícil de haver erros.

8.5.3 Lendo um arquivo de texto usando a biblioteca padrão de C++

Exemplo:

```
#include <fstream.h>
void main() {
    float pi;
    double d;
    ifstream myfile;
    char* fname = "fileout.dat";
    myfile.open (fname);
    if (!myfile) {
        cout << "File " << fname << " not open";
        exit(0);
    }
    myfile >> pi >> d;
    cout << "pi=" << pi << endl
        << "d=" << d << endl;
    myfile.close();
}
```

Resultado:

```
pi=3.14
d=2.3456
```

8.5.4 Dica para leitura de arquivo de texto.

Caso se queira ler um arquivo de texto até a sua última linha, é preciso lembrar-se que o flag de “end-of-file” somente é ativado caso se tente ler a linha depois da última. O programa abaixo é um exemplo simples que copia um arquivo de texto para o console.

```
#include <fstream.h>
#include <vblib.h>
void main () {
    const char* fName = "c:/xxx/t.txt";
    ifstream myFile(fName);
    if (!myFile) {
        cout << "Could not open file" << endl;
    }
    VBString a;
    while (true) {
        myFile >> a; // read a line to a
        if (myFile.eof()) break;
        cout << a << endl;
    }
}
```

Admitindo que existe um arquivo chamado “t.txt” no diretório “c:\xxx”, e que esse arquivo contém o que é mostrado abaixo (repare que existe um “return” após a última linha com conteúdo).

```
Esse é o arquivo t.txt.  
Qualquer coisa pode ser escrita aqui.  
Mesmo linhas muitíssimo longas podem ser escritas sem nenhum problema . . . . .  
Depois dessa última linha, há um “return”. Isso pode ser comprovado pelo fato  
de que o cursor pode ser posicionado na linha abaixo.
```

A saída do programa será como mosrado abaixo. Igual ao arquivo de texto t.txt, mas sem a última linha.

```
Esse é o arquivo t.txt.  
Qualquer coisa pode ser escrita aqui.  
Mesmo linhas muitíssimo longas podem ser escritas sem nenhum problema . . . . .  
Depois dessa última linha, há um “return”. Isso pode ser comprovado pelo fato  
de que o cursor pode ser posicionado na linha abaixo.
```

Esse tipo de laço de programa é muito útil para se ler dados de um arquivo de texto, cujo comprimento não se sabe a priori. Por exemplo: leia um vetor de inteiros do disco e coloque o resultado num vetor alocado dinamicamente. Não se sabe antecipadamente a quantidade de dados.

8.5.5 Lendo um arquivo de texto usando a biblioteca padrão de C

Exemplo:

```
#include <stdio.h>  
void main() {  
    float pi;  
    double d;  
    FILE *myfile;  
    char* fname = "fileout.dat";  
    myfile = fopen (fname, "r");  
    if (!myfile)  
        printf("File %s not open\n",fname);  
    fscanf(myfile,"%f\n%lf",&pi,&d);  
    printf("pi=%f\nd=%lf",pi,d);  
    fclose(myfile);  
}
```

Resultado:

```
pi=3.140000  
d=2.345600
```

8.6 Acesso a disco (Arquivos binários para leitura/escrita)

8.6.1 Escrevendo um arquivo binário usando a biblioteca padrão de C++

```
#include <fstream.h>  
void main() {  
    float pi=3.14;  
    // to write, binary mode  
    ofstream myfile;  
    char* fname = "fileout.dat";  
    myfile.open (fname,ios::binary);  
    if (!myfile)  
        cout << "File " << fname << " not open";  
    myfile.write((unsigned char*)&pi,sizeof(float));  
    myfile.close();  
}
```

8.6.2 Escrevendo um arquivo binário usando a biblioteca padrão de C.

```
#include<stdio.h>
void main() {
    float pi=3.14;
    FILE *myfile;
    myfile = fopen ("f1.dat","wb");
    if (!myfile)
        printf("File Not Open\n");

    fwrite(&pi,sizeof(float),1,myfile);

    fclose(myfile);
}
```

Prática: Faça um programa que grave um vetor de float com 10.000 posições em 2 arquivos – um arquivo em formato binário e outro arquivo em formato texto.

Resposta:

```
#include<fstream.h>

#define SIZE 10000
#define TYPE float

void main() {
    TYPE vetor[SIZE];
    int i;
    for (i=0 ; i<SIZE ; i++)
        vetor[i] = i+0.1234567890123; // carrega o vetor com qualquer coisa

    // grava o arquivo em binário
    ofstream myfile;
    char* fname = "f_bin.dat";
    myfile.open (fname,ios::binary);
    if (!myfile)
        cout << "File " << fname << " not open";

    // para gravar um por um dos elementos do vetor
    for (int i=0 ; i<SIZE ; i++)
        myfile.write((unsigned char*)&vetor[i],sizeof(TYPE));

    // para gravar o vetor todo de uma vez
    // myfile.write((unsigned char*)&vetor[0],SIZE*sizeof(TYPE));

    myfile.close();

    // grava o arquivo em forma de texto
    fname = "f_txt.dat";
    myfile.open (fname);
    if (!myfile)
        cout << "File " << fname << " not open";

    for (i=0 ; i<SIZE ; i++)
        myfile << vetor[i] << endl;
    myfile.close();
}
```

Resultado:

Foram criados 2 arquivos. Um é binário, chamado `f_bin.dat`, que não pode ser visto em editor de texto, com tamanho de 40.000 bytes. Outro é chamado `f_txt.dat`, que é em formato texto (portanto pode ser visto em editor de texto), mas seu tamanho é 90.001 bytes.

8.6.3 Lendo um arquivo binário usando a biblioteca padrão de C++

```
#include <fstream.h>
void main() {
    float pi;
    ifstream myfile;
    char* fname = "fileout.dat";
    myfile.open (fname);
    if (!myfile)
        cout << "File " << fname << " not open";
    myfile.read((unsigned char*)&pi,sizeof(float));
    // usa a variável pi
    myfile.close();
}
```

8.6.4 Lendo um arquivo binário usando a biblioteca padrão de C

```
#include<stdio.h>
void main() {
    float pi;
    FILE *myfile;
    myfile = fopen ("f_bin.dat","rb");
    if (!myfile)
        printf("File Not Open\n");
    fread(&pi,sizeof(float),1,myfile);
    // usa a variável pi
    fclose(myfile);
}
```

Prática: Faça um programa para ler os arquivos `f_bin.dat` e `f_bin.dat` feitos na prática anterior.

Resposta:

```
#include<fstream.h>

#define SIZE 10000
#define TYPE float

void main() {
    TYPE vetor[SIZE];
    int i;
    // lê o arquivo em binário
    ifstream myfile;
    char* fname = "f_bin.dat";
    myfile.open (fname);
    if (!myfile)
        cout << "File " << fname << " not open";

    // para ler o vetor todo de uma vez
    myfile.read((unsigned char*)&vetor,sizeof(vetor));

    // para ler um por um dos elementos do vetor
    // for (i=0 ; i<SIZE ; i++)
    //     myfile.read((unsigned char*)&vetor[i],sizeof(TYPE));

    // usa o vetor para alguma coisa

    myfile.close(); // fecha o arquivo

    // lê o arquivo em forma de texto
    fname = "f_txt.dat";
    myfile.open (fname);
    if (!myfile)
        cout << "File " << fname << " not open";
}
```

```
// Lê um por um os elementos do vetor
for (i=0 ; i<SIZE ; i++)
    myfile >> vetor[i];

// usa o vetor para alguma coisa
myfile.close(); // fecha o arquivo
}
```

Capítulo 9) Programação genérica (template)

A palavra `template` existe desde a versão 3 da linguagem C++ (não confundir com versão de algum compilador). Todos os compiladores atuais dão suporte a `template`, que pode ser traduzido como fôrma¹³, molde, gabarito. `template` é um recurso muito versátil de criação de funções e classes de tipo genérico, de tal forma que se pode criar vários modelos repetidos quase idênticos. Veja a sintaxe.

```
template < template-argument-list > declaration
```

Uma classe pode ser definida contendo um ou mais tipos genéricos (classes genéricas) definidas em `template`. Assim como a simples definição de uma classe não gera código, a definição de uma classe com `template` também não gera. Uma classe com `template` é apenas uma informação para o compilador de que uma “fôrma de bolo” (classe com `template`) foi definida. Após essa definição, é permitido que se defina o “sabor do bolo”, isto é, aplique-se um argumento definido para a classe genérica. Após ter aplicado um “sabor ao bolo”, obtém-se um tipo, que pode ser instanciado gerando objetos no programa.

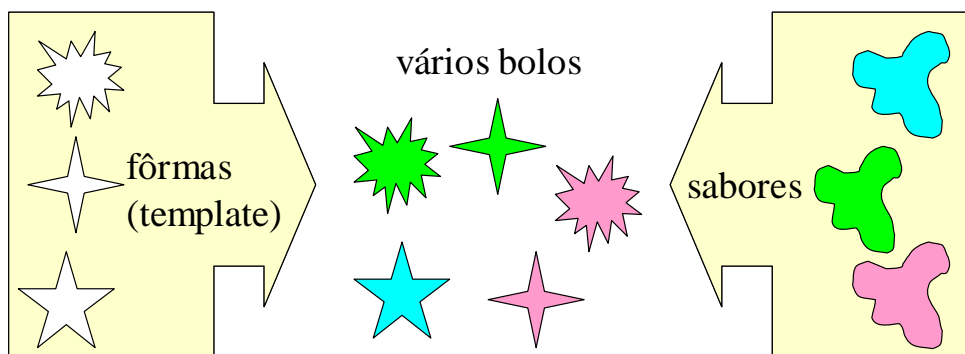


Figura 6: Analogia entre template e fôrma de bolo

Com `template`, pode-se desenvolver 2 tipos de elementos para programação genérica:

- Funções globais genéricas
- Classes genéricas

No exemplo abaixo, está definida uma matriz de dimensão fixa 3×3 com `template`. Essa é a “fôrma do bolo”. A partir dessa definição pode-se criar vários “bolos com fôrmas iguais e sabores diferentes”. No caso os “sabores” são os tipos `int`, `float`, `double`, `long double` e `complex`. Na classe `vb_matrix33` definida abaixo, o único método definido é `operator()`, que retorna um `lvalue`. Assim, os objetos das classes derivadas de `vb_matrix33` poderão usar o `operator()` de ambos os lados do sinal de `=`.

```
#include <complex.h>    // complex
#include <iostream.h>
// vou fazer uma matriz 3 por 3 com tipo generico. É a "forma do bolo"
template <class T>
class vb_matrix33 {
    T v[3][3];
public:
```

¹³ Na grafia moderna esta palavra perdeu o acento, porém a forma antiga de grafia foi mantida para diferenciar de fôrma (também sem acento).

```

    T & operator() (int i, int j) {return v[i][j];};
}

// definindo os tipos derivados, são os
// "bolos com fôrmas iguais e sabores diferentes"
typedef vb_matrix33 <int>          i_matrix;
typedef vb_matrix33 <float>        f_matrix;
typedef vb_matrix33 <double>       d_matrix;
typedef vb_matrix33 <long double>  ld_matrix;
typedef vb_matrix33 <complex>      c_matrix;

void main () {
    // um objeto de cada tipo
    i_matrix im1;
    f_matrix fm1;
    d_matrix dm1;
    ld_matrix ldml;
    c_matrix cml;
    // para testar, coloco um elemento em cada uma das matrizes
    i_ml(0,0) = 3;          cout << i_ml(0,0) << "\n";
    f_ml(1,1) = 4.44444;    cout << f_ml(1,1) << "\n";
    d_ml(1,1) = 1.11111;    cout << d_ml(1,1) << "\n";
    ld_ml(2,2) = 2.22222;    cout << ld_ml(2,2) << "\n";
    c_ml(1,1) = complex(1.1,6.6); cout << c_ml(1,1) << "\n";
}

```

Outro exemplo interessante é a função genérica com `template`. Pensemos na função `swap` genérica, que troca qualquer coisa com qualquer outra. Pode-se fazer a fôrma da função e deixar que o compilador transforme-a em realidade para os tipos que precisar. Veja o exemplo.

```

// generic template of function swap
template <class T> void swap(T & a, T & b) {
    T temp = a; a = b; b=temp;
}

void main () {
    double d1=1.1, d2=2.2;
    int i1=1, i2=2;
    swap (d1,d2);
    swap (i1,i2);
}

```

9.1 Uma lista encadeada simples com template

Pode-se escrever uma lista encadeada de uma classe genérica (isto é, usando `template`). No exemplo abaixo, a classe `VBList` é escrita usando `template`, e implementa uma lista simplesmente encadeada com alguns recursos. Note que dentro da classe `VBList` há uma outra classe chamada `node`, que é a classe que evamente vai ser replicada na lista. Essa classe `node` contém um dado do tipo genérico e um ponteiro para `node` `p_next`. A interface da `VBList` contém os seguintes métodos:

- construtor implícito (*default constructor*), e destrutor.
- `GetFirst` e `GetNext`, usados para varrer a lista.
- `add`, para acrescentar um elemento na lista, usando `operator=`.
- `remove`, para remover todos os elementos da lista que retornam verdadeiro com o `operator==`.
- `deleteAll`, para remover toda a lista.

O código fonte da `VBList` está mostrado abaixo.

```

// linked list template class
template <class dataType>
class VBList

```

```

{
class node
{
public:
    dataType data;
    node *p_next;
}; // end of class node

node *p_first, *p_last, *p_nextSave;
// void (*error_function)(); // pointer to function

public:
    // default constructor
    VBList()
    {
        p_first = p_last = p_nextSave = NULL;
        // error_function=error_fun;
    }; // end of VBList();

    dataType *GetFirst()
    {
        if (p_first)
        {
            p_nextSave = p_first->p_next;
            return &p_first->data;
        }
        else
            return NULL;
    };

    dataType *GetNext()
    {
        node *ret = p_nextSave;
        if (p_nextSave)
        {
            p_nextSave = p_nextSave->p_next;
            return &ret->data;
        }
        else
            return NULL;
    };

    // add data to list (addTail)
    bool add(const dataType &newData)
    {
        node *p_new;
        // ASSERT(p_new=new node);
        if ((p_new = new node) == NULL)
            return (1);
        // return true if allocation fails

        p_new->data = newData; // copies data to list
        // if new element is the first element of list
        if (!p_first)
        {
            p_first = p_last = p_new;
            p_new->p_next = NULL;
        }
        // if new element is NOT the first element of list
        else
        {
            p_last->p_next = p_new; // previous p_last points to new element
            p_new->p_next = NULL; // finnish list
            p_last = p_new; // now p_last points to new element
        }
    }
};

```

```

    }
    return (0);
}; // end of void add(dataType & newData)

// delete the whole list
void deleteAll()
{
    node *p_t1, *p_t2;
    if (p_first) // if list exists
    {
        p_t1 = p_first;
        p_t2 = p_first->p_next;
        do
        { // since list exists, at least 1 delete
            delete p_t1;
            p_t1 = p_t2;
            if (p_t2)
                p_t2 = p_t2->p_next;
        } while (p_t1 != NULL);
        p_last = p_first = NULL;
    }
}; // end void deleteAll()

// delete one data from list
void remove(const dataType & dataToDelete) {
    node *p_t1, *p_t2, *p_t3;
    if (p_first) // if list exists
    {
        p_t1 = p_first;
        p_t2 = p_first->p_next;

        // if data to delete is the first one
        // dataType must have operator== defined
        if (p_first->data == dataToDelete) {
            // for debug
            // cout << "DEBUG Deleted:"<< p_first->data << endl;
            delete p_first;
            p_first = p_t2;
            remove(dataToDelete); // recursively calls remove, to
            // be sure
        }
        else { // the data to delete is not the first one

            // this loop will seek for data to delete and delete it
            while (p_t1->p_next != NULL)
            { // since list exists, at least 1 delete

                if (p_t2->data == dataToDelete) {
                    // for debug
                    // cout << "DEBUG Deleted:"<< p_t2->data << endl;
                    p_t3 = p_t2->p_next;
                    delete p_t2;
                    p_t1->p_next = p_t3; // re-links the list (bypass p_t2)
                    p_t2 = p_t3; // to keep going
                }
                else {
                    // move pointers one step into the list
                    // only if no remove was made.
                    p_t1 = p_t2;
                    p_t2 = p_t2->p_next;
                }
            } // while
        } // else
    } // if p_first
}

```

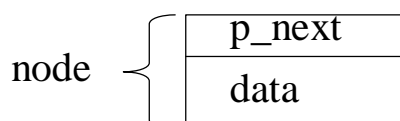
```

} // remove

// default destructor
~VBList()
{
    deleteAll();
}; // end of ~VBList();
}; // end of template <class dataType>

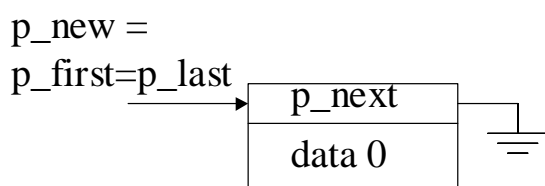
```

A lista é um encadeamento de nodes, sendo que um node é uma classe que contém 2 campos. Um é um ponteiro para node, que é carregado com o endereço do próximo node da lista (se for o último da lista usa-se o valor NULL, isto é, zero). A visualização gráfica de um node está mostrado na figura abaixo.



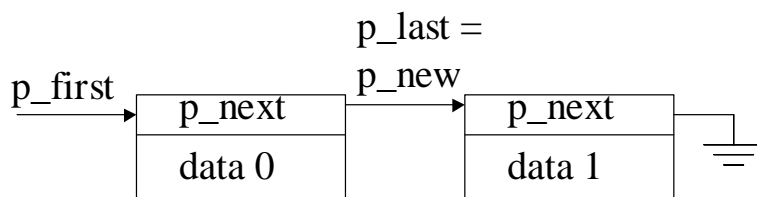
Após o construtor, `p_first` e `p_last` estão inicializados com NULL (zero). O método `add` faz a lista alocar espaço para mais um dado e encadea-lo aos demais. Para fazer isso o método `add` aloca um node e faz uma variável local tipo (ponteiro para node) chamada `p_new` apontar para o novo node.

Se o node alocado for o primeiro de todos (data 0), os dados membro da classe `VBList` chamados `p_first` e `p_last` serão inicializados apontando para esse único node. O campo `p_next` desse node será inicializado com NULL e o dado será copiado para o campo de dados do node.

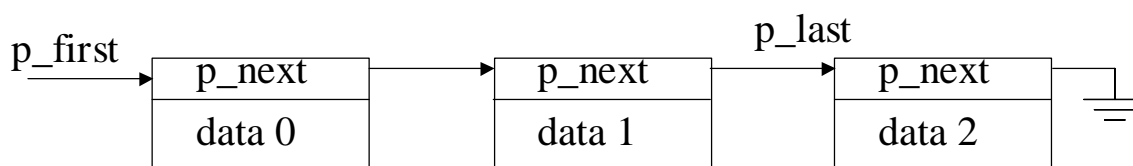


Com a instrução `p_new->data = newData;` copia-se os dados para o campo `data`. Lembre-se que os dados são copiados de forma genérica, isto é, com `template`. Quando se escreve a classe de lista, que nesse caso chama-se `VBList`, não se sabe a priori qual é o método que copia os dados, justamente porque a classe que chama esse método é genérica (`template`). Apenas sabe-se que para que uma classe de entrada que seja usada nessa lista, é preciso que o método `operator=` esteja definido. Ao efetuar a compilação, mesmo que não haja erro de sintaxe na classe `VBList` (a forma do bolo), pode surgir um erro de compilação quando se aplicar nessa lista uma classe de entrada (sabor do bolo) que não tenha o método `operator=` definido, ou que tenha bug nesse método.

Com a entrada do próximo dado na lista (data 1), o método `add` aloca novo node, faz o ponteiro que era o último da lista (nesse caso o ponteiro do node 0), apontar para o novo node, faz o próximo do node novo apontar para zero e atualiza a variável membro da classe `VBList` chamada `p_last` para que aponte para o novo último node da lista.



Chamando mais uma vez o método add (data 2), a lista ficará com a representação gráfica como mostrado na figura abaixo. Nessa representação, não é mostrado a variável temporária `p_new`.



Por várias razões pode-se querer varrer a lista desde o primeiro elemento até o último. Isso pode ser feito com o laço de programação mostrado abaixo, que usa um ponteiro para `dataType` e os métodos `GetFirst` e `GetNext`. No exemplo abaixo, o laço varre a lista e coloca cada elemento no console, usando o operador inserção (`operator<<`).

```

void g() {
    typedef int TYPE;
    VBList<TYPE> listObj;
    // add data to list
    TYPE *p; // a pointer to dataType
    for (p = listObj.GetFirst() ; p ; p = listObj.GetNext() ) {
        cout << *p << endl;
    }
}

```

Suponha que se deseje procurar um elemento da lista com uma característica específica, digamos cujo conteúdo do campo “telephone” seja igual a “222-2222”. Essa busca pode ser feita fazendo a comparação em questão dentro do loop que varre a lista.

No exemplo abaixo, definiu-se uma classe chamada `data_tel_name`, que contém 2 campos de dados do tipo `VBString`, contendo nome e telefone. Para essa classe, definiu-se o `operator=` e `operator==`, além de alguns outros métodos. Para simplificar, definiu-se um construtor com 2 parâmetros, que permite fazer a entrada de dados diretamente (sem a necessidade de um objeto separado).

```

#include "vblib.h" // VBString, VBList

// the data class must have operator= and operator== defined
class data_tel_name {
    VBString m_telephone, m_name;
public:
    // default constructor
    data_tel_name() {
        m_name = "no name";
        m_telephone = "no telephone";
    };

    // constructor with parameters
    data_tel_name(VBString name, VBString tel) {
        m_name = name;
        m_telephone = tel;
    };

    VBString getName () { return m_name; }
    VBString getPhone () { return m_telephone; }

    void operator=(const data_tel_name & a) {
        m_telephone = a.m_telephone;
        m_name = a.m_name;
    };

    // return true if a==inner. return false if a!=inner
    // compares only telephone, ignores other fields
    bool operator==(const data_tel_name & a) {

```

```

    return m_telephone == a.m_telephone;
};
}; // end of data_tel_name

void main () {
    VBList<data_tel_name> listObj; // listObj is instance of myTelNameList

    // add some data
    listObj.add(data_tel_name("Sergio","222-2222"));
    listObj.add(data_tel_name("Marcia","333-3333"));
    listObj.add(data_tel_name("Camila","444-4444"));
    listObj.add(data_tel_name("Mateus","555-5555"));

    VBString searchPhone;
    cout << "Enter telephone to search: ";
    cin >> searchPhone;

    data_tel_name *p; // a pointer to the data type
    bool found;
    // this loop scans the entire list
    for (p = listObj.GetFirst() ; p ; p = listObj.GetNext() ) {
        data_tel_name a = data_tel_name("",searchPhone);
        found = (a == *p);
        if (found) break;
    }

    if (found)
        cout << "The name of person with telephone \"" << p->getPhone() <<
            "\" is \"" << p->getName() << "\"<< endl;
    else
        cout << "No person was found with telephone \"" <<
            searchPhone << "\"<< endl;
}

```

Nesse programa, um objeto do tipo lista é criado, e alguns dados são acrescentados à lista. Em seguida, pergunta-se por um telefone. O laço de varredura da lista é usado para procurar elemento a elemento por um retorno válido do operador== (que por sua vez, compara apenas o campo de telefone). Caso o elemento seja encontrado, termina-se a procura. Caso contrário, faz-se a varredura até o fim.

Após a varredura, mostra-se o que foi encontrado ou informa-se que o elemento não foi encontrado. Um exemplo de saída poderia ser como abaixo.

```

Enter telephone to search: 222-2222
The name of person with telephone "222-2222" is "Sergio"

```

Outro exemplo poderia ser como abaixo.

```

Enter telephone to search: 222-3333
No person was found with telephone "222-3333"

```

9.2 VBMath - uma biblioteca de matemática matricial usando classe template

Um exemplo interessante de uso de classe genérica é a elaboração de classes para manipulação de matrizes (ou vetores). Nessa seção, será mostrado o funcionamento de uma biblioteca de manipulação matemática matricial usando `template`. Essa biblioteca chama-se `VBmath` (Villas-Boas math), e pode ser encontrada na Internet no endereço abaixo.

<http://www.del.ufrj.br/~villas/cpplibs/vbmath/>

Na listagem abaixo, está mostrado um esqueleto de uma classe de matriz usando um parâmetro como classe genérica.

```
template <class REAL>
class vb_matrix {
    // member data
    REAL *data;
    unsigned rows;    // number of rows.
    unsigned cols;    // number of columns.
public:
    // methods ...
};
```

Uma classe de matriz definida assim é uma “forma de bolo”, cujos “sabores” poderão ser por exemplo: float, double, long double, etc. Aplicando-se os sabores a forma do bolo, cria-se um tipo para cada sabor. Isso pode ser feito com o código abaixo.

```
typedef vb_matrix<long double> vb_longDoubleMatrix;
typedef vb_matrix<double> vb_doubleMatrix;
typedef vb_matrix<float> vb_floatMatrix;
```

Tendo definido os tipos, pode-se instanciar os objetos e usa-los, com suas funções membro. Abaixo estão listados alguns exemplos didáticos de uso da `VBmath`. O primeiro exemplo é a simples alocação criação de uma matriz de double, com dimensão 2×2. Essa matriz é carregada com constantes e em seguida é exteriorizada pela stream diretamente em `cout`. A dimensão da matriz é passado para o objeto como parâmetro. O objeto `M` do tipo `vb_doubleMatrix` aloca a memória necessária para os dados no construtor. O objeto `M` pode usar normalmente os dados até que seja terminado o seu escopo, quando será chamado o destrutor, que libera a memória alocada. Entre outros, está definido o método `operator()`, que retorna um lvalue do tipo `REAL` &, sendo que `REAL` é o tipo genérico (template) de elemento da matriz. Por isso, pode-se acessar os elementos da matriz de ambos os lados de um operador de designação (*assignment operator*). O método `operator<<` contém informações que permitem um objeto da classe ser colocado numa stream.

```
// vb_matrix (unsigned rows, unsigned cols=1);
// friend ostream & operator<< (ostream & stream, vb_matrix<REAL> & obj);
#include "vbmath.h" // main include file

void main() {
    int s=2;
    vb_doubleMatrix M(s,s);
    M(0,0) = 1.1;
    M(0,1) = 2.1;
    M(1,0) = 3.1;
    M(1,1) = 4.1;
    cout << M;
}
```

Resultado

```
(2,2)
+1.1000 +2.1000
+3.1000 +4.1000
```

No exemplo abaixo, demonstra-se o método `operator=`, que faz a cópia elemento a elemento do objeto `M` para o objeto `N`. O resultado é o mesmo anterior. No caso da `VBmath`, o método `operator=` retorna `void` (isto é, não retorna).

```
// void operator=(vb_matrix<REAL> & x);
#include "vbmath.h" // main include file

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1.1;
    M(0,1) = 2.1;
    M(1,0) = 3.1;
    M(1,1) = 4.1;
    vb_doubleMatrix N=M;
```

```
    cout << N;
}
```

Em C/C++ o método `operator=` padrão retorna o mesmo tipo da classe. Com isso, é possível escrever `a=b=c=d;` por exemplo. Por motivo de desempenho, não é recomendável que o método `operator=` retorne dessa forma no caso de uma biblioteca matricial.

No exemplo abaixo, demonstra-se o método `operator+()`, nesse caso somando a matriz com um `REAL` (o mesmo tipo do elemento da matriz). Nesse caso, a soma é feita elemento a elemento.

```
// friend vb_matrix<REAL> operator+(vb_matrix<REAL> & mat, REAL re);
#include "vbmath.h" // main include file

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1;
    M(0,1) = 2;
    M(1,0) = 3;
    M(1,1) = 4;
    vb_doubleMatrix N=M+1.1;
    cout << N;
}
```

Resultado

```
(2,2)
+2.1000 +3.1000
+4.1000 +5.1000
```

No exemplo abaixo, demonstra-se o método `operator*`, nesse caso multiplicando a matriz com um `REAL` (o mesmo tipo do elemento da matriz). Nesse caso, a multiplicação é feita elemento a elemento.

```
// friend vb_matrix<REAL> operator*(vb_matrix<REAL> & mat, REAL re);
#include "vbmath.h" // main include file

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1.1;
    M(0,1) = 2.1;
    M(1,0) = 3.1;
    M(1,1) = 4.1;
    vb_doubleMatrix N=M*2.5;
    cout << N;
}
```

Resultado

```
(2,2)
+2.7500 +5.2500
+7.7500 +10.2500
```

No exemplo abaixo, demonstra-se o método `operator*`, nesse caso multiplicando a matriz com outra matriz. Nesse caso, a operação é a multiplicação matricial seguindo as normas matemáticas. O retorno da função é um objeto tipo matriz com as dimensões adequadas. Caso a operação de multiplicação matricial seja impossível por problema de dimensão dos argumentos, um erro de execução é gerado.

A resposta da multiplicação é armazenado no objeto `k1`, criado diretamente, ou no objeto `k2` criado inicialmente e posteriormente recebendo o retorno da função. A tentativa de multiplicar as matrizes de forma incompatível com suas dimensões resulta em erro, tratado pela própria biblioteca.

```
// vb_matrix<REAL> operator* (vb_matrix<REAL> & arg);
#include "vbmath.h" // main include file

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1.1;
    M(0,1) = 2.1;
```

```

M(1,0) = 3.1;
M(1,1) = 4.1;
vb_doubleMatrix N(2,1);
N(0,0) = 3.3;
N(1,0) = 7.3;
vb_doubleMatrix K1=M*N;
vb_doubleMatrix K2; K2=M*N;
cout << M << N << K1 << K2;
vb_doubleMatrix J=N*M; // execution error, incompatible simensions
}

```

Resultado

```

(2,2)
+1.1000 +2.1000
+3.1000 +4.1000

```

```

(2,1)
+3.3000
+7.3000

```

```

(2,1)
+18.9600
+40.1600

```

```

(2,1)
+18.9600
+40.1600

```

```

=====
VBmath - The Villas-Boas library for mathematics in C++ multiplatform
Version 1.0 of October 27, 1999
http://www.del.ufrj.br/~villas/cpplib/vbmath/
Error:arg1.cols must be equal to arg2.rows to perform matrix multiplication
=====

```

No exemplo abaixo, demonstra-se o método `operator+`, nesse caso somando a matriz com outra matriz. Também mostra-se que um objeto matriz pode ser atribuído seguidamente sem problemas. Na primeira vez que o objeto recebe uma atribuição, aloca-se memória para os dados. Na vez seguinte que o objeto recebe atribuição, a memória é liberada e nova memória é alocada, tudo automaticamente.

```

// vb_matrix<REAL> operator+ (vb_matrix<REAL> & arg);
#include "vbmath.h" // main include file

```

```

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1.5;
    M(0,1) = 2.5;
    M(1,0) = 3.5;
    M(1,1) = 4.5;
    vb_doubleMatrix N(2,2);
    N(0,0) = 1.1;
    N(0,1) = 2.2;
    N(1,0) = 3.3;
    N(1,1) = 4.4;
    vb_doubleMatrix K=M-N;
    K=M+N;
    cout << M << N << K;
}

```

Resultado

```

(2,2)
+1.5000 +2.5000
+3.5000 +4.5000

```

```

(2,2)

```

```
+1.1000 +2.2000
+3.3000 +4.4000
```

```
(2,2)
+2.6000 +4.7000
+6.8000 +8.9000
```

O exemplo abaixo demonstra o método `inv`, que calcula a inversa da matriz. A matriz M multiplicada pela sua inversa K é a matriz identidade J .

```
#include "vbmath.h" // main include file

void main() {
    vb_doubleMatrix M(2,2);
    M(0,0) = 1.1;
    M(0,1) = 2.1;
    M(1,0) = 3.1;
    M(1,1) = 4.1;
    vb_doubleMatrix K;
    K=M.inv();
    vb_doubleMatrix J=K*M;
    cout << M << K << J;
}
```

Resultado

```
(2,2)
+1.1000 +2.1000
+3.1000 +4.1000
```

```
(2,2)
-2.0500 +1.0500
+1.5500 -0.5500
```

```
(2,2)
+1.0000 +0.0000
+0.0000 +1.0000
```

O exemplo abaixo tem uma função `t`, cujo parâmetro é a dimensão de uma matriz quadrada. A função preenche a matriz com dados aleatórios e calcular a inversa dessa matriz. Em seguida, multiplica a matriz original pela sua inversa (o que gera a matriz identidade com dimensão igual ao parâmetro de entrada). Em seguida, mostra as 3 matrizes.

```
#include "vbmath.h" // main include file

void t(unsigned size=3) {
    unsigned i,j;
    vb_longDoubleMatrix M(size,size);
    for ( i = 0; i < size; ++i )
        for ( j = 0; j < size; ++j )
            M(i,j) = rand()/10000.0;
    vb_longDoubleMatrix K;
    K=M.inv();
    vb_longDoubleMatrix J=K*M;
    cout << M << K << J;
}

void main() {
    t(5);
}
```

Resultado

```
(5,5)
+0.0041 +1.8467 +0.6334 +2.6500 +1.9169
+1.5724 +1.1478 +2.9358 +2.6962 +2.4464
+0.5705 +2.8145 +2.3281 +1.6827 +0.9961
+0.0491 +0.2995 +1.1942 +0.4827 +0.5436
```

```
+3.2391 +1.4604 +0.3902 +0.0153 +0.0292
```

```
(5,5)
```

```
-0.2580 +0.3976 +0.0471 -0.9724 +0.1225
```

```
+0.6938 -0.9660 -0.1837 +2.2124 +0.4669
```

```
-0.5682 +0.4117 +0.4011 -0.5699 -0.2611
```

```
-1.7145 +2.4306 +1.6755 -7.8903 -1.3533
```

```
+2.4118 -2.5664 -2.2720 +8.9669 +1.5070
```

```
(5,5)
```

```
+1.0000 +0.0000 +0.0000 -0.0000 -0.0000
```

```
-0.0000 +1.0000 -0.0000 +0.0000 -0.0000
```

```
+0.0000 -0.0000 +1.0000 -0.0000 -0.0000
```

```
+0.0000 +0.0000 +0.0000 +1.0000 +0.0000
```

```
+0.0000 -0.0000 -0.0000 +0.0000 +1.0000
```

Capítulo 10) Tratamento de exceção (*exception handling*)

Quando se usa programação orientada a objetos, o programador pode (e deve) usar objetos, que podem ter construtor e destrutor. O tratamento de exceções geralmente é usado no tratamento de algum tipo de erro que foi detectado durante a execução. Possivelmente esse erro conduz a um procedimento de finalização que leva ao fim do programa. Na programação procedural, o tratamento de exceção pode ser feito a partir de um simples goto, mas isso não pode ser feito na programação orientada a objetos, pois pode haver objetos antes do goto que requerem a chamada de destrutor. O goto pura e simplesmente não faz chamar os destrutores dos objetos.

A forma recomendada pela programação orientada a objetos para o tratamento de exceções é o uso de estruturas baseadas nas palavras reservadas abaixo.

- try (tentar)
- catch (capturar)
- throw (jogar)

Durante a execução de um programa, a execução ocorre normalmente de cima para baixo, e pode haver um “bloco try-catch”. Esse bloco é caracterizado por um bloco try seguido de um ou mais blocos catch. Ao fim do último catch após o try, termina-se o “bloco try-catch”. No exemplo abaixo, há bloco try-catch com um bloco try seguido de dois blocos catch. O programa vem sendo executado antes do bloco try-catch, executa o bloco try-catch (de acordo com as regras que discute-se nesse capítulo), e segue sendo executado de cima para baixo.

```
// program14

try {
    // code can throw an exception
}

catch (someProblem) {
    // code for someProblem
}

catch (someOtherProblem) {
    // code for someOtherProblem
}

// program
```

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

¹⁴ someProblem = algumProblema, someOtherProblem = algumOutroProblema, code=código, code for someProblem = código para algumProblema, code can throw an exception = o código pode jogar uma exceção

```
class CatchHandler {
public:
    const char *ShowExplanation() const {
        return "CatchHandler Explanation";
    }
};

class SomeClass {
public:
    SomeClass() {cout << "SomeClass constructor" << endl;};
    ~SomeClass(){cout << "SomeClass destructor" << endl;};
};

void FunctionUnderTest() {
    SomeClass D;
    throw CatchHandler(); // the throw produces an exception
    SomeClass D2;
}

void main() {
    cout << "In main." << endl;

    try {
        cout << "In try block, calling FunctionUnderTest()." << endl;
        FunctionUnderTest();
    }

    catch( CatchHandler E ) {
        cout << "CatchHandler exception type: "
            << E.ShowExplanation() << endl;
    }

    cout << "Back in main. Execution resumes here." << endl;
}
```

Resultado

```
In main.
In try block, calling FunctionUnderTest ().
SomeClass constructor
SomeClass destructor
CatchHandler exception type: Catch Handler Explanation
Back in main. Execution resumes here.
```

Capítulo 11) RTTI – Identificação em tempo de execução

11.1 Introdução

Um recurso relativamente recente da linguagem C++ é a RTTI (*Real Time Type Identification*), ou identificação em tempo de execução. Sem esse recurso, a identificação de tipo somente podia ser feita em tempo de compilação. Mas usando RTTI, desde que é possível a identificação de tipo em tempo de execução, novos recursos lógicos podem ser incorporados aos programas.

As novas palavras reservadas (ou identificadores da biblioteca padrão) que foram criadas para RTTI são:

- typeid
- dynamic_cast
- static_cast
- const_cast

11.2 typeid

Pode-se saber o nome de um tipo diretamente, ou comparar dois tipos com typeid. No exemplo abaixo, o nome do tipo de 3 variáveis é retornado, e mostrado no console. Tanto tipos padrão quanto tipos do usuário podem ser obtidos. Além disso, os tipos podem ser comparados. No caso, uma função global printSameType recebe um parâmetro booleano e imprime se os tipos são os mesmos ou não no console. O programa de exemplo chama essa função global duas vezes para mostrar o caso em que os tipos em comparação são iguais e o caso em que não são iguais.

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

class myClass {
    // ...
};

void printSameType(bool b) {
    if (b)
        cout << "Same type" << endl;
    else
        cout << "NOT SAME TYPE" << endl;
}

void main () {
    int i,j;
    float f;
    myClass a;
    cout << "The type of i is " << typeid(i).name() << endl;
    cout << "The type of f is " << typeid(f).name() << endl;
    cout << "The type of a is " << typeid(a).name() << endl;
    bool sameType = (typeid(i) == typeid(j));
    printSameType(sameType);
    sameType = (typeid(i) == typeid(a));
```

```
    printSameType(sameType);  
}
```

A saída do programa é como mostrado abaixo.

```
The type of i is int  
The type of f is float  
The type of a is class myClass  
Same type  
NOT SAME TYPE
```

A obtenção dos tipos em tempo de execução pode ser de interesse particularmente grande quando se está usando união tardia (*late bind*) entre um ponteiro e um objeto. Como foi explicado na seção 7.10 (página 114).

Capítulo 12) namespace

12.1 Introdução

O conceito de namespace foi introduzido na linguagem C++ para que se evite confusões com nomes de identificadores globais (com visibilidade para todo o programa). Observa-se na prática que há certos identificadores que são “manjados”, isto é, candidatos óbvios a certas aplicações. Identificadores globais tais como `matrix`, `vector`, `list`, `complex`, `string`, `errorHandler`, etc. podem facilmente já ter sido definidos por algum outro desenvolvedor.

No exemplo abaixo, são definidos dois namespaces, sendo que nos dois há um identificador global chamado `errorHandler`, que é uma função. No programa principal, a função `errorHandler` é chamada duas vezes, sendo que o namespace a que pertence a função é explicitado pelo operador de escopo (`::`).

```
#include <iostream.h>
namespace oneNamespace {
    void errorHandler() {
        cout << "This is the error handler of oneNamespace" << endl;
    }
}
namespace otherNamespace {
    void errorHandler() {
        cout << "This is the error handler of otherNamespace" << endl;
    }
}
void main () {
    oneNamespace::errorHandler();
    otherNamespace::errorHandler();
}
```

A saída do programa é como mostrado abaixo.

```
This is the error handler of oneNamespace
This is the error handler of otherNamespace
```

Para simplificar o uso de namespace, existe a palavra reservada “`using`”. Essa palavra faz referência a um namespace e declara para o compilador que a partir daquele ponto o namespace em questão deve ser tomado implicitamente. Veja o programa acima como ficaria se o namespace “`oneNamespace`” fosse declarado implícito com “`using`”.

```
#include <iostream.h>
namespace oneNamespace {
    void errorHandler() {
        cout << "This is the error handler of oneNamespace" << endl;
    }
}
namespace otherNamespace {
    void errorHandler() {
        cout << "This is the error handler of otherNamespace" << endl;
    }
}
using namespace oneNamespace;
void main () {
    errorHandler(); // uses namespace oneNamespace by default
    otherNamespace::errorHandler();
}
```

Na “gramática” do C++, nada impede o uso implícito de mais de um namespace, desde que não haja ambiguidade. Por exemplo: o programa abaixo é ambíguo, e não compila;

```
#include <iostream.h>
namespace oneNamespace {
    void errorHandler() {
        cout << "This is the error handler of oneNamespace" << endl;
    }
}
namespace otherNamespace {
    void errorHandler() {
        cout << "This is the error handler of otherNamespace" << endl;
    }
}
using namespace oneNamespace;
using namespace otherNamespace;
void main () {
    errorHandler(); // error ! ambiguous.
}
```

Caso a função global em questão tivesse nome diferente, então não haveria ambiguidade e nesse caso o programa compilaria normalmente com mais de um namespace implícito.

```
#include <iostream.h>
namespace oneNamespace {
    void errorHandler() {
        cout << "This is the error handler of oneNamespace" << endl;
    }
}
namespace otherNamespace {
    void errorHandlerNewName() {
        cout << "This is the error handler of otherNamespace" << endl;
    }
}
using namespace oneNamespace;
using namespace otherNamespace;
void main () {
    errorHandler();
    errorHandlerNewName();
}
```

A saída do programa é como mostrado abaixo.

```
This is the error handler of oneNamespace
This is the error handler of otherNamespace
```

12.2 Namespace aberto

Uma característica interessante do namespace é o fato de que ele é de escopo aberto, ou melhor, expansível. Isso significa que a qualquer momento pode-se expandir o escopo de um namespace, e não é necessário que todos os nomes do escopo de um namespace sejam declarados juntos. No exemplo abaixo, o namespace oneNamespace é declarado com uma função global, em seguida o namespace otherNamespace é declarado com uma função global de mesmo nome, e posteriormente o namespace oneNamespace é expandido com mais uma função global.

```
#include <iostream.h>
namespace oneNamespace {
    void errorHandler() {
        cout << "This is the error handler of oneNamespace" << endl;
    }
}
namespace otherNamespace {
    void errorHandler() {
```

```
        cout << "This is the error handler of otherNamespace" << endl;
    }
}
namespace oneNamespace { // continuation of oneNamespace
    void otherFun() {
        cout << "otherFun of oneNamespace" << endl;
    }
}
void main () {
    oneNamespace::otherFun();
}
```

A saída do programa é como mostrado abaixo.

```
otherFun of oneNamespace
```

12.3 Biblioteca padrão de C++ usando namespace

A própria biblioteca padrão é uma grande usuária de identificadores “manjados”. Portanto, a partir de que o conceito de namespace foi introduzido em C++, os identificadores da biblioteca padrão foram isolados num namespace, chamado de “std” (abreviatura de *standard*, isto é, “padrão”). Muitos headers padrão foram re-escritos para acomodarem-se ao padrão de uso de namespace. Quando um header padrão é re-escrito, o seu nome torna-se o mesmo, apenas eliminando-se a extensão “.h” no final. O programa típico “hello world” (olá mundo), em C++ sem namespace é escrito como abaixo.

```
// hello world in C++ without namespace
#include <iostream.h>
void main () {
    cout << "Hello" << endl;
}
```

Esse mesmo programa usando namespace é fica como abaixo.

```
// hello world in C++ WITH namespace
#include <iostream>
using namespace std;
void main () {
    cout << "Hello" << endl;
}
```

Outra versão do programa hello world usando namespace é o programa abaixo.

```
// hello world in C++ WITH namespace
#include <iostream>
void main () {
    std::cout << "Hello" << endl;
}
```

Para quem quer ou precisa usar namespace a partir da biblioteca padrão, a forma mais fácil e imediata de fazê-lo é substituir a inclusão dos headers padrão com extensão “.h” pelas versões de mesmo nome sem a extensão “.h”, e acrescentar o comando para uso implícito do namespace std, com a linha abaixo (após a inclusão dos headers padrão).

```
using namespace std;
```

O motivo para se usar a versão da biblioteca padrão com namespace é o fato de que algumas funcionalidades mais novas dessa biblioteca somente estão disponíveis na versão com namespace. Por exemplo: a classe de string da biblioteca padrão somente pode ser usada com namespace. Outro exemplo: a classe complex (para números complexos) sofreu algumas melhorias na versão que é usada com namespace. Desde que se use uma versão moderna do compilador (o que não costuma ser difícil), o uso de namespace não representa qualquer problema.

Há alguns detalhes a serem observados quando se usa namespace std (da biblioteca padrão). O mais importante é que não se deve misturar header sem namespace (com extensão “.h”) com header que tem versão para namespace (sem extensão “.h”). Em outras palavras: caso se esteja usando a biblioteca padrão na versão com namespace, todos os headers padrão devem ser da nova versão. Exemplo:

```
// certo, todos os headers na versão com namespace
#include<iostream>
#include<string>
using namespace std;
// ...
```

Outro exemplo:

```
// errado, misturando versões de header com namespace e sem namespace
#include<iostream>
#include<string.h>
using namespace std;
// ...
```

12.4 Adaptando uma biblioteca existente para uso de namespace std

Seja o caso de um programador que seja autor de uma biblioteca não padrão, foi desenvolvida originalmente sem uso de namespace. Digamos que essa biblioteca se chama mylib, e é composta por dois arquivos: mylib.cpp e mylib.h.

Suponha que essa biblioteca precise da inclusão de algum header padrão, apenas para poder compilar o seu próprio header mylib.h. Isso pode ocorrer caso exista uma classe nessa biblioteca para a qual se sobrecarregou o insersor (operator<<) (veja a seção 8.3, na página 126). No exemplo abaixo, a classe myClass declarou um protótipo da sobrecarga do insersor na sua descrição. Mas para seja possível compilar, o identificador “ostream” precisa estar declarado antes da classe myClass. Como trata-se de um identificador da biblioteca padrão, a solução simples é incluir o header padrão iostream.h. Essa inclusão pode ser feita dentro ou fora do header mylib.h. Vamos supor que seja feita dentro, como mostrado abaixo.

```
// mylib.h
#include <iostream.h>
class myClass {
    // ...
public:
    friend ostream & operator<<(ostream & s, const myClass & obj);
};
```

Nesse caso, para usar a classe myClass, basta incluir o header da biblioteca, como mostrado abaixo.

```
// main.cpp
#include "mylib.h"
void main () {
    myClass a;
    cout << a << endl;
}
```

A questão é: como uma biblioteca como essa deve ser adaptada caso se deseje usar namespace std, sendo que é sabido que não se pode misturar o header padrão com namespace std e sem namespace std? Uma solução para esse problema é criar uma versão alternativa do header da biblioteca comandada por um define, e compilada de uma forma ou de outra com diretivas de compilação. No exemplo abaixo, a existência da definição de MYLIB_USE_NAMESPACE_STD comanda as duas versões possíveis para o header de mylib.

```
// mylib.h
#ifdef MYLIB_USE_NAMESPACE_STD
    #include <iostream>
    using namespace std;
#else
    #include <iostream.h>

```

```
#else
    #include <iostream.h>
#endif
class myClass {
    // ...
public:
    friend ostream & operator<<(ostream & s, const myClass & obj);
};
```

Na versão abaixo de utilização da biblioteca mylib, sem uso de namespace std, nenhuma modificação é requerida em relação a primeira versão da biblioteca (desenvolvida sem o conceito de namespace). Se arquivos adicionais da biblioteca padrão forem necessários, basta incluí-los antes da inclusão de mylib.h.

```
// main.cpp, versão sem namespace std
#include <fstream.h> // inclusão de outros headers padrão na versão sem namespace std
#include <string.h>  // inclusão de outros headers padrão na versão sem namespace std
#include "mylib.h"
void main () {
    myClass a;
    cout << a << endl;
    // ...
}
```

Usando o mesmo header mylib.h anterior, agora numa utilização que requer o uso de namespace std, basta declarar o define MYLIB_USE_NAMESPACE_STD antes da inclusão de mylib.h. Essa declaração fará com que o header mylib.h use a versão compatível com namespace std. Assim, a compilação é possível, mesmo se for necessário incluir (antes de mylib.h) outros headers padrão na versão com namespace.

```
// main.cpp, versão COM namespace std
#include <fstream> // inclusão de outros headers padrão na versão COM namespace std
#include <string>  // inclusão de outros headers padrão na versão COM namespace std
#define MYLIB_USE_NAMESPACE_STD
#include "mylib.h"
// daqui para baixo, o programa é igual à versão sem namespace std
void main () {
    myClass a;
    cout << a << endl;
    // ...
}
```

Capítulo 13) STL - Standard Template Library

13.1 Introdução

Esse capítulo fala de forma resumida sobre STL, ou Standard Template Library - que em português significa algo como “biblioteca padrão genérica”. O termo “genérica” nesse caso significa “programação genérica”, como explicado no capítulo de “Programação genérica (template)”, na página 140. O leitor interessado em STL deve considerar o exposto nesse capítulo apenas como uma introdução, e procurar referências mais completas para uma leitura mais aprofundada.

Para quem usa Visual C, procure por exemplos de STL procurando no help por “STL sample programs”.

STL é atualmente considerada como parte da biblioteca padrão de C++. A idéia básica do STL é implementar um conjunto de componentes típicos com template, de forma a que esses componentes possam ser usados de forma genérica. STL dá ênfase no uso de classes contenedoras (*container*) e iteradoras (*iterator*)¹⁵, e também na implementação de algoritmos genéricos. As classes contenedoras são listas encadeadas, vetores, mapas, etc. As classes iteradoras são aquelas que permitem apontar para os elementos das classes contenedoras, e varrer esses elementos. Os algoritmos são `find`, `sort`, `binary_search`, etc.

A biblioteca STL é parte da biblioteca padrão de C++, mas merece atenção separada, por sua estrutura, sua abrangência e sua generalidade. O conceito de programação genérica (com template) é independente do conceito de namespace. A biblioteca STL pode ser usada com ou sem namespace `std`. Algumas das funcionalidades mais novas e sofisticadas da biblioteca padrão de C++ somente estão implementadas com namespace `std` (e.g classe de string, novidades na classe `complex`, etc.). Desde que o uso de STL também é de certa forma um assunto sofisticado, estou assumindo que quem está interessado em STL tem acesso a um compilador C++ moderno que tenha suporte pleno a namespace. Todos os exemplos desse capítulo usam namespace `std`. Caso o leitor procure referências adicionais sobre STL, com certeza irá encontrar exemplos em que STL é usado sem namespace. Como já foi dito, STL é baseado no conceito de programação genérica (com template), que não tem relação com o conceito de namespace.

13.1.1 Classes contenedoras

STL provê diferentes tipos de classes contenedoras, sempre formuladas com template para que possam ser usadas de forma genérica. As classes contenedoras - `vector`, `list`, `map` - tem um método chamado `size()`, por exemplo, que retorna o número de elementos que estão contidos na classe contenedora. Há também outros métodos, tais como `begin()` e `end()`, que são usados para localizar o primeiro elemento e a primeira posição *após* o último elemento. Uma classe contenedora vazia possui valores iguais para o retorno dos métodos `begin()` e `end()`.

¹⁵ Em latim, *iterare* significa repetir. Não confundir com “interar” (verbo que não existe, mas que lembra “interação”, ou “interativo”).

13.1.2 Classes iteradoras

Os iteradores são como ponteiros. São usados para acessar os elementos de uma classe contenedora. Os iteradores podem mover-se de um elemento para o outro, sendo que a implementação do código dessa movimentação é implícita para quem usa os iteradores. Por exemplo, usando um objeto da classe contenedora `vector`, o operador `++` sobre um objeto iterador significa passar para a posição de memória onde encontra-se o próximo elemento armazenado.

13.1.3 Algoritmos

Os algoritmos implementados funcionam com iteradores acessando objetos de classes contenedoras.

Os objetos de classes contenedoras podem ser acessados por iteradores, e os algoritmos usam os iteradores. Dessa forma, os algoritmos podem ser usados de forma excepcionalmente clara.

contenedores \Leftrightarrow iteradores \Leftrightarrow algoritmos

13.2 Preâmbulo

Para prepararmos-nos para a programação no estilo STL, vamos considerar o exemplo abaixo. No exemplo, cria-se uma classe contenedora com dados do tipo `'int'`. Cria-se um iterador para esse tipo. Escreve-se a função `my_find` para encontrar um elemento na classe contenedora. No programa principal enche-se a classe contenedora com dados (números pares). A função `my_find` encontra a posição relativa a ocorrência dos valores entrados pelo usuário.

A elegância desse código está no fato de que o algoritmo de encontrar um valor foi escrito sem que se considere o tipo a ser procurado (no caso `'int'`), ou que se considere como o iterador vai para a próxima posição (no caso `'begin++'`).

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

// creating the new type for Iterator, a pointer to 'int'
typedef int* Iterator;

// implementing a find algorithm. my_find is a global function
Iterator my_find(Iterator begin, Iterator end, const int & Value) {
    while (begin != end && // pointer comparison
           *begin != Value) // object comparison
        begin++; // next position
    return begin;
}

void main () {
    const int max = 100;
    int aContainer[max]; // the container object.
                        // it contains max elements of type 'int'
    Iterator begin = aContainer; // point to the begin
    Iterator end = aContainer + max; // point to the position after the last element

    // fill the container with some data
    for (int i=0; i < max ; i++)
        aContainer[i] = 2*i;

    int Number=0;
    while (1) { // for ever
        cout << "Enter required number (-1 end):";
        cin >> Number;
        if (Number == -1) break;
        Iterator position = my_find(begin,end,Number);
```

```

        if (position != end)
            cout << "Found at position " << (position - begin) << endl;
        else
            cout << Number << " not found" << endl;
    }
}

```

Um exemplo do uso do programa é mostrado abaixo.

```

Enter required number (-1 end):12
Found at position 6
Enter required number (-1 end):44
Found at position 22
Enter required number (-1 end):33
33 not found
Enter required number (-1 end):-1

```

Uma variação interessante desse programa seria re-escrever a função `my_find` usando `template`. Essa variação pode ser feita sem que o resto do programa precise ser alterado.

```

// implementing a find algorithm. my_find is a global function
template <class iteratorType, class valueType>
iteratorType my_find(iteratorType begin, iteratorType end, const valueType & Value) {
    while (begin != end && // pointer comparison
           *begin != Value) // object comparison
        begin++; // next position
    return begin;
}

```

No exemplo abaixo, estamos de fato usando o STL. O header ‘`algorithm`’ é necessário para incluir a definição da função global com `template` ‘`find`’, e o header ‘`vector`’ é necessário para a definição da classe genérica ‘`vector`’. Para isolar o mais possível o problema do tipo que se está usando nos dados, criou-se uma macro `TYPE`, que significa ‘`int`’. O tipo ‘`Iterator`’ poderia continuar sendo um ‘`TYPE*`’, mas é mais elegante usar a definição pelo STL, como mostrado no programa.

Outra coisa boa: o objeto contenedor agora é alocado (e liberado) automaticamente. Portanto, o número de elementos ‘`max`’ não precisa mais ser um `const`. Pode ser um `int` normal (isto é, pode ser definido em tempo de execução; antes não podia).

O objeto `aContainer` agora é do tipo `vector`, quando a essa classe genérica (forma do bolo) é aplicado o tipo (sabor do bolo) ‘`TYPE`’, isto é ‘`int`’. Ou seja, `aContainer` é um vetor de `int`. Esse vetor é carregado com dados da mesma forma que antes, e a forma de usar o `find` é muito parecido com o `my_find` anterior.

```

#include <iostream>
#include <algorithm> // STL, find
#include <vector> // STL, vector
using namespace std;
#define TYPE int
// creating the new type for Iterator, a pointer to 'int'
// typedef TYPE* Iterator; // directly define the iterator type
typedef vector<TYPE>::iterator Iterator; // define iterator type through STL

void main () {
    int max = 100;
    vector<TYPE> aContainer(max); // the container object.

    // fill the container with some data
    for (int i=0; i < max ; i++)
        aContainer[i] = 2*i;

    int Number=0;
    while (1) { // for ever

```

```

        cout << "Enter required number (-1 end):";
        cin >> Number;
        if (Number == -1) break;
        Iterator position = find(aContainer.begin(), aContainer.end(), Number);
        if (position != aContainer.end())
            cout << "Found at position " << (position - aContainer.begin()) <<
endl;
        else
            cout << Number << " not found" << endl;
    }
}

```

13.2.1 Classes e funções auxiliares

Essa seção descreve algumas ferramentas de STL que serão necessárias para compreensão posterior.

13.2.1.1 Par (pair)

Um par no sentido de STL é o encapsulamento de dois objetos que podem ser de tipos diferentes. Os pares são componentes fundamentais, e que serão usados mais tarde nos exemplos de STL. São definidos como uma `struct` pública no header `<utility>`:

```

template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair(){}; // empty default constructor
    pair(const T1 & a, const T2 & b) // call constructor of parent class
        : first(a), second(b) {};
};

```

O segundo construtor padrão faz os elementos a serem inicializados pelos construtores de seu tipo.

Para a classe `pair`, os `operator==` e `operator<` são os comparadores globais definidos no STL.

```

template <class T1, class T2>
inline bool operator==(const pair<T1, T2> & x, const pair<T1, T2> & y) {
    return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<(const pair<T1, T2> & x, const pair<T1, T2> & y) {
    return x.first < y.first ||
(!x.first < y.first) && x.second < y.second;
}

```

Em `operator<`, quando o primeiro objeto é igual, o retorno é determinado pela comparação do segundo objeto do par. Contudo, de forma a fazer o mínimo de exigências possível para um objeto poder fazer parte de um par, `operator==` não é usado em `operator<`.

Para facilitar a criação de pares há a função global abaixo.

```

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1 & x, const T2 & y) {
    return pair<T1, T2>(x, y);
}

```

13.2.2 Operadores de comparação

O `operator!=` é definido tendo como base o `operator==`.

```

template <class T>
inline bool operator != (const T & x, const T & y) {
    return !(x==y);
}

```

O `operator>` é definido tendo como base o `operator<`.

```
template <class T>
inline bool operator< (const T & x, const T & y) {
return y < x;
}
```

O `operator<=` é definido tendo como base o `operator<`.

```
template <class T>
inline bool operator<= (const T & x, const T & y) {
return !(y < x);
}
```

O `operator>=` é definido tendo como base o `operator<`.

```
template <class T>
inline bool operator>= (const T & x, const T & y) {
return !(x < y);
}
```

Em tese o `operator==` poderia ser definido a partir exclusivamente de `operator<`, conforme abaixo. Portanto, o termo “igualdade” não deveria ser usado, e deveria ser substituído pelo termo “equivalência”. Mas há resistências ao uso do `operator==` dessa forma.

```
// not part of STL
template <class T>
inline bool operator== (const T & x, const T & y) {
return !(x < y) && !(y < x);
}
```

13.2.3 Classes de comparação

STL possui várias classes de comparação, que podem ser usadas para várias finalidades. A tabela abaixo mostra as classes de comparação definidas no header `<functional>`.

Definição do Objeto	Chamada	Retorno
<code>equal_to<T> X;</code>	<code>X(x,y)</code>	<code>x == y</code>
<code>not_equal_to<T> X;</code>	<code>X(x,y)</code>	<code>x != y</code>
<code>greater<T> X;</code>	<code>X(x,y)</code>	<code>x > y</code>
<code>less<T> X;</code>	<code>X(x,y)</code>	<code>x < y</code>
<code>greater_equal<T> X;</code>	<code>X(x,y)</code>	<code>x >= y</code>
<code>less_equal<T> X;</code>	<code>X(x,y)</code>	<code>x <= y</code>

Classes de comparação do STL

A idéia é que essas classes forneçam ponteiros para funções de comparação, de forma que seja fácil (entenda-se: com interface uniforme) escrever uma função que implementa um algoritmo qualquer (que usa comparação). Assim, pode-se escrever um algoritmo que recebe como um dos parâmetros, um ponteiro para a função de comparação. Isso significa que pode-se usar o mesmo algoritmo com a função de comparação que se desejar.

Com a finalidade de padronizar os nomes nas em todas as funções de comparação, STL definiu uma função de comparação base, da qual as demais herdam. Essa função é `binary_function`, mostrada abaixo. Para classes unárias, uma classe correspondente chamada `unary_function` também é definida. O leitor interessado poderá vê-la examinando o header `<functional>`.

```
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

A implementação da classe de comparação `equal_to` é exemplificada abaixo. A palavra reservada “const” colocada antes de “{” na terceira linha significa que o ponteiro para função que corresponde

ao operator() somente poderá ser passado como argumento para uma função que tenha “const” nesse argumento.

```
template <class T>
struct equal_to : binary_function<T, T, bool> {
    bool operator()(const T & x, const T & y) const {
        return x==y;
    }
};
```

No programa abaixo, implementa-se o famoso algoritmo de ordenação “bolha” (bubble_sort) usando template e com baseado numa função de comparação passada ao algoritmo como ponteiro para função.

A forma com que se escreveu o algoritmo bubble_sort é considerado elegante, pois pode-se usar a mesma função para ordenar um array (qualquer) com a função de comparação que se queira. Tanto pode-se usar as funções de comparação derivadas das classes de comparação do STL, quanto pode-se criar funções de comparação do usuário nos mesmos moldes do STL e usar o mesmo algoritmo de ordenação.

```
#include <iostream>
#include <functional> // less<T>
#include <cstdlib> // abs
using namespace std;

template <class T, class CompareType>
void bubble_sort(T* array, int n, const CompareType & compare) {
    for (int i=0 ; i < n ; i++)
        for (int j=i+1 ; j < n ; j++)
            if (compare(array[i],array[j])) {
                // swap (array[i], array[j])
                T temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
}

// my comparison function
template <class T>
struct absolute_less : binary_function<T,T,bool> {
    bool operator()(const T & x, const T & y) const {
        return abs(x) < abs(y);
    }
};

// a global function to help display the array
void display(int *array, int n) {
    for (int i=0 ; i < n ; i++) {
        cout.width(7);
        cout << array[i];
    }
    cout << endl;
}

void main () {
    int table1[] = {-3, -41, -5, -7, 9, -9, 11, -13, 17};
    const int num = sizeof(table1)/sizeof(int);

    cout << "unsorted" << endl;
    display(table1,num);

    // variation 1: create explicitly a compare function and pass it to
    // the bubble_sort function as a parameter
    less<int> lessCompareFunction;
```

```

bubble_sort(table1,num,lessCompareFunction);

cout << "sorted with lessCompareFunction" << endl;
display(table1,num);

// variation 2: don't create explicitly a compare function and
// call the bubble_sort function directly
bubble_sort(table1,num,less<int>());
// the same as bubble_sort(table1,num,less<int>.operator());

cout << "sorted with less<int>()" << endl;
display(table1,num);

// variation 3: call user comparison function, if wanted
bubble_sort(table1,num,absolute_less<int>());

cout << "sorted with absolute_less<int>()" << endl;
display(table1,num);
}

```

Saída:

```

unsorted
-3    -41    -5    -7    9    -9    11    -13    17
sorted with lessCompareFunction
17    11    9    -3    -5    -7    -9    -13    -41
sorted with less<int>()
17    11    9    -3    -5    -7    -9    -13    -41
sorted with absolute_less<int>()
-41    17    -13    11    9    -9    -7    -5    -3

```

13.2.4 Classes aritméticas e lógicas

De forma muito semelhante às classes de comparação, STL também possui classes para funções aritméticas e lógicas. A tabela abaixo ilustra-as.

Definição do Objeto	Chamada	Retorno
<code>plus<T> X;</code>	<code>X(x,y)</code>	<code>x + y</code>
<code>minus<T> X;</code>	<code>X(x,y)</code>	<code>x - y</code>
<code>multiplies<T> X;</code>	<code>X(x,y)</code>	<code>x * y</code>
<code>divides<T> X;</code>	<code>X(x,y)</code>	<code>x / y</code>
<code>modulus<T> X;</code>	<code>X(x,y)</code>	<code>x % y</code>
<code>negate<T> X;</code>	<code>X(x)</code>	<code>-x</code>
<code>logical_and<T> X;</code>	<code>X(x,y)</code>	<code>x && y</code>
<code>logical_or<T> X;</code>	<code>X(x,y)</code>	<code>x y</code>
<code>logical_not<T> X;</code>	<code>X(x)</code>	<code>!x</code>

Classes de aritméticas e lógicas do STL

13.2.5 Complexidade de um algoritmo

É importante que se tenha uma noção de como medir a eficiência de um algoritmo (por exemplo um algoritmo de busca). Para isso uma das notações mais usadas é a notação O , que dá a ordem de grandeza de como cresce o tempo máximo de processamento do algoritmo em relação ao número de elementos contidos. As classificações típicas que se pode definir com a notação O são: $O(1)$, $O(n)$, $O(n^2)$, $O(\log n)$, $O(n \log n)$, etc.

Definição da ordem de complexidade O :

Seja $f(n)$ a função que retorna o tempo de execução de um algoritmo que contém n elementos. Esse algoritmo possui complexidade $O(g(n))$ se e

somente se existem as constantes positivas c e n_0 tais que $|f(n)| \leq c|g(n)|$ é verdadeiro para qualquer $n \geq n_0$.

Exemplos:

Algoritmo	Frequência	complexidade (de tempo)
$x = x + y$	1	constante
for i=1 to n do $x = x + y$ end do	n	linear
for i=1 to n do for j=1 to n $x = x + y$ end do end do	n^2	quadrático
n = (inteiro) k = 0 while n > 0 do $n = n / 2$ $x = x + y$ end do	$\log n$	logaritmico

13.2.5.1 Algumas regras

Regra	Exemplo
$O(\text{const} * f) = O(f)$	$O(2n) = O(n)$
$O(f * g) = O(f) * O(g)$	$O((22n) * n) = O(22n) * O(n) = O(n) * O(n) = O(n^2)$
$O(f / g) = O(f) / O(g)$	$O((4n^3) / n) = O(4n^2) = O(n^2)$
$O(f + g) = \text{dominante}(O(f), O(g))$	$O(n^5 + n^3) = O(n^5)$

Exemplos:

1. Algoritmo tipo “busca linear”

Seja um problema como esse. Há n fichas telefônicas desordenadas. O trabalho é encontrar uma ficha que de um nome de entrada. A busca é linear. Como a localização no nome é aleatória nas fichas, na média pode-se dizer o tempo de resposta é o tempo de procurar $n/2$ fichas. Portanto a ordem de complexidade é $O(n/2) = O(n)$.

2. Algoritmo tipo “busca binária”

Vejamos agora como fica o algoritmo anterior no caso de fichas ordenadas. A partir de um nome de entrada, procura-se no meio das fichas. A cada inferência, ou encontra-se a ficha, ou reduz-se o espaço de procura pela metade. Não é difícil provar que nesse caso a ordem de complexidade é $O(\log n)$.

13.3 Iterador (iterator)

Iteradores são usados por algoritmos para referenciar objetos dentro de contenedores. O iterador mais simples são ponteiros. Essa seção descreve outros tipos de iteradores e suas propriedades em geral. Os

iteradores funcionam em parceria muito próxima com contenedores, que são explicados em maior detalhe na próxima seção.

As propriedades essenciais dos iteradores, como mostrado na seção 13.2 são avançar (++), referenciar (*) e comparação (!= e ==). Se o iterador não é um ponteiro comum, mas um objeto de uma classe de iterador, essas propriedades são implementadas por meio de funções de operação.

13.3.1 Iteradores padrão para inserção em contenedores

O programa abaixo ilustra o funcionamento de `back_insert_iterator` e `front_insert_iterator`, que são usados para criar iteradores para o objeto contenedor `myListObject`, que por sua vez foi criado a partir da classe genérica STL `list`, aplicando-se a ela o tipo `double`.

Para facilitar a exibição do objeto contenedor, foi criada a função global genérica (com template) chamada `VBShowContainer`.

O programa cria uma lista vazia inicial, com 2 zeros. São criados iteradores para inserção no fim (`myBackInsertIterator`) e no início (`myFrontInsertIterator`). Em seguida, o programa insere elementos no início da lista, e posteriormente insere elementos no final da lista. Em todos os casos, o conteúdo da lista é mostrado, para que se acompanhe o que está acontecendo com a lista. Preste atenção na ordem com que os elementos são inseridos.

```
#include <iostream>
#include <list> // STL, list
#include <iterator> // STL, back_insert_iterator
using namespace std;
#define TYPE double
typedef list<TYPE> myListType;

template <class containerType>
void VBShowContainer(const containerType & containerObj) {
    typename containerType::const_iterator iteratorObj;
    if (containerObj.empty()) {
        cout << "====Container empty" << endl;
        return;
    }
    cout << "====show the container contents, size="
        << containerObj.size() << endl;
    for (iteratorObj = containerObj.begin() ;
        iteratorObj != containerObj.end() ;
        iteratorObj++)
        cout << *iteratorObj << endl;
};

void main () {
    int i;

    myListType myListObject(2); // 2 zeros

    // create user defined iterators
    back_insert_iterator<myListType> myBackInsertIterator(myListObject);
    front_insert_iterator<myListType> myFrontInsertIterator(myListObject);

    VBShowContainer(myListObject);

    // insertion by means of operations *, ++, =
    for (i = 1 ; i < 3 ; i++)
        *myBackInsertIterator++ = i;

    VBShowContainer(myListObject);
}
```

```

        // insertion by means of operations *, ++, =
        for (i = 10 ; i < 13 ; i++)
            *myFrontInsertIterator++ = i;

        VBShowContainer(myListObject);
    }

```

Saída do programa.

```

=====show the container contents, size=2
0
0
=====show the container contents, size=4
0
0
1
2
=====show the container contents, size=7
12
11
10
0
0
1
2

```

Uma forma alternativa de se criar um iterador tipo “back” e tipo “front” é usar o `insert_iterator`. Por exemplo: as linhas abaixo poderiam ter sido substituídas

```

back_insert_iterator<myListType> myBackInsertIterator(myListObject);
front_insert_iterator<myListType> myFrontInsertIterator(myListObject);

```

por essas linhas

```

insert_iterator<myListType> myBackInsertIterator(myListObject,myListObject.end());
insert_iterator<myListType> myFrontInsertIterator(myListObject,myListObject.begin());

```

13.4 Contenedor (container)

Em resumo pode-se dizer que um contenedor é um objeto q ue contém outros objetos. Os contenedores mais importantes do STL estão mostrados abaixo.

13.4.1 Vector

O vetor é um contenedor que pode ser referenciado diretamente com `operator[]`. Além disso, pode-se acrescentar dados ao vetor com `back_insert_iterator` (mas não com `front_insert_iterator`). Como todo contenedor, possui métodos `size`, `begin`, e `end`.

```

#include <iostream>
#include <vector> // STL, vector
#include <iterator> // STL, back_insert_iterator

// the same VBShowContainer global function as above

using namespace std;

#define TYPE double
typedef vector<TYPE> myVectorType;

void main () {
    myVectorType myVectorObject(3);    // the container object. 5 zeros
    myVectorType::iterator myVectorIterator;

    VBShowContainer(myVectorObject);

    // create user defined iterators

```

```

    back_insert_iterator<myVectorType> myBackInsertIterator(myVectorObject);

    // insertion by means of operations *, ++, =
    for (int i = 10 ; i < 13 ; i++)
        *myBackInsertIterator++ = i;

    VBShowContainer(myVectorObject);

    // referencing directly the vector with operator[]
    cout << "----- Referencing directly" << endl;
    myVectorObject[1] = 55; // load some data directly
    int size = myVectorObject.size();
    for (i = 0 ; i < size ; i++)
        cout << myVectorObject[i] << endl;
}

```

Saída do programa.

```

=====show the container contents, size=3
0
0
0
=====show the container contents, size=6
0
0
0
10
11
12
----- Referencing directly
0
55
0
10
11
12

```

13.4.2 List

No exemplo abaixo, os métodos `push_front`, `push_back`, `assign`, `empty` e `erase` são mostrados.

Um outro exemplo de uso de `list` foi mostrado na seção 13.2.5.

```

#include <list>
#include <iostream>
// the same VBShowContainer global function as above
using namespace std ;
typedef list<int> myListType;
void main() {
    myListType listOne;
    myListType listAnother;

    // Add some data to list one
    listOne.push_front (2);
    listOne.push_front (1);
    listOne.push_back (3);
    VBShowContainer(listOne); // see list one

    // Add some data to list another
    listAnother.push_front(4);
    listAnother.assign(listOne.begin(), listOne.end());

    VBShowContainer(listAnother); // see list another
    listAnother.assign(4, 1);
    VBShowContainer(listAnother); // see list another
    listAnother.erase(listAnother.begin());
    VBShowContainer(listAnother); // see list another
}

```

```
listAnother.erase(listAnother.begin(), listAnother.end());
VBShowContainer(listAnother); // see list another
}
```

Saída do programa:

```
=====show the container contents, size=3
1
2
3
=====show the container contents, size=3
1
2
3
=====show the container contents, size=4
1
1
1
1
=====show the container contents, size=3
1
1
1
=====Container empty
```

13.4.3 Pilha (Stack)

Uma pilha (stack) é um elemento contenedor em que “o primeiro que entra, é o último que sai”. Para colocar dados numa pilha, usa-se o método `push`. Somente pode-se acessar o último objeto que entrou na pilha. O acesso a esse objeto é feito pelo método `top` (tanto para ler quanto para escrever). Para jogar fora quem está no topo e pegar o que está em baixo, há o método `pop`. Há ainda o método `empty`, que verifica se a pilha está vazia ou não.

```
#include <stack>
#include <iostream>
using namespace std ;
#define TYPE float
typedef stack<TYPE> myStackType;
void main() {
    myStackType stackObject;
    cout << "stackObject is " <<
        (stackObject.empty() ? "empty": "full") << endl;

    stackObject.push(2); // push some data
    stackObject.push(4); // push some data
    stackObject.push(6); // push some data
    stackObject.push(8); // push some data

    cout << "stackObject is " <<
        (stackObject.empty() ? "empty": "full") << endl;

    // Modify the top item.
    if (!stackObject.empty()) {
        stackObject.top()=22;
    }

    // Repeat until stack is empty
    while (!stackObject.empty()) {
        const TYPE & t=stackObject.top();
        cout << "stack object = " << t << endl;
        stackObject.pop();
    }
}
```

Saída do programa

```

stackObject is empty
stackObject is full
stack object = 22
stack object = 6
stack object = 4
stack object = 2

```

13.4.4 Fila (Queue)

Uma fila (queue) permite que se insira dados por uma ponta e retire esse dado pela outra ponta.

A interface da fila contém os métodos abaixo:

- `bool empty();` retorna true se a fila está vazia.
- `size_type size() const;` retorna o tamanho da fila.
- `value_type & front();` retorna o valor no início da fila.
- `const value_type & front() const;` retorna o valor no início da fila.
- `value_type & back();` retorna o valor no fim da fila.
- `const value_type & back() const;` retorna o valor no fim da fila.
- `void push(const value_type & x);` acrescenta o objeto x a fila.
- `void pop();` apaga o primeiro objeto da fila.

```

#include <iostream>
#include <queue>
using namespace std ;
#define TYPE int
typedef queue<TYPE> queueIntType;
typedef queue<char*> queueCharType;
void main()
{
    int size;
    queueIntType myIntQueueObject;
    queueCharType myCharQueueObject;

    // Insert items in the queue(uses list)
    myIntQueueObject.push(1);
    myIntQueueObject.push(2);
    myIntQueueObject.push(3);
    myIntQueueObject.push(4);

    // Output the size of queue
    size = myIntQueueObject.size();
    cout << "size of int queue is:" << size << endl;

    // Output items in queue using front(), and delete using pop()
    while (!myIntQueueObject.empty()) {
        cout << myIntQueueObject.front() << endl; // read object from queue
        myIntQueueObject.pop(); // delete object from queue
    }

    // Insert items in the char queue
    myCharQueueObject.push("Maria");
    myCharQueueObject.push("Alan");
    myCharQueueObject.push("Sergio");
    myCharQueueObject.push("Marcia");
    myCharQueueObject.push("Paulo");

    // Output the item inserted last using back()

```

```

    cout << "The last element of char* queue is " <<
        myCharQueueObject.back() << endl;

    // Output the size of queue
    size = myCharQueueObject.size();
    cout << "size of char* queue is:" << size << endl;

    // Output items in queue using front(), and delete using pop()
    while (!myCharQueueObject.empty()) {
        cout << myCharQueueObject.front() << endl; // read object from queue
        myCharQueueObject.pop(); // delete object from queue
    }
}

```

Saída do programa:

```

size of int queue is:4
1
2
3
4
The last element of char* queue is Paulo
size of char* queue is:5
Maria
Alan
Sergio
Marcia
Paulo

```

13.4.5 Fila com prioridade (priority queue)

A fila com prioridade é diferente da fila comum por permitir atribuir prioridades aos objetos da fila. Quando alguém é retirado da fila, é respeitada a ordem de prioridades estabelecida.

No programa abaixo, foi criada uma classe `myClass`, que armazena o nome e idade de uma pessoa. Para essa classe, definiu-se o `operator<` e `operator>` baseados na observação dos campos de idade apenas (esse será o critério de prioridade). Definiu-se também nessa classe o operador `insersor` (`operator<<`) para permitir fácil exibição no console.

Em seguida, criou-se um `queueObject` a partir do aparentemente estranho tipo ressaltado na linha abaixo.

```
priority_queue<myClass , vector<myClass> , greater<myClass> >
```

Trata-se de uma classe definida em template, sendo que o primeiro tipo é a classe a ser armazenada, o segundo tipo é uma classe contenedora e o terceiro tipo é uma classe de comparação STL (veja tabela na página 165). No caso, escolheu-se a classe `greater<T>` para comparação, que é baseada no `operator>` da classe em questão.

```

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

class myClass {
public:
    myClass(char *name, int age) { m_name = name; m_age = age; }
    char *m_name;
    int m_age;
    friend bool operator>(myClass x, myClass y) { return !(x.m_age > y.m_age); }
    friend bool operator<(myClass x, myClass y) { return !operator>(x,y); }
    friend ostream & operator<< (ostream & stream , const myClass & obj ) {
        stream << "Name:" << obj.m_name
            << "\tAge:" << obj.m_age;
        return stream;
    }
}

```

```
};

void main () {
    priority_queue<myClass , vector<myClass> , greater<myClass> > queueObject;
    queueObject.push(myClass("Sergio", 36));
    queueObject.push(myClass("Alan", 26));
    queueObject.push(myClass("Marcia", 22));
    queueObject.push(myClass("Paulo", 28));
    queueObject.push(myClass("Ana", 21));

    // Output the size of queue
    int size = queueObject.size();
    cout << "size of int queue is:" << size << endl;

    // Output items in queue using front(), and delete using pop()
    while (!queueObject.empty()) {
        cout << queueObject.top() << endl; // read object from queue
        queueObject.pop(); // delete object from queue
    }
}
```

A saída do programa é mostrada abaixo. Repare que os objetos entraram desordenados na fila, e saíram ordenados de acordo com a prioridade escolhida (no caso sai primeiro quem tem idade menor).

```
Name:Ana      Age:21
Name:Marcia   Age:22
Name:Alan     Age:26
Name:Paulo    Age:28
Name:Sergio   Age:36
```

Caso se deseje mudar a prioridade para sair primeiro que tem idade *maior*, basta mudar a classe de comparação de `greater<T>` para `less<T>`. Se isso for feito, a saída do programa fica como mostrado abaixo.

```
size of int queue is:5
Name:Sergio   Age:36
Name:Paulo    Age:28
Name:Alan     Age:26
Name:Marcia   Age:22
Name:Ana      Age:21
```

13.4.6 Contenedores associativos ordenados

São contenedores que permitem acesso rápido aos dados através de chaves que não necessariamente coincidem com os dados propriamente ditos, isto é, podem ser chaves estrangeiras.

O STL armazena as chaves de forma ordenada, apesar de isso não ser requisito para o funcionamento das tarefas. Essa característica é apenas um detalhe de implementação, que balanceia o armazenamento dos objetos (chaves). Devido ao ordenamento, o acesso aos elementos é muito rápida. O contenedor das chaves é geralmente uma árvore (tree), que fica balanceada devido ao ordenamento. Uma alternativa para tornar o acesso ainda mais rápido é usar um hash¹⁶ para armazenar as chaves. Caso o hash seja usado, o uso de memória é maior, mas a ordem de acesso é $O(1)$, ao invés de $O(\log N)$ que ocorre no caso de se usar árvore.

Em set e multiset, os dados eles mesmo são usados como chaves. Em map e multimap, as chaves e os dados são diferentes. Os 4 tipos de contenedores associativos ordenados estão mostrados abaixo.

¹⁶ hash significa “carne picada”, ou “picar”. Não creio que sejam termos razoáveis para serem usar em português, portanto o termo usado será em inglês mesmo.

13.4.6.1 Set

No set, as chaves (key) coincidem com os dados. Não pode haver elementos com a mesma chave.

No programa abaixo, um setObject é definido, e alguns dados são acrescentados. A função global myCheck testa se o objeto existe. Para isso, é chamado o método `find`, que retorna o iterador apontando para o objeto que foi encontrado, ou apontando para o fim caso o objeto não tenha sido encontrado. A variável booleana `exist` armazena a informação de existência ou não do objeto procurado. A função global myShowExist retorna `char*` informando “existe” ou “não existe” de acordo com o argumento recebido.

Repare que o mesmo dado (12) é inserido duas vezes. Na segunda vez que o dado é inserido, ele é ignorado.

Esse programa gera alguns warnings quando programado em Visual C++. Esses warnings podem ser ignorados. Em g++ não é gerado qualquer warning.

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

// the same VBShowContainer global function as above

#define TYPE int
typedef set<TYPE> setType;

char *myShowExist(bool b) {
    if (b)
        return " exists";
    else
        return " does not exist";
}

// argument can not be "const setType & setObject"
void myCheck(setType & setObject, const TYPE & k) {
    setType::iterator it; // iterator
    it = setObject.find(k);
    bool exist = it!=setObject.end();
    cout << "Item " << k << myShowExist(exist) << endl;
}

void main() {
    setType setObject; // container object
    setType::iterator it; // iterator

    // fill with some data
    setObject.insert(5);
    setObject.insert(8);
    setObject.insert(12);
    setObject.insert(12); // insert same data twice. Ignored
    setObject.insert(7);
    VBShowContainer(setObject);

    // erasing an object
    int i=7;
    it = setObject.find(i);
    setObject.erase(it);

    // check if data exists in the container object
    myCheck(setObject,8);
    myCheck(setObject,6);
}
```

Saída do programa.

```

=====show the container contents, size=3
5
8
12
Item 8 exists
Item 6 does not exist

```

Um outro exemplo interessante usando set é mostrado abaixo. Repare que um array de dados é passado para o construtor do setObject, e com isso, todos os dados entram no objeto. Como deveria ser, os dados repetidos não entram.

```

#include <set>
#include <iostream>
// the same VBShowContainer global function as above
#define TYPE double
void main () {
    TYPE array[] = { 1.0, 2.0, 2.0, 5.0, 6.5, 6.8, 7.0, 1.0, 2.2 };
    int count = sizeof(array) / sizeof(TYPE);
    set<TYPE> setObject(array, array + count);
    VBShowContainer(setObject);
}

```

Saída do programa.

```

=====show the container contents, size=7
1
2
2.2
5
6.5
6.8
7

```

Exercício:

Faça um programa que leia um arquivo de texto, em que cada linha contém um email. Mas é sabido que há emails repetidos na lista. O programa deve abrir esse arquivo como leitura e salvar outro arquivo com o mesmo conteúdo, apenas eliminando os emails repetidos.

13.4.6.2 Multiset

O multiset é parecido com o set, exceto que aqui é permitido dados com mesma chave. O programa abaixo é uma repetição do último programa com set, apenas mudando para multiset. A diferença é que os elementos repetidos são permitidos.

```

#include <set>
#include <iostream>
// the same VBShowContainer global function as above
#define TYPE double
void main () {
    TYPE array[] = { 1.0, 2.0, 2.0, 5.0, 6.5, 6.8, 7.0, 1.0, 2.2 };
    int count = sizeof(array) / sizeof(TYPE);
    multiset<TYPE> setObject(array, array + count);
    VBShowContainer(setObject);
}

```

Saída do programa.

```

=====show the container contents, size=9
1
1
2
2
2.2
5
6.5
6.8

```

7

13.4.6.3 Map

O map é parecido com o set, exceto que os dados e a chave são distintos. No map, não é permitido dados com mesma chave. O map é um contenedor de um par (veja 13.2.1.1) de elementos.

No programa de exemplo abaixo, um objeto map contém os elementos (e os mapeamentos) de long e string. É criado um tipo para armazenar o mapa e um tipo que armazena valor, no caso `valuePairType`. O programa cria o objeto map, carrega-o com dados pelo código fonte. Para exemplificar, um dos dados tem a mesma chave, portanto não será efetivamente incluído. Em seguida, um laço mostra o conteúdo total do map.

A próxima parte do programa pede ao usuário para entrar uma chave pelo console (no caso um long). Com essa chave, busca-se no map o elemento (par) que corresponde com o método `find`. O algoritmo de busca é feito com ordem $O(\log N)$. Depois que o iterador é encontrado, o acesso ao elemento é feito com $O(1)$. Outra possibilidade é usar o operador[], que retorna o elemento `second` do par com a mesma ordem $O(\log N)$.

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

typedef map<long, string> mapType;
typedef mapType::value_type valuePairType;

void main() {
    mapType mapObject;
    mapType::iterator it;    // iterator

    // add some data.
    mapObject.insert(valuePairType(1234567,"Sergio"));
    mapObject.insert(valuePairType(232134,"Marcos"));
    mapObject.insert(valuePairType(3423423,"Ana"));
    mapObject.insert(valuePairType(938423,"Marcia"));
    mapObject.insert(valuePairType(23434534,"Mateus"));
    mapObject.insert(valuePairType(746545,"Camila"));

    // Attention: Sergio and Gabriel have the same key !
    // So, Gabriel is not input !
    mapObject.insert(valuePairType(1234567,"Gabriel"));

    int size = mapObject.size();
    cout << "==== map size = " << size << endl;
    // loop to show the map
    it = mapObject.begin();
    while (it != mapObject.end()) {
        cout << it->first << " \t: "    // number
              << it->second           // name
              << endl;
        it++;
    }

    cout << "Enter a number:";
    long number;
    cin >> number;

    it = mapObject.find(number);    // O(logN)

    if (it != mapObject.end()) {
        cout << "Number found" << endl;
        cout << "Correspondent name is: "
```

```

                << it -> second << endl // O(1)
                << "again: " << mapObject[number] << endl; // O(logN)
            }
            else
                cout << "Number not found" << endl;
        }

```

Saída do programa.

```

===== map size = 6
232134      : Marcos
746545      : Camila
938423      : Marcia
1234567     : Sergio
3423423     : Ana
23434534    : Mateus
Enter a number:23434534
Number found
Correspondent name is: Mateus
again: Mateus

```

13.4.6.4 Multimap

O multimap é parecido com o map, exceto que é permitido dados com mesma chave.

13.5 Algoritmos

13.5.1 remove_if

O algoritmo `remove_if()` é parecido com o algoritmo `remove`. Os dois primeiros argumentos marcam o início e o final da sequência. O terceiro argumento é o predicado. Diferentemente de `remove()`, o algoritmo `remove_if()` usa o predicado dado para selecionar os elementos que removerá. Com isso, o usuário do algoritmo poderá introduzir uma regra para a remoção, de acordo com a sua própria lógica. O uso do `remove_if()` é similar ao `remove()`, sendo a única diferença a necessidade de se definir o predicado.

No exemplo abaixo, o algoritmo `remove_if()` é usado para remover vogais de um objeto do tipo `string`. O predicado deve identificar (retornar booleano) se uma letra é vogal ou não.

Lembre-se que o `remove_if()` realmente não apaga os elementos da string. Essa função simplesmente move os elementos para o fim e retorna um iterator para a posição. Utilizando-se desse iterator uma nova string, que não contém vogais, é criada.

```

#include <string>
#include <iostream>
#include <functional> // unary_function
#include <algorithm> // for remove_if
using namespace std;

/* prototype of remove_if
template <class ForwardIterator, class T>
ForwardIterator remove_if (ForwardIterator first,
                          ForwardIterator last,
                          Predicate pred);

*/

template <class T>
class is_vowel: public unary_function<T,T>
{
public:
    bool operator()(T t) const {
        if ((t=='a')||(t=='e')||(t=='i')||(t=='o')||(t=='u'))

```

```
        return true; //t is a vowel
    }
    return false; // t is not a vowel
};

void main() {
    string original;
    original = "abcdefghijklmnopqrstuvwxyz";

    // Next, we call remove_if() to remove all the vowels from the input string.
    // We create a temporary object of the specialization is_vowel and pass
    // it as the predicate of remove_if():
    // move vowels to end and store the position where the vowels start
    string::iterator it= remove_if(original.begin(),
                                   original.end(),
                                   is_vowel<char>());

    // create new string from original using previous iterator
    string no_vowels(original.begin(),it);
    cout << no_vowels << endl;
}
```

A saída do programa é como mostrado abaixo.

```
bcd fghjklmnpqrstvwxyz
```

Capítulo 14) Componentes de Programação

Nesse capítulo são mostrados alguns componentes de programação com utilidades diversas. Alguns são para Windows & DOS, outros são para unix (testado em Linux).

14.1 Para Windows & DOS

Programas que usem diretamente o barramento de entrada e saída da CPU, o que equivale a instruções de in e out em assembly, não funcionam em Windows NT (nem em Windows 2000 que é baseado em tecnologia NT). Mas funciona normalmente em DOS em Windows 9x (que é baseado em tecnologia DOS).

14.1.1 Programa para listar o diretório corrente (Visual C++)

```
#include <windows.h>
#include <iostream.h>    // cout

void main () {
    WIN32_FIND_DATA f;
    HANDLE h;
    bool done;

    h = FindFirstFile("*.*", &f);
    done = (h != 0);
    while (!done) {
        cout << f.cFileName << endl;
        done = !FindNextFile(h, &f);
    }
}
```

Resultado

```
.
..
card_1cpp.cpp
teste6.dsp
teste6.plg
cardcpp_2.cpp
```

Um código alternativo, para listar arquivos, eliminando-se o arquivo “.”, “..” e os arquivos que são diretórios.

```
#include <windows.h>
#include <iostream.h>    // cout

void main () {
    WIN32_FIND_DATA f;
    HANDLE h;
    bool done;

    h = FindFirstFile("*.*", &f);
    done = (h != 0);
    while (!done) {
        // while not finished the loop to get files under current directory
        if ((strcmp(f.cFileName, ".") && (strcmp(f.cFileName, ".."))) {
            DWORD att = GetFileAttributes(f.cFileName); // get file attributes
            int is_directory = att & FILE_ATTRIBUTE_DIRECTORY;
            // check if file is a directory
            if (!is_directory) { // only display if it is not directory
```

```

        cout << f.cFileName << endl;
    }
}
done = !FindNextFile(h, &f);
} // end while
}

```

Resultado

```

card_1cpp.cpp
teste6.dsp
teste6.plg
cardcpp_2.cpp

```

14.1.2 Porta Serial

// todo

14.1.3 Porta Paralela

A pinagem da porta paralela pode ser vista na tabela abaixo.

Descrição dos pinos			
Pino	Sinal (* significa lógica inversa)	Direção	Descrição
1	STROBE*	OUT	sinal de Controle
2	DATA 0	OUT	bit de dado D0
3	DATA 1	OUT	bit de dado D1
4	DATA 2	OUT	bit de dado D2
5	DATA 3	OUT	bit de dado D3
6	DATA 4	OUT	bit de dado D4
7	DATA 5	OUT	bit de dado D5
8	DATA 6	OUT	bit de dado D6
9	DATA 7	OUT	bit de dado D7
10	ACKNOWLEDGE*	IN	sinal de controle
11	BUSY	IN	sinal de controle
12	PAPER END	IN	sinal de controle
13	SELECT OUT	IN	sinal de controle
14	AUTO FEED*	OUT	sinal de controle
15	ERROR*	IN	sinal de controle
16	INITIALIZE PRINTER	OUT	sinal de controle
17	SELECT INPUT*	OUT	sinal de controle
18	GROUND	-	Terra (ground)
19	GROUND	-	Terra (ground)
20	GROUND	-	Terra (ground)
21	GROUND	-	Terra (ground)
22	GROUND	-	Terra (ground)
23	GROUND	-	Terra (ground)
24	GROUND	-	Terra (ground)
25	GROUND	-	Terra (ground)

O programa abaixo joga constantes byte na porta paralela usando diretamente a instrução outportb.

```

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

```

“C / C++ e Orientação a Objetos em Ambiente Multiplataforma”, versão 5.1

Esse texto está disponível para download em www.del.ufrj.br/~villas/livro_c++.html

```

#define LPT1_DATA 0x378

// the user out port function
void outportb(unsigned short port, unsigned short val) {
    // calls the outport function of the compiler
    _outp(port,val);    // _outp is VisualC only
}

// the user in port function
int inportb(unsigned short port) {
    return (_inp(port)); // _inp is VisualC only
}

const unsigned char init_val = 128;    // 1000 0000

void main () {
    unsigned char value = init_val;
    int i=0;
    while (1) {
        outportb(LPT1_DATA,value);
        cout << i++ << " ) press any key to for next byte, space to stop" << endl;
        char ch = getch();
        if (ch == ' ') break;
        value = value >> 1;    // bitwise shift
        if (value==0)
            value = init_val;
    }
}

```

14.2 Componentes para unix (inclui Linux)

14.2.1 Entrada cega (útil para entrada de password)

```

// compile with "g++ thisFile.cpp vblib.cpp"
// because this source uses VBString (in vblib.cpp)
// vblib.h also required.
#include <iostream.h> // cout, cin
#include <unistd.h>    // read
#include <termios.h>   // tcgetattr
#include "vblib.h"    // VBString

void main () {
    // set terminal for no echo
    struct termios myTerm; // a structure to store terminal settings
    int fd = 1; // anything
    tcgetattr(fd,&myTerm); // get terminal settings
    tcflag_t lflagOld = myTerm.c_lflag; // save flag
    myTerm.c_lflag &= !ECHO; // set flag for no echo
    int optional_actions = 1; // anything
    tcsetattr(fd,optional_actions,&myTerm); // set terminal

    VBString password;
    cout << "Enter password:";
    cin >> password;

    // restore original terminal
    myTerm.c_lflag = lflagOld;
    tcsetattr(fd,optional_actions,&myTerm);
    cout << endl;
}

```

14.3 Elementos de programação em tempo real

14.3.1 Conceitos de programação em tempo real

Até agora, todos os programas são executados com a máxima velocidade que a CPU e o sistema operacional permitem. Em princípio, “quanto mais rápido melhor”. Contudo, há alguns casos em que é importante que se leve em conta o tempo de execução explicitamente para se atingir algum objetivo. Nesses casos não é verdade que “quanto mais rápido melhor”, mas “executar o procedimento no momento certo”. Esses casos são chamados de programação em tempo real.

Um exemplo simples de um programa em tempo real é um programa que mostra um relógio. Um programa desses deve atualizar o mostrador do relógio a cada segundo (ou a cada minuto), no momento certo. A velocidade da CPU precisa ser rápida suficiente para atualizar o mostrador na velocidade solicitada; qualquer velocidade adicional de CPU é irrelevante.

Outro exemplo é um sistema de monitoração de informações baseado em computador ligado a sensores. Num sistema desses, geralmente há um menu de configuração onde se informa quantas vezes por segundo o sensor deve ser amostrado, e seu resultado armazenado (ou mostrado).

Outro exemplo ainda é o caso de um computador que controla uma planta qualquer (como a posição de um canhão ou a posição de um lançador de torpedos). O computador nesse caso é parte integrante de um laço de controle e precisa amostrar os dados do sensor e comandar o atuador em períodos de amostragem pré-definidos. Nesse caso, o computador claramente opera com um programa em tempo real.

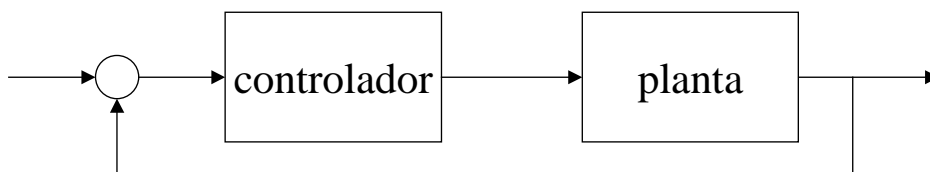


Figura 13: Diagrama do computador como controlador em tempo real

14.3.2 Programa em tempo real com “status loop”

Para um programa operar em tempo real, há uma técnica chamada de “*status loop*”, que em português seria “laço de situação”. Trata-se de um laço de programa que consulta uma base de tempo (que é atualizada por um hardware separado da CPU) para saber se é ou não a hora de executar um procedimento.

O PC é um projeto que contém um “timer tick”, que pode ser consultado. O timer tick é um relógio que interrompe periodicamente a CPU. Isso é muito importante para o sistema DOS, para que se pudesse criar o efeito de multiprocessamento, mas é relativamente pouco importante no Windows, que já possui multiprocessamento internamente.

O DOS inicializa a interrupção do timer tick no boot. Um programa pode capturar

No exemplo abaixo, consulta-se o relógio interno do PC para se saber o instante atual.

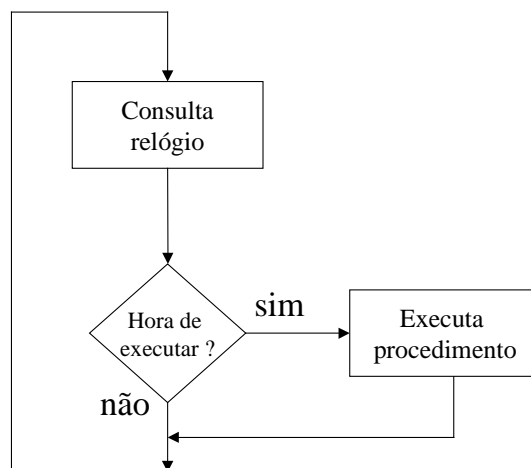


Figura 14: diagrama de um laço para programação em tempo real usando *status loop* (laço de situação)

```

/*=====
NOTE: This is Borland specific source code,
that is, non portable C source code.
=====*/

#include <time.h>
#include <dos.h>
#include <iostream.h>

void t1() {
    clock_t start, end;
    start = clock();

    // um for para perder tempo
    for (int i=0; i<10000 ; i++)
        for (int k=0; k<10000 ; k++)
            {};

    end = clock();
    cout << "O tempo de espera foi de " << (end - start) / CLK_TCK
         << "segundos" << endl;
}

/*
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};

struct time {
    int ti_sec;
    int ti_min;
    int ti_hour;
    int ti_hund;
};
*/

void printLocalTime2() {

```

```

time_t timer;
struct tm *tblock, t;
timer = time(NULL); // gets time of day
tblock = localtime(&timer); // converts date/time to a structure
t = *tblock;
cout << "Hora: " << t.tm_hour << ":" << t.tm_min << ":"
      << t.tm_sec << endl;
}

void printLocalTime() {
    struct time t;
    gettime(&t);
    cout << "Hora: " << (int)t.ti_hour << ":" << (int)t.ti_min << ":"
          << (int)t.ti_sec << ":" << (int)t.ti_hund << endl;
}

void main() {
    float period=1.0; // in seconds
    float dt;
    clock_t c1, c2;
    c1 = clock();
    while (1) { // laço infinito
        c2 = clock(); // pega o relógio atual
        dt = (c2-c1)/CLK_TCK; // calcula o atraso
        if (dt > period) { // se o atraso for maior que o período
            c1=c2; // relógio é atualizado para a próxima interação
            // a seguir, faz-se alguma coisa em tempos controlados
            cout << "<ctrl-c> to stop" << endl;
            printLocalTime();
        } // end if
    } // end while
} // end main

```

14.3.3 Programa em tempo real com interrupção

O projeto do PC inclui um relógio chamado *timer tick*, que interrompe a CPU regularmente, com frequência de aproximadamente 17 vezes por segundo. O sistema operacional possui uma rotina padrão para tratar essa interrupção. Geralmente, as rotinas de interrupção devem ser muito curtas e como o DOS não é reentrante, é proibido que qualquer função do DOS seja chamada dentro da rotina que trata a interrupção do timer tick.

Para se fazer um programa em tempo real no PC baseado na interrupção do timer tick, é preciso que se escreva uma rotina de interrupção no programa e direcionar a interrupção do timer tick para essa rotina. Para não afetar outras funcionalidades do sistema operacional, a rotina de interrupção deve terminar chamando a rotina original de tratamento do timer tick. Quando o programa termina, a rotina original deve ser restaurada.

O diagrama de um laço de programação em tempo real usando a interrupção do timer tick é mostrada abaixo.

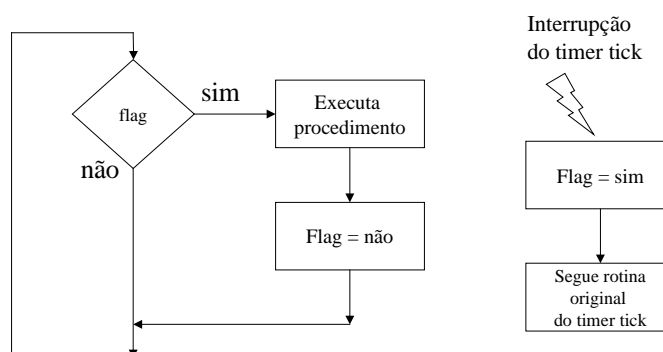


Figura 15: diagrama de um laço para programação em tempo real usando interrupção do timer tick

No programa de exemplo abaixo, a rotina de interrupção conta até 17 antes de setar o flag. Com isso, produz-se um efeito como se a rotina do timer tick tivesse uma frequência 17 vezes menor, o que corresponde a aproximadamente 1 segundo. Além disso, o programa redireciona o tratamento de ctrl-c, para que se garanta o restabelecimento da rotina original de timer tick no caso de o programa terminar com o usuário pressionando ctrl-c.

```

/*=====
NOTE: This is an interrupt service routine.
You can NOT compile this program with
Test Stack Overflow turned on and get an
executable file that will operate correctly.

NOTE2: This is Borland specific source code,
that is, non portable C source code.

NOTE3: if you have problems in compiling
this code, try using the command line compiler
bcc.
=====*/

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <dos.h>
#include <iostream.h>

#define INTR 0x1C    // The timer tick interrupt

#ifdef __cplusplus
    #define __CPPARGS ...
#else
    #define __CPPARGS
#endif

/*=====
CMAX delta_t(seconds)
17      0.99
18      1.04
=====*/
#define CMAX 17

// global variables
void interrupt ( *oldhandler)(__CPPARGS); // pointer to function
long int count=CMAX;
char flag_timeToRun=0;

#define ABORT 0
int c_break() {
    char *str="=====";
    cout << endl << str << endl;
    cout << "Control-Break pressed." << endl
         << "Restoring timer tick handler and Program aborting ...";
    cout << endl << str << endl;

    setvect(INTR, oldhandler); // reset the old interrupt handler
    return (ABORT);
}

// don't call DOS inside interrupt handler function
void interrupt interrupt_handler(__CPPARGS) {
    count++;
    if (count > CMAX) {

```

```

    count = 0;
    flag_timeToRun = 1;
}
oldhandler(); // call the old routine
}

void printLocalTime() {
    struct time t;
    gettime(&t);
    cout << "Hora: " << (int)t.ti_hour << ":" << (int)t.ti_min << ":"
        << (int)t.ti_sec << ":" << (int)t.ti_hund << endl;
}

void main(void){
    oldhandler = getvect(INTR); // save the old interrupt vector
    setvect(INTR, interrupt_handler); // install the new interrupt handler
    ctrlbrk(c_break); // register ctrl-c hadling routine

    flag_timeToRun = 0; // initialize flag
    for (int i=0; i<15000; i++) {
        for (int k=0; k<30000; k++) { // long loop
            if (flag_timeToRun) {
                flag_timeToRun = 0;
                cout << endl << "CTRL-C to abort    ";
                printLocalTime();
            } // end if
        } // end for k
    } // end for i

    setvect(INTR, oldhandler); // reset the old interrupt handler
} // end main

```

Os programas em tempo real que usam interrupção são um pouco mais problemáticos de se depurar e entender que os programas que usam o *status loop*. Um erro no programa com interrupção pode levar o computador a travar completamente, só se recuperando com um reset geral. Além disso, não se pode rodar debug com um programa que usa interrupção.

Em alguns casos é possível atingir velocidades maiores de execução usando interrupção. Se um problema em questão requer que se use a velocidade de processamento no limite máximo da velocidade, o uso de interrupção pode ser uma boa alternativa. Caso contrário, é mais livre de problemas utilizar a técnica de *status loop*.

14.3.4 Programas tipo “watch dog” (cão de guarda)

Um programa de cão de guarda é um programa que fica “rodando em paralelo” com demais programas com a finalidade de checar a integridade de um sistema. Uma aplicação para esse tipo de programa é gerar um alerta ou alarme no caso em que uma condição seja atingida.

Capítulo 15) Boa programação × má programação

15.1 Introdução

Um compilador C++ é uma ferramenta de propósito absolutamente genérico. Serve para fazer projetos muito sérios, e também para experiências de qualquer natureza. A flexibilidade muito grande que o compilador propicia leva ao programador a desenvolver diversos estilos de programação.

Refletindo sobre os objetivos que uma pessoa tem ao fazer um programa, e revendo os conceitos de programadores experientes, chega-se a um conjunto de regras, chamadas de “regras de boa programação”. Essas regras não são impostas pelo compilador, no sentido que se não forem seguidas, ainda assim o compilador consegue compilar. Mas são regras muito recomendáveis, pois a experiência mostra que a falha em seguir essas regras geralmente causa muitos problemas mais tarde.

Os objetivos de quem programa são mais ou menos os seguintes (não estão em ordem de prioridade):

- Desenvolver programas que solucionem a solicitação de clientes. Faze-lo de forma mais rápida e objetiva possível, evitando-se dentro do possível “re-inventar a roda”.
- Facilitar o provável upgrade, isto é, ser capaz de adaptar um programa, no caso de uma mudança de especificação solicitada pelo cliente. Chama-se isso de “manutenção de programa”.
- Aprender mais sobre a tecnologia de programação. Conhecer as tecnologias em vigor e as tendências.
- Facilitar que outra pessoa, que não necessariamente é conhecida, altere o programa. E essa pessoa pode ser você mesmo após algum tempo.
- Considerar a comercialização do fonte de um programa, e não apenas do executável.

Portanto, desenvolver programas é muito mais que apenas fazer um código que atenda a uma especificação. Caso o programador não se concientize disso, seu código fica de “baixa qualidade” (o que não quer dizer que não funcione).

Mas os programas que se faz hoje em dia são muito complexos, e é imperioso “jogar em time”. Definitivamente não é mais possível que uma pessoa sozinha faça um programa razoavelmente grande de forma completa, sem nenhum tipo de ajuda, ainda que indireta. Na prática, os programadores trocam muitas informações uns com os outros. Na era da Internet, essa troca de informações é em grande parte feita por email ou web.

Nesse contexto, é muito importante que o código fonte seja facilmente compreendido pela comunidade de programadores com os quais se pretende trocar informações. Por exemplo: durante o desenvolvimento de um programa, pode acontecer que um trecho de programa simplesmente não funcione. O programador resolve recorrer a uma lista de email para expor o trecho e pedir ajuda. Para fazer isso, a maneira mais fácil é cortar-e-colar o trecho em questão para o email. Mas se o programa está mal programado (no sentido de ser difícil de entender), então será muito mais difícil obter ajuda.

15.2 Itens de boa programação

15.2.1 Identação correta

A identação do programa deve ajudar o programador a entender o nível de laço que se está programando. Quando o programa tiver laços dentro de laços, cada laço deve estar identado em relação ao anterior. O mesmo vale para if. O exemplo abaixo está ilustra o conceito. Repare que a chave que fecha o escopo de for k está exatamente embaixo do for k.

```
void t() {
    const int row = 5;
    const int col = 15;
    int data[row][col];
    for (int i=0 ; i < row ; i++)
        for (int k=0 ; k < col ; k++) {
            data[i][k] = i + 10*k + 6;
            if (k==4) {
                data[i][k] = 200; // this case is special
            }
        } // for k
}
```

Os marcadores de escopo ({ e }) devem seguir um dos dois padrões abaixo.

```
if (k==4)
{
    data[i][k] = 200;
}
```

ou

```
if (k==4) {
    data[i][k] = 200;
}
```

15.2.2 Não tratar strings diretamente, mas por classe de string

Uma fonte de bugs muito comum em C/C++ ocorre no tratamento de strings diretamente, ou seja, criar um buffer de char com um tamanho fixo e usar funções de biblioteca padrão que manipulam esse buffer, tais como strcpy ou strcat. Caso ocorra bug, o comportamento do programa é aleatório, isto é, pode funcionar normalmente, pode funcionar com erro (e.g. copiar a string faltando um pedaço), pode ocorrer mensagem de erro no console, pode travar a máquina, etc.

15.2.2.1 Motivos pelos quais é má programação tratar strings diretamente

Fazer um programa que trata strings diretamente é portanto considerado como má programação. Os motivos explícitos de considera-lo assim estão expostos abaixo.

1. O compilador não tem como distinguir entre um ponteiro para buffer, ponteiro alocado e ponteiro não alocado. O programa abaixo contém um bug e o compilador não sinaliza qualquer warning.

```
// programa com bug !
#include <string.h> // strcpy
void main () {
    char *buffer=0;
    strcpy(buffer,"abc"); // bug. data is being copied to non allocated buffer
}
```

2. Caso seja usado um buffer de char com tamanho fixo, somente não ocorre bug caso a string nunca ultrapasse o valor desse tamanho de buffer. Portanto o programa sempre tem uma certa possibilidade de conter um bug. Isso é particularmente problemático ao se escrever um componente de programação

de propósito genérico. Usando a função `strcpy` ou `strcat`, o compilador não verifica se o buffer tem espaço suficiente para a cópia. Caso o espaço não seja suficiente, ocorre bug.

```
// programa com bug !
#include <string.h> // strcpy
void main () {
    char buffer[10];
    strcpy(buffer,"abcdefghijklmnopqrst"); // bug. data is larger than buffer
}
```

3. Caso uma função retorne `char*`, esse retorno não pode ser feito com referência a uma variável local, a menos que seja “static”.

```
// programa com bug !
#include <iostream.h>
#include <string.h>

char *htmlPath(char *userName) {
    char *path = "/home/usr";
    char *html = "/public_html";
    char buffer[30]; // this line
    strcpy(buffer,path);
    strcat(buffer,userName);
    strcat(buffer,html);
    return buffer; // BUG, returning reference to local variable
}

void main() {
    char *completePath = htmlPath("fred");
    cout << completePath << endl;
}
```

Esse programa pode ser corrigido acrescentando-se a palavra “static” ao buffer de retorno. Dessa forma, o buffer passa a ter escopo global (e não mais escopo local), apesar de ter visibilidade apenas local. Trocando-se a linha marcada com “this line” pela linha abaixo, obtém-se o efeito desejado.

```
static char buffer[30]; // this line
```

Mas mesmo assim persiste o problema do item anterior, isto é, a função apenas não tem bug se em nenhum caso o conteúdo do buffer ultrapassar o tamanho definido. Pode-se minimizar a possibilidade de bug (não anular essa possibilidade) por fazer o tamanho do buffer ser muito grande. Mas isso faz consumir desnecessariamente recursos do computador.

15.2.2.2 Solução recomendada para tratamento de strings: uso de classe de string

Há mais de uma classe de string que se pode usar. Isso ocorre porque a classe de string não é parte integrante da estrutura da linguagem C++, mas escrita em cima da própria linguagem C++. Portanto, cada programador pode usar/escrever sua própria classe de string de acordo com sua conveniência.

Abaixo, estão citados 3 exemplos de classes de string, com suas características

1. `CString` - classe que vem junto com a biblioteca MFC e Visual C++. É uma classe muito boa, mas somente funciona em Windows. Não funciona em unix.
2. `string` - classe padrão de C++, que funciona igualmente em unix e Windows. É bastante boa, mas requer o uso de namespace. Somente compiladores mais modernos dão suporte a namespace (mas isso não é grande problema, porque é muito comum usar sempre as últimas versões dos compiladores). O (pequeno) problema é que nem sempre o programador quer usar namespace, ou sabe fazê-lo.
3. `VBString` - é uma classe feita pelo Villas-Boas, em código aberto. Para usa-la, é preciso incluir no projeto o arquivo `vblib.cpp`, e incluir no fonte `vblib.h`. É uma classe de string que funciona

igualmente em Windows e unix, e tem várias funções extras (em relação a uma classe de string padrão) que podem ser muito úteis.

Usando classe de string, evita-se o uso de constantes com dimensão de buffer de array of char.

```
// programa sem bug, bem programado
#include "vblib.h" // VBString
void main () {
    VBString buffer; // não se indica o tamanho do buffer.
    buffer = "abcdefghijklmnpqrst";
}
```

Concatenação de strings pode ser feita diretamente.

```
// programa sem bug, bem programado
#include "vblib.h" // VBString
void main () {
    VBString buffer; // não se indica o tamanho do buffer.
    buffer = "abcdefghijklmnpqrst";
    buffer += "123";
    cout << buffer << endl; // abcdefghijklmnpqrst123
}
```

Uma função pode retornar um objeto do tipo string sem problemas

```
// programa sem bug, bem programado
#include <iostream.h>
#include "vblib.h"

VBString htmlPath(VBString userName) {
    char *path = "/home/usr";
    char *html = "/public_html";
    VBSrtring buffer;
    buffer = path;
    buffer += userName;
    buffer += html;
    return buffer;
}

void main() {
    VBString completePath = htmlPath("fred");
    cout << completePath << endl;
    // ou
    cout << htmlPath("fred") << endl;
}
```

Uma classe de string possui um buffer interno para armazenar o conteúdo da string. Esse buffer não é acessado diretamente, mas apenas por operadores e outros métodos da interface da classe de string. A classe cuida de alocar o buffer interno na dimensão exata para a medida do que está sendo guardado. Obviamente, a classe também cuida de liberar a memória na destruição dos objetos string, para evitar vazamento de memória. Portanto, um objeto de classe de string pode conter uma string de qualquer tamanho, mesmo que seja patologicamente grande. Essa característica é particularmente útil no tratamento de arquivos de texto, por exemplo. Pode-se ler uma linha de um arquivo de texto diretamente para uma string, despreocupando-se com o tamanho que a linha possa ter.

No programa de exemplo abaixo, um arquivo de texto é lido linha a linha, armazenado numa string e depois enviado ao console. A elegância está no fato de que não há constante de dimensão de array of char, e o programa funciona mesmo para arquivos que possam ter linhas de dimensão patologicamente grande. Como cada linha é lida e posteriormente enviada ao console, pode-se pensar em fazer algum processamento com essa linha, o que dá maior utilidade ao programa.

```
// programa sem bug, bem programado
#include <fstream.h> // ifstream
#include "vblib.h" // VBString
```

```
void main () {
    ifstream myFile("filename.dat");
    VBString str;
    while (!myFile.eof()) {
        myFile >> str;
        // do some processing, if needed
        cout << str << endl;
    }
}
```

15.2.3 Acrescentar comentários elucidativos

Os comentários devem explicar para que serve um trecho de programa, ou uma função. Os comentários *não* devem explicar a linguagem, a menos em situações específicas (como um tutorial).

```
void main () {
    int i;
    // Exemplo de comentário ruim, pois apenas explica a linguagem
    i = i + 1; // soma 1 a variável i
}
```

15.2.4 Evitar uso de constantes relacionadas

Facilitar o upgrade, ao permitir uma nova versão apenas por mudar um único lugar do programa. O programa abaixo é um exemplo de programa mal escrito, pois a constante 5 de `int dataObj[5];` é relacionada com a constante 5 da linha seguinte `for (i = 0 ; i < 5 ; i++)` {. Caso o programa seja alterado e a linha `int dataObj[5];` passe a ser `int dataObj[4];`, o programa passa a estar inadequado, e o seu comportamento é aleatório (pode funcionar perfeitamente, pode não funcionar, pode travar a máquina, etc).

```
// Exemplo de programa mal escrito por ter constantes relacionadas17
void main () {
    int i;
    int dataObj[5];
    // fill dataObj with data
    for (i = 0 ; i < 5 ; i++) {
        dataObj[i] = i + 4;
    }
}
```

A forma correta de escrever esse programa seria como mostrado abaixo. Nesse programa, a dimensão do vetor e a dimensão no qual o vetor é preenchido são relacionadas pela macro `NUM`. Caso seja necessário um upgrade, há apenas uma linha para se modificar.

```
// Exemplo de programa bem escrito por não ter constantes relacionadas
#define NUM 5
void main () {
    int i;
    int dataObj[NUM];
    // fill dataObj with data
    for (i = 0 ; i < NUM ; i++) {
        dataObj[i] = i + 4;
    }
}
```

15.2.5 Modularidade do programa

Um bom programa deve ser “modularizado”, isto é, o programa deve conter muitos “módulos”, que são componentes de programação. O oposto disso é o programa que é todo escrito, de cima a baixo,

¹⁷ fill dataObj with data = preencher objetoDeDados com dados

numa única função, muito longa. Como regra prática, uma função não deve ter em geral muito mais que 30 linhas. Se tiver, provavelmente é mais fácil de ser compreendida se parte dessas 30 linhas transformar-se num “módulo”, isto é, uma função separada.

Outra coisa que se deve evitar é o uso exagerado de loop dentro de loop diretamente. Escrever o código assim dificulta a manutenção do mesmo. Do ponto de vista de legibilidade do código é bastante elegante que o número de níveis de laços dentro de laços (ou if's) seja limitado a cerca de 5 ou 6. Se for necessário mais que isso (digamos 10 níveis), o ideal é criar uma função no meio para “quebrar” a sequência de níveis e facilitar a legibilidade.

Uma outra forma de dizer a mesma coisa é que o programador deve focar a programação mais na biblioteca (conjunto de componentes) e menos na aplicação em si. Os módulos devem ser escritos da forma mais geral possível, de forma que possam ser usados em muitas situações. Assim, esse módulo torna-se um componente de programação. O ideal é que a cada programa feito, o programador aumente seu conhecimento sobre componentes de programação que possui cópia, e que sabe usar.

Caso o programador já tenha componentes de programação (classes, funções, macros, etc.) que estejam sendo usadas por mais de um programa, essas funções devem ser separadas num arquivo separado de fonte (*.cpp) e outro arquivo de header (*.h). A longo prazo, esse arquivo deverá ser a biblioteca pessoal desse programador. Veja o exemplo abaixo. Nesse exemplo é necessário que o projeto contenha os arquivos application.cpp e mylib.cpp. Caso haja uma outra aplicação a ser desenvolvida que use novamente a classe myClass, basta fazer outro arquivo (digamos application2.cpp), usando novamente a mesma classe myClass.

```

////////////////////////////////////
// application.cpp
#include "mylib.h"
void main () {
    myClass a; // myClass is declared in mylib.h
    a.someFunction(); // the prototype of someFunction() is in mylib.h
    // the code of someFunction() is in mylib.cpp
}

////////////////////////////////////
// mylib.h
class myClass { // declaration of myClass
public:
    void someFunction(); // prototype of someFunction
// etc
};

////////////////////////////////////
// mylib.cpp
void myClass::someFunction() {
// code of someFunction
}

```

15.2.6 Uso de nomes elucidativos para identificadores

Os identificadores são nomes de funções, de variáveis, de classes, de macros, de métodos, etc. Esses nomes devem significar o que representam. Dessa forma, o programa fica intrinsecamente legível. Algumas regras sobre isso:

- Nomes curtos que não significam nada (e.g. i, k, etc.), podem ser usados para iteradores inteiros simples.
- Atributos de uma classe (variáveis membro) devem começar com m_.
- Nomes com significado devem ser em inglês (veja a sub-seção abaixo).

15.2.7 Programar em inglês

Sem considerações ideológicas, o fato é que inglês é o idioma internacional. Se não o é para todas as áreas, com certeza é para tecnologia da informação. Portanto, o fato é que os melhores livros, listas de discussão, gurus, páginas web, etc. que falam sobre tecnologia de informação estão em inglês.

Considerando o que já foi dito no sentido da importância de se trocar informações com uma ampla comunidade, de desenvolver e usar bibliotecas, etc. a consequência é que programar pensando em inglês é a forma mais objetiva de se atingir o melhor nível de programação. Nesse idioma será mais fácil obter ajuda, internacional se necessário. Se o seu programa for considerado muito bom, será mais fácil vender o código fonte.

Para quem fala português como nós, pode parecer um aviltamento a recomendação de se programar em inglês. Eu me defendo disso dizendo que a experiência me mostrou que programar em inglês é a forma mais prática de se obter desenvoltura no mercado de tecnologia, que é ultra-competitivo, globalizado e em constante mudança. Eu já tive a oportunidade de examinar programas muito interessantes escritos por programadores que falam alemão, e que falam japonês. Em alguns casos, os programas foram feitos “em inglês”, e em outros casos essa recomendação não foi seguida. Quando a recomendação não era seguida, o programa era útil apenas a nível de execução, mas não a nível de reutilização de código. Se nós programarmos em português, pessoas que não falam português pensarão o mesmo do nosso programa. Na prática, isso significa que o nosso trabalho terá menor valor.

Há que se distinguir muito bem o “programador” (ou “profissional de tecnologia de informação”), do “usuário” que vai simplesmente operar o programa. O profissional de tecnologia de informação é pessoa em geral muito mais treinada e educada que o usuário. Quem tem que falar o idioma nativo é o usuário, não o profissional de tecnologia de informação. O profissional de tecnologia de informação vive num mundo globalizado literalmente, enquanto o usuário geralmente não o faz.

Para dar exemplos concretos: no desenvolvimento do Windows, ou do Linux, ou do Office da Microsoft. Em todos esses casos claramente há um “elemento central” (kernel) do programa, que é feito por profissionais muito qualificados. Com certeza, esses códigos fonte foram escritos em inglês. Mas todos esses programas são usados por usuários no mundo inteiro. Para que os programas possam ser melhor disseminados, são feitas versões deles em vários idiomas. Mas as versões em outros idiomas são meras traduções de tabelas de strings. Não se traduziu o fonte do programa propriamente dito. E estando em inglês, o programa está “bem programado” no sentido que é mais fácil obter ajuda no descobrimento de bugs ou preparação para uma próxima versão.

Programar “pensando em inglês” significa usar nomes de identificadores em inglês, comentários em inglês, nomes de arquivos em inglês, etc. O fonte abaixo está em português, mostrado como exemplo:

```
// em português, má programação !
// mostra_dados.cpp
#include <iostream.h>
#define NUMERO_MAXIMO 5
void main () {
    int i;
    int meus_dados[NUMERO_MAXIMO];
    cout << "Veja os dados" << endl; // sinaliza para o usuário
    // enche meus_dados de dados
    for (i = 0 ; i < NUMERO_MAXIMO ; i++) {
        meus_dados [i] = i + 4;
        cout << meus_dados [i] << endl;
    }
}
```

Esse programa poderia ser melhor escrito assim como abaixo. Repare que apenas os identificadores e comentários foram mudados. A mensagem para o usuário é mantida em português.

```
// in english, good programming !
// show_data.cpp
#include <iostream.h>
#define MAXIMUM_NUMBER 5
void main () {
    int i;
    int my_data[MAXIMUM_NUMBER];
    cout << "Veja os dados" << endl; // send message to user
    // fill my_data with data
    for (i = 0 ; i < MAXIMUM_NUMBER ; i++) {
        my_data [i] = i + 4;
        cout << my_data [i] << endl;
    }
}
```

Capítulo 16) Erros de programação, dicas e truques

Nesse capítulo mostra-se os erros mais comuns de programação, dicas e truques em C++. Os problemas são selecionados de acordo com o nível de dificuldade. 1 é o mais fácil, 10 é o mais difícil.

16.1 Cuidado com o operador , (vírgula)

Dificuldade: 6

Tente completar o programa incompleto abaixo. É preciso que existam 2 funções chamadas f e g. A questão é: quantos argumentos possui a função g ?

```
// add code here
void main () {
    int a=3, b=4;
    f(a,b);
    g((a,b));
}
```

Resposta: a função g possui 1 argumento apenas, esse argumento é o retorno do operador vírgula, que no caso é o conteúdo da variável b. Seja o programa completo de exemplo abaixo.

```
#include <iostream.h>
void f(int x, int y) {
    cout << "x=" << x << " ; y=" << y << endl;
}
void g(int z) {
    cout << "z=" << z << endl;
}
void main () {
    int a=3, b=4;
    f(a,b);
    g((a,b));
}
```

A saída do programa é:

```
x=3 ; y=4
z=4
```

16.2 Acessando atributos privados de outro objeto

O encapsulamento pode ser “contornado” se um método de uma classe faz acessar um atributo privado de um objeto recebido como parâmetro. Geralmente isso é má programação, e deve ser evitado. Um exemplo disso é mostrado abaixo.

```
#include <iostream.h>

class myClass {
    int n; // private member
public:
    void fun(myClass & obj) { obj.n=0; } // another object's private member!
    void operator=(int i) { n = i; }
    friend ostream & operator<<(ostream & s, const myClass & obj) {
        s << obj.n;
        return s;
    }
}
```

```
};

void main() {
    myClass a,b;
    a = 3;
    b = 4;
    cout << "b=" << b << endl;
    a.fun(b); // a changes b's n
    cout << "b=" << b << endl;
}
```

A saída do programa é:

```
b=4
b=0
```

16.3 Entendendo o NaN

NaN significa “not a number”, ou seja, “não é um número”.

// é preciso falar mais aqui.

16.4 Uso de const_cast

Em princípio não se deve converter um parâmetro const para um não const. Mas se isso for necessário, a forma mais elegante de fazê-lo é através de const_cast. O mesmo vale para volatile.

```
#define TYPE int
void main () {
    const TYPE i = 5;
    TYPE* j;
    j = (TYPE*)&i; // explicit form (not elegant)
    j = const_cast<TYPE*>(&i); // Preferred (elegant)

    volatile TYPE k = 0;
    TYPE* u = const_cast<TYPE*>(&k);
}
```

16.5 Passagem por valor de objetos com alocação de memória

Quando um objeto possui memória alocada, esse objeto não pode ser passado por valor para nenhuma função. Caso seja passado, os

```
#include <iostream.h>

// a class that allocs memory
class myClass {
    double *m_d;
public:
    myClass(int i=10) { m_d = new double [i]; }
    ~myClass() { delete [] m_d; }
};

// parameter by value: bug !
void myFun(myClass c) {
}

// correct version: parameter by reference
void myFun2(myClass & c) {
}

void main () {
    myClass a;
    myFun(a);
    myFun2(a);
}
```

```
}
```

16.6 Sobrecarga de insersor e extrator quando se usa namespace

Um programa simples que faz sobrecarga de `operator<<` (insersor) ou `operator>>` (extrator) pode não funcionar se passado para uso de namespace. Por exemplo, veja o programa abaixo.

```
#include <iostream.h>

class myClass {
public:
    int i;
    friend ostream & operator<< (ostream & s , const myClass & obj);
};

ostream & operator<< (ostream & s , const myClass & obj) {
    s << "i=" << obj.i;
    return s;
}

void main () {
    myClass c;
    c.i = 4;
    cout << c << endl;
}
```

Esse programa compila e liga, e gera como resultado o que é mostrado abaixo.

```
i=4
```

Mas se o mesmo programa for adaptado para usar namespace, como mostrado abaixo

```
// programa com erro (de ambiguidade)
#include <iostream>
using namespace std;

class myClass {
public:
    int i;
    friend ostream & operator<< (ostream & s , const myClass & obj);
};

ostream & operator<< (ostream & s , const myClass & obj) {
    s << "i=" << obj.i;
    return s;
}

void main () {
    myClass c;
    c.i = 4;
    cout << c << endl;
}
```

O novo programa, como mostrado acima, compila mas não liga. Durante a compilação, ocorre erro de ambiguidade com o `operator<<`. Esse erro pode ser corrigido como na versão modificada do programa, mostrada abaixo. Nessa versão, o `operator<<` é definido dentro da classe `myClass`.

```
// programa sem erro
#include <iostream>
using namespace std;

class myClass {
public:
    int i;
    friend ostream & operator<< (ostream & s , const myClass & obj) {
        s << "i=" << obj.i;
        return s;
    }
};
```

```
    }  
};  
  
void main () {  
    myClass c;  
    c.i = 4;  
    cout << c << endl;  
}
```

16.7 Inicializando um array estático, membro de uma classe

Em princípio, não é permitido definir um array estático como atributo de uma classe e inicializá-lo dentro da definição da classe.

```
struct myclass {  
    static int array[2] = { 1, 2 }; // error  
}
```

Mas a inicialização pode ser feita após a definição da classe, como mostrado abaixo.

```
struct myclass {  
    static int array[2];  
};  
int myclass::array[2] = { 1, 2 }; // OK
```

16.8 SingleTon

Em alguns casos, pode ser importante que uma determinada classe seja necessariamente intanciada apenas uma vez, isto é, que haja somente uma cópia do objeto no programa. Algumas pessoas chamam esse tipo de classe de “singleTon”. Para que isso seja possível, é necessário que não exista *default constructor* público. Além disso, é preciso existir uma cópia do objeto na classe, e que um método retorne um ponteiro para esse objeto.

É isso que ocorre no exemplo abaixo. A classe MySingleton possui *default constructor* como privado (portanto não há *default constructor* público). Além disso, há alguns atributos (no caso m_i e m_d), e também um método público chamado GetObject, que retorna um ponteiro estático para o objeto, que existe de forma estática dentro do método GetObject.

No programa principal, não se pode instanciar diretamente essa classe (gera erro de compilação). Mas pode-se pegar um ponteiro para o objeto, e acessar os atributos do objeto. Se existir um outro ponteiro que também pegue o ponteiro para o objeto, estará apontando para o mesmo objeto, como pode ser comprovado.

```
#include <iostream.h>  
  
class MySingleton {  
public:  
    static MySingleton * GetObject() {  
        static MySingleton obj; // the actual object  
        return &obj;  
    }  
  
    // general attributes  
    int m_i;  
    double m_d;  
  
private:  
    // default constructor as private not to allow direct  
    // instantiation of this class  
    MySingleton() {};  
};
```

```

void main() {
    // MySingleton a; // ERROR can not directly instantiate a singleton

    // get a pointer to singleton object
    MySingleton *pObject = MySingleton::GetObject();

    // place some data to singleton attributes
    pObject->m_i = 3;
    pObject->m_d = 4.4;

    // get another pointer to the same object
    MySingleton *pAnother = MySingleton::GetObject();

    // show attributes of object
    cout << "i=" << pAnother->m_i << endl;
    cout << "d=" << pAnother->m_d << endl;
}

```

A saída do programa é como mostrado abaixo.

```

i=3
d=4.4

```

16.9 Slicing in C++

Slicing (fatiamento) é o fenômeno de comportamento indesejado de funções virtuais no caso de se perder informação do atributo de um objeto derivado a partir da passagem de parâmetro para uma função que tem como argumento o objeto base.

Seja o programa abaixo. O programa compila e linka ? Resposta: sim. Desde que a classe Circle é derivada de Shape, o compilador gera um *default assignment operator* (operador de atribuição implícito) e irá copiar os campos da classe base (ou seja, os atributos comuns entre a classe base e a classe derivada). Como o atributo m_iRadius pertence a classe derivada, ele não será copiado pelo default assignment operator. Além disso, não há acesso a esse atributo pelo objeto da classe derivada, na função globalFunc.

```

#include <iostream.h>

class Shape {
public:
    Shape() { };
    virtual void draw() {
        cout << "do drawing of Shape (base class)." << endl;
    };
};

class Circle : public Shape {
private:
    int m_iRadius;
public:
    Circle(int iRadius){ m_iRadius = iRadius; }
    virtual void draw() {
        cout << "do drawing of Circle (derived class)."
            << "The radius is " << m_iRadius << endl;
    }
    int GetRadius(){ return m_iRadius; }
};

// a global function, and the base argument is passed by value
void globalFuncByValue(Shape shape) {
    cout << "(by value) do something with Shape" << endl;
    Circle *c = (Circle*)&shape;
    c->draw();
}

```

```
// a global function, and the base argument is passed by pointer
void globalFuncByPointer(Shape * shape) {
    cout << "(by pointer) do something with Shape" << endl;
    Circle *c = (Circle*)shape;
    c->draw();
}

void main () {
    Circle circle(2);
    cout << "=== Circle draw" << endl;
    // call the derived class method directly
    circle.draw();

    cout << "=== by value" << endl;
    // pass a derived object as parameter,
    // when function expects a base object
    // slicing occurs
    globalFuncByValue(circle);

    cout << "=== by pointer" << endl;
    // pass a pointer to derived object as parameter,
    // when function expects a pointer to base object
    // slicing does not occur
    globalFuncByPointer(&circle);
}
```

A saída do programa é como mostrado abaixo.

```
=== Circle draw
do drawing of Circle (derived class). The radius is 2
=== by value
(by value) do something with Shape
do drawing of Shape (base class).
=== by pointer
(by pointer) do something with Shape
do drawing of Circle (derived class). The radius is 2
```

No caso, há comportamento indesejado na função `globalFunc`, pois o `Circle` está sendo traçado como um `Shape`.

O slicing é comum em tratamento de exceções (*exception handling*), quando exceções são capturadas usando-se a classe base.

```
class CException {};
class CMemoryException : public CException {};
class CFileException: public CException{};

try{ /*do something silly*/ }
catch(CException exception) { /*handle exception*/ }
```

Para se evitar o slicing, faça funções globais que chamam ponteiros para classe base (como em `globalFunc2`), e não para objetos da classe base (como em `globalFunc`). Dentro da função, mascare o ponteiro para a classe derivada, como mostrado.

16.10 Dicas de uso de parâmetro implícito

Uma função com argumento (parâmetro) implícito (*default argument*) não pode ter o valor implícito redefinido no mesmo escopo. Por isso ocorre o ERROR 1 no programa abaixo. Mas é possível redefinir o parâmetro implícito desde que seja em outro escopo, como ocorre no OK 1 do programa abaixo.

Pode-se também desabilitar o parâmetro implícito, como ocorre no OK 2 do programa abaixo. Após desabilitar o parâmetro implícito de uma função, a linha marcada com ERROR 2 não compila.

Além disso, pode-se redefinir uma função que não tem parâmetro implícito para que tenha, como ocorre em OK 3 no programa abaixo.

```
#include <iostream.h>

void foo_default_parameter(int i=1) {
    cout << i << endl;
}

// ERROR 1. Can not redefine default parameter in the same scope
void foo_default_parameter(int i=3);

void fun1() {
    void foo_default_parameter(int i=3); // OK 1. redefinition of default parameter
    foo_default_parameter(); // calling function with default parameter 3
}

void fun2() {
    void foo_default_parameter(int); // OK 2. disable default parameter
    foo_default_parameter(10); // ok
    foo_default_parameter(); // ERROR 2. because default parameter is disabled
}

void foo_no_default_parameter(int i) {
    cout << i << endl;
}

void fun3() {
    void foo_no_default_parameter(int=4); // OK 3. enable default parameter
    foo_no_default_parameter(14); // ok
    foo_no_default_parameter(); // calling function with default parameter 4
}

void main () {
    fun1();
    fun2();
    fun3();
}
```

A saída do programa é como mostrado abaixo.

```
3
10
14
4
```

Capítulo 17) Incompatibilidades entre compiladores C++

17.1 Visual C++ 6.0 SP5

17.1.1 for não isola escopo

A forma correta de se compilar o for é mapea-lo de acordo com o código abaixo (veja página 88).

```
for (<inic> ; <test> ; <set>)  
    <statement>;
```

Mapeado para:

```
{  
    <inic>;  
    while (<test>) {  
        <statement>;  
        <set>;  
    }  
}
```

Mas no Visual C++, o mapeamento é feito sem o isolamento de escopo, ou seja, conforma o código abaixo.

```
<inic>;  
while (<test>) {  
    <statement>;  
    <set>;  
}
```

Portanto, o código abaixo, que é perfeitamente legal em C++, não compila em Visual C++. Devido ao mapeamento errado, a variável “i” é declarada 2 vezes, o que é ilegal.

```
void main () {  
for (int i=0 ; i < 10 ; i++)  
    cout << "i=" << i << endl;  
for (int i=0 ; i < 10 ; i++)    // error, i redefinition  
    cout << "i=" << i << endl;  
}
```

17.1.2 Comportamento do ifstream quando o arquivo não existe

Quando o visual C++ tenta abrir como leitura um arquivo que não existe, é criado um arquivo com conteúdo zero (se o usuário tiver direito de escrita no diretório). Como o arquivo é criado, o ponteiro para o arquivo retorna válido, isto é, o programa não entra no procedimento para o caso de o arquivo não existir. Isso é um erro do compilador.

```
#include <fstream.h>  
void main () {  
    ifstream myFile("non_existent_file.txt");  
    if (!myFile) {  
        // handle case when file does not exist  
    }  
}
```

Pode-se contornar esse erro primeiro testando o arquivo, usando o flag ios::nocreate, que força a não criação do arquivo.

```
#include <fstream.h>
void main () {
    ifstream myFile("non_existent_file.txt",ios::nocreate);
    if (!myFile) {
        // handle case when file does not exist
    }
}
```

17.1.3 ios::nocreate não existe quando fstream é usado com namespace std

No Visual C++, quando um arquivo é criado com fstream e namespace, o ios::nocreate não compila.

```
#include <fstream>
using namespace std;
void main () {
    ifstream myFile("non_existent_file.txt",ios::nocreate); // error
    if (!myFile) {
        // handle case when file does not exist
    }
}
```

17.1.4 Compilador proíbe inicialização de variáveis membro estáticas diretamente

```
static const int k = 5;          // global, ok

struct A {
    static const int i = 5;      // member, error in VC
};
```

17.1.5 “Namespace lookup” não funciona em funções com argumento

O compilador deveria deduzir o namespace (“namespace lookup”) de uma função com argumento no mesmo namespace, mas não o faz em Visual C++.

```
namespace myNamespace {
    struct myStruct {};
    void myFun(myStruct obj);
}

void g() {
    myNamespace::myStruct a;
    myFun(a);          // ERROR: 'myFun': undeclared identifier
    myNamespace::myFun(a); // OK
}
```

17.1.6 Encadeamento de “using” não funciona

```
void f();
namespace N {
    using ::f;
}
void g()
{
    using N::f; // C2873: 'f': the symbol cannot be used in a using-declaration
}
```

17.1.7 Erro em instanciamento explícito de funções com template

```
// wrong only for Visual C++ (6.0 SP4)
#include <iostream.h>
template<class T>
void f() {
    cout << sizeof(T) << endl;
}

void main() {
    f<double>();          // correct output: "8"   VC output "1"
    f<char>();            // correct output: "1"   VC output "1"
```

}

17.2 Borland C++ Builder (versão 5.0, build 12.34)

17.2.1 Inabilidade de distinguir namespace global de local

```
// error in Borland C++, Visual C++. OK for gnu C++
namespace myNamespace {
    int myFun() {return 1;};
}
using myNamespace::myFun;
using namespace myNamespace;
void main() {
    int i = myFun();    // Ambiguous overload
}
```

17.2.2 Erro na dedução de funções com template a partir de argumentos const

A dedução de funções com template deveriam omitir a existência de “const” no tipo. O exemplo abaixo mostra que isso não ocorre no compilador Borland C++.

```
template<class T>
void myFun(T x) {
    x = 1; // works
    T y = 1;
    y = 2; // error
}

void main() {
    const int i = 1;
    myFun(i); // error
    int j = 1;
    myFun(j); // OK
}
```

17.2.3 Erro na conversão de “const char *” para “std::string”

A conversão implícita de argumentos tipo “const char *” para “std::string” falha quando funções em template são instanciadas explicitamente.

```
#include <string>
using namespace std;
template<class T>
void myFun(string & s) {};
void main() {
    myFun<double>("hello"); // error
}
```

17.3 Gnu C++ (versão 2.91.66)

17.3.1 Valor do elemento em vector<double> de STL

Em visual C++, pode-se escrever o programa abaixo.

```
// in Visual C++
#include <vector>
using namespace std;
void main () {
    int val = 1;
    vector<double> vectorObject(5,val); // 5 val, OK
    vector<double> vectorObject2(5); // OK
}
```

em gnu C++ (g++), o compilador não aceita o val.

```
// in gnu C++
```

```
#include <vector>
using namespace std;
void main () {
    int val = 1;
    vector<double> vectorObject(5,val); // error
    vector<double> vectorObject2(5); // OK
}
```

É interessante que o problema somente ocorre quando o tipo é `vector<double>`, mas não ocorre com o tipo `vector<int>`.

Capítulo 18) Glossário

Formato livre (*free format*) - O termo que significa que o compilador observa o texto fonte sem atenção a posicionamento relativo dos caracteres. Em FORTRAN, que por exemplo não possui formato livre, o código fonte só pode ser escrito a partir da coluna 7 do texto e o return no final da linha significa que o comando acabou. A maioria das linguagens modernas como C e C++ podem ser escritas em formato livre o que significa também que no lugar de 1 espaço podem haver qualquer número de espaços, returns e tabs sem qualquer alteração na compilação.

Identificador (*identifier*) - O termo se refere as *strings* que o programador coloca no programa para dar nome a suas variáveis, funções, tipos, etc. Em C e C++, que são linguagens fortemente tipadas, todos identificadores devem ser declarados antes de serem usados.

MDI - Multiple Document Interface - interface de documentos múltiplos

Recurso Windows - refere-se a menus, caixas de diálogo (dialog box), tabelas de strings, aceleradores, etc.

Capítulo 19) Bibliografia

[1] Link com referências sobre rpm, <http://rpm.redhat.com/RPM-HOWTO/>

[2] Libtool - ferramenta da Gnu para gerenciamento de bibliotecas -
<http://www.gnu.org/software/libtool/>

[3] VBMcgi - biblioteca para programação CGI em C++ - www.vbmcgi.org

// é preciso melhorar a bibliografia. Isso fica para a próxima versão

Capítulo 20) Índice remissivo

// é preciso melhorar o índice remissivo. Isso fica para a próxima versão

Biblioteca	Componente de programação, 96
(como conjunto de componentes de programação), 96	Porta Paralela, 182
Complexidade de um algoritmo	Porta Serial, 182
Notação O , 167	string
	classe de, 190

O conhecimento é como a luz. É sem peso e intangível. Pode facilmente viajar o mundo, iluminando a vida das pessoas em todos os lugares.

O conhecimento quando compartilhado não se divide, mas se multiplica. O conhecimento quando usado não se gasta, mas se renova.